ETAS ASCET V6.4

Getting Started

www.etas.com

ASCET V6.4 I Getting Started R09 EN I 06.2024

Contents

1 Introduction 6
1.1 Intended Use 6
1.2 Target Group 6
1.3 Classification of Safety Messages 7
1.4 Safety Information 7
1.5 Data Protection 8
1.6 Data and Information Security 8
1.6.1 Data and Storage Locations 8
1.6.2 Technical and Organizational Measures 10
2 About ASCET 11
2.1 Features at a Glance 12
2.1.1 ASCET-MD 12
2.1.2 ASCET-RP 12
2.1.3 ASCET-SE 12
2.1.4 ASCET-SCM 13
2.1.5 ASCET-DIFF 13
2.2 Supporting Functions 13
2.2.1 Monitor Window 13
2.2.2 Keyboard Assignment 13
2.3 Finding Out More 13
3 Embedded Automotive Control Software Development with ASCET 15
3.1 Model-Based Design 16
3.1.1 Control Algorithm Development 17
3.1.2 Rapid Prototyping 20
3.1.3 Implementation and ECU Integration of Control Algorithms 22
3.1.4 Reuse of Control Algorithm in Different Kinds of Projects 26
3.1.5 Testing Technical System Architecture in the Lab 28
3.1.6 Testing and Honing of Technical System Architecture in the Vehicle 28
3.2 Using ASCET in a Production Environment 29
3.3 Summary 31
4 Tutorial 32
4.1 Preparations 32
4.1.1 Tutorial Database or Workspace 33
4.1.2 Summary 34
4.2 Lesson 1: A Simple Block Diagram 35
4.2.1 Preparatory Steps 35
4.2.2 Specifying a Class 36
4.2.3 Summary 43
4.3 Lesson 2: Experimenting with Components 43

4.3.1 Starting the Experimentation Environment . . . 43
4.3.2 Setting up the Experimentation Environment . . . 44
4.3.3 Using the Experimentation Environment . . . 47
4.3.4 Summary . . . 49

4.4 Lesson 3: To Specify a Reusable Component . . . 49
4.4.1 Creating the Diagram . . . 50
4.4.2 Experimenting with the Integrator . . . 55
4.4.3 Summary . . . 57

4.5 Lesson 4: A Practical Example – Controller . . . 57
4.5.1 Specifying the Controller . . . 58
4.5.2 Experimenting with the Controller . . . 60
4.5.3 Project . . . 61
4.5.4 Setting Up the Project . . . 61
4.5.5 Experimenting with the Project . . . 63
4.5.6 Summary . . . 64

4.6 Lesson 5: Extending the Project . . . 64
4.6.1 Specifying the Signal Converter . . . 64
4.6.2 If...Then...Else in the Signal Converter . . . 66
4.6.3 Experimenting with the Signal Converter . . . 68
4.6.4 Integrating the Signal Converter into the Project . . . 70
4.6.5 Summary . . . 73

4.7 Lesson 6: Modeling a Continuous Time System . . . 73
4.7.1 Motion Equation . . . 73
4.7.2 Model Design . . . 74
4.7.3 Summary . . . 77

4.8 Lesson 7: Process Model . . . 78
4.8.1 Specifying the Process Model . . . 78
4.8.2 Integrating the Process Model . . . 81
4.8.3 Online Experiment with the Process Model . . . 83
4.8.4 Summary . . . 86

4.9 Lesson 8: State Machines . . . 86
4.9.1 Specifying the State Machine . . . 86
4.9.2 How a State Machine Works . . . 91
4.9.3 Experimenting with the State Machine . . . 92
4.9.4 Integrating the State Machine in the Controller . . . 93
4.9.5 Summary . . . 95

4.10 Lesson 9: Hierarchical State Machines . . . 95
4.10.1 Specifying the State Machine . . . 96
4.10.2 Experimenting with the Hierarchical State Machine . . . 100
4.10.3 How Hierarchical State Machines Work . . . 101
4.10.4 Summary . . . 101

5 Support Function for Feedback to ETAS in Case of Errors ... 102

6 Troubleshooting General Problems ... 104

6.1 Network Adapter Cannot be Selected via Network Manager ... 104

6.2 Search for Ethernet Hardware Fails ... 104

7 Tool Classification for ISO26262 ... 109

8 Contact Information ... 112

Glossary ... 113

Figures ... 124

Index ... 125

1 Introduction

In this chapter, you can find information about the intended use, the addressed target group, and information about safety and privacy related topics.

Please adhere to the ETAS Safety Advice (accessible via **Help > Product Disclaimer**) and to the safety information given in the user documentation.

ETAS GmbH cannot be made liable for damage which is caused by incorrect use and not adhering to the safety information.

1.1 Intended Use

The ASCET tools support model-based software development. In model-based development, you construct an executable specification – the model – of your system and establish its properties through simulation and testing in early stages of development. When a model behaves as required, it can be converted automatically to production-quality code.

ASCET provides a multi-paradigm modeling framework, providing integrated support for a number of different modeling notations. These modeling notations abstract from low-level details, separating the concerns of what the system software must do from how it is realized in code executing in the ECU. ASCET can also interface directly with C code as a "low-level" specification language.

ASCET provides a systematic way to augment the high-level specification (referred to as the *physical model*) with the necessary information for target implementation (referred to as the *implementation model*). The implementation model covers the low-level details required to make the model run on target hardware.

The physical and implementation models are clearly separated in ASCET so that the design specification is not corrupted with implementation details that may change from project to project. Maintaining this separation also allows ASCET to support multiple implementation models for a single physical model, keeping the number of model variants low.

1.2 Target Group

This manual addresses qualified personnel working in the fields of automobile control unit development and calibration. Specialized knowledge in the areas of measurement and control unit technology is required.

ASCET users should be familiar with the Microsoft Windows® 10 operating system. Knowledge of a programming language, preferably ANSI-C, can be helpful when using ASCET.

1.3 Classification of Safety Messages

Safety messages warn of dangers that can lead to personal injury or damage to property:

DANGER

DANGER indicates a hazardous situation that, if not avoided, will result in death or serious injury.

WARNING

WARNING indicates a hazardous situation that, if not avoided, could result in death or serious injury.

CAUTION

CAUTION indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.

NOTICE

NOTICE indicates a situation that, if not avoided, could result in damage to property.

1.4 Safety Information

Observe the following safety information when using the NVRAM capabilities of the ASCET-RP or ASCET-SE targets, to avoid injury to yourself and others as well as damage to property:

> **WARNING**
>
> **Harm or property damage due to unpredictable behavior of vehicle or test bench**
>
> Wrongly initialized NVRAM variables (NV variables) can lead to unpredictable behavior of a vehicle or a test bench. This behavior can cause harm or property damage.
>
> ASCET projects that use the NVRAM possibilities of ASCET-RP targets 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).

Adhere to the ETAS Safety Advice and the safety information given in the online help and user guides. You can open the ETAS Safety Advice from the main ASCET window with **Help › Product Disclaimer**. A PDF version is available on the installation medium: `Documentation\ETAS Safety Advice.pdf`

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

Further safety advice for this ETAS product is available in the ASCET V6.4 safety manual, available at ETAS upon request.

1.5 Data Protection

If the product contains functions that process personal data, legal requirements of data protection and data privacy laws shall be complied with by the customer. As the data controller, the customer usually designs subsequent processing. Therefore, he must check if the protective measures are sufficient.

1.6 Data and Information Security

To securely handle data in the context of this product, see the next sections about data and storage locations as well as technical and organizational measures.

1.6.1 Data and Storage Locations

The following sections give information about data and their respective storage locations for various use cases.

License Management

When using the ETAS License Manager in combination with user-based licenses that are managed on the FNP license server within the customer's network, the following data are stored for license management purposes:

Data

- Communication data: IP address
- User data: Windows user ID

Storage location

- FNP license server log files on the customer network

When using the ETAS License Manager in combination with host-based licenses that are provided as FNE machine-based licenses, the following data are stored for license management purposes:

Data

- Activation data: Activation ID

  Used only for license activation, but not continuously during license usage

Storage location

- FNE trusted storage

  `C:\ProgramData\ETAS\FlexNet\fne\license\ts`

Problem Report

When an error occurs, ASCET offers to send an error report to ETAS for troubleshooting. ETAS uses the personal information to have a contact person in case of system errors.

The problem report may contain the following personal data or data category:

Data

- Communication data: IP address
- User data: Windows user ID, user name

Storage location:

- `EtasLogFiles`*`<index number>`*`.zip` in the ETAS-specific log files directory

Additionally to the problem information that is entered by the users themselves, ASCET collects the available product-related log files in a zip archive to support the bug fixing process at ETAS. The zip file is named according to the pattern `EtasLogFiles`*`<index number>`*`.zip`. See also chapter 5 "Support Function for Feedback to ETAS in Case of Errors" on page 102.

All ETAS-related log files in the ETAS-specific log files directory and the zip archives created by the Problem Report feature can be removed after closing all ETAS applications if they are no longer needed.

1.6.2 Technical and Organizational Measures

We recommend that your IT department takes appropriate technical and organizational measures, such as classic theft protection and access protection to hardware and software.

Locations for Generated Files

Names and paths of files generated by ASCET may contain personal data, if they refer to the current user's personal directory or subdirectories (e.g., `C:\Users\<UserId>\Documents\...`).

If you do not want personal information to be included in the generated files, make sure of the following:

- The workspace of the product points to a directory without personal reference.
- All settings in the product (accessed via the menu function **Tools > Options** in the product) refer to directories and file names without personal reference.
- All project settings in the projects (accessed via the menu function **File > Properties** in the ASCET project editor) refer to directories and file names without personal reference.
- Windows environment variables (such as the temporary directory) refer to directories without personal reference because these environment variables are used by the product.

In this case, please also make sure that the users of this product have read and write access to the newly set directories.

2 About ASCET

The ASCET tools support model-based software development. In model-based development, you construct an executable specification – the model – of your system and establish its properties through simulation and testing in early stages of development. When the model behaves as required, it can be converted automatically to production-quality code.

The key advantage of model-based development is that the software system can be designed by domain experts, using domain-specific notions, independently from knowing any details how it will be realized by an implementation. You can learn more about model-based design in section 3.1.

ASCET provides a multi-paradigm modeling framework, providing integrated support for a number of different modeling notations, each providing support for a different type of modeling need:

- Block diagrams (occasionally abbreviated to BD) – to model continuous control systems
- State machines (occasionally abbreviated to SM) – to model event-triggered systems
- Conditional and Boolean tables – to model complex mathematical expressions
- Embedded Software Description Language (ESDL) – a textual modeling language

The modeling languages abstract from low-level details, separating the concerns of what the system software must do from how it is realized in code executing in the ECU. ASCET can also interface directly with C code as a "low-level" specification language.

ASCET provides a systematic way to augment the high-level specification (referred to as the "physical model" in ASCET) with the necessary information for target implementation (referred to as the "implementation model" in ASCET). The implementation model covers the low-level details required to make the model run on target hardware, including conversion between real-number arithmetic on the model and fixed-point arithmetic on the target, interfacing to interpolation routines for maps and curves, integration of optimized arithmetic service implementations, integration with a real-time operating system for run-time scheduling, memory mapping for embedded devices etc.

The physical and implementation models are clearly separated in ASCET so that the design specification is not corrupted with implementation details that may change from project to project. Maintaining this separation also allows ASCET to support multiple implementation models, each containing different implementation characteristics, for a single physical model, keeping the number of model variants low during the overall life cycle of a software function.

2.1 Features at a Glance

The ASCET product family consists of a number of products that provide interfaces to simulation processors, third-party software packages and for remote access to ASCET. The following products are available for the current version of ASCET.

Various additional customer-specific products can be integrated in ASCET. More detailed information is available upon request.

2.1.1 ASCET-MD

- Model-based development of automotive software, including AUTOSAR software components
- Hierarchical, object-based modeling architecture
- Support for systematic conversion from real-number to fixed-point arithmetic
- Creation of custom block set libraries
- Import and export of AUTOSAR software component descriptions
- Support for calibration parameters, including maps and curves
- Automatic documentation generation for archiving the design model
- PC-hosted, offline simulation of application software

2.1.2 ASCET-RP

- Hardware configuration for support for the ES910 experimental target
- Support for hardware-in-the loop simulation and rapid prototyping applications

2.1.3 ASCET-SE

- Automatic generation of fully modular, high-performance, low-overhead, production-ready MISRA-C:2004 compliant C code that is easily traceable to the parent model
- Integration of 3rd-party interpolation and arithmetic service routines.
- Configuration of memory sections and systematic application of compiler intrinsic in generated code to support embedded microcontrollers
- Platform integration configuration to interface ASCET code with OSEK operating systems (e.g. RTA-OSEK) or with an AUTOSAR RTE (e.g. RTA-RTE) and ensure correct use of platform concurrency control mechanisms
- "Additional programmer" mode to generate source code and data for integration with a 3rd-party build environment
- "Integration platform" mode to provide "one-click-build" of an ECU executable image for a wide range of compilers and microcontrollers, with full user-side customization
- Generation of ASAM-MCD-2MC data description files for calibration tools (e.g. INCA)
- Generation of AUTOSAR XML code

2.1.4 ASCET-SCM

- Interaction with 3rd-party version management tools from within ASCET

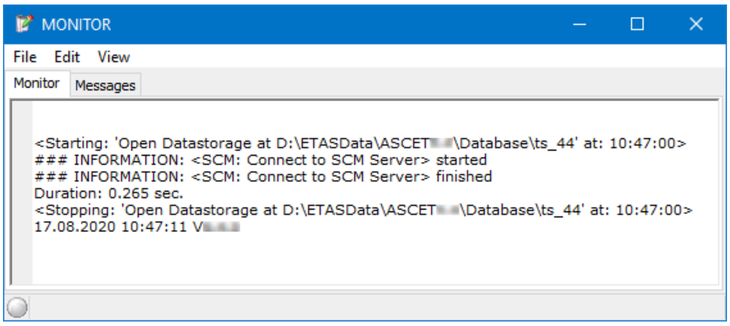
2.1.5 ASCET-DIFF

- Graphical and textual comparison of ASCET models

2.2 Supporting Functions

2.2.1 Monitor Window

The monitor window (see the ASCET online help) is used to log the working steps performed by ASCET. All actions, including errors and notifications, are logged. As soon as an event is logged, the monitor window is displayed in the foreground.



In addition to displaying information, the monitor window also provides the functionality of an editor.

- The display field in the "Monitor" tab of the monitor window can be freely edited. This way, your own notes and comments can be added to the ASCET messages.
- The ASCET messages can be saved as text files along with your comments.
- Other ASCET text files already stored can be loaded so that you can compare specific working steps.

2.2.2 Keyboard Assignment

You can display an overview of the keyboard commands currently used at any time by pressing <CTRL> + <F1>.

For more details, see the ASCET online help, section "Operation Hints".

2.3 Finding Out More

If not specified otherwise during installation, the following PDF manuals are available in the `ETAS\ETASManuals` folder after installing ASCET and ASCET-MD:

- ASCET Getting Started (this manual; `ASCET V6.4 Getting Started.pdf`)
- ASCET Installation Guide (`ASCET V6.4 Installation.pdf`)
- ASCET Icon Reference Guide (`ASCET V6.4 Icon Reference Guide.pdf`)

- ASCET AUTOSAR User Guide (`ASCET V6.4 AUTOSAR User Guide.pdf`)
- AUTOSAR to ASCET Importer User Guide (`ASCET V6.4 AUTOSAR To ASCET Converter User Guide.pdf`)

> **NOTE**
>
> The cooperation of ASCET and AUTOSAR requires the installation of the ASCET-SE target *`ANSI-C`*.

When you install ASCET-RP, ASCET-SE, ASCET-SCM, or ASCET-DIFF, further documentation becomes available:

- ASCET-RP
  - user guide (`ASCET-RP V6.4 User Guide.pdf`)
  - separate online help; accessible via the **Help** menu and ‹F1› in the hardware configurator
- ASCET-SE
  - ASCET-SE user guide (`ASCET-SE V6.4 User Guide.pdf`)
  - EHOOKS Add-On user guide (`ASCET-SE V6.4 EHOOKS Add On User Guide.pdf`)
- ASCET-SCM
  - online help (integrated in the main ASCET online help)
- ASCET-DIFF
  - ASCET-DIFF installation guide
  - separate online help; accessible via the **Help** menu and ‹F1› in the ASCET-DIFF windows

ETAS offers efficient trainings in the use of ASCET in order to provide an even more thorough knowledge of ASCET, especially if users have to gain a comprehensive insight in the functionality of ASCET in a very short period of time.

3 Embedded Automotive Control Software Development with ASCET

Embedded automotive software development is an interdisciplinary task requiring cooperation between the different vehicle domains (infotainment, chassis, body, powertrain) as well as between different companies, i.e. the vehicle manufacturer and the supplier. Furthermore, embedded automotive software is an integral part of a mechanical subsystem which means that it

- implements control algorithms which read data from sensors, and calculate control values which are sent to an actuator.
- runs typically in so-called electronic control units (ECUs for short), employing one or more microcontrollers and additional electrics and electronics.
- will normally not be changed during the lifetime of a vehicle.
- has to obey all requirements with respect to safety and reliability of the mechanical subsystems.

As a result, creating a common understanding of the functionality which has to be implemented in software is the basis for a seamless integration and non-functional optimizations, e.g. resource consumption. The latter point becomes apparent if one keeps in mind that ECUs are produced in large quantity. Small cost reduction of a single ECU may hence result in significant savings of the series' overall costs. For example, saving of memory resulting in a cheaper derivate of a microcontroller will lessen the overall cost even though the cost for a single ECU changes only marginally.

A graphical model of the function frequently serves as the basis for the common understanding described above. On the one hand, the graphical model is more abstract than embedded C code, while on the other it is formal, i.e. unambiguous without leeway for interpretations compared to a non-formal textual specification. It can be executed on a computer in a simulation. It can be experienced in a vehicle at an early point in time by means of rapid prototyping. For short, a graphical model of a function serves as digital specification.

Using automatic code generation, graphical functional models can be transformed to embedded automotive software. To accomplish this, functional models must be enhanced by adding dedicated design information that includes non-functional product properties like safety and resource consumption measures.

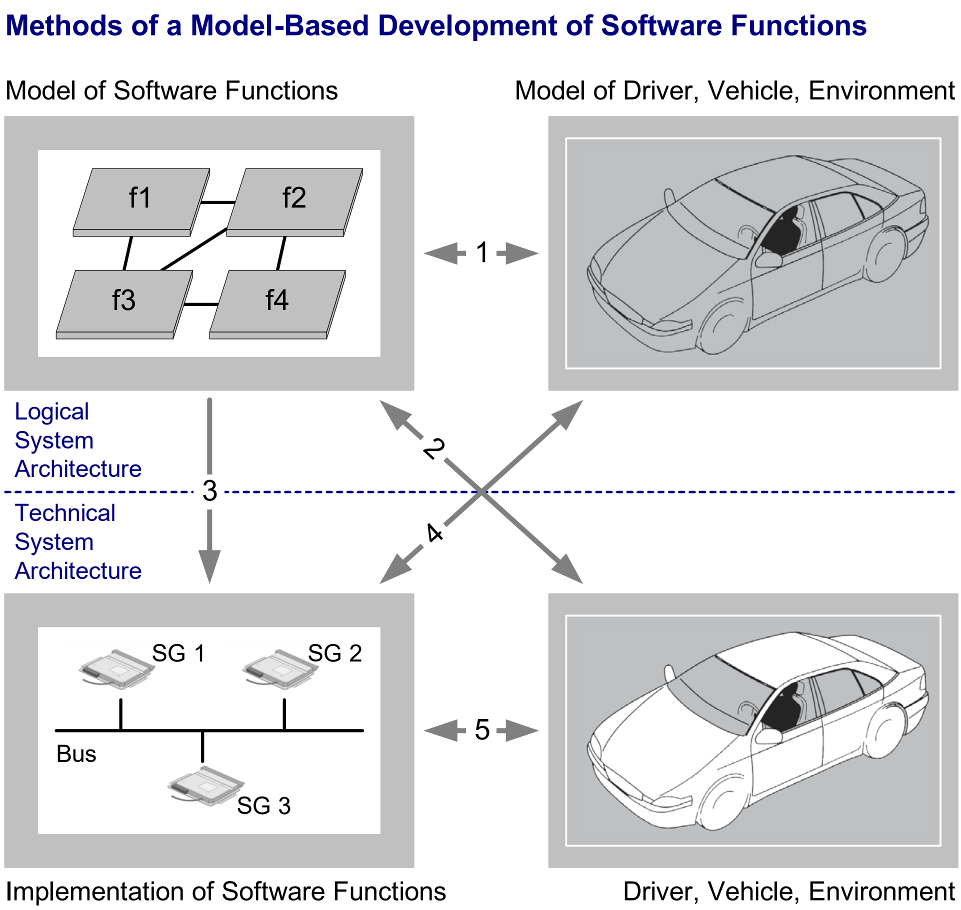
The operating environment of ECUs can be simulated by means of hardware-in-the-loop test systems (HiL for short) which facilitate early testing of ECUs in the laboratory. HiL-testing of ECUs offers a greater flexibility in generating test-cases than in-vehicle tests typically provide.

The calibration of embedded automotive software often can be finalized only at some point toward the end of the development process. In many cases, this procedure is carried out in the vehicle with all systems (i.e. mechanical systems embedding automotive software of all domains) running, and requires support of dedicated tools and methods, which have also to be considered during the generation of the software.

Section 3.1 describes in detail the stages of model-based design and explains the abstraction mechanisms employed in ASCET to create a graphical model of a function. Section 3.2 shows how ASCET models can be used in an ECU production development environment while section 3.3 summarizes the major topics.

3.1 Model-Based Design

The development of embedded automotive control software is characterized by several development steps which can be summarized by using the V-model. One starts with the analysis and design of the logical system architecture, i.e. defines the control functions, proceeds with defining the technical architecture, which is a set of networked ECUs, and then proceeds with software implementation on an ECU. The software components will be integrated and tested, then the ECU is integrated in the vehicle network and, last but not least, the system running the implemented functions is fine-tuned by means of calibration. However, this is not a top-down process, but requires early feedback by means of simulation and rapid-prototyping.



1. Modeling and simulation of software functions as well as vehicle, driver, and environment
2. Rapid prototyping of software functions in the real vehicle
3. Design and implementation of software functions
4. Integration and test of software functions with lab vehicles and test benches
5. Test and calibration of software functions in the vehicle

Fig. 3-1 Model-Based Development of a Software Function

3.1.1 Control Algorithm Development

At first, control algorithms are developed. This is mainly a control-engineering task. It starts with the dynamic analysis of the system to be controlled, i.e. the plant. A plant-model is a model of the vehicle (including the sensors and actuators), its environment (e.g. the road conditions), and the driver. Typically, only subsystems of the vehicle are considered in special scenarios like the engine with the power-train and the driver, or the chassis with the road-conditions. These models can be either analytical, such as an analytically solved differential equation, or a simulation model, i.e. a differential equation to be solved numerically. In practice, a plant-model is often a mixture of both.

Then, according to some quality criteria, the control law is applied. Control laws compensate the dynamics of a plant. There are a lot of rules to find good control laws. Automotive control algorithms very often combine closed-loop control laws with open-loop control strategies. The latter are often automatons or switching constructs. This means that control algorithms are hybrid systems from a system-theory point of view. Typically, the control law consists of set-point generating function with controlling and monitoring functions, all realized by software (see section 3.1.1.1 "Software Realization of Control Algorithms").

The first step is to design a control algorithm for a vehicle subsystem which is represented as a simulation model. Both the control algorithm and the plant model are running on a computer. The plant is typically realized as a quasi-continuous-time model while the control algorithm is modeled in discrete-time. The value range of both models is continuous, i.e. the state variables and parameters of the control algorithm and the plant are realized as floating-point variables in the simulation code. This model is depicted in the upper part of Fig. 3-1 on page 16. The logical system architecture represents the control algorithm which is coupled to a model of the driver, the vehicle & the environment. The arrow labeled 1 represents the control algorithm design step. Control algorithm modeling is based on the use of shared signals. This means that one component shares the signal in a provide role while other components share the signal in a require role.

3.1.1.1 Software Realization of Control Algorithms

Control algorithms are hybrid systems, i.e. a mixture of open- and closed-loop systems where the open-loop parts are quite often non-linear, discrete systems, for example finite-state-machines. If the control algorithms run on a microcontroller, they have to be transformed in a sequential programming language, e.g. C. The easiest way for a realization of the control algorithm is to construct a main-loop, which is triggered by an interrupt, and to call several subroutines, which contain the sequential program. Data exchange between the subroutines is performed by global variables. Triggering the main-loop by interrupts realizes a reoccurring execution of the sequential-program. If the interrupt is a timing interrupt, the main-loop realizes a sampled system.

This kind of straight-forward realization of control algorithms in software runs into its limits if multi-rate systems are considered, i.e. systems having different sample rates, which are realized by several tasks instead of one main-loop. These multi-tasking systems require a proper exchange of signal data between the tasks. Furthermore, it is quite difficult on C code level to distinguish between state variables, parameters, input and output signals. Realization of control algorithms in ASCET closes the gap between the control-engineering view and the implementation view of the control algorithm. Instead of simply using variables and subroutines, ASCET provides the following control algorithm modeling constructs:

- Modules
- Classes
- Projects

Combinations of these constructs allow the construction and execution of complex control algorithms on several targets. Targets are a PC, a rapid-prototyping system or a microcontroller. Execution is performed by first transforming the ASCET model to C code and afterwards transforming the C code to executable code on the respective target. All modeling constructs are maintained in a database or workspace.

3.1.1.2 Modules

Modules provide means for sequential statements, for (state) variables, parameters, input and output signals. Sequential statements are realized in a block diagram editor (BDE) by variables with sequence calls. These sequence calls assign the result of an expression to the variable. An alternative to the BDE in ASCET to realize statements is the ESDL programming language. Sequential statements can be grouped to processes. Processes represent subroutines.

Input signals are modeled as so-called *receive messages*. Expressions can read from receive messages and use the actual value of that message for further calculations. Output signals are modeled as so-called *send messages*. The result of an expression can be assigned (written) to a send message. In the block diagram editor, the assignment to a message is realized by a sequence call similar to variables.

Parameters have an own representation. Their value can only be read by an expression, but assignments are not allowed.

To summarize, a module consists of send and receive messages for data exchange with other modules. It has several processes which cluster sequential statements. Besides messages, a module contains variables and parameters. Receive-message reading can be shared by the processes of the modules, while message-writing requires disjoint access by the processes. There might be messages which are only exchanged between processes within a module. These dedicated messages are called *send-receive messages*.

3.1.1.3 Classes

If a process is running, it might want to store data to process-internal variables, e.g. the state of a control algorithm. From a computer science point of view, internal variables are typed. Clustering types results in compound types. Furthermore, statements can be defined on the elements of a compound type. These operations can themselves be clustered in sub-functions, or methods. In particular, methods can have arguments which decouple the access to the data elements of a compound type from the actual data manipulation. A compound type with methods is called a class. Since a class is a type, it can be instantiated like the definition of a variable. In ASCET, variables and instances of classes can be defined in classes or modules.

If a class is defined as instance in the scope of another, i.e. outer class, the methods of the instantiated class can be called by the methods of the outer class. If an "instantiated method" realizes a calculation, e.g. a filtering algorithm, its results can be used in the calculations of the calling methods. Using this mechanism, one can represent control algorithms as a typed object hierarchy. Calling a method of the top-level class, i.e. the outermost class which is not instantiated in another class, will result in the deliverable of the output value(s) of a method. For the calculation of the result, methods of embedded instances will be called sequentially and yield their results which will be used by other calculations. From this point of view, the execution of a top-level method is equal to the sequential execution of an object-oriented program.

3.1.1.4 Parameters

From a computer-scientist point of view, parameters are a special kind of internal variables because they can only be read while writing is forbidden. From the control-engineering point of view, parameters are used to trim the control algorithm to a dedicated vehicle. Parameters are set before the start of the control algorithm execution and remain fixed[1)] during the run-time of the control algorithm. Because parameters are a special kind of variables, they can be grouped in a similar way as variables.

Classes might contain parameters (they can be seen as elements of a compound type). Since classes can be instantiated several times, these parameters will exist several times, too. However, as a rule, parameters are not initialized by dedicated methods (e.g. constructors) in a start-up phase, but typically exist in read-only memory. This means that an initial set of values has to be provided before run-time, e.g. at design time. This set of values is called data set. If the allocation of parameter values to instances of behavioral classes is done at design time, a data set has to be associated to a particular instance. In ASCET, at design time of the class, the data sets for tentative instances have to be defined, too, while the association to a particular instance is done when the instance is created.

3.1.1.5 Employing Classes in Modules

As written above, the sequential execution of a control algorithm starts with calling the method of a top-level class. This method call is initiated by the execution of a process. The arguments of a method are typically fed by the receive messages of the process, while the return value of the method will be fed to a send-message (Of course, these methods might also be fed by internal variables of a module).

From a real-time perspective, the process calling a method of a top-level class generates a sequential call stack of methods which belong to encapsulated instances. Even the methods of leave instances are executed in the context of the task the process is mapped to. Making the call stack of methods deep might com-

1) Adaptive parameters are not considered here.

promise reactivity to events. Therefore, when designing classes and employing them into real-time components, one has to find an appropriate balance between object-oriented reusability and reactiveness in a task-schedule.

3.1.1.6 Continuous Time Blocks for Plant Modeling

ASCET provides dedicated blocks for the modeling of continuous time systems. These continuous-time blocks (CT blocks) have two flavors:

- Basic blocks, which describe the dynamics of elementary systems
- Structure blocks, which group elementary blocks

Basic blocks assume a non-linear system in normal form of the following equations:

$$\dot{x} = f(x,u)$$
$$y = g(x,u)$$

Basic blocks specify the dynamic behavior in an object-oriented manner. There is an initialization and termination method, input, update and derivative methods to realize `f` as well as direct and non-direct output methods to realize `g`. Furthermore, there is a state-event detection method as well as an event method describing what to do in case of a state-event. Last but not least there is a method to resolve dependent parameters. The expressions can be specified either in ESDL code or in C code.

3.1.1.7 Projects for Closed-Loop Simulations

An ECU composition is a set of communicating modules and an operating system. The operating system configuration defines the tasks and their schedule, while the operating system itself realizes the tasks as well as the messages. The task-schedule contains the assignment of processes to tasks. To perform closed-loop simulations on a PC, CT blocks (cf. section “Continuous Time Blocks for Plant Modeling” on page 20) are attached to the real-time components of the control algorithm. Binding between the messages of the real-time components and the CT blocks has to be done explicitly, i.e. by connecting ports graphically and not by name-matching. The methods of a CT block are called from the numerical integration algorithms. The integration algorithms will be executed as separate task in the resulting operating system configuration.

After mapping the processes to tasks and creating the appropriate CT block tasks, the OS configuration will be translated to executable code. In case of a closed-loop simulation on a PC, a simulation environment with appropriate event queues and numerical solvers will be generated. The simulation environment is no real-time execution environment.

3.1.2 Rapid Prototyping

Unfortunately, the employed plant models are typically not detailed enough to serve as a unique reference throughout the design process. Therefore, the control algorithm has to be checked in a real vehicle. This is the first time the control algorithm will run in real-time. The execution entry points of the software components are mapped to operating system tasks while dedicated software components for hardware access have to be created and connected with the software components of the control algorithm. This step is shown in Fig. 3-1 on page 16 in linking the logical system architecture to the real vehicle which is driven by a driver in a real environment, represented by the arrow labeled 2.

There are many ways to realize this step. First of all, one can use a dedicated rapid prototyping system with dedicated I/O boards to interface with the vehicle. The rapid prototyping systems (RP system) consist of a powerful processor board and I/O boards. The boards are connected via an internal bus system, e.g. VME. Compared to a production ECU, these processor boards are in general more powerful; they have floating-point arithmetic units, and provide more ROM and RAM. Interfacing with sensors and actuators via bus-connected boards provides flexibility in different use cases. For short, priority is on rapid prototyping of control algorithms and not on production cost of ECUs.

The interfacing needs of the rapid prototyping systems often result in dedicated electrics on the boards. This limits flexibility, and an alternative is therefore to interface to sensors and actuators using a conventional ECU with its microcontroller peripherals and ECU electronics. A positive side-effect is that the software components of the I/O-hardware abstraction layer can be reused for series production later on. Fig. 3-2 shows that the control and monitoring functions run on a bypass system, which is connected to the vehicle via sensors and actuators.

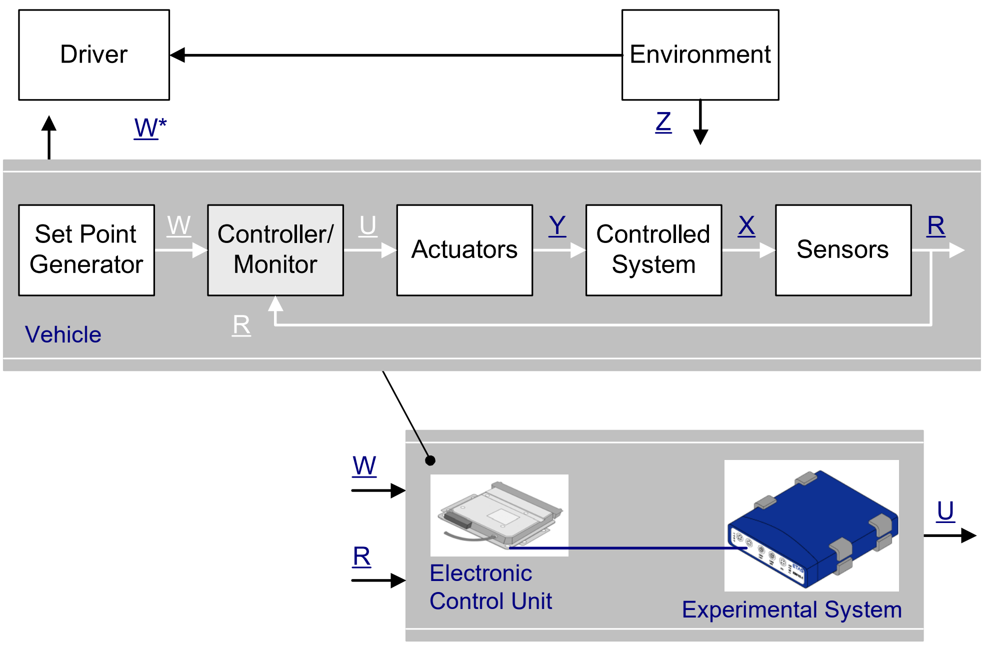


Fig. 3-2 typical rapid prototyping system

For rapid prototyping in bypass configuration, as shown in Fig. 3-2, the ECU's microcontroller-peripherals are used to drive the sensors and actuators. This means that the control algorithm still runs in the rapid prototyping hardware whereas the I/O-drivers are running on the production ECU.

The signals W, R, and U are digital values representing the set-point, the sampled reaction of the plant, and the digital actuator signal. The actuator signal is transformed to an electrical or mechanical signal Y driving the vehicle in the state prescribed the driver's wish W*. W is the corresponding sampled digital signal. The actual state of the vehicle in terms of mechanical or electrical signals X is sampled and fed to the control algorithm as digital signal R. Furthermore, there are noise signals Z like the road conditions which are not directly taken into account by the control algorithm as measured input signal, but also influence the behavior of the vehicle.

Provided no other vehicle signals are used directly, the RP system uses only a dedicated communication board in addition to the processor board. The sensor values R, the set-point values W, and actuator values U are transmitted over the high-speed link. In most cases, the ECU hardware is modified with dedicated facilities to accommodate the high-speed communication link.

From the software development point of view, structured interfaces of the software running on the production ECU as well as in the control algorithm development improves the efficiency of rapid prototyping considerably.

3.1.2.1 Hardware Configuration Component

For rapid prototyping experiments, dedicated hardware will be used. Besides a high-performance microprocessor, there are means available for communication and I/O.

From a certain point of view, a rapid prototyping system represents a reconfigurable embedded system. In particular, the communication and I/O hardware facilities need basic software modules as glue between the hardware and the control algorithm. These basic software modules are configurable. In ASCET, all basic software modules for the communication and I/O are represented in a hardware configuration component (see the ASCET-RP user guide for further information). For example, there will be a process reading signals from the CAN buffer and providing the signals as send messages. This process will be scheduled in an operating system task. The signal name as well as the CAN-frame ID can be configured in an editor.

If a control algorithm shall be tested on an ETAS rapid prototyping system, the real-time-I/O code has to be generated from the configuration parameters given in the hardware configuration component. This component has to be attached to the other real-time components, i.e. ASCET modules, to form a running rapid prototyping control algorithm.

3.1.2.2 Projects for Rapid Prototyping

A project for rapid prototyping does not contain a plant model represented by continuous time blocks. Instead, it contains a real-time I/O configuration in the shape of a dedicated hardware configuration component. This real-time I/O configuration is configured for the rapid prototyping project. On the model level, the real-time I/O configuration communicates with the control algorithm modules via messages. Depending on the real-time I/O configuration, there are several processes to be hooked to an operating system task.

3.1.3 Implementation and ECU Integration of Control Algorithms

After the rapid-prototyping step, the control algorithm is valid for use in the vehicle. The code that was generated for rapid prototyping systems relied on the special features of the processing board, such as RAM resources and the floating-point unit. To make the control algorithm executable under limited memory and computational resources, the model of the control algorithm has to be re-engineered. For example, computation formulas are transformed from floating-point to fixed-point control algorithms, and efficiency, scalability, modularity and other concerns are addressed. The adapted design can be automatically transformed to production code in a code generation step.

3.1.3.1 Floating-Point to Fixed-Point Conversion

A physical plant, e.g. a vehicle, deals with physical quantities, like vehicle-speed and acceleration, coolant temperature, yaw rate, battery voltage, etc. In simulation models, these physical quantities are realized by variables of type float, either in 64 or 32 bit guise. The simulation models represent a closed-loop control system, which means that both the vehicle model and the model of the control algorithm are represented in floating point. However, floating-point units are expensive and their use in automotive microcontrollers is not common. This means, implementation of a control algorithm on an automotive microcontroller involves a floating-point to fixed-point conversion.

Example

The coolant temperature might range from -50° Celsius to 150° Celsius. Fitting these values to a 16 bit integer straight forward would be quite inefficient. Only 0.3% of the available bits would be used, as shown in Fig. 3-3(a), and the resolution of the temperature would only be 1° Celsius per bit, resulting in a measured temperature of 83.4° Celsius represented as 83° Celsius in the control software.

This can be changed by multiplying every temperature value by 217.78 thus having a resolution of approximately 0.0046° Celsius per Bit, as shown in Fig. 3-3(b). Unfortunately, this adaptation will end up in a floating-point multiplication itself and is therefore not desirable.

An alternative would be to limit the resolution to 0.0078125° Celsius per bit. Now the multiplication operation can be expressed by a 7 bit left-shift operation. Applying this operation to the temperature range yields bit-patterns from -6400 to 19200, thus using a 16 bit integer variable by 39%. This scaling is shown in Fig. 3-3(c).

An even better utilization can be achieved by using an unsigned 16-bit integer value and a resolution of 0.00390625° Celsius per bit with an offset. This offset is set to -12800. The temperature range can now be used from -12800 to 38400, thus using a range from 51200 values and hence providing a utilization of more than 78%, as shown in Fig. 3-3(d). However, the offset requires an additional subtraction.

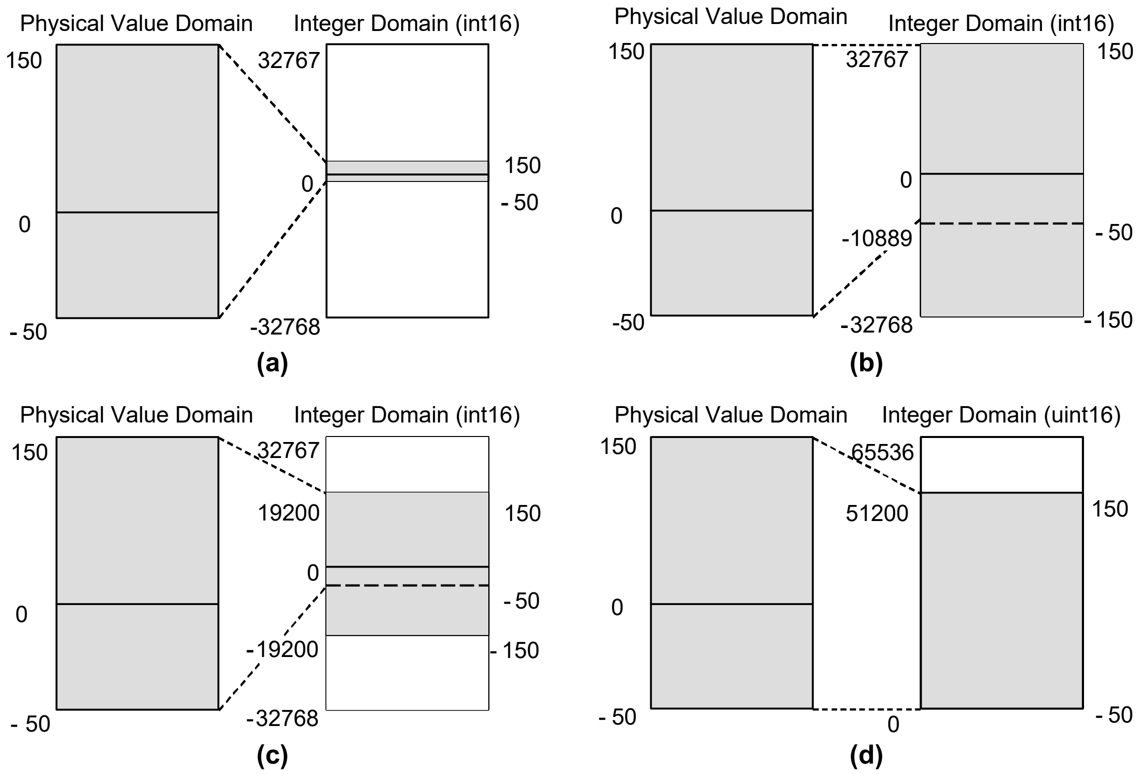


Fig. 3-3 Unscaled Mapping (a), Arbitrary Mapping (b), 2^7 Scaling (c), 2^8 Scaling with Offset (d)

The relationship can be expressed by the linear relationship:

```
impl_value = f_impl(phys_value) = phys_value * 256 + 12800
```

or, more generally, by

```
impl = scal * phys_value + x
```

where `scal` is the scaling factor and `x` the offset. The resolution is the reciprocal scaling factor, which means that the physical value is represented by an implementation value of

```
phys_value = impl_value / scal - x
```

3.1.3.2 Arithmetic with Fixed-Point Values

Associating an implementation formula to every variable has a heavy impact on the statements, i.e. expressions and assignments, of methods or processes. Even the simple assignment of two variables representing physical values

```
a = b
```

is not a trivial operation if the implementations, i.e. the associated implementation formulas, are different. Let `a` and `b` be implemented by the following implementation formulas as unsigned 8 bit variables (range from 0 to 255):

```
a = 2 * a_impl, b = 3 * b_impl
```

meaning that the physical value of `a` has a resolution of 2 while the physical value of `b` has a resolution of 3. Representing the assignment `a=b` in implementation terms yields:

```
2* a_impl = 3* b_impl
```

This is followed by a simple substitution:

```
a_impl = (3 / 2)* b_impl
```

Compared to the original statement `a = b`, we have now an adapted statement `a_impl = (3/2) * b_impl`. With respect to implementation formulas[1)], the adaptations are merely arithmetic operations with constants. However, care must be taken with the series of adaptive operations in order to consider the requirement for maximum precision. If one, as shown above, first performs the division, the various conversion equations would be ineffective due to the integer computation, and the results would be about 50% incorrect. A better way to express the adapted statement would be:

```
a_impl = 3 * b_impl / 2
```

As a result, statements of physical variables adapted by implementation operations often take into account more than just a simple operation.

The question of overflow must be taken into account. This means that if one first multiplies by 3, there is an overflow as soon as `b_impl` becomes greater than 255 / 3 = 85. Similarly, one must always be careful of underflows and rounding errors. If one first divides by 2, this is equivalent to a right shift operation, i.e. the last bit is dropped. No distinction can then be made whether `b_impl` has the value 1 or 0. In both cases, the result for `a_impl` and thus also for `a` is the value 0. In fact, the assignment `a = b` only makes sense if the physical ranges are identical (here max. 0 to 510). `b_impl` can therefore assume the maximum value 510 / 3 = 170. An overflow can occur here and must be avoided at all costs. One might think of making a case distinction in the code generation, i.e. first multiply for values from `b_impl` to 170 and first divide for values from `b_impl` greater than 170. But this leads to a requirement for more code. So here, one must accept a negligible error in precision of max. 1.5. within the entire value range. It is clear that the situation itself can become more difficult with regular arithmetic operations with few operands, not to mention complex links and expressions.

3.1.3.3 C Code Classes and Modules

For the migration of legacy code or for microcontroller peripheral access, one might define classes with the internal behavior of the method specified in C code as well as modules with the internal behavior of processes specified in C. Both C code classes and C code modules already represent implemented code. This code will be integrated verbatim into the executable for the target. Therefore, C code classes and modules are target-dependent. If one changes the target of a project, one has to provide the C code for the actual target, too.

---

1) At least the formulas ASCET supports

3.1.3.4 Projects for Embedded Microcontrollers

As written above, C code classes and modules can be used to access the peripherals of a microcontroller. The ASCET project editors allow to fully configure and generate an operating system. Together with the modules representing the control algorithms, projects for embedded microcontrollers can be used as integration platform. In this case, the code generator will examine the OS schedule and the message communication between the modules and generate the tasks, the messages and the access code[1)] of processes to messages. The resulting C code for the project and all its contained modules can be transformed to a `*.hex` file and flashed onto the microcontroller. Needless to say that an ASAM-MCD-2MC file will be generated, too, containing all variables to be measured as well as all parameters to be calibrated.

However, there are many cases where a build environment and dedicated basic software modules are used for a series production ECU. In this case, typically only the application software, i.e. the control algorithm, is modeled in ASCET[2)]. The messages are generated – including the access code of processes – as well as so-called task bodies, i.e. a sequence of processes as specified in the OS editor. This task body can then be copied to an appropriate OS configuration editor (external to ASCET).

3.1.4 Reuse of Control Algorithm in Different Kinds of Projects

As written above, all ASCET modeling elements are maintained in a database or workspace. Furthermore, projects for different targets differ in the number and kind of modules for the same control algorithm.

- Project for closed-loop simulation:

  This project references the modules for the control algorithm as well as CT blocks.

- Project for rapid prototyping:

  This project references the modules for the control algorithm (which are the same modules as for the closed-loop simulation) and the hardware configuration component.

- Project for embedded microcontroller:

  This project references the modules for the control algorithm as well as the (C code) modules for the peripheral access. If one wants to obtain fixed-point code, one has to attach implementation formulas to modules, classes and projects. Before generating code, one has to the select the appropriate implementation for the project.

ASCET projects can be executed on different execution targets, which might be a PC, a rapid prototyping system, or a production ECU[3)]. To run experiments, ASCET provides an integrated experiment environment (or EE for short) if the project runs

1) Typically realized as macro
2) This use case is often called *additional programmer*
3) or an evaluation board

on a PC or rapid-prototyping system. For ECU experiments, an EE is integrated in the measurement and calibration system INCA[1)] because ECU experiments are to some extent similar to the fine-tuning[2)] of a control algorithm in the vehicle.

From a software perspective, there are the following kinds of experiments:

A Physical Experiment

B Quantized Experiment

C Implementation Experiment

D Object-Based Controller Implementation Experiment

E Object-Based Controller Physical Experiment

Only the physical experiment does not need any implementation information. The quantized experiment needs the quantization, the implementation and object-based implementation experiments need additionally the limits and, more important, an integer base type.

ASCET control algorithm models are composed of statements whose generated code looks differently depending on the type of the target and the selected experiment.

In *physical experiments*, the physical statements will be resolved to real64 variables with no quantization effects.

The *quantized experiment* uses also real64 variables as basis, but coerces the physical statements in a way that quantization effects will become visible.

The *implementation experiment* uses the full implementation information and is based on integer types. This means that the types of the variables in the generated code are the chosen base-types of the implementation and the operators in the physical statements have been transformed to implementation statements.

The *object-based controller implementation experiment* uses the types and implementation statements of the implementation experiment, but the structure of the modules and classes is resolved in a different way. For example, it is possible for every variable in ASCET not only to attach base types, limits and implementation formulas, but also memory classes. The memory classes reflect the memory layout of the employed microcontroller. However, as written above, the object-based controller implementation experiment can only be chosen for production ECUs, and online experimentation can only be performed by INCA or any other measurement & calibration tool.

The *object-based controller physical experiment* generates controller code, but ignores the implementation specification.

When working with a PC or rapid-prototyping target, and all the implementation information regarding base type, limit, offset and quantization has been attached to all elements, one can study the effects of implementation formulas or integer base types with respect to the physical environment by just switching the experiment type.

---

1) If the ASCET project consists of CT blocks only and the project runs on a PC or rapid prototyping hardware, the EE is integrated into LABCAR operator.

2) Because of the limited ECU resources for experimenting, dedicated means are necessary which are not in the scope of this section.

3.1.5 Testing Technical System Architecture in the Lab

The result of the implementation and integration phase is the technical system architecture, i.e. networked ECUs. These ECUs are tested against plant-models in real-time. The plant models themselves are augmented by models of the sensors and actuators and dedicated boards being able to simulate the electrical signals as they are expected by the ECU electronics. These kind of systems are called Hardware-in-the-Loop systems (or HiL systems for short) and consist of processing and I/O boards. The plant model is initialized with different values simulating typical driving maneuvers. Then, the driving maneuver is simulated on the HiL and providing ECU sensor data as output and accepting ECU actuator data as input. This way it can be checked whether the ECU integration was successful. HiL testing is represented by the arrow labeled 4 in Fig. 3-1 on page 16.

3.1.6 Testing and Honing of Technical System Architecture in the Vehicle

As written above, there are many use cases where plant models are not detailed enough to represent the vehicle's dynamics. Though a lot of calibration activities can nowadays be done by means of HiL systems, final honing of a vehicle's control algorithm still needs to be done with the production software in a production ECU in a real vehicle. This requires that the technical system architecture is built into a vehicle and tests are done on a proving ground. This kind of fine-tuning only concerns the parameter setting of the control algorithm.

3.2 Using ASCET in a Production Environment

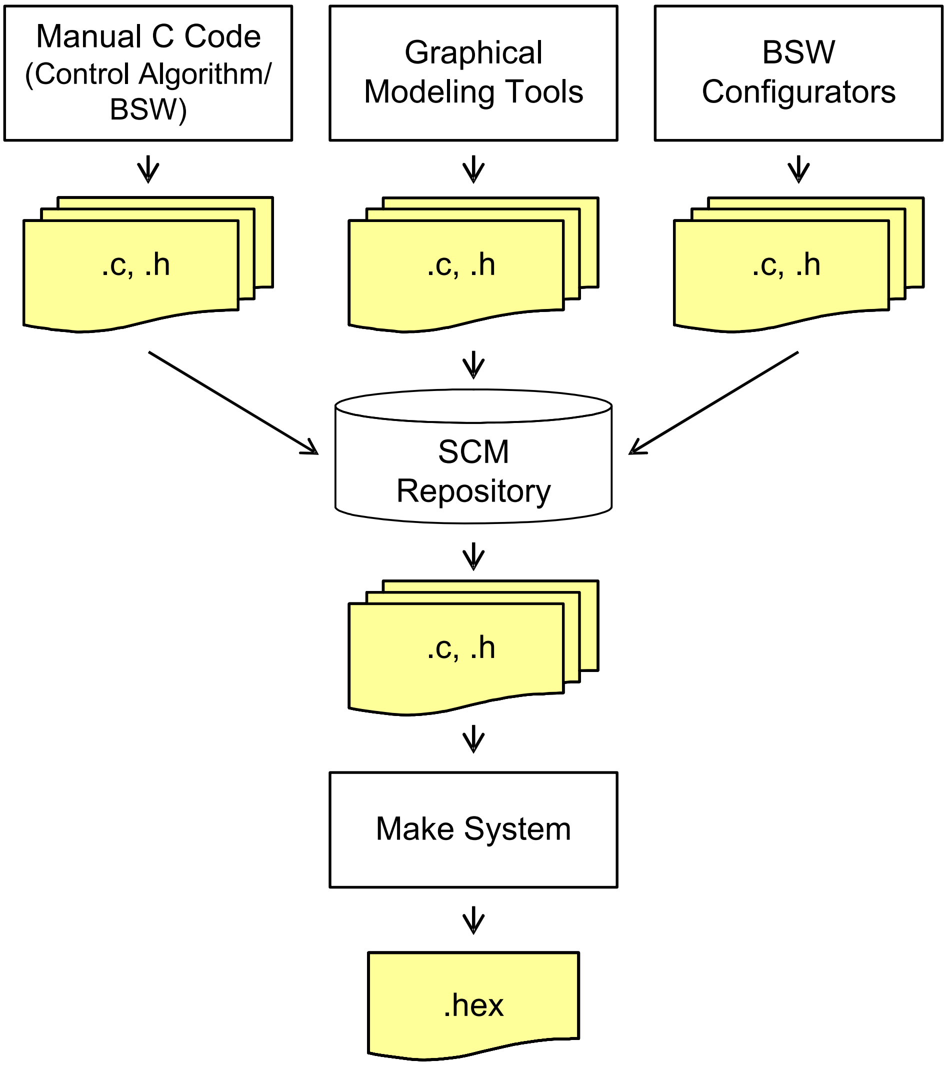


Fig. 3-4 Advanced Software Production Environment

In a manual coding environment, there are typically several software developers providing the C code for the control algorithm as well as for the basic software modules, including the operating system. Then there is an ECU integrator collecting all necessary source code files and starting the so-called make toolchain, which starts the compiler and linker. The C code is transferred between the software developers by using the file system on the one hand and a source-code management (SCM) system[1)] on the other. The latter is a database holding different versions of the source code files but also allowing the creation and maintenance of configurations. The latter are used as a baseline to generate/integrate ECU software. To see differences between two versions of a C code file, difference browsers highlighting the changes in the program text are used. In the last decade, intensive use of SCM systems and difference browsing contributed considerably to the enhanced quality of embedded automotive software.

1) Typical SCM systems are CVS and SubVersion

In advanced software production environments, some of the C files for control algorithms are generated from control algorithm models, e.g. an implemented ASCET model, while a lot of C files for basic-software modules, e.g. OS and COM stack, are generated by so-called configurators. Leaving the ASAM-MCD-2MC file generation aside, such an advanced production environment is shown in Fig. 3-4 on page 29. It shows the C code-generating entities, the SCM database as well as the make system. Looking deeper in such an advanced production environment, and focusing on the model-based generation of C code for control algorithms with ASCET, one will realize that the models, which are the basis for the source code, will evolve in the course of the control algorithm development, e.g. incorporating the results of rapid prototyping. Hence, the models have to be maintained in the SCM database too.

ASCET components are stored in a local database or workspace. The local database/workspace holds exactly one version of the model. The ASCET-SCM interface establishes a link from the local database/workspace to the SCM repository and enables the model exchange. This model exchange is shown in part (a) of Fig. 3-5. Since, in source-code development, difference-browsing between different versions is indispensable, a similar feature is highly desirable in model-based development, too. The ASCET-SCM interface can be enhanced by ASCET-DIFF (a model difference browser), thus highlighting, e.g., an additional message in the block diagram editor of a module.

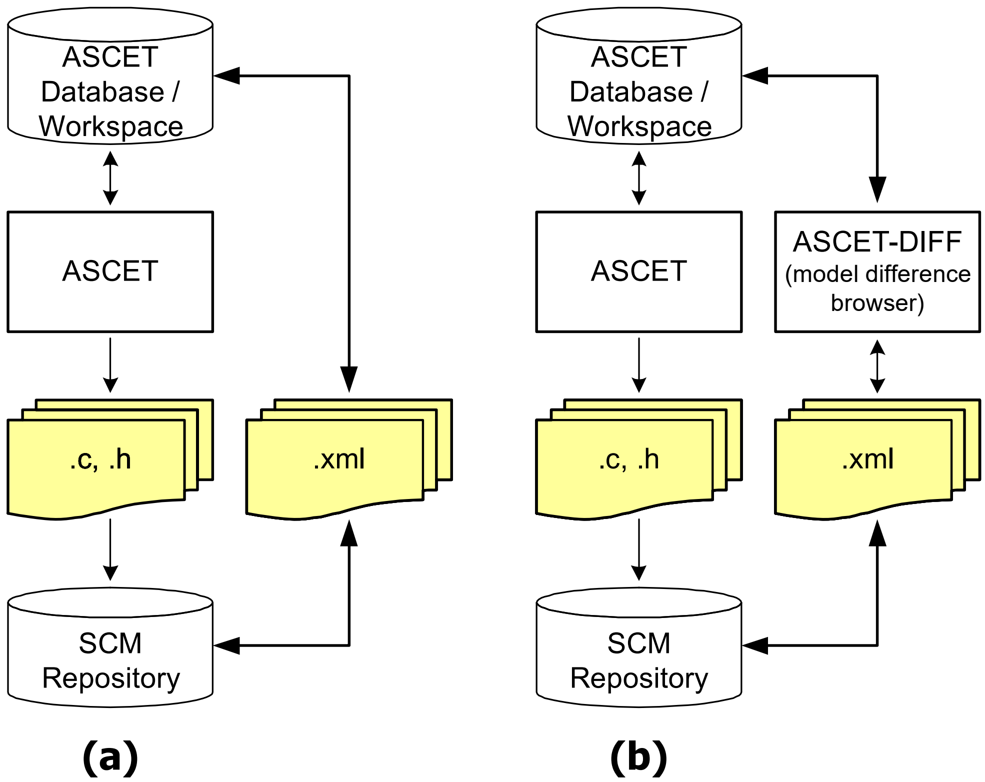


Fig. 3-5 ASCET-SCM interface without (a) and with (b) ASCET-DIFF

3.3 Summary

Model-based design and implementation of control algorithms is supported by ASCET for several development stages. The employed abstraction means allow to use the physical control algorithm model as backbone for all subsequent implementation annotations throughout the course of development. In particular, no blocks need to be replaced when changing the target. Employing the SCM interface with difference browsing, ASCET can be seamlessly integrated in an ECU production development environment.

4 Tutorial

Users who are not familiar with ASCET will learn all the basic working steps in this tutorial. The tutorial does not require any knowledge of ASCET, but does assume that the user is familiar with the Windows operating system.

Before you start working on the tutorial, you should have read chapter "Embedded Automotive Control Software Development with ASCET" on page 15.

All components and projects for lessons 1 – 9 of this tutorial can be found in the folder named `ASCET_Tutorial_Solutions` in the database named `Tutorial`. It is therefore not necessary to specify all the components described here yourself.

It is, however, advisable to specify at least the components of lessons 1, 3 and 4, to get some practice using ASCET.

These databases are available in the database directory of your ASCET installation (e.g. `D:\ETASData\ASCET6.4\database\Tutorial`) and in the export files `Tutorial.*`[1] and `INTECRIO_Tutorial.*` in the Export directory of your ASCET installation (e.g. `C:\etas\ASCET6.4\export`).

4.1 Preparations

Before you can start the tutorial, you have to prepare your computer.

The computer you want to use for the tutorial has to have an executable ASCET installed. ASCET can either be launched using the icon on the desktop or using the Windows **Start** menu.

> **NOTE**
>
> You can perform the tasks in all lessons without any hardware.
>
> Lesson 7 also describes an online experiment, which requires a working experimental target.

At the start of ASCET, the Component Manager opens. If you open ASCET for the first time, the `Tutorial` database opens. (If you open ASCET later, the most recently used database/workspace opens.)

Fig. 4-1 on page 34 shows the Component Manager with an empty database or workspace. The top left field, "Database" or "Workspace", will show the folders and components in the database. The bottom left field is used for general comments about the item selected in the "Database" or "Workspace" pane. The right field, "Contents", will show the content of the item selected in the "Database" or "Workspace" pane.

Before you can start, you have to open a database or workspace to work in. All components of this tutorial will be stored in this database/workspace.

1) `.*` = **`.exp`** (binary export format) or **`.axl`** (compressed XML-based export format)

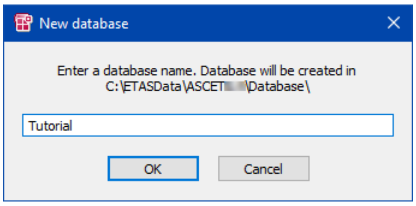
4.1.1 Tutorial Database or Workspace

It is recommended that you use a separate database/workspace – either a newly created one or the `Tutorial` database shipped with ASCET – for the tutorial to keep the data transparent.

To create a new database

1. In the Component Manager, select **File › New Database**.

   The "New database" window opens.

2. Enter the name, e.g., `Tutorial`.
3. Click **OK**.

   The new database (**A** in Fig. 4-1), containing only the database name, opens.

To create a new workspace

1. In the Component Manager, select **File › New Workspace**.

   The "Save new workspace as" window opens. The ASCET workspace path1), the ASCET workspace file type (`*.aws`), and the default name `workspace.aws` are preselected.
2. In that window, create a new subfolder in the `Workspaces` folder and name it, e.g., `Tutorial`.
3. Open the `Tutorial` subfolder.
4. Enter the file name `Tutorial.aws` and click **Save**.

   The new workspace (**B** in Fig. 4-1), containing only the workspace name, opens in the Component Manager.

1) e.g., `C:\ETASData\ASCET6.4\Workspaces`

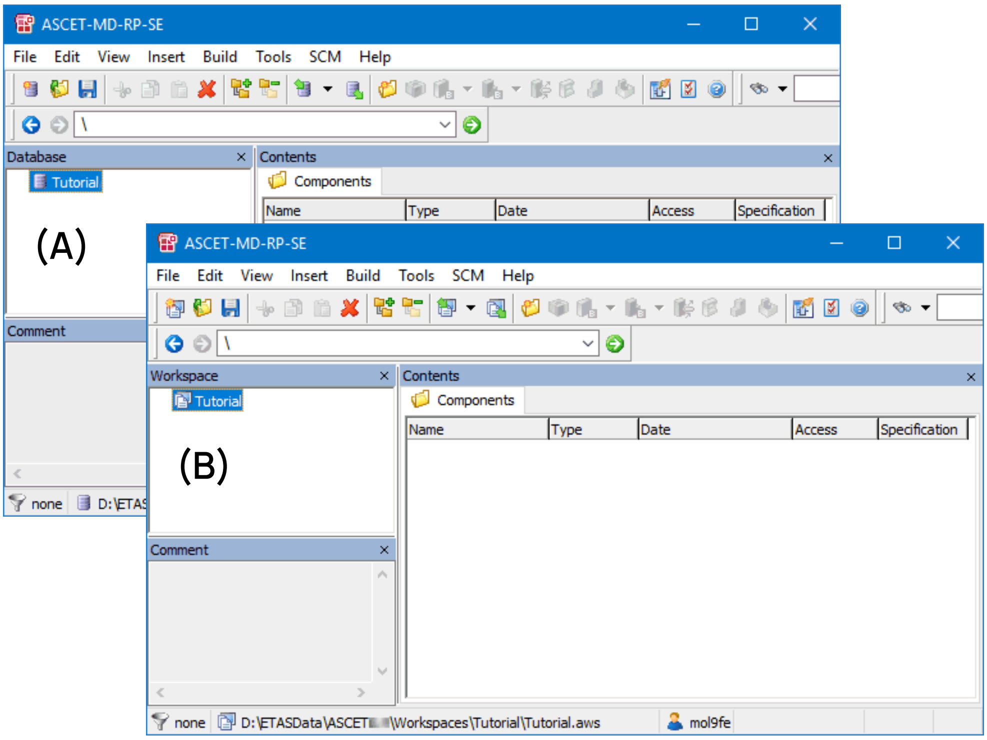


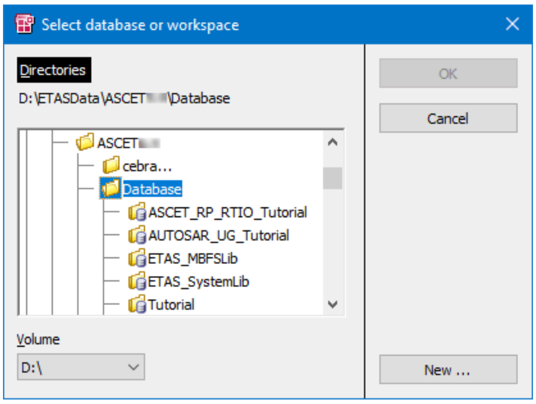
Fig. 4-1 ASCET Component Manager (A: with empty database, B: with empty workspace)

<u>To open a database</u>

To use the existing `Tutorial` database, proceed as follows:

1. In the Component Manager, select **File > Open**.

   The "Select database or workspace" window opens. It shows the current database path and the databases found there.

2. Select the `Tutorial` database and click **OK**.

   The `Tutorial` database opens in the Component Manager.

4.1.2 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating a new database or workspace

- Opening an existing database

4.2 Lesson 1: A Simple Block Diagram

In ASCET you use components, such as classes and modules, as the main building blocks of your applications. You can either use predefined components, which come with ASCET or have been developed earlier, or create your own, which is what you will be doing in this tutorial.

In ASCET components are usually specified graphically. Once all the components have been specified, they are assembled into a project, which forms the basis of an ASCET software system. A software system consists of C code that has been generated from the graphical model description, and which can be run on a micro-controller or experimental target computer.

4.2.1 Preparatory Steps

The first step in creating your own components is to create a new top level folder named `Tutorial` and a subfolder named `Lesson1`.

<u>To create new folders</u>

1. In the "Database" or "Workspace" pane, select the database/workspace name.

2. Click the **Insert Folder** button.

   A new top-level folder named `Root` appears.
3. Change the name of the top-level folder to `Tutorial`.

   You can type over the highlighted name and then press <ENTER>.
4. Select the folder `Tutorial`.
5. Add a subfolder named `Lesson1` to `Tutorial`.

All components you create in this tutorial will be stored in a folder. You should create a new folder `Lesson<n>` for every lesson.

> **NOTE**
>
> All folder and item names and the names of variables and methods they contain must comply with the ANSI C standard.

You can proceed by creating your first component in the `Lesson1` folder.

<u>To create a component</u>

1. In the "Database" or "Workspace" pane, click the folder `Lesson1`.
2. Select **Insert > Class > Block Diagram**.

   A new component named `Class_Block_Diagram` appears in the "Database" or "Workspace" pane under the `Lesson1` folder. This component is of type *class*, which is frequently used in ASCET.
3. Change the component name to `Addition`.

4.2.2 Specifying a Class

After you have created a new component in the `Tutorial\Lesson1` folder, you can specify its functionality. First define the interface for the component, i.e. its methods, arguments and return values. Then draw a block diagram that specifies what the component does.

To prepare the class

1. In the "Database" or "Workspace" pane, select the component `Addition`.
2. To open the component, select **Edit > Open Component**.

   The block diagram editor opens. This is the main window for specifying component functionality.

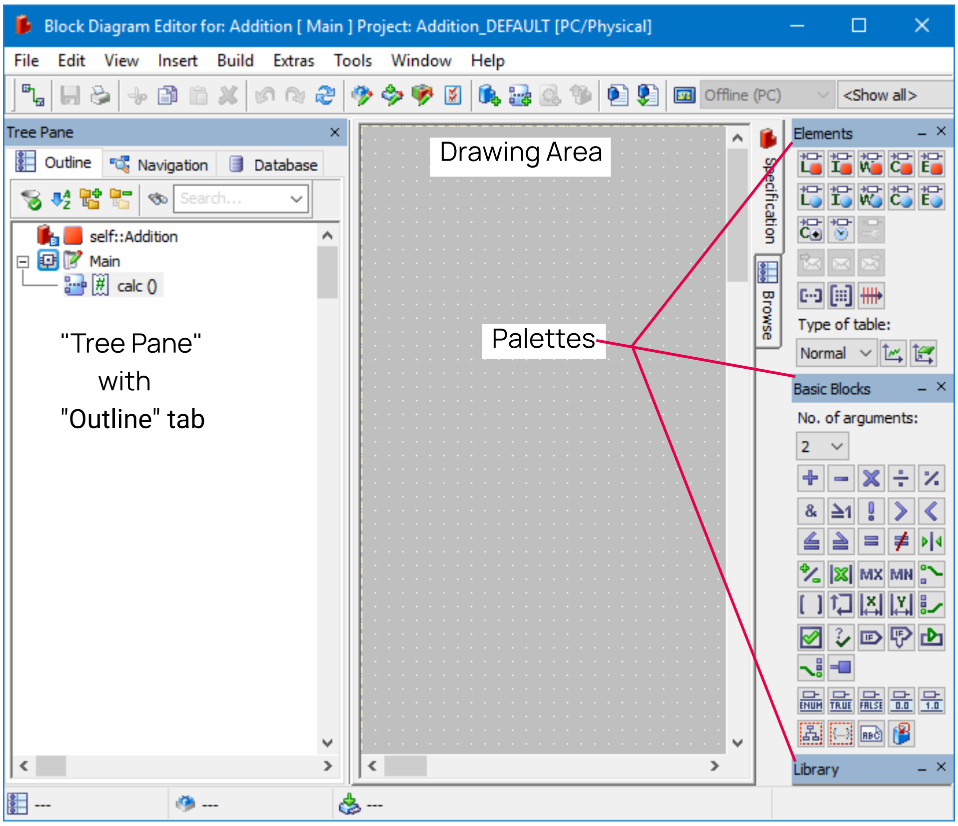


Fig. 4-2 ASCET block diagram editor (Specification view)

3. In the "Outline" tab, select the method `calc`.

   This method is created by default.
4. Select **Edit > Rename**.

   The name of the method `calc` is highlighted.
5. Change the name of the method to `DoAddition`.

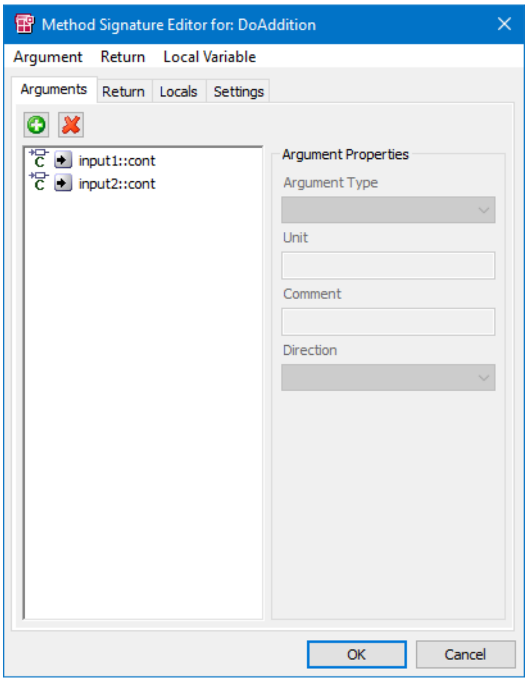
Every class needs at least one method. Methods in ASCET are similar to methods in object-oriented programming, or functions in procedural programming languages. A method can have several arguments and one return value (these are all optional). Arguments are used to transmit data to a component. Return values are used to return results of calculations within the component to the "outside".

To specify the method signature, you will add two arguments of type `continuous` and a return value using the signature editor.

To specify the method signature

1. Double-click the method name `DoAddition`.

   The signature editor for the method opens.

2. In the signature editor, select **Argument** › **Add**.

   A new argument called `arg` is created.

3. Change the name of the argument to `input1`.
4. Add another argument called `input2`.

   By default, the data type of the arguments is set to *continuous* (or *cont* for short), which is what you need in the example.

5. In the "Return" tab of the signature editor, activate the **Return Value** option.

   The type of the return value is also set to *cont* by default.

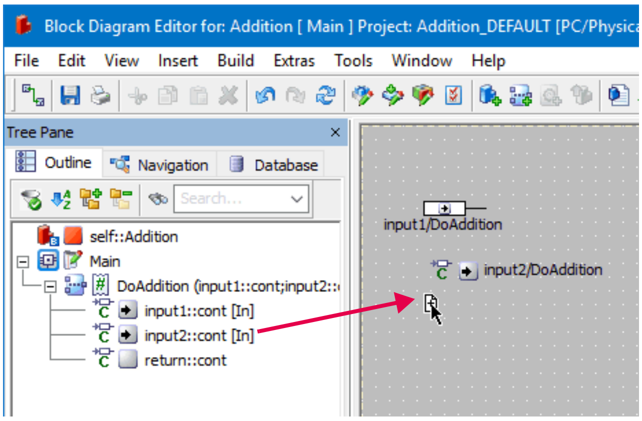
6. Click **OK** to close the signature editor.

The names of the arguments and the return value for the method `DoAddition` appear below the method in the "Outline" tab on the left of the block diagram editor. Now you can specify the functionality of the component by drawing a block diagram.

To specify the functionality of the component **`Addition`**

1. Drag the first argument from the "Outline" tab and drop it onto the drawing area of the block diagram editor.

   The symbol for the argument appears in the drawing area.

2. Now add the other argument and the return value to the diagram.

3. Click the **Addition** button in the "Basic Blocks" palette.

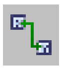
   The mouse is loaded with an addition operator.

4. Click inside the drawing area, between the symbols for the argument and for the return value.

   An addition symbol is displayed. By default it has two input pins (indicated by arrows) and one output pin. The output pin is located on the right.

You can now arrange the elements and the operator by dragging them to their places on the drawing area. Next, you need to connect the elements to specify the flow of information.

To connect the diagram elements



1. Click the **Connect** button in the "General" toolbar.

   Alternatively, you can right-click in the drawing area (but not on an element).

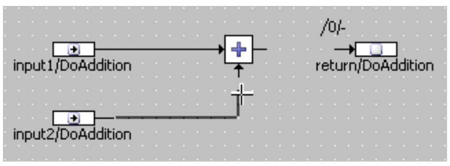
   The cursor changes to a crosshair when it is inside the drawing area.

2. Click the output pin of the first argument symbol to begin a connection.

   Now, as you move the mouse cursor, a line is drawn after it. Every time you click inside the drawing area, the line remains fixed up to that point. That way you can determine the path of the connection line.

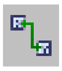
3. Click the left input of the addition symbol.

   The argument symbol is now connected to the input of the addition symbol.

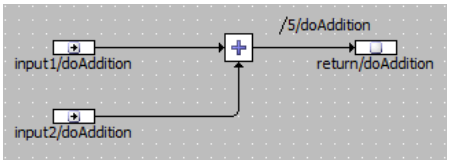
4. Connect the second argument symbol with the other input and the return value with the output of the addition symbol.

   The connection between the addition operator and the return value is displayed as a green line to indicate that the sequencing for this operation needs to be determined.



5. End the connection mode with another click on the **Connect** button in the "General" toolbar.
6. Double-click the empty sequence call `/0/-` to determine the addition sequence automatically.

   The connection between the addition operator and the return value is displayed as a black line.



Component specification is now complete. The last step in editing your component is to specify its layout, i.e., the way it is displayed when used within other components.

To edit the layout of a component

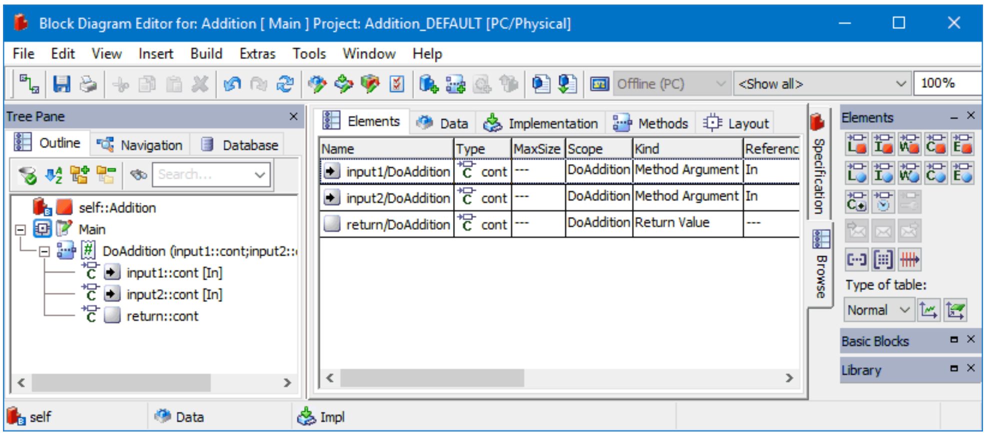
There are two ways to edit a layout:

1. Select **Edit › Component › Layout**.

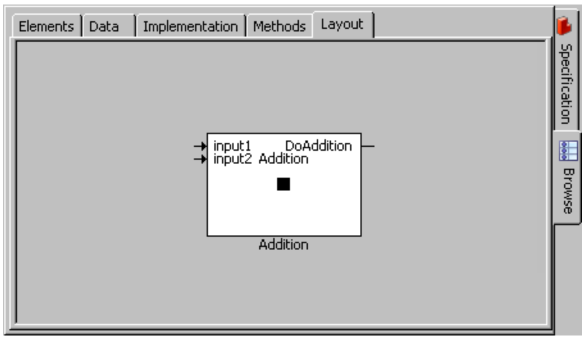
Or – as an alternative –

2. Do the following:

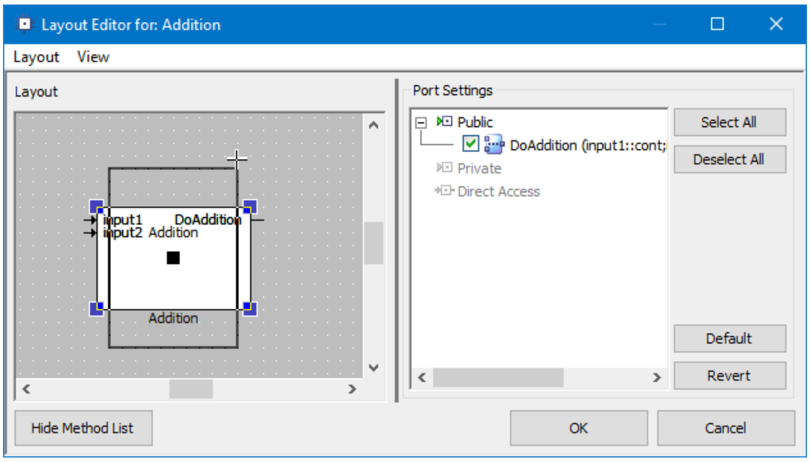
   i. Use the **Browse** tab to open the "Browse" view.



ii. Double-click in the "Layout" tab.



The Layout Editor opens.



3. Click the block and drag the handles to resize it.
4. Drag the pins of the arguments and the return value to create a symmetrical design.
5. Click **OK**.

Now that you have finished your component, the last step in this lesson is to save the component in the database or workspace.

<u>To save the component **`Addition`**</u>

1. Select **File > Save**.
2. Close the block diagram editor with **File > Close**.

   When you select **Save** in the block diagram editor, the changes are only stored in the cache memory. It is therefore advisable to click **Save** in the Component Manager regularly as work progresses.

3. In the Component Manager, click the **Save** button.

   Your work is not written to disk until you perform this operation.

   You can have your changes saved automatically by activating the appropriate user options (see the ASCET online help) for your ASCET session.

As an optional exercise, you could now model the same functionality in *ESDL* (ESDL: *Embedded Software Description Language*). If you continue with this exercise, you will learn how to use the external source code editor.

The first step is to copy the module interface to a new module with type ESDL and rename it. Then create the functionality you want either directly in the ASCET ESDL editor or use the external text editor.

To copy and specify the component **`AdditionESDL`**

1. In the "Database" or "Workspace" pane of the Component Manager, right-click the component `Addition` and select **Reproduce As > ESDL** from the context menu.

   A copy of the component is created; it is named `Addition1`.
2. Name the new component `AdditionESDL`.
3. Double-click the name of the new component.

   The ESDL editor for `AdditionESDL` opens, making various functionalities available for editing.

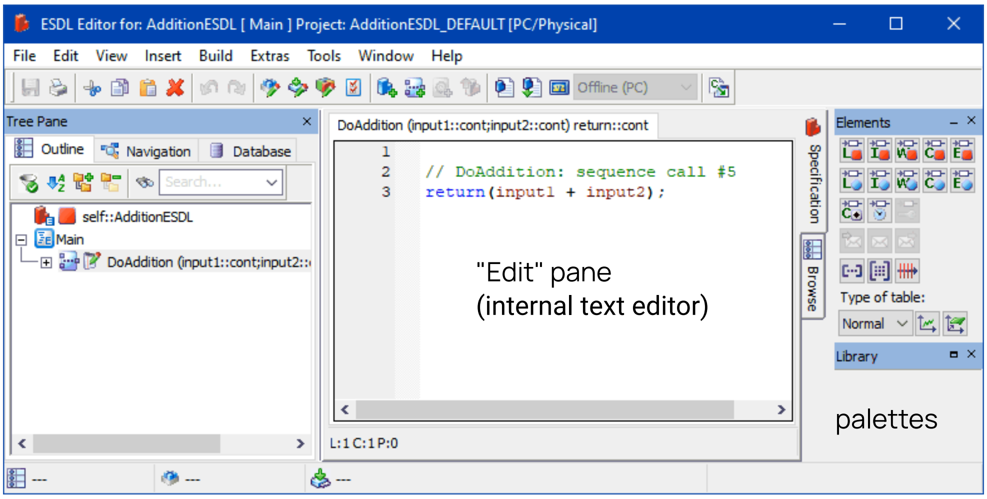


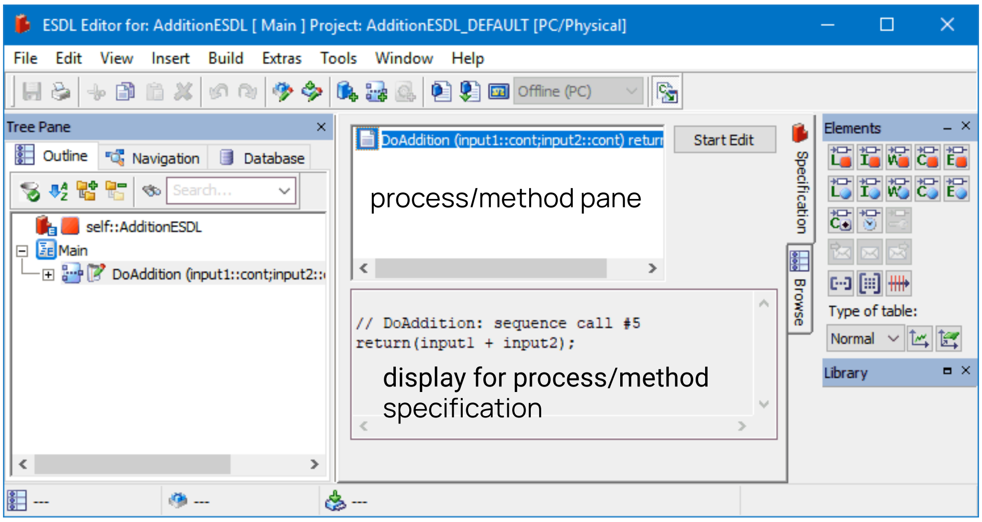
Fig. 4-3 ASCET ESDL editor (Specification view)

4. Use the **Activate External Editor** button to switch to external editor mode.

If unsaved changes exist, you are asked if you want to save your changes.

5. Confirm with **Yes**.

   The changes are saved, and the ESDL editor switches to "external editor" mode. The editor looks different in "external editor" mode.



6. In the process/method pane, select the method or process you want to specify.

   The functionality entered previously appears in the specification field, and the **Start Edit** button is activated.

7. Activate the external editor with **Start Edit**.

> **NOTE**
>
> When the external editor starts up, the application associated with the file endings `*.c` and `*.h` in the operating system register database is called. Data transfer is done via temporary files; this is why you have to save the files before you close the external editor or end the "external editor" mode of the ESDL editor.

8. Edit the functionality in the external editor.

9. Save the functionality in the external editor.

   With that, your changes are transferred to the ESDL editor. You do not have to close the external editor to continue working in ASCET.

10. Click **Activate External Editor** a second time to end the "external editor" mode.

    A message window opens. Read the text carefully.

11. Click **OK** to continue.

12. Select **Build > Analyze Diagram** to check the code you entered.

    Errors are listed in the ASCET monitor window.

4.2.3 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating and naming a folder
- Creating and naming a component
- Defining the interface for a method
- Placing diagram elements on the drawing area
- Connecting diagram elements
- Editing the layout of a component
- Switching between Specification and Browser views
- Saving a component
- Copying a component interface
- Using the ESDL editor
- Using the external editor

4.3 Lesson 2: Experimenting with Components

You can now experiment with the `Addition` or `AdditionESDL` components. Experimentation allows you to see how the component works, just as it would in a real application. The experimentation environment provides a variety of tools that can show the values of inputs, outputs, parameters and variables within a component.

4.3.1 Starting the Experimentation Environment

The experimentation environment is called from the block diagram or the ESDL editor. First open it with the component you want to experiment with.

<u>To start the experimentation environment:</u>

1. From the ASCET Component Manager, open the editor for the class `Addition` or `AdditionESDL`.
2. In the editor, select **Build > Experiment**.

   The code for the experiment is generated. ASCET analyses the model in your specification and generates C code that implements the model. It is possible to generate specific code for different platforms.

   In your example, you simply use the default settings to generate code for the PC.

After the code has been generated and compiled, the experimentation environment opens.

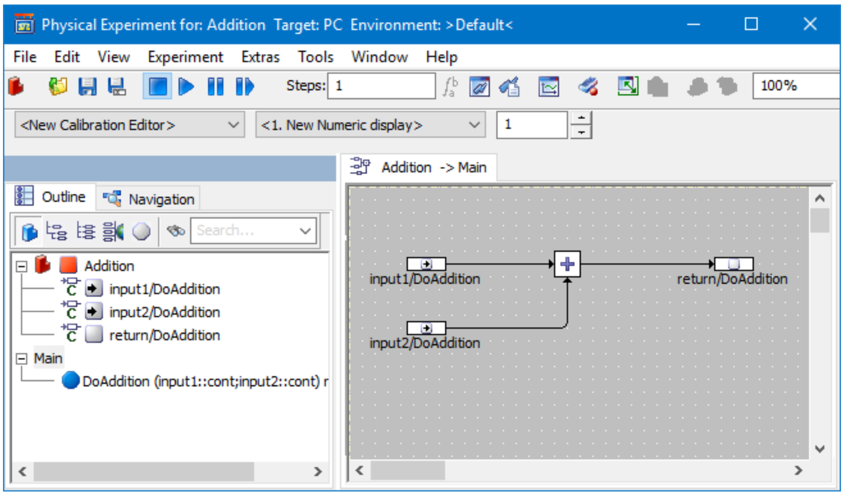


Fig. 4-4 ASCET offline experimentation environment

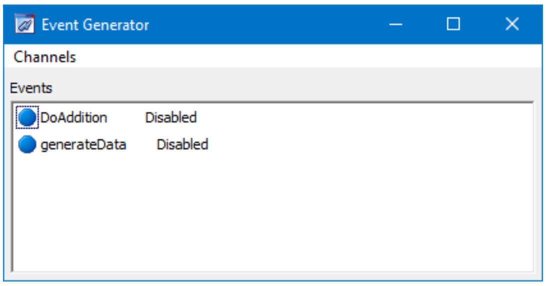
4.3.2 Setting up the Experimentation Environment

Before you can start experimenting, you have to set up the environment, which means determining the input values generated for the experiment and how you want to view the results. You have to carry out three steps. First, you set up the event generator, then the data generator, and finally the measurement system.

<u>To set up the Event Generator</u>

1. Click the **Open Event Generator** button.

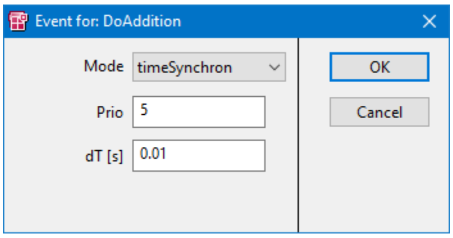
   The "Event Generator" window opens. You need to create an event for each method to be simulated, and also a `generateData` event. The events simulate the scheduling performed by the operating system of a real application.



2. Select the event `DoAddtion`.
3. Select **Channels › Enable**.
4. Select the event `DoAddtion` again.

5. Select **Channels > Edit**.

   The "Event" dialog window opens.

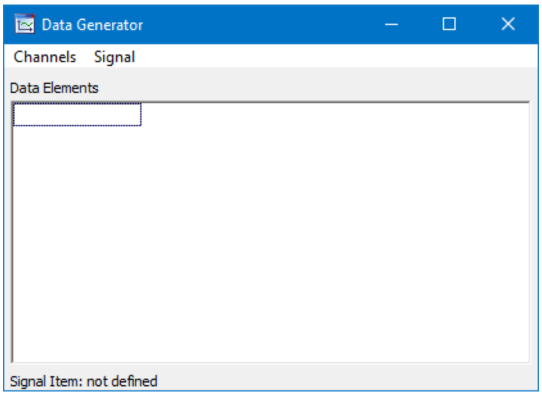


6. Set the `dT` value to `0.001`.
7. Click **OK**.
8. In the event generator, select the `generateData` event and set its `dT` value to `0.001`.
9. Close the "Event Generator" window.

<u>To set up the Data Generator</u>

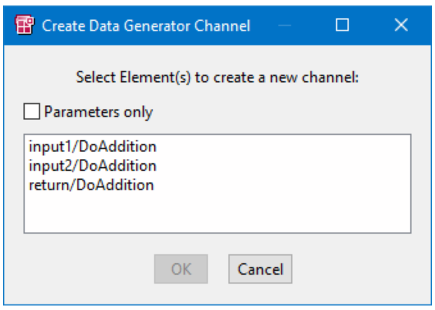
1. Click the **Open Data Generator** button.

   The "Data Generator" window opens.



2. Select **Channels > Create**.

   The "Create Data Generator Channel" dialog window opens.

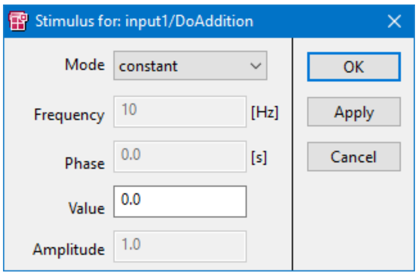


3. Select the entries `input1/DoAddition` and `input2/DoAddition` from the list.
4. Click **OK**.

   Now both inputs are listed in the "Data Elements" pane of the "Data Generator" window.

5. Select `input1/DoAddition` in the "Data Elements" pane.
6. Select **Channels › Edit**.

   The "Stimulus" dialog window opens.



7. Set the values for `input1` and `input2` as follows.

Mode:	`sine`
Frequency:	`1.0 Hz`
Phase:	`0.0 s`
Offset:	`0.0`
Amplitude:	`1.0`

8. Click **OK** to close the "Stimulus" dialog window.
9. Set the values for `input2` as follows:

Mode:	`sine`
Frequency:	`2.0 Hz`
Phase:	`0.0 s`
Offset:	`0.0`
Amplitude:	`2.0`

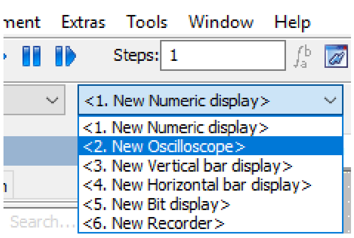
10. Close the "Data Generator" window.

With these settings you get two sine waves with different frequencies and different amplitudes. The `Addition` component adds the two waves and displays the resulting curve.

In order to see the three curves displayed on an oscilloscope, you will now set up a measurement system.

<u>To set up the measurement system</u>

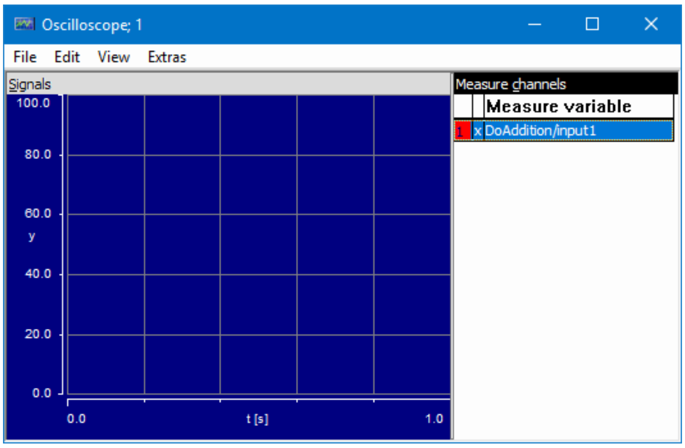
1. In the "Physical Experiment" window, select `<2. New Oscilloscope>` as data display type from the "Measure View" combo box.



2. In the "Outline" tab, select `input1/DoAddition`.

3. Select **Extras > Measure**.

   An oscilloscope window opens with `input1` as measurement channel. The "Measure view" list in the experimentation environment is updated to display the title of the measurement window.

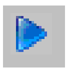
4. Add `input2/DoAddition` and `return/doAddition` to the same oscilloscope.
5. In the experimentation environment, select **File > Save Environment**.

Now the experimentation environment is set up, and you are ready to start the experiment. Since you have saved the experiment, it is automatically reloaded next time you start the experimentation environment for this component.

4.3.3 Using the Experimentation Environment

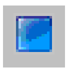
The experimentation environment provides a set of tools that allow you to view the values of all the variables in your component and also change the setup while the experiment is running. You can also adjust the way the values are displayed and choose from several ways of displaying them.

To start the experiment



1. In the "Physical Experiment" window, click the **Start Offline Experiment** button.

   The experiment starts running and the results are displayed in the oscilloscope.



2. Click the **Stop Offline Experiment** button to stop the experiment.

You will only see a small portion of the curves on the oscilloscope. To display the curves on the oscilloscope, you need to alter the scale on the value axis.

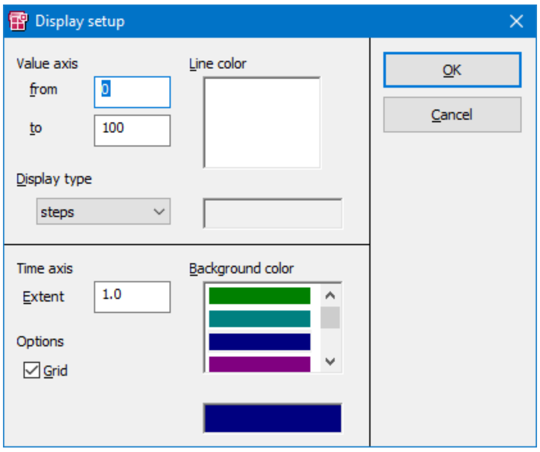
To change the scale on the oscilloscope

1. Select all three channels from the "Measure Channels" list in the oscilloscope window.

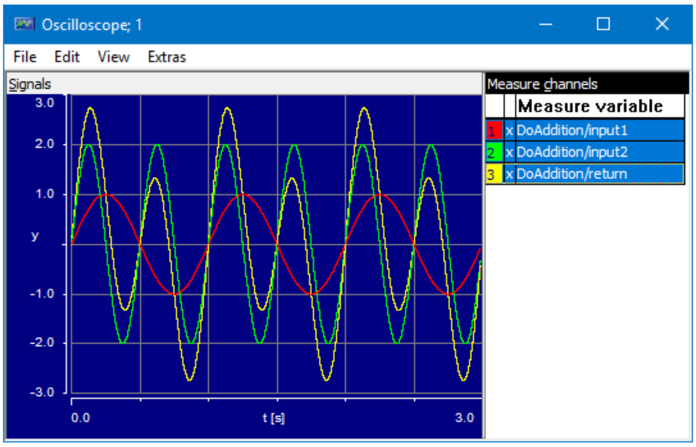
   With that, the changes you make will affect all three of them.

2. Select **Extras > Setup**.

   The "Display Setup" dialog window opens.



3. Set the "Value Axis" to a range of -3 to 3.
4. Set the "Time Axis Extent" to 3.
5. Select a background color in the "Background color" list.
6. Press <ENTER>.



The oscilloscope now shows the values with the appropriate scaling on the value axis. You will see the two input sine waves, together with the wave resulting from their addition. You can now adjust the input values to see how the output is affected.

<u>To change the input values for experimentation</u>

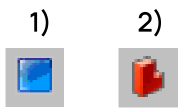
1. In the "Physical Experiment" window, open the "Data Generator" window.
2. In the data generator, select the variable you want to change.
3. Select **Channels > Edit**.

   The "Stimulus" dialog window opens.

4. Adjust the values you want to change.
5. Click **Apply**.

The curves in the oscilloscope change according to the new settings. You can change all the settings in the experimentation environment while the experiment is running.

To end the experiment



1. To stop the experiment, click the **Stop Offline Experiment** button.
2. To return to the component editor, click the **Exit to Component** button.

4.3.4 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

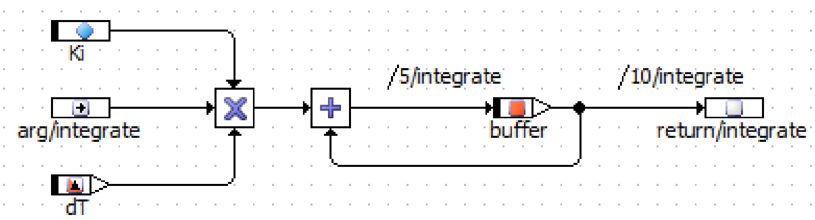
- Calling the experimentation environment
- Setting up the event generator
- Setting up the data generator
- Setting up the measuring system
- Starting and stopping the experiment
- Saving the experiment
- Changing stimuli while the experiment is running

4.4 Lesson 3: To Specify a Reusable Component

In this lesson you will create a class that implements an integrator, a standard piece of functionality that is often used in microcontroller software. While this is a slightly more complex diagram, the techniques for creating and experimenting with it are the same ones you have learned already.

In this example, you specify an integrator that calculates the distance covered where time and speed are known. The input value will be given in meters per second, and at each interval multiplied with a `dT` in seconds. The value for each time slice is added up in an accumulator. The accumulator stores the distance in meters that has been covered after a certain length of time.

In ASCET, a standard block, such as an accumulator, can be realized with a simple diagram.



4.4.1 Creating the Diagram

Before you start working on the diagram, you should perform the same steps as for the `Addition` component. First create a new folder in the `Tutorial` folder, then add a new class. Finally, you can specify the interface of the methods, then the block diagram and the layout.

You will start by creating the folder and the new class.

To create the integrator class

1. In the Component Manager, open the `Tutorial` folder.
2. Create a new folder and call it `Lesson3`.
3. In the `Lesson3` folder, create a new class and call it `Integrator`.

To define the integrator interface

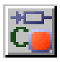
1. Open the `Integrator` class in the block diagram editor.
2. Rename the method `calc` to `integrate`.
3. Edit the method `integrate` and add one argument (type `cont`) and a return value (type `cont`).
4. Place the argument and return value of `integrate` on the drawing area.

The integrator uses two new types of elements: a variable and a parameter.

Variables are used in the same way as they are used in programming languages; you can store values in them and read the values for further calculations. In contrast, *parameters* are read-only. They can only be changed from outside, e.g. they can be calibrated in the experimentation environment, but they cannot be overwritten by any of the calculations within the component itself.

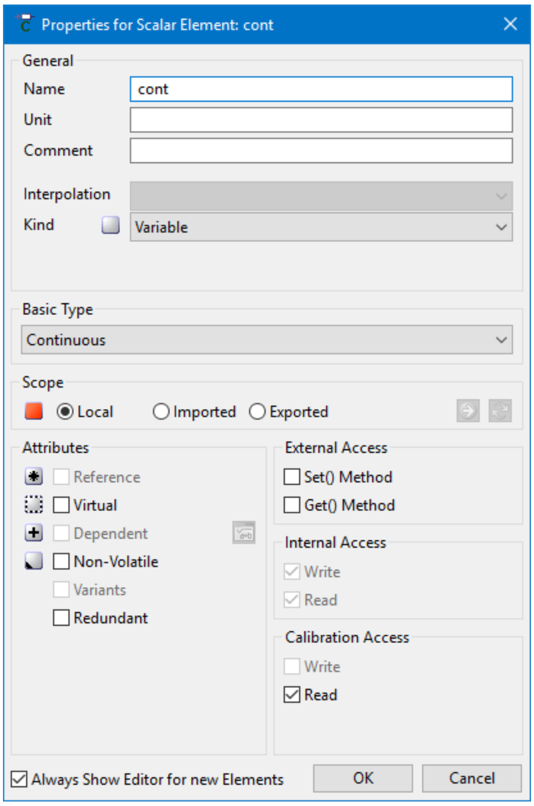
In addition, you may specify a dependent parameter in this example. However, it is irrelevant for the functionality of the integrator. A dependent parameter is dependent on one or several parameters, i.e. its value is calculated based on a change in another one. The calculation or dependency is only carried out on specification, calibration or application. A dependent parameter behaves in exactly the same way in the target code as a normal parameter.

To create a variable



1. Click the **Continuous Variable** button in the "Elements" palette.

   The properties editor opens.



2. Name the variable `buffer`, then click **OK**.

   The cursor shape changes to a crosshair. It is loaded with the continuous variable.

3. Click inside the drawing area to place the variable.

   The variable is placed in the drawing area. Its name is highlighted in the "Outline" tab.

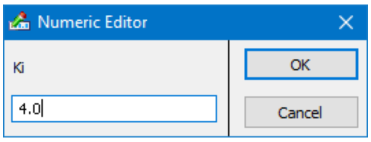
If the properties editor does not open automatically, place the variable in the drawing area. Afterwards, double-click the variable to open the properties editor manually. Make the required settings and activate the **Always show dialog for new elements** option. The next time you create an element, the properties editor opens automatically.

To create a parameter

1. Use the **Continuous Parameter** button to create a parameter named `Ki`.
2. Click inside the drawing area to place the parameter.

3. Right-click the parameter and select **Data** from the context menu.

   A data configuration window (numeric editor) opens.

4. Set the value in the window to `4.0` and click **OK**.

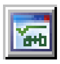
   This value becomes the default value for the parameter. You can assign default values to all parameters or variables in a diagram.

<u>To create a dependent parameter</u>

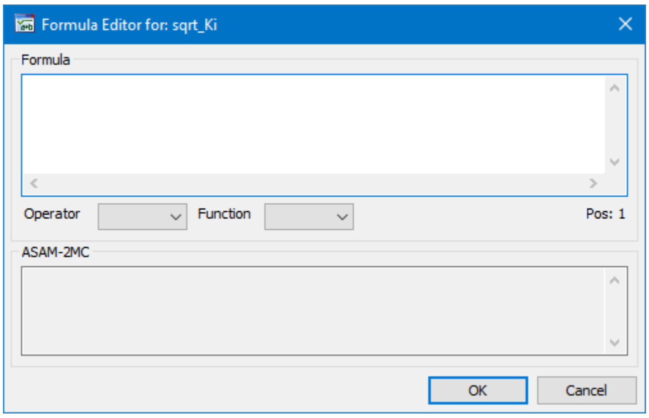
1. Click the **Continuous Parameter** button.

   The properties editor opens.

2. Name the parameter `sqrt_Ki`.
3. In the "Attributes" field, activate the option **Dependent**.



4. Open the formula editor using the **Formula** button.

   The "Formula" field is used to specify the formula for a dependent parameter. A formula consists of functions, operators, and formal parameters.

5. In the "Formula" field, specify the calculation rule.

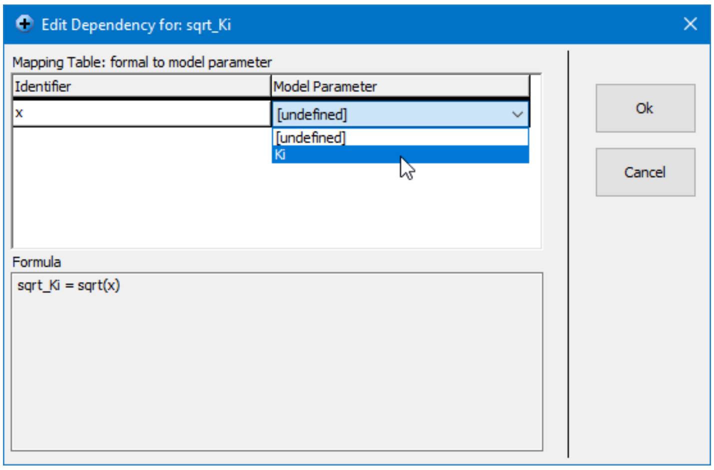
   You can select different operators and functions from the "Operator" and "Function" combo boxes.

   For the example here, select the root calculation of the formal parameter.

   Formal Parameter: `x`
   Formula: `sqrt(x)`

6. Exit with **OK**, and close the properties editor, too.
7. Click into the drawing area to place the parameter.
8. Right-click the `sqrt_Ki` parameter and select **Data** from the context menu.

9. In the "Edit Dependency" window, assign a model parameter from the combo box to the formal parameter (in this example `Ki`).



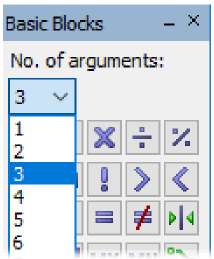
10. Complete data entry with **OK**.

    You have now specified a parameter dependent on the parameter `Ki` which on calibration will automatically be calculated based on `Ki`. Later on in the experiment, you can check the dependency or the calculation.

Now that you have added all the elements, you need to specify an integrator. You can proceed by creating the remainder of the diagram.

To create the diagram

1. In the "No. of arguments" combo box in the "Basic Blocks" palette, set the current value to 3 to specify the number of input values for the multiplication operator.



2. Create a multiplication operator and place it on the drawing area.

3. Click the **Delta t** button to create a dT element.

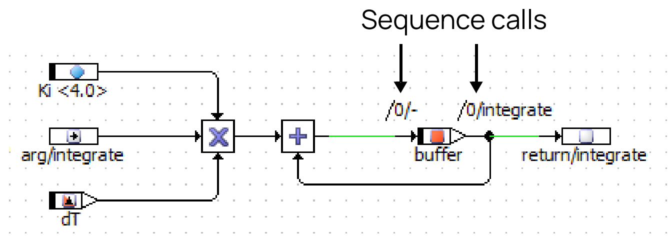
   The properties editor opens. All setting options are deactivated.
4. Close the properties editor with **OK**.
5. Place the dT element inside the drawing area.
6. Create an addition operator with *two* inputs and place it on the drawing area.
7. Connect the elements as shown below.

   The input lines for both the buffer and the return value are displayed in green.

Now all elements of the diagram are in place. Next, you need to determine the sequence of calculation by specifying the sequence calls.

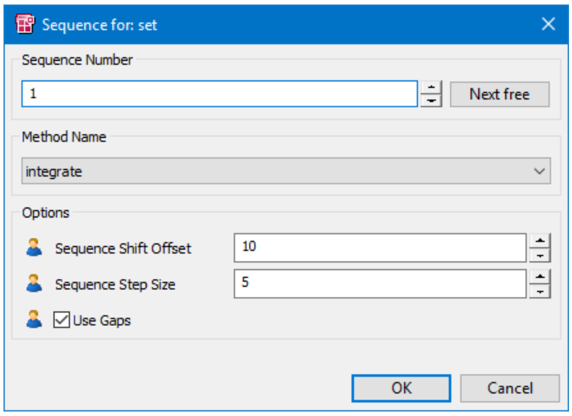
To assign a value to a sequence call

1. Right-click the sequence call above the variable `buffer`.



2. Select **Edit** from the context menu.

   The sequence editor opens.



3. Enter a number in the "Sequence Number" field.
4. Click **OK** to accept the settings.

   The assignment comes first in the algorithm for your integrator.

To adjust the sequence number in a sequence call

1. Open the sequence editor for the sequence call above the return value for `integrate`.
2. In the sequence editor, set the value for "Sequence Number" to a value higher than the first sequence number.
3. Click **OK**.

   The return value is assigned only after the variable `buffer` has been updated.

To adjust the layout

1. Select **Edit** > **Component** > **Layout**.

   The layout editor opens.
2. Drag the argument to the middle of the left-hand side of the block.
3. Drag the return value to the middle of the right-hand side of the block.
4. Click **OK**.

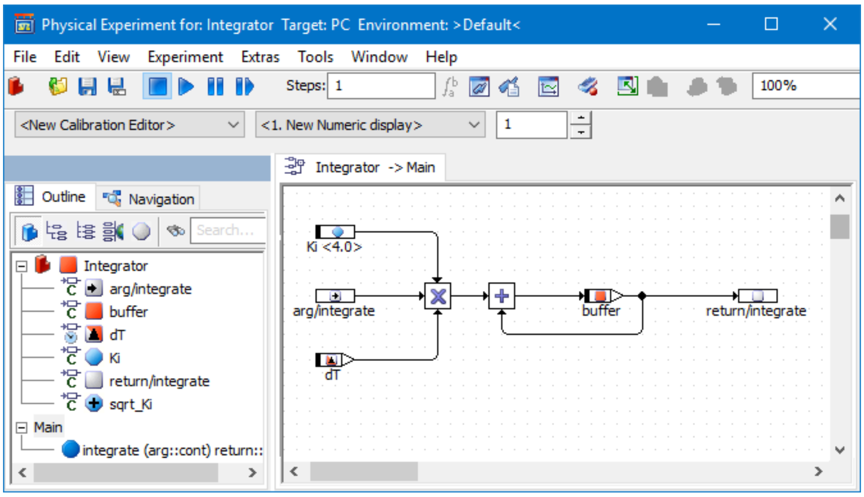
With that, the diagram for the integrator class is complete. Now save the changes to the diagram and to the database/workspace.

4.4.2 Experimenting with the Integrator

Again, first set up the event generator, then the data generator and finally the measurement system.

<u>To set up the experimentation environment for the integrator</u>

1. Start the experimentation environment by selecting **Build > Experiment.**

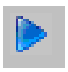


2. Open the "Event Generator" window and activate the event `integrate` using the default `dT value` of 0.01.
3. Close the "Event generator" window.

4. Open the "Data Generator" window and create a data channel for the argument of the `integrate` method.
5. Set the values as follows:

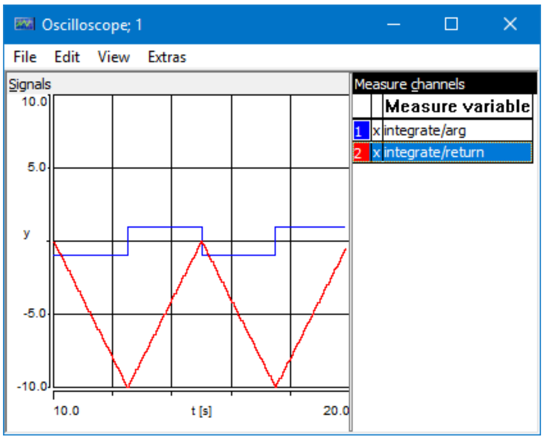
Mode:	`pulse`
Frequency:	`0.2 Hz`
Phase:	`0.0 s`
Offset:	`-1.0`
Amplitude:	`2.0`

6. Close the Data Generator.
7. Open an oscilloscope window with the `arg` and `return` values from the `integrate` method.
8. Set the value axis for both values to a range from -10 to 10 and the time axis extent to 10 seconds.



9. Start the experiment.

The output value of the `integrate` method increases when the argument is positive, and decreases when it is negative. Because the positive and negative parts of the input curve are equal, the output remains within stable boundaries.



To reset an experiment

If you stop an experiment, the current values of variables and parameters are stored; they are used again when the experiment is restarted. It may be desirable to reset all variables or parameters to their initial values.

1. In the "Physical Experiment" window, select **Extras** > **Reinitialize** > **Variables** or **Parameters** or **Both**.

   Depending on your selection, either all variables or all parameters, or both, are reset to their initialization values.

Next, you should experiment with various settings to illustrate the function of the integrator. You can adjust the `Ki` parameter and change the input.

To experiment with the integrator

1. In the "Outline" tab, expand the `Integrator` element.
2. Select the parameter `Ki`.
3. Select **Extras** > **Calibrate**.

   A numerical editor opens for the parameter.
4. Set the value to `5`.

   The output curve on the oscilloscope becomes steeper.
5. Set the value to `3`.

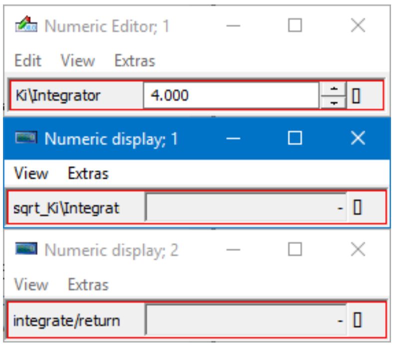
   The output curve now becomes flatter again.
6. Set the parameter back to `4` and close the numerical editor.
7. Open the "Data Generator" window.
8. Set the offset of the input pulse to -0.5.
9. Click **OK**.

Now the positive part is greater, so the output will start to increase. At some point it will exceed the oscilloscope limits. You can adjust the scale of the oscilloscope for each value individually by selecting only that value when you make changes. You can also open a numerical display window to see the output value.

To display a value numerically

1. Select <1.New Numeric Display> in the "Measure View" combo box in the experimentation environment.
2. In the "Outline" tab, select the `return` value from the `integrate` method.
3. Select **Extras** › **Measure**.

   A "Numeric display" window shows the current return value.
4. Also display the dependent parameter `sqrt_Ki`.



5. Change `Ki` and watch `sqrt_Ki` changing automatically.

When you have finished, you can stop the experiment and return to the block diagram editor.

4.4.3 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating a parameter
- Creating and specifying a dependent parameter
- Creating a variable:
- Creating an operator with multiple inputs
- Setting the sequence number of a sequence call
- Assigning a default value
- Calibrating a value during experimentation
- Displaying values in a "Numeric display" window

4.5 Lesson 4: A Practical Example – Controller

In this lesson you will create a controller based on a slightly enhanced standard PI filter. The controller will be used to keep the rotational speed of an idling car engine constant.

When controlling the idling speed of an engine, you have to make sure that the actual number of revolutions `n` stays close to the nominal value for idling `n_nominal`. The value `n` is subtracted from `n_nominal` to determine the deviation that is to be controlled.

The deviation in the actual number of revolution forms the basis for calculating the value of `air_nominal`, which determines the throttle position, i.e. the amount of air the engine gets.

4.5.1 Specifying the Controller

The steps in creating the diagram for your controller are the same as earlier:

- adding a new folder and creating the component in the Component Manager,
- defining the interface and drawing the block diagram.

The major difference is that you will implement the controller as a module. Modules are used as the top-level components in projects. They define the processes that make up a project.

To create the controller component

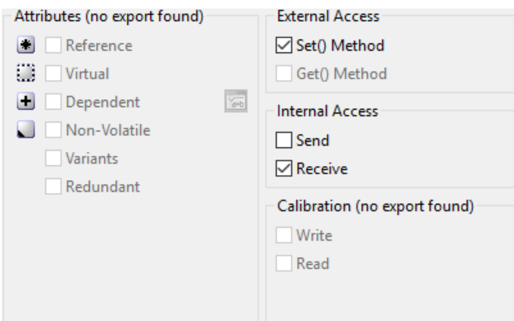
1. In the Component Manager, add a new subfolder named `Tutorial\Lesson4`.
2. Select the `Lesson4` folder and select **Insert > Module > Block diagram** to add a new module.
3. Rename the new module `IdleCon` and open the block diagram editor.
4. In the "Outline" tab, rename the process `process` to `p_idle`.

The functionality of modules is specified in processes, which correspond to the methods in classes. Unlike methods, processes do not have arguments or return values. Data exchange (communication) between processes is based on directed messages, which are referred to as *Receive messages* (inputs) and *Send messages* (outputs) in ASCET.

In your controller, you will use a receive message to process the actual number of revolutions `n` and a send message to adjust the throttle position to `air_nominal`.

To specify the interface of the controller

1. Create a receive message by clicking on the **Receive Message** button, and name it `n`.
2. In the properties editor for the message `n,` activate the **Set() Method** option.



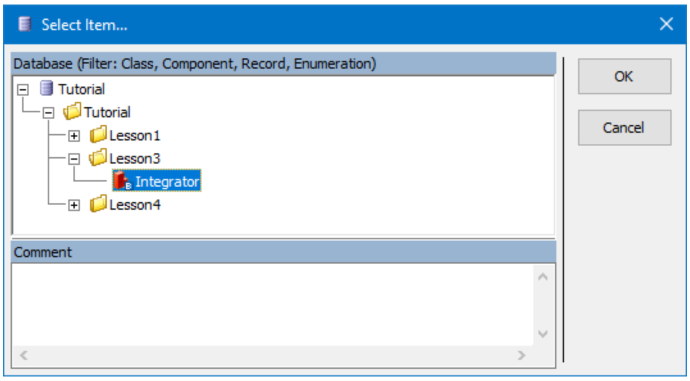
3. Click the **Send Message** button to create a send message `air_nominal`.
4. In the properties editor for the message `air_nominal`, activate the **Get() Method** option.

5. Place both messages in the drawing area.

The controller element uses the integrator you created in Lesson 4.

To add the Integrator to the controller

1. Select **Insert › Component** to open the "Select item" dialog window.



2. In the "Database" or "Workspace" pane, select the item `Integrator` from the `Tutorial\Lesson3` folder and click **OK**.

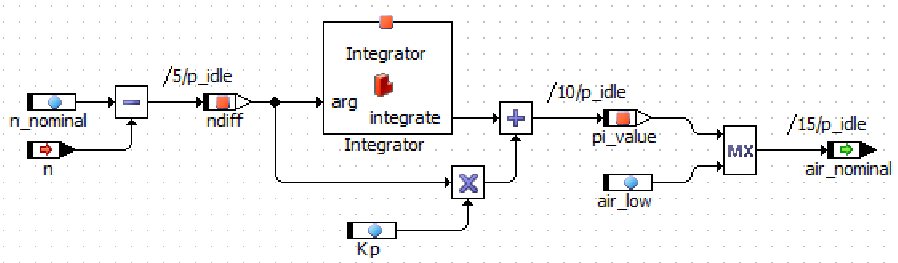
   The integrator is included in the component `IdleCon`. A component is included by reference, i.e., if you change the original specification of the integrator, that change will be reflected in the included component.

In addition to the elements you have added so far, you need to add the following elements to your controller:

- two continuous variables named `ndiff` and `pi_value`
- three continuous parameters named `n_nominal`, `Kp`, and `air_low`

To specify the functionality of the controller

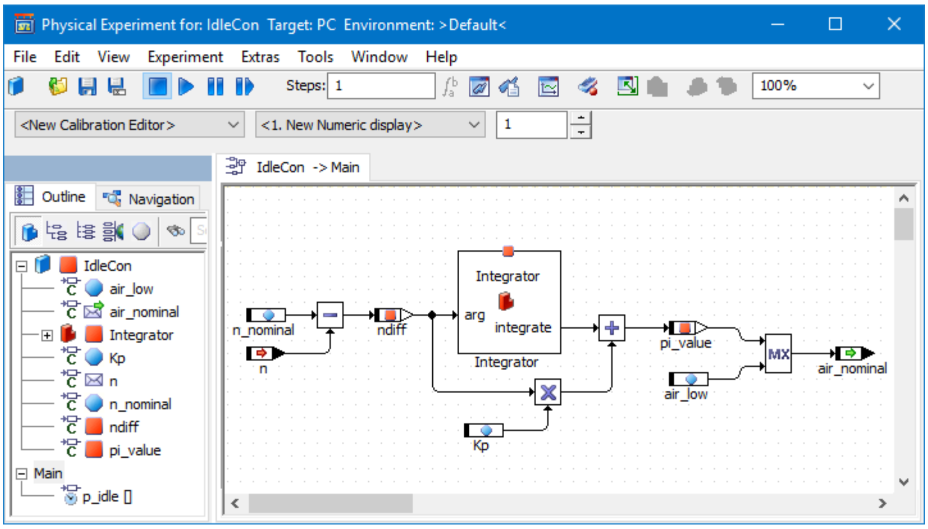
1. Create the operators and the other elements needed, then connect them as shown in the block diagram below.



2. In the "Outline" tab, select the `n_nominal` parameter, then select **Edit › Data**.
3. Set the value for `n_nominal` to 900.
4. Set the value for `Kp` to 0.5.
5. Save your specification in the diagram and apply the changes to the database/workspace.

4.5.2 Experimenting with the Controller

Experimentation with modules works like experimentation with other components. First the data and event generators and then the measurement system are set up.



To set up the experimentation environment

1. Start the experimentation environment.
2. Open the "Event Generator" window and enable the event for the process `p_idle` using the default value of 0.01 for `dT`.

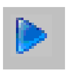
   An event for a process works the same as an event for a method.
3. Open the "Data Generator" window and set up the channel for the receive message `n` with the following values:

Mode:	`pulse`
Frequency:	`1.0 Hz`
Phase:	`0.0`
Offset:	`800.0`
Amplitude:	`200.0`

4. Set up an oscilloscope with the variables `ndiff` and `air_nominal`.
5. In the oscilloscope, set the value axis to -500 to 500 and the time axis extent to 2.
6. Save the environment.

The experiment is now set up to display the relationship between the deviation in the number of revolutions and the throttle position.

To experiment with the controller



1. Start the experiment.
2. Open a calibration window for the parameters `Ki` and `Kp`. From here, you can adjust the values `Ki` and `Kp` and observe their effect on the output.

   From time to time, you may need to reinitialize the model in order to get back to meaningful values.

4.5.3 Project

A project is the main unit of ASCET software representing a complete software system. This software system can be executed on experimental or microcontroller targets in real-time with an online experiment. Individual components can only be tested in the offline experimentation environment.

Every experiment runs in the context of a project. Whenever code is generated for a project, the operating system code is also generated. The operating system specification is required to run an ASCET software system in real-time. Running a software system in real-time is called *Online experimentation*. So far, you have experimented *offline* only, i.e. not in real-time.

NOTE

All ASCET experiments – both online and offline – run within the context of a project. This is clearly seen with offline experiments, which use an (otherwise invisible) default project. Creating and setting up a project for the express purpose of specifying an operating system is only required for online experiments. However, you also have the option of configuring the default project for your own application.

4.5.4 Setting Up the Project

The project is created in the Component Manager. You can add it to the same folder as the `IdleCon` module.

<u>To create a project</u>

1. In the Component Manager, click **Insert Project** to add a new project.
2. Name the project `ControllerTest`.
3. Double-click the project.

   The project editor opens for the project.

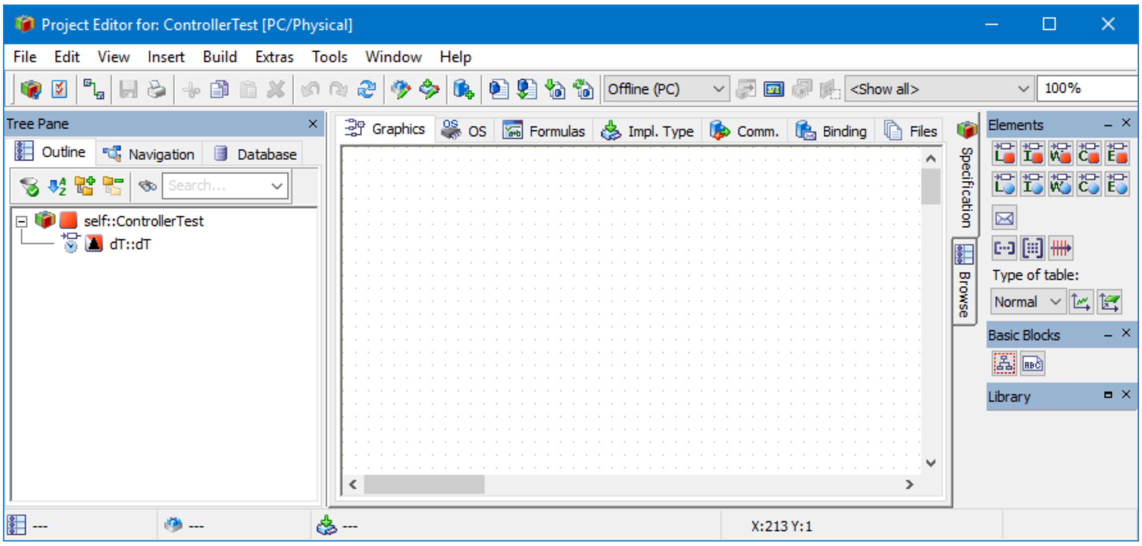


Fig. 4-5 ASCET project editor (Specification view)

4. Add the `IdleCon` controller to the project (cf. Page 59).

   The name of the controller is shown in the "Outline" tab of the project editor.

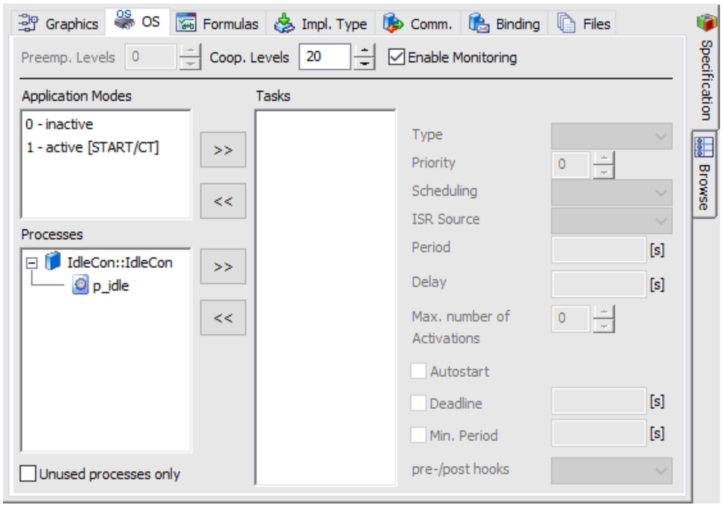
Components are included by reference, i.e. if you change the diagram of an included component, that change will also be effective in the project.

The operating system schedules the tasks and processes of a project. Before you can generate code for the project, you have to create the necessary tasks and assign the processes to them.

The operating system schedule is specified in the "OS" tab of the project editor. You will now specify the operating system schedule to have the `p_idle` process activated every 10 ms.

To set up the operating system schedule for the project

1. Click the "OS" tab.



2. Select **Task > Add** to create a new task.
3. Name it `Task10ms`.

   Newly created tasks are by default alarm tasks, i.e. they are periodically activated by the operating system.
4. Assign the task a period of 0.01 seconds in the "Period" field.

   The period determines how often the task is activated, which is every 10 ms in this case.
5. In the "Processes" list, expand the `IdleCon` item.
6. Select the process `p_idle` and select **Process > Assign**.

   The process is assigned to the `Task10ms` task. It is displayed beneath the task name in the "Tasks" list.

In projects, imported and exported elements are used for inter-process communication. They are global elements that correspond to the send and receive messages in the modules. Global elements must be declared in the project and linked to their respective counterparts in the modules included in the project.

To define global elements

1. In the project editor, select **Extras › Resolve Globals**.

   The necessary global elements are created and automatically linked to their counterparts. Elements with the same name are automatically linked to each other.

4.5.5 Experimenting with the Project

You will now run an offline experiment with this project. Offline experimentation can be performed on the PC without the connection of any additional hardware. Projects run on the PC by default. Therefore you do not have to adjust any settings. Offline experimentation with projects works like offline experimentation with components.

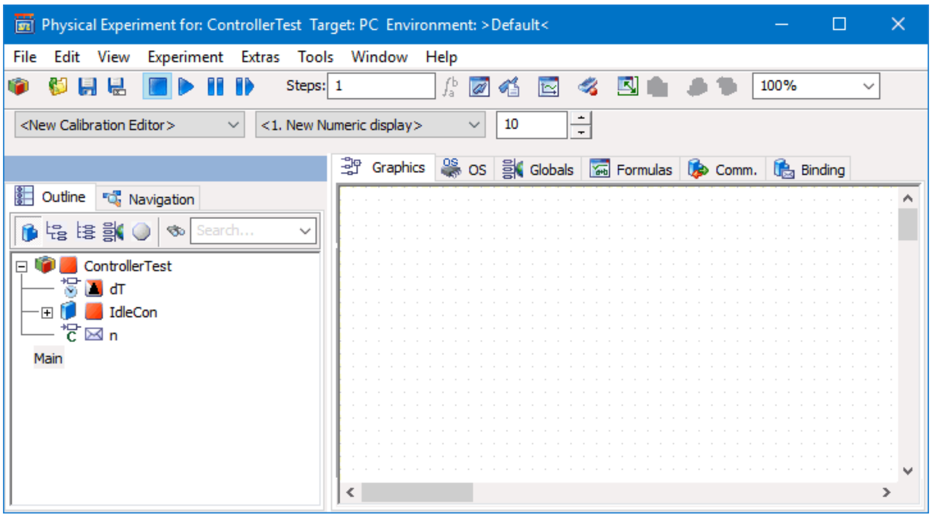
To set up the experimentation environment

1. In the Component Manager, select **File › Save**.

   It is always a good idea to apply your changes to the database/workspace before you start the experimentation environment.

2. In the project editor, select **Build › Experiment**.

   Code for the project is generated and the offline experimentation environment opens.



3. Open the "Event Generator" window.

   In the event generator you see an event for each task you can use in the experiment, rather than for each method or process, as in experimentation with components.

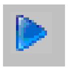
4. Enable the task `generateData` from the event generator and use the default `dT` value of 0.01 seconds.

   The task `Task10ms` is already enabled by default, and both events now have 0.01 seconds as their `dT` value; therefore you do not need to make any further adjustments.

5. Close the event generator.

6. Set up the data generator and measurement system with the same values as in the previous experiment (cf. “Experimenting with the Controller” on page 60).
7. Save the environment.

<u>To run the experiment</u>



1. Click the **Start Offline Experiment** button.
2. Adjust the `Ki` and `Kp` parameters as in the previous section to see the effect of your changes in the output.

4.5.6 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating modules
- Creating messages in modules
- Using components from the Component Manager in a block diagram.
- Creating a project
- Including components in projects.
- Creating tasks and assigning processes to them
- Experimenting with projects

4.6 Lesson 5: Extending the Project

In this lesson you will add some refinements to make your controller more realistic. You will create a signal converter that converts sensor readings into actual values. Many sensors, used for instance in automotive applications, return a voltage that corresponds to a particular measurement value, such as temperature, position or number of revolutions per minute. The relationship between the voltage and the measured value is not always linear. ASCET provides characteristic lines and maps to model this kind of behavior efficiently.

4.6.1 Specifying the Signal Converter

The first step in modeling the signal converter is to create a folder and a module that specifies the functionality. The signal converter uses two characteristic lines to map its input values to the corresponding outputs.

<u>To create the module</u>

1. In the Component Manager, create a new folder `Tutorial\Lesson5`.
2. Create a new module and name it `SignalConv`.
3. Double-click `SignalConv` to open the block diagram editor.
4. In the block diagram editor, select **Insert > Process** to create a second process.
5. Name the processes `n_sampling` and `t_sampling`.
6. Create two receive messages with Set method[1)], `U_n` and `U_t`, and two send messages with Get method[1)], `t` and `n`.

7. Create a characteristic line by clicking the **OneD Table** button.

   The properties editor opens.

8. In the properties editor, do the following:

   i. Name the characteristic line `t_sensor`.

   ii. In the "x" part of the "Dimension" field, enter the value 13.

   The characteristic line can now span a maximum of 13 columns.

   As you have created a (one-dimensional) characteristic line, there is no "y" part of the "Dimension" field.

   iii. In the "Interpolation" combo box, select `Linear` interpolation.

   iv. Click **OK** to close the properties editor.

9. Then click in the drawing area to place the table.

   The table is added to the "Outline" tab.

10. Create a second table named `n_sensor` with maximal 2 columns and linear interpolation.

11. Connect the elements as shown and edit the sequencing to assign the corresponding processes.

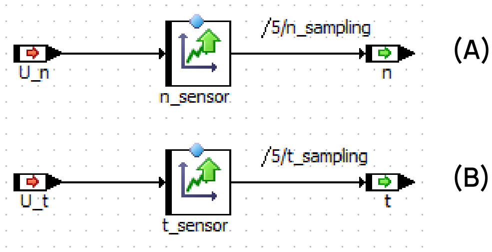


Fig. 4-6 Signal Converter

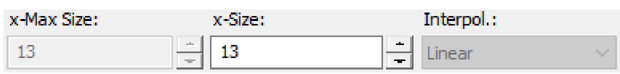
The next step is to edit the data for the two characteristic lines. ASCET provides a table editor for editing arrays, matrices and characteristic lines/maps.

To edit the tables

1. Right-click `t_sensor` and select **Data** from the context menu.

   The table editor opens.

2. Adjust the size of the table as follows:



   The table is extended to 13 columns with all Values set to 0 by default.

1) If you need help, see “To specify the interface of the controller” on page 58.

3. Enter the values listed in the following table. The top row corresponds to the "X" row, the bottom row to the "Values" row.

0.00	0.08	0.30	0.67	1.17	2.5	5.00	7.50	8.83	9.33	9.70	9.92	10.00
-40.0	-26.0	-13.0	0.0	13.0	40.0	80.0	120.0	146.0	160.0	173.0	186.0	200.0

   You should edit the table by entering the sample points (X values) first, starting from left to right.

4. Click an X value and then enter the new value in the dialog box.

   The new X value must be between the limits set by the adjacent sample points.

5. Then enter the output values by clicking a value and typing over the highlighted value.

6. Edit the second table in the same way using the following data:

0.0	10.0
0.0	6000.0

7. Save your specification in the diagram and apply the changes to the database/workspace.

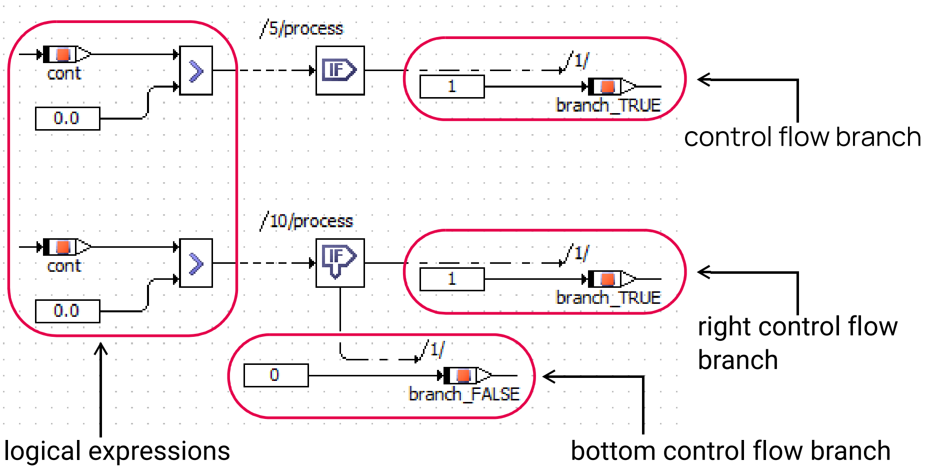
In this example, the second table represents a linear relationship between input and output, therefore it needs only two sample points. This works because you have specified the interpolation mode between values as linear.

In linear interpolation, for an input value between two sample points, the output value is determined from a straight line. In this case, an input of 0 returns 0 and an input of 10 returns 6000. If the input value is 5, the return value is interpolated accordingly as 3000.

4.6.2 If...Then...Else in the Signal Converter

The `If...Then` block evaluates a logical expression and activates a control-flow branch if the result is `true`. The control flow output is connected to one or more sequence calls which are triggered whenever the control flow branch is activated. Whenever the input expression evaluates to `true`, the connected sequence calls are executed.

`If...Then...Else` is similar to `If...Then`, but has two control flow branches. Depending on the value of the logical expression, the right or the bottom branch is executed. The right branch is executed if the expression evaluates to `true`, the bottom branch is executed if the expression evaluates to `false`.

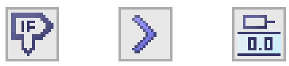


You will now insert an If...Then...Else block to the Signal Converter and use it to change the behavior of `t_sensor` so that `U_t` values `≤ 0.025` produce a temperature `t` of `-40.0`.

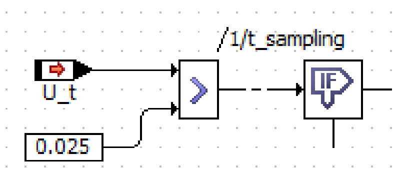
First, you specify the logical expression.

<u>To specify the logical expression for the If...Then...Else block</u>

1) 2) 4)



1. Create an If...Then...Else block by clicking on the **If Then Else** button.
2. Create a **Greater** operator.
3. Create a new graphical occurrence of `U_t`: Drag `U_t` from the "Outline" tab and drop it on the drawing area.
4. Create a literal.
5. Double-click the literal.
6. In the "Literal" editor, enter a value of `0.025`, then click **OK**.
7. Connect `U_t` with the upper input of the operator and the literal with the lower input of the operator.
8. Connect the operator output with the input of the If...Then...Else block.
9. Edit the sequence call as shown in the figure below.

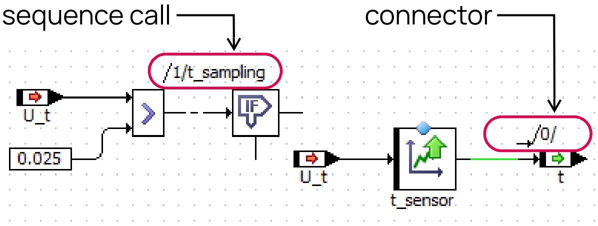


Next, you specify the control flow branches. The expression for the right branch is already there (**A** in Fig. 4-6), it just needs to be connected to the If...Then...Else block.

To specify the control-flow branches

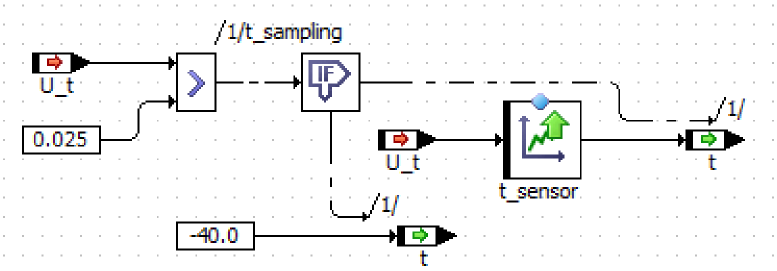
1. To connect an expression to a control flow branch, convert the expression's sequence call to a connector.
   i. Right-click the sequence call and select **Connector** from the context menu.

      A pin appears at the left of the sequence call, the number is reset to 0, and the process name disappears.



   ii. Connect the right branch of the If...Then...Else block to the connector.
2. Create a new graphical occurrence of `t`.
3. Create a literal with value `-40.0` and connect it to the new occurrence of `t`.
4. Connect this expression to the bottom branch of the If...Then...Else block.

The temperature-related part of the diagram should now look like this:

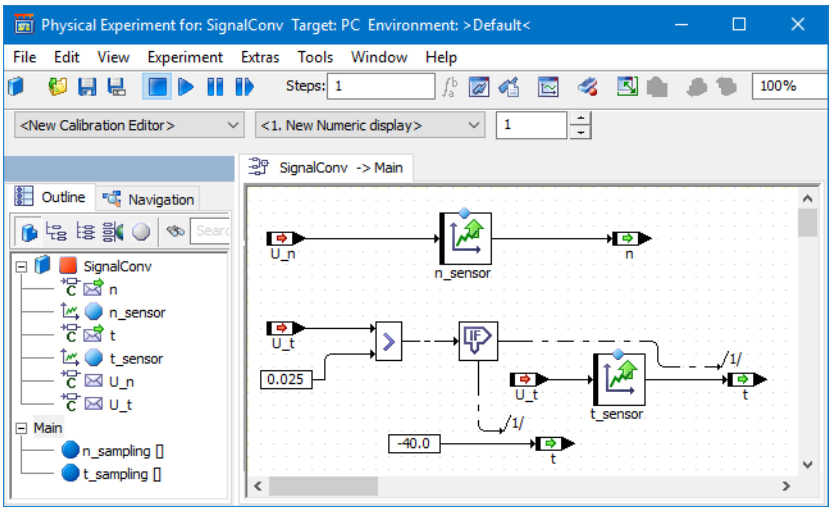


4.6.3 Experimenting with the Signal Converter

You can now experiment with the new component to observe the behavior of the tables. Since the two tables have different value ranges, you will set up a separate oscilloscope window for each of them.

To set up the experimentation environment

1. Start the experimentation environment.



2. Create an event for each process in the component (`n_sampling`, `t_sampling`, `generateData`) and assign a `dT` value of 4 ms to each event.
3. In the data generator, create a channel for the message `U_n` and one for `U_t` and set up both channels with the following values:

Mode:	`sine`
Frequency:	`2.0 Hz`
Phase:	`0.0`
Offset:	`5.0`
Amplitude:	`5.0`

4. Create an oscilloscope window with the messages `n` and `U_n` and a second oscilloscope with the messages `t` and `U_t`.

The resolution of the sampling points and their corresponding interpolation values differs so much that you should configure each channel in the two oscilloscopes individually in order to optimize the way the behavior of the two tables is displayed.

To set up the oscilloscopes for measuring

1. In the oscilloscope for the process `n_sampling` (channels `U_n` and `n`), select the message `n` and select **Extras > Setup**.

   The "Display Setup" dialog window for the message `n` is displayed.
2. Set the range of the value axis to 0 to 6000 and the time axis to 0.5
3. Open the "Display Setup" dialog window for the message `U_n`.
4. Set its value axis to a range from -1 to 11.

   The time axis must be the same for all variables in an oscilloscope window, so you do not have to change that.

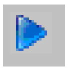
5. Set up the channels in the oscilloscope for the process `t_sampling` as follows:

	U_t	**t**
Min	-1	-40
Max	11	200
Extent	0.5	0.5

6. Save the experimentation environment.

You are now ready to run the experiment and see how your signal converter works. Observe the differences between the two conversion modes.

<u>To run the experiment</u>



1. Start the offline experiment.

   The `n_sensor` table changes only the amplitude of the input sine wave. The input here is a voltage signal ranging from 0 to 10 volts, this is mapped to the rotational speed, ranging from 0 to 6000 revolutions per minute.

   The `t_sensor` table does not represent a linear relationship between the input voltage and the output temperature. It matches the characteristic behavior of temperature sensors commonly used in the automotive industry.

   The influence of the If...Then...Else block can be seen at the lower peak of the `t` curve.

2. Change the data generator channels to different waveforms and observe the effect on both output curves.

4.6.4 Integrating the Signal Converter into the Project

After you have specified the signal converter, you can integrate it in the project you created in Lesson 4. The output signal for the signal converter is used as the input signal for the motor controller.

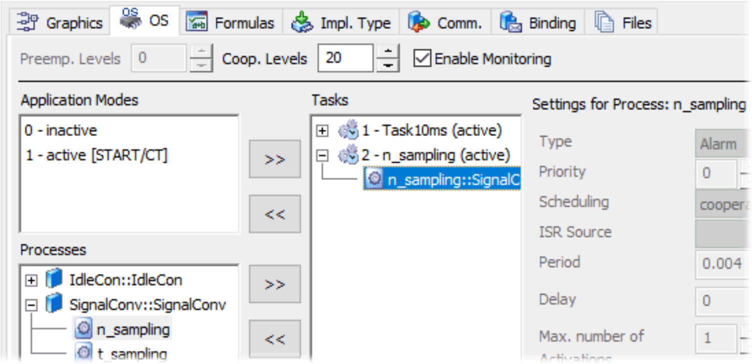
To integrate the signal converter in the project, you will set up another task in the operating system schedule for the new processes and declare and link the global elements necessary for the processes to communicate.

<u>To add the signal converter to the project</u>

1. Open the project editor for `ControllerTest`.

2. Add the module `SignalConv` to the project (cf. Page 59).
3. Click the "OS" tab to activate the operating system editor.
4. Create a new task `n_sampling` with a period of 0.004 seconds.

5. Assign the process `n_sampling` to the task `n_sampling`.

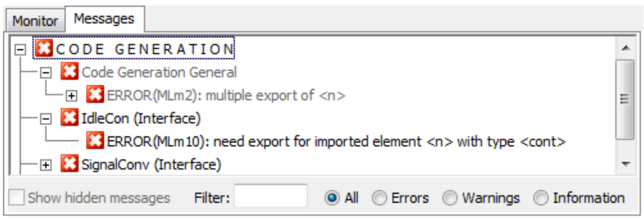


The project now has two tasks. The first task is activated every 10 milliseconds, the second one every 4 milliseconds. All processes assigned to a given task are executed at the interval specified. In the example, each task has only one process, but it is possible to have any number of processes per task.

The next step in integrating the signal converter is to resolve communication between the modules. Communication between the processes works through global elements. All global elements used within a project have to be defined as messages in the corresponding modules.

By default, send messages are defined in a module, while receive messages are normally only imported into a module so they have to be defined now within the context of the project.

Each global element must be defined only once in the project context. Multiple definitions cause code generation errors.

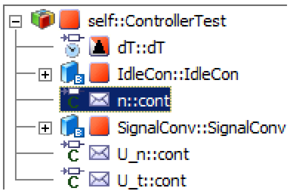


To set up the global elements

1. Select **Extras › Resolve Globals** to set up automatic links.

   All necessary global elements are created and linked automatically to the corresponding elements with a matching name. The global message `U_n`, for instance, is automatically linked to the message `U_n` in `SignalConv`.

2. Delete the message `n` from the project.



   This message was defined in lesson 4 in the project. Now, it is defined in the module `SignalConv`, and it is used for communication between the processes of the modules. The definition in the project is no longer needed.

3. The project may contain unused global elements. To search and delete them, proceed as follows.

   i. Select **Extras > Show Unused Elements**.

      The "Search Results" view opens below the tabs. (See the online help for details.) This view lists all unused elements at the project level and in the modules.

   ii. In the "Elements" tab of the "Search Results" view, select all elements on project level you want to delete and press <DEL>.

<u>To experiment with the project</u>

1. Open the experimentation environment.
2. Open the event generator.

   The new task `n_sampling` is already enabled by default, with 0.004 seconds as `dT` value. You do not need to make any adjustments.

   During offline experimentation with projects, the event generator simulates the scheduling that is performed by the operating system during online experimentation.

3. Open the data generator and delete the existing data channel.
4. Set up a new channel for the message `U_n`:

Mode:	`pulse`
Frequency:	`1.0 Hz`
Phase:	`0.0`
Offset:	`1.333333`
Amplitude:	`0.333333`

   `U_n` is the output voltage of the rotational speed sensor.

   The signal converter converts the voltage value into the actual value for `n` using the characteristic table `n_sensor`.

   The values given above produce an output range for `n` that matches the range from the previous experiment (without signal processing).

5. Save the environment, then start the experiment.

The output curves should be the same as in the example without signal processing. The stimulus created by the data generator is different, but it is then processed in the table so that it looks the same as before.

4.6.5 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating and using characteristic lines
- Creating and using an If...Then...Else block
- Adding components to a project
- Defining the communication between different components in a project

4.7 Lesson 6: Modeling a Continuous Time System

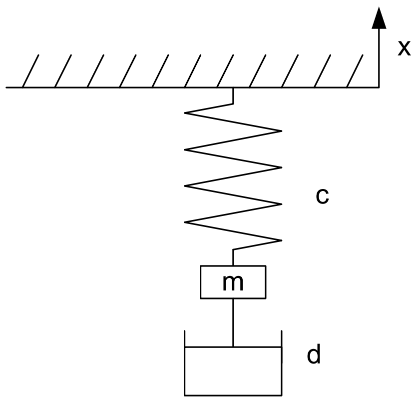
The realistic modeling of physical, mechanical, electrical, and mechatronical processes, often described by differential equations, requires continuous time methods. Before integrating a method like this in the project created in the previous lessons, this lesson covers modeling a continuous time system using a detailed example.

ASCET supports the modeling and simulation of continuous time systems by means of so-called CT blocks. CT stands for "Continuous Time" and refers to items that are modeled or calculated in quasi-continuous time intervals. The continuous time modeling in ASCET is based on state space representation, the standard description form used in the design of continuous time systems. This representation allows the description of CT basic blocks by nonlinear ordinary first-order differential equations and nonlinear output equations. ASCET provides several real-time integration methods to find optimal solutions to these differential equations (refer to the ASCET online help for more information).

The procedure for modeling a continuous time system will now be explained using the example of a mass-spring pendulum with attenuation by the earth's gravity.

4.7.1 Motion Equation

The mass `m` shown in the following illustration is subject to the following forces:



- gravity: $F_g = -mg$
  (g = gravitational acceleration)
- Spring force: $F_F = -c\ (x + l0)$
  (c = spring rate, $l0$ = length of spring at rest, and x = position of mass m)

- Attenuation $F_D = - d\ x'$
  (d = attenuation constant and x' = velocity of mass)

This gives the motion equation as follows:

$mx'' = -mg + F$ or $x'' = -g + F/m$ (with $F = F_F + F_D$)

Breaking the second-order differential equation into two first-order differential equations ($x = x$, $v = x'$) results in:

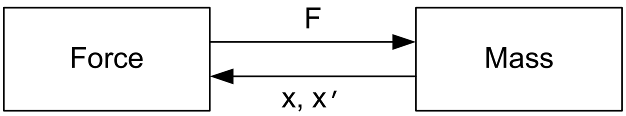
$x' = v$

$v' = -g + F/m$

These differential equations will be used in the following model design.

4.7.2 Model Design

For simplicity, the model of the mass-spring pendulum can be designed using a single CT block. However, to illustrate the "direct pass-through" or "non-direct pass-through" properties and to demonstrate how to avoid an algebraic loop by skillful setting of these properties, we will design this model using two blocks.



- The `Force` block calculates spring force `F` from the position of the pendulum's mass `m` and the friction force from the velocity `x'`.
- From the spring force `F` the `Mass` block calculates the acceleration `x''` from the integration of which the velocity `x'` and the position `x` result.

At first sight, this system looks like an algebraic loop: each block expects an input value from the other block in order to calculate an output value required by the other block.

This algebraic loop can be avoided by clever setting of the *direct pass-through* or *non-direct pass-through* properties:

- In the `Force` block, the output variable `F` depends directly on the input variables `x` and `x'`; see the following equation:

  $F = -c(x + l_0) - dx'$

  This block is thus defined as having a direct pass-through.
- In the `Mass` block however, the output variables `x` and `x'` do not depend directly on the input variable `F`, but on the internal state variables of the block. These, at least at the start, have initial values from which the output variables `x` and `x'` can be calculated, when the input variable `F` is unknown. Otherwise the output variables are calculated using the following differential equations:

  $x' = v$
  $v' = -g + F/m$

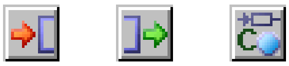
  This block is thus defined as having a non-direct pass-through.

<u>To create the model</u>

1. In the Component Manager, create a folder `Lesson6`.

2. In this folder, use **Insert > Continuous Time Block > ESDL** to create a block `Force` and a block `Mass`.
3. Double-click the `Force` block to open the ESDL editor.

4) 5) 6)



4. Use the **Input** button to create two inputs `x` and `v` (type `continuous`).
5. Use the **Output** button to create an output `F` (type `continuous`).
6. Use the **Parameter** button to create the parameters `c` (spring rate), `d` (attenuation constant) and `l0` (length of the spring at rest).

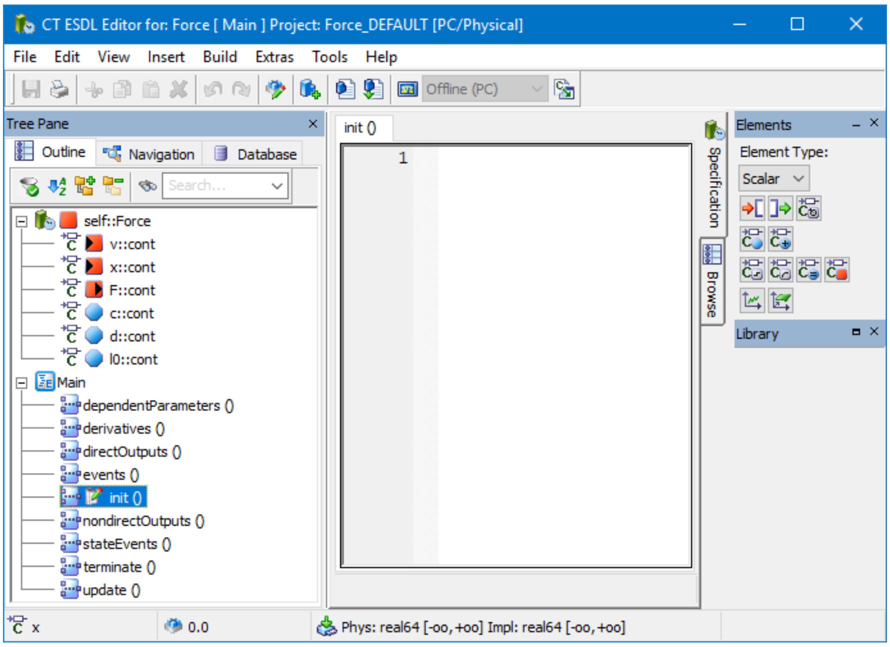
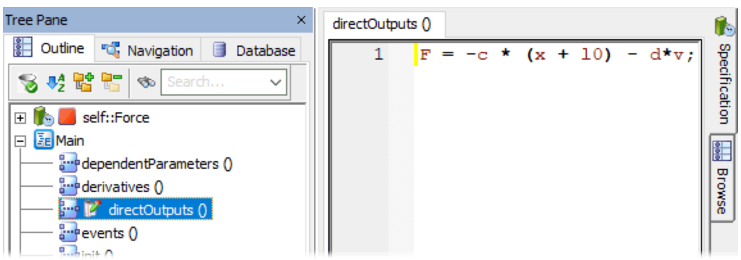


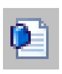
Fig. 4-7 ASCET ESDL editor for CT blocks (Specification view)

The methods in the "Outline" tab are fixed.

7. Assign realistic values to the constants (e.g., 5.0 to the spring rate `c`, 1.0 to the attenuation constant `d`, and 2.0 to the length of the spring at rest `l0`).
8. In the "Outline" tab, click the method `directOutputs()`.
9. In the edit field, specify the formula used to calculate the force:

```
F = -c * (x + l0) - d*v;
```

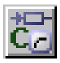




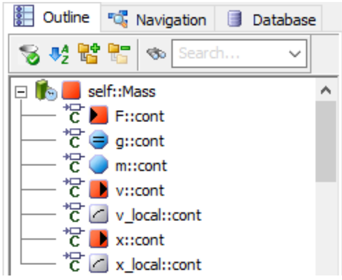
10. Click the **Generate Code** button.

The CT block `Force` is compiled.

11. Double-click the `Mass` block to open the ESDL editor.
12. As above, create an input `F`, two outputs `x` and `v`, one parameter `m` (mass), and one parameter or constant `g` (gravitational acceleration).
13. Assign values to `g` and `m` (9.81 to `g` and, e.g., 2.0 to the mass `m`).



14. Click the **Continuous State** button to create state variables `x_local` and `v_local` for the internal calculation of the outputs.



15. For the `derivatives()` method, specify the differential equations required for the calculation:

```
x_local.ddt(v_local);
v_local.ddt(-g + F/m);
```

16. In `nondirectOutputs()`, pass the state variables `x_local` and `v_local` to the outputs `x` and `v`:

```
x=x_local;
v=v_local;
```

17. In the `init()` method, you can provide the system with realistic initial values for `x` and `v` using the `resetContinuousState()` function.

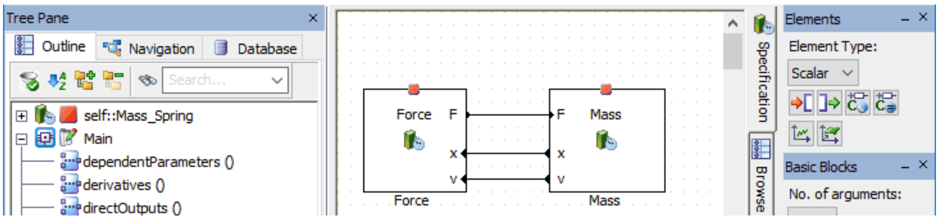
```
resetContinuousState(x_local,0.0);
resetContinuousState(v_local,0.0);
```

18. Click the **Generate Code** button to compile the `Mass` CT block.
19. Adjust the layout of both blocks.

<u>To combine the two basic CT blocks</u>

The combination of the two basic CT blocks into one CT structure block is done using the block diagram editor (BDE).

1. In the Component Manager, `Lesson6` folder, select **Insert** > **Continuous Time Block** > **Block Diagram** to create a new block `Mass_Spring`.
2. Open the new block in the block diagram editor.
3. Add the `Mass` and `Force` blocks to the `Mass_Spring` block.
4. Connect the corresponding inputs and outputs with each other.

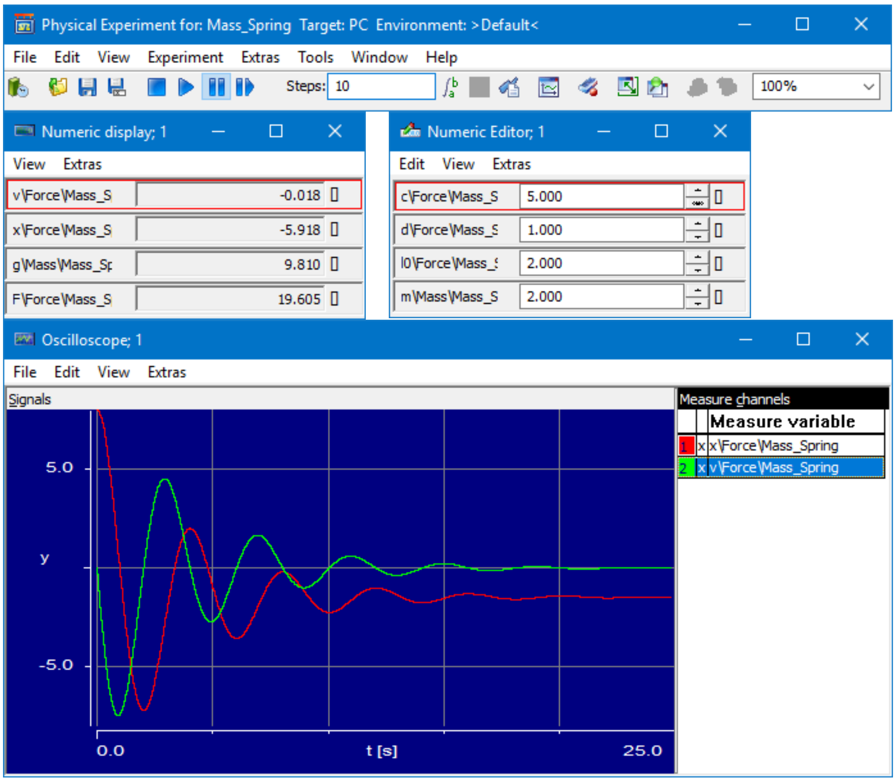


> **NOTE**
>
> Double-clicking a CT basic block opens it in the respective editor. Note, however, that any modification to the blocks affects the entire library, i.e., all structure blocks that use these basic blocks.

5. Select **Build › Experiment**.

   The CT block is now compiled, and the experiment is started.

6. Create the experimentation environment required with numeric editors for the parameters and graphical displays for `x` and `v`.



7. Scale the channels in the oscilloscope separately, from -10 to 0 for `x`, from -8 to +8 for `v`.

8. Set the extent of the time axis to 25 s.

9. Perform the experiment.

4.7.3 Summary

After finishing this lesson, you should be able to carry out the following tasks in ASCET:

- Creating a model to simulate a process
- Using the ESDL editor to create CT blocks with direct and non-direct pass-through
- Using the block diagram editor to combine CT blocks
- Performing the physical experiment

4.8 Lesson 7: Process Model

Following the introduction of CT blocks in the previous lesson, you will now use a CT block for testing your controller. In ASCET you can develop a model of the technical process to be controlled, and then experiment with a closed control loop. This means the controller can be thoroughly tested before it is used in a real vehicle.

In the current example, the motor is the technical process. It returns a value `U_n` which is a sensor reading of the rotational speed of the engine. This value is processed by the controller, which returns a value `air_nominal`. The controller output value determines the throttle- position of the engine, and thus in turn influences the rotational speed.

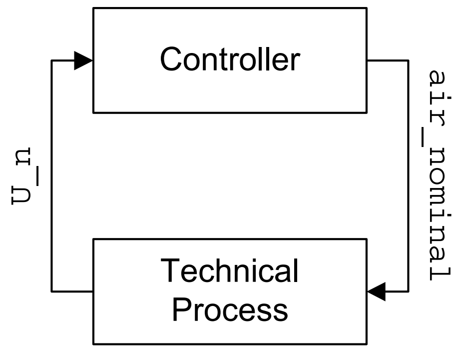


Fig. 4-8 Closed-loop experiment

You will use a CT block for this process model. This type of component is particularly suitable for process models. The model is based on the following differential equation, which models a PT2 system:

$$T^2 s'' + 2DTs' + s = Ku$$

Equ. 4-1 A PT2 - system

The parameters T, D and K have to be set up with appropriate values.

4.8.1 Specifying the Process Model

Creating continuous time components is different from creating other components. They have inputs and outputs, which are the equivalent of arguments and return values. The main difference is that a continuous time block can have multiple inputs and outputs, which are not tied to a particular method. There is a fixed set of methods defined in each continuous time block, that cannot be modified by the user.

You will use ESDL Code for the example here. The syntax of the ESDL code is similar to C++ or Java. An object method is called with the name of the object, a dot, the name of the method and the arguments in brackets followed by a semicolon. The method used for deriving is called `ddt()`. For example, the equation $sp = \dot{s}$ is equivalent to the ESDL statement `s.ddt(sp);`.

<u>To create a continuous time component</u>

1. In the Component Manager create the folder `Tutorial\Lesson7`.
2. To add a continuous time block, select **Insert > Continuous Time Block > ESDL**.
3. Name the new component `ProcModel`.
4. Open `ProcModel` in the ESDL editor.

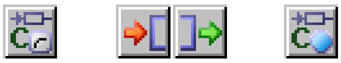
   You can, of course, also use the external text editor. There are instructions for this in the first part of the tutorial.

To edit the process model, first add the elements required and then edit the methods `derivatives` and `non directOutput`.

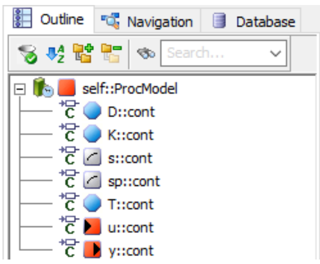
<u>To edit the process model</u>

1. In the ESDL editor, create two continuous states `s` and `sp`.
2. Create an input `u` and an output `y`; both of type `cont`.
3. Create the parameters `D`, `K` and `T`.

   1) **2)** **3)**

   The "Outline" tab for the process model should look like this:

4. Adjust the parameter values as follows:

   ```
   D = 0.4
   K = 0.002
   T = 0.05
   ```

5. In the "Outline" tab, select the `derivatives` method and edit the code as follows:

```
s.ddt(sp);
sp.ddt((K*u - 2*D*T*sp - s) / (T*T));
```

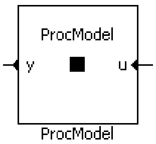
> **NOTE**
>
> See the ASCET online help for specifying CT blocks for information on how to resolve a differential equation.

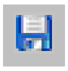
6. Select the `nondirectOutputs` method and type in the following text.

```
y = s;
```

7. Adjust the layout in the layout editor.

   Note that in a process model it is preferable to put the outputs on the left and the inputs on the right.





8. Save the process model.

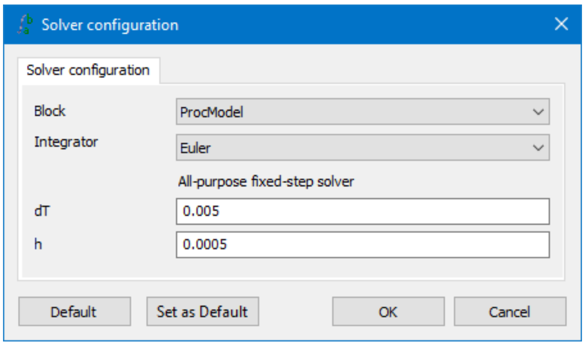
You can now start experimenting with the new model.

To experiment with the model

1. Open the experimentation environment.

2. Click the **Open CT Solver** button.

   The "Solver Configuration" window opens.



3. Enter a dT value of 0.005, then click **OK**.
4. Open the data generator and create a channel for the input `u`.
5. Set up the channel `u` with the following values:

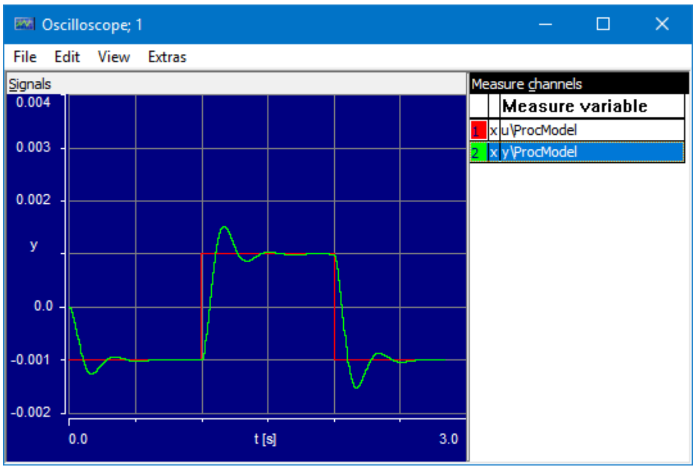
Mode:	`pulse`
Frequency:	`0.5 Hz`
Phase:	`0.0 s`
Offset:	`-0.5`
Amplitude:	`1.0`

6. Open an oscilloscope window with the channels `u` and `y`.
7. Set the measure channels for the oscilloscope as follows:

	u	y
Min	-1	-0.002
Max	2	0.004
Extent	3.0	3.0

8. Save the environment.
9. Start the experiment.

   The output should look like this:



4.8.2 Integrating the Process Model

To create a closed control loop, you will now integrate the process model into the controller project you created in Lesson 4. The steps required are the same as before: including the module, setting up the operating system and linking the global elements.

> **NOTE**
>
> The process model is added to the same project for simplicity. This is often useful in the early stages of testing closed loop simulation. In regular projects, the process model would be distributed over a network in another project since they are not part of the same embedded system.

<u>To include the process model</u>

1. Open the project editor for `ControllerTest`.
2. In the project editor, add the component `ProcModel` to the "Outline" tab.
3. Go to the "OS" tab of the project editor to specify the scheduling for the CT tasks.
4. Select the task `simulate_CT1` and set the value in the "Period" field to 0.01 s.

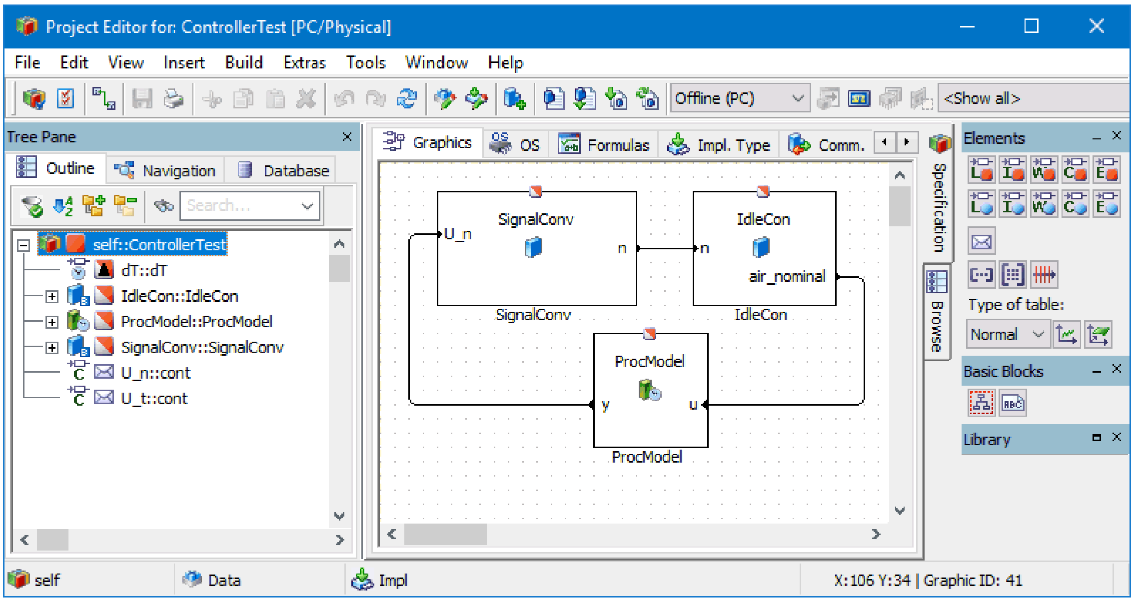
   The controller and the process model both run in the same time interval.

Linking the continuous time blocks and the modules cannot be done automatically. They have to be connected explicitly in a block diagram.

To adjust the linking between modules and CT block

1. Go to the "Graphics" tab.
2. From the "Outline" tab, drag the three components and drop them into the drawing area.
3. Connect the messages of the modules with the corresponding input and output of the CT block.

   To construct the example, connect the output `y` of `ProcModel` with the global message `U_n`, and connect the input `u` of `ProcModel` with the global message `air_nominal`.
4. Right-click each component and select **Ports > Remove Unconnected Ports** to remove these ports from the diagram.



Linking the messages for communicating between modules is done automatically. Messages that have the same name are linked with each other.

The project is now complete and ready for experimentation.

To experiment with the process model

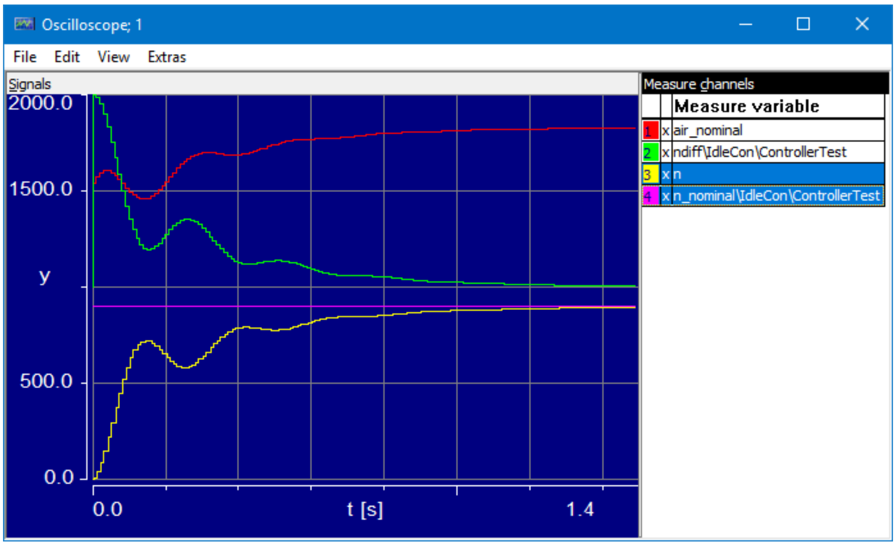
1. Open the experiment environment.
2. Remove all existing channels from the data generator.
3. Open numeric editors for `n_nominal`, `Ki` and `Kp`.
4. Create an oscilloscope window with the elements `air_nominal`, `ndiff`, `n_nominal` and `n`.

5. Set up the channels according to the following table:

	Min.	Max.	Extent
`air_nominal` `ndiff`	-900	900	1.5 s
`n_nominal` `n`	0	2000	1.5 s

6. Save the environment.
7. Start the experiment.

   The values for `air_nominal` and `ndiff` should quickly approach finite values and stay there. The value for `n` should quickly approach `n_nominal` and stay there.



8. Modify `n_nominal` in the numeric editor.

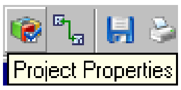
   The value `n` should change in line with the changes to `n_nominal`.
9. You can optimize the behavior of the control loop by adjusting the `Ki` and `Kp` parameters.

4.8.3 Online Experiment with the Process Model

If desired, you will now experiment online, which requires an ASCET-RP installation and an ES910 real-time target. If you do not have both, you have to skip this section.

To set up the project for online experimentation

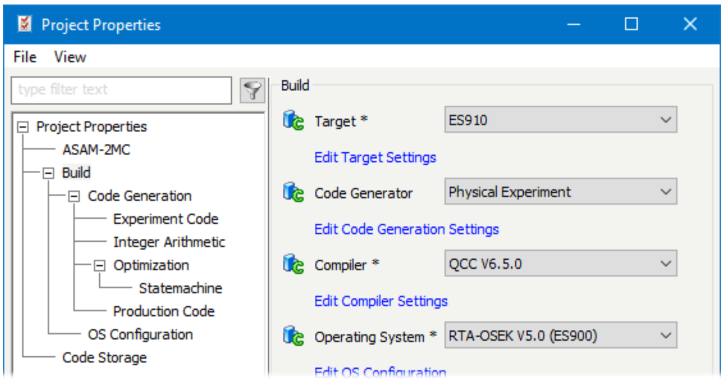
1. Click the **Project Properties** button.



2. In the "Project Properties" dialog window, "Build" node, select the following options:

Target:	`ES910`
Code Generator:	`Physical Experiment`
Compiler:	`QCC V6.5.0`
Operating System:	`RTA-OSEK V5.0 (ES900)`

   These options specify the hardware and the corresponding compiler for code generation.

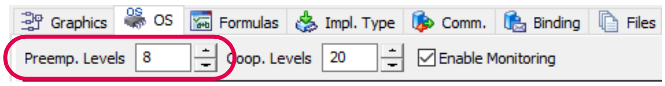


3. Click **OK** to close the "Project Properties" dialog window.

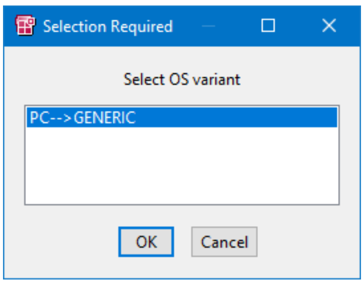
   The buttons **Reconnect to Experiment of selected Experiment Target** and **Select Hardware** are now available.

4. Click the "OS" tab to activate the operating system editor.

5. Set the number of preemptive levels to 8.



6. To copy the schedule you created earlier, select **Operating System › Copy From Target**.



7. From the "Selection Required" dialog, select `PC-->GENERIC` and click **OK**.

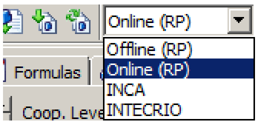
   The project for the new target now has the same scheduling as that specified before for the offline PC simulation.

There are several differences from the offline experiment. In the online experiment, there is no event or data generator. The event generator serves to simulate the scheduling of the operating system tasks generated for online experiments.

In the online experiment, the experiment and the measurements are started separately, and have separate buttons in the toolbar. This is because the measurements may influence the real-time behavior of the experiment, so it may sometimes be necessary to switch them off.

<u>To experiment online with the project</u>

1. Select `Online (RP)` from the "Experiment Target" combo box.

   `Offline (RP)` is intended for offline experiments on the target.

2. Select **Build › Experiment**.

   The code for the experiment is generated and the experiment opens with the environment defined in the offline experiment (see Page 82).

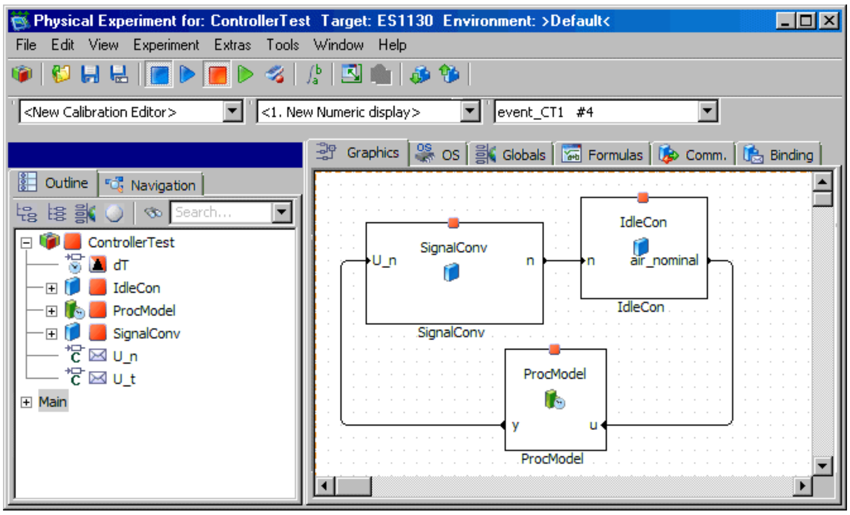
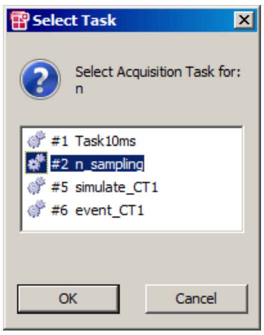


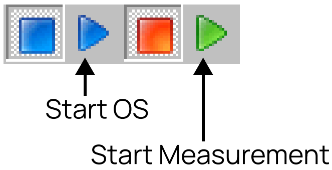
Fig. 4-9 ASCET online experimentation environment

If your project contains several tasks, you may be prompted to select one acquisition task for each measure value.



3. In the "Selection Required" window, select the `#3 simulate_CT1` task and click **OK**.

4. Click the **Start Measurement** button and then click the **Start OS** button.



The experiment starts and the results are displayed on the oscilloscope. The value for `n` should quickly approach `n_nominal` and stay there.

5. Modify `n_nominal` in the numeric editor.

   The value `n` should change in line with the changes to `n_nominal`.

6. You can optimize the behavior of the control loop by adjusting the `Ki` and `Kp` parameters.

4.8.4 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating and specifying continuous time blocks
- Experimenting with continuous time blocks
- Integrating continuous time blocks in a project
- Creating variable links
- Switching between different targets
- Experimenting offline and online with a project

4.9 Lesson 8: State Machines

State machines are useful for modeling systems that move between a limited number of distinct states. ASCET provides a powerful mechanism for specifying components as state machines. In this lesson, you will specify and test a simple state machine that implements a temperature dependent change in the nominal number of revolutions of an idling engine. That state machine will then be integrated into our project. In the next lesson (section 4.10 on page 95), you will then construct a hierarchical state machine.

If the engine is cold, it has to idle at a higher speed to keep it turning over. Once the engine has warmed up, the rotational speed for idling can be decreased to reduce fuel consumption. Our state machine thus has two states: one when the engine is cold, and one when it is warm. It represents a two- phase control.

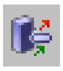
4.9.1 Specifying the State Machine

A state machine consists of the state graph itself and a number of specifications of actions and conditions. The actions and conditions can be specified using either block diagrams or ESDL code. They determine what happens in the various states and during the transitions between states.

The diagrams for actions and conditions are specified in the block diagram editor or ESDL editor. Another possibility is to write ESDL code directly in a text editor, which can be opened for every state and every transition (i.e., *without* opening the ESDL editor). State machines have inputs and outputs for data transfer with other components.

To create a state machine

1. In the Component Manager, create the folder `Tutorial\Lesson8`.



2. Click the **Statemachine** button to create a new state machine.
3. Name it `WarmUp`.
4. Open the state machine in the state machine editor.

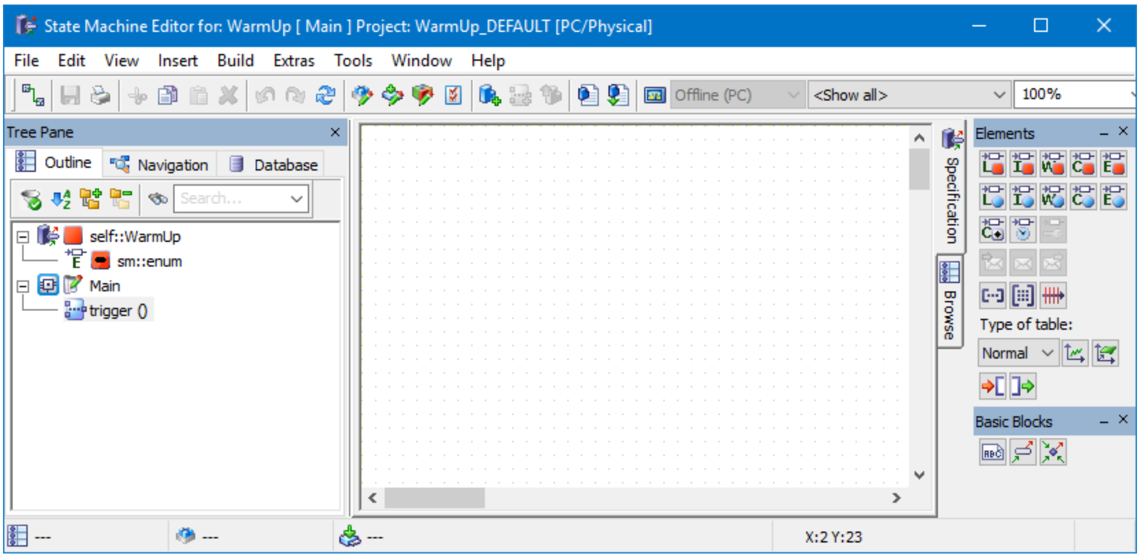
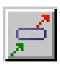


Fig. 4-10 ASCET state machine editor (Specification view)

When you create a state machine, you specify the state diagram first and then define the various actions and conditions associated with states and state transitions.

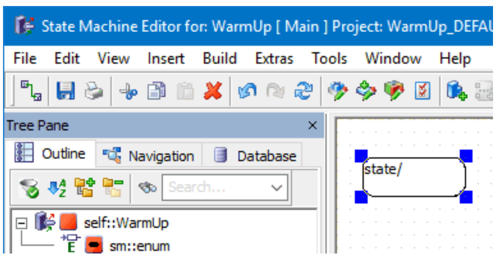
The state machine controlling your motor has two states: one for when the motor is cold and one for when the motor is warm.

To specify the states



1. Click the **State** button to load the cursor with a state item.
2. Click inside the drawing area, where you want to place the state.

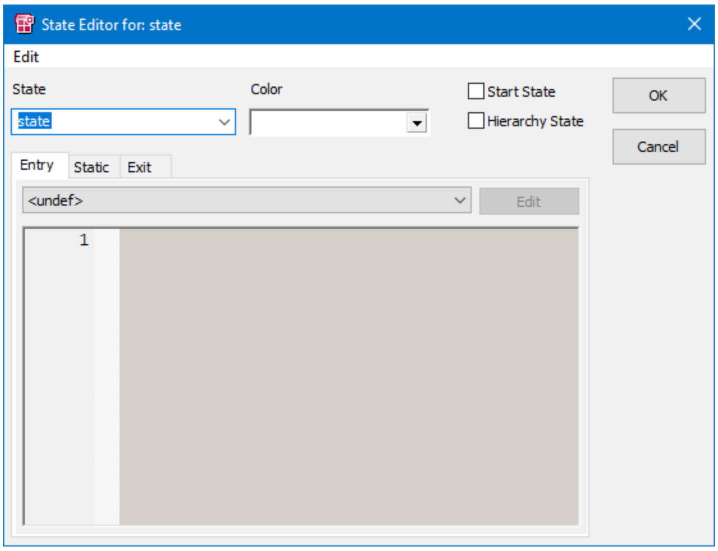
   A state symbol is drawn where you clicked.



3. Create a second state and place it below the first one in the drawing area.

4. Right-click the first state and select **Edit State** from the context menu.

   The State Editor opens.



5. In the "State" field, enter the name `coldEngine`.
6. Activate the **Start State** option to determine the state the machine is in when it is first started.

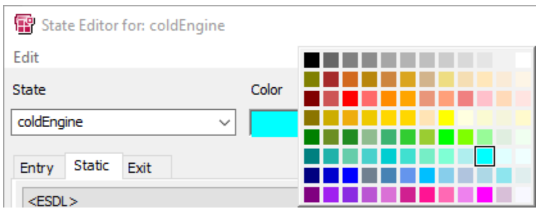
   Each state machine must have one start state.
7. Click **OK** to close the State Editor.

   The name is displayed in the state symbol.
8. Name the second state symbol `warmEngine`.

If desired, change the colors of the individual states to improve clarity.

<u>To set state colors</u>

1. Open the State Editor and select the color in the "Color" combo box.



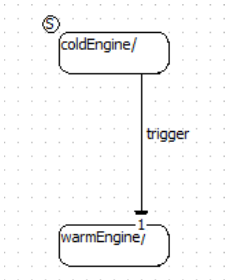
2. Close the State Editor with **OK**.

<u>To create the transitions</u>

1. Right-click in the drawing area, outside any symbol, to activate the connection mode.

2. Click in the right half of the `coldEngine` state symbol to begin a connection, then click in the right half of the `warmEngine` state symbol to connect the two states.

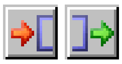
   A line is drawn between the two state symbols. It has an arrow at one end, pointing from the top to the bottom symbol. The lines represent possible transitions between states.

3. Create another transition from `warmEngine` to `coldEngine`.

The next step in building the state machine is to specify its interface. You need an input for the temperature value and an output for the number of revolutions. In addition, parameters are required that specify high and low temperature and number of revolutions per minute.

<u>To specify the interface of the state machine</u>



1. Create an input `t` and an output `n_nominal`.
2. Create four continuous parameters:

   `t_up` = 70

   `t_down` = 60

   `n_cold` = 900

   `n_warm` = 600

You can now proceed by specifying the actions and conditions for both the states and the transitions between states. You can specify three actions for each state:

- The entry action is executed each time the state is entered.

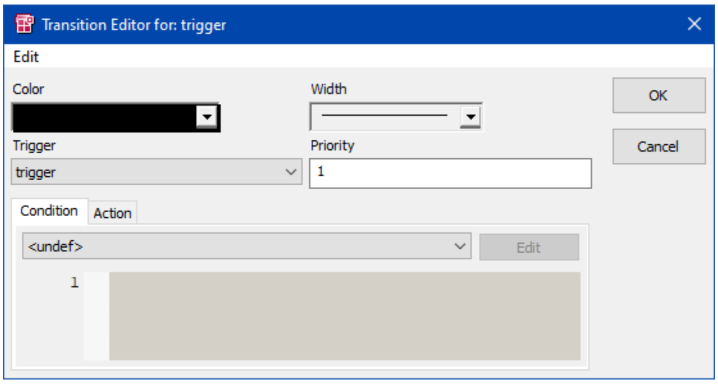
  Exception: Upon first activation of the state machine, the entry action of the start state is *not* executed.
- The exit action is executed each time the state is left.
- The static action is executed while the state machine remains stationary.

Similarly, a trigger event, a condition, a priority and an action can be specified for each transition. The name of the trigger and of the condition appear next to the transition. One trigger is automatically created when the state machine is created.

The actions and conditions are specified in ordinary diagrams or in ESDL code. In this example you will use ESDL code.

<u>To specify the trigger actions and conditions</u>

1. Right-click the transition from the `coldEngine` state to `warmEngine` and select **Edit Transition** from the context menu.

   The condition for a transition from cold to warm is that the actual temperature value `t` is greater than `t_up`.

2. On the "Condition" tab, select `<ESDL>` from the combo box.

   You can influence the default selection in this combo box via the "State Machine" node in the ASCET options window.

3. Enter the following code in the code pane of the condition:

   > **NOTE**
   >
   > In the Transition Editor, the condition is *not* terminated with a semicolon. This is also true for regular ESDL code where conditions appear in parentheses.

   ```
   // reached warmup temperature
   t > t_up
   ```

   The first line is a comment, the second line is the condition.

   If the condition evaluates to `true`, the idle speed of the engine is set to `n_warm`.

   By default, the code of the condition is shown in the state diagram. Here, the comment in the first line is used as an alias name for the condition instead.

4. Select `<ESDL>` for the action, too, and enter the following code:

   ```
   n_nominal = n_warm;
   ```

5. Click **OK** to close the Transition Editor.

6. Look at the diagram. Note that the condition and the action of the transition can be seen.

7. Edit the transition from `warmEngine` to `coldEngine` and enter the following ESDL code for action and transition:

condition:	`t < t_down`
action:	`n_nominal = n_cold;`

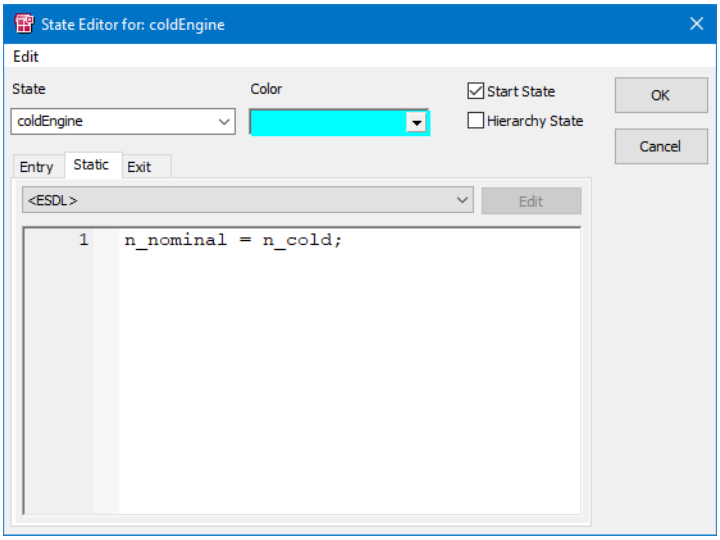
8. Close the transition editor and save the state machine.

You can also specify the actions and conditions as block diagrams instead of ESDL code. See the ASCET online help for details.

The initial value for the output `n_nominal` is still missing. Unlike the parameter values, this cannot be set. Instead, you need to specify an action for the `coldEngine` start state. Since the entry action of the start state is *not* executed at the fist activation of a state machine, you have to specify the initial value in the static action.

<u>To specify a static action</u>

1. Open the `coldEngine` state in the State Editor.



2. Select `<ESDL>` from the combo box on the "Static" tab to specify the static action.

   You can influence the default selection in this combo box via the "State Machine" node in the ASCET options window.

3. Enter `n_nominal = n_cold;` in the code pane to set the initial value of `n_nominal` to 900.
4. Close the State Editor and save the state machine.

That completes the specification of your state machine. Before you start experimenting with it, you should understand the way it works.

4.9.2 How a State Machine Works

While it is usually easy to understand what a standard component does from its graphical specification, the function of a state machine may, at first, be less obvious. This section explains the principles of state machines using the example from the previous section. A detailed description of state machines and their functionality is given in the ASCET online help for the state machine editor.

Each state of a state machine has a name, an entry action, a static action and an exit action. It has transitions to and from other states. Each transition has a priority, a trigger, an action and a condition. All actions are optional.

Each state machine needs a start state. When the state machine is first called up, it is in the start state. It then checks the conditions in all the transitions pointing away from it. In our example there is just one such transition with the condition `t > t_up`. This condition checks whether the input value exceeds the value of the `t_up` parameter. If that is the case, the condition is true, and a transition takes place.

The parameters `t_up` and `t_down` determine the temperature that the engine has to reach, before the nominal rotational speed can be changed. In our example, if the engine temperature rises above 70 degrees, the speed can be reduced to 600 revolutions per minute. If it then falls below 60 degrees, the nominal speed must be reset to 900 revolutions per minute.

Whenever a transition takes place, the transition action specified for the transition is executed. In this example the transition action `n_nominal = n_warm`, which is executed when a transition from `coldEngine` to `warmEngine` takes place, sets the variable `n_nominal` to 600. The transition action `n_nominal = n_cold` sets it to 900 in the reverse case. When a transition occurs, the state machine also executes the exit action of the state it leaves, and the entry action of the state it enters. In our example, these are empty and nothing happens.

Once the state machine has entered the second state, it stays in that state until the condition in the transition from the second to the first state is fulfilled. While the state machine stays in one state, the static action is executed every time the state machine is triggered. Triggering is always an outside event which starts one pass through the state machine.

A pass through a state machine consists of first testing all the conditions on transitions leading away from the current state. Transitions and their conditions are tested in order of their priorities. If a condition is true, the corresponding transition is performed and the exit, transition and entry actions are executed. Once the first condition checks out true, any other transitions leading from the same state but having lower priorities are not tested. If no condition is true, the machine remains in the current state and performs the static action once for each pass.

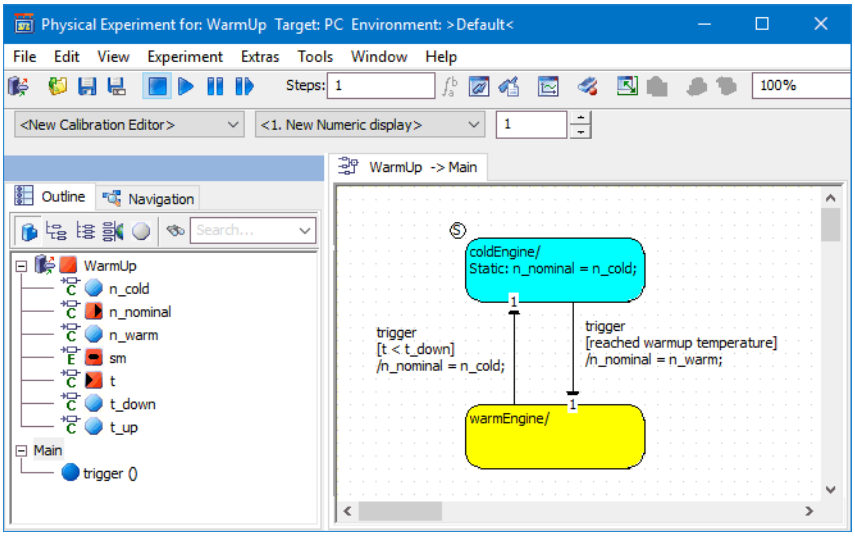
Once the condition in the second transition of our state machine is true, i.e. if the input value falls below the threshold, the state machine returns to the first state. The machine then remains in that state until the input value grows larger than the threshold again.

4.9.3 Experimenting with the State Machine

The experimentation environment works the same for state machines as for other types of components. One extra feature for experimenting with state machines is their animation, i.e. the current state is highlighted in the state machine diagram while the experiment is running.

To experiment with the state machine

1. Open the experimentation environment.



2. Right-click one of the states and select **Animate States** from the context menu.
3. Enable the `trigger` event.
4. In the data generator, create a channel for the variable `t`.
5. Assign a sine wave with frequency `1 Hz`, offset `70`, and amplitude `20` to the channel.
6. Open an oscilloscope window for `t` and `n_nominal`.
7. Experiment with the state machine.

The value of `n_nominal` changes according to whether the sine-wave exceeds or falls below the corresponding temperature threshold value. You can change the threshold using the calibration system to observe the effect of different values on the output. Also, in the state diagram the current state is highlighted.

4.9.4 Integrating the State Machine in the Controller

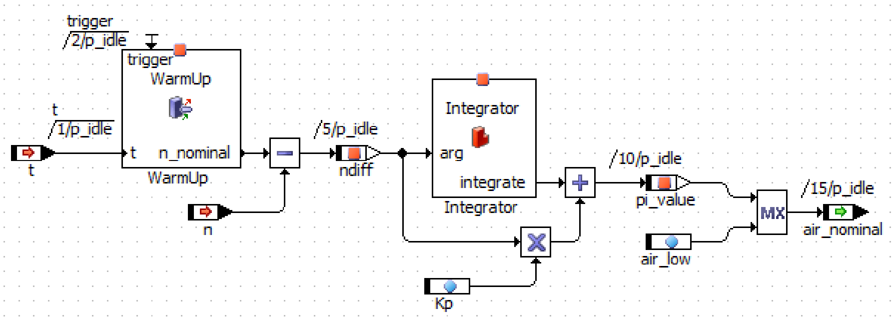
Like other components in ASCET, a state machine can be used as a building block within another component of any type. You can now integrate the state machine into your controller module to adjust the rotational speed to the engine temperature.

To integrate the state machine

1. Open `IdleCon` in a block diagram editor.
2. Remove the parameter `n_nominal` from the diagram and then from the "Outline" tab.

   You will replace the parameter with the state machine in the block diagram.
3. Add the state machine to the controller.
4. Create a receive message and name it `t`.

5. Connect the output of the `WarmUp` component with the subtraction operator in place of the deleted parameter, and connect the input of `WarmUp` with the receive message `t`.
6. Adjust the diagram as shown below. Be sure to adjust the sequencing in the diagram to include all items in the correct order.



7. Save `IdleCon`.

In order to make the modified controller work with your project, you have to make some adjustments to the project. At this point you will also integrate the temperature sensor, which has been left unused so far.

To modify the project

1. Open the project editor for the project `ControllerTest`.
2. Switch to the "OS" tab.
3. Assign the process `t_sampling` to the task `Task10ms`.
4. Use the command **Task** › **Move Up** to make the process `t_sampling` the first process in that task.

To experiment with the modified project

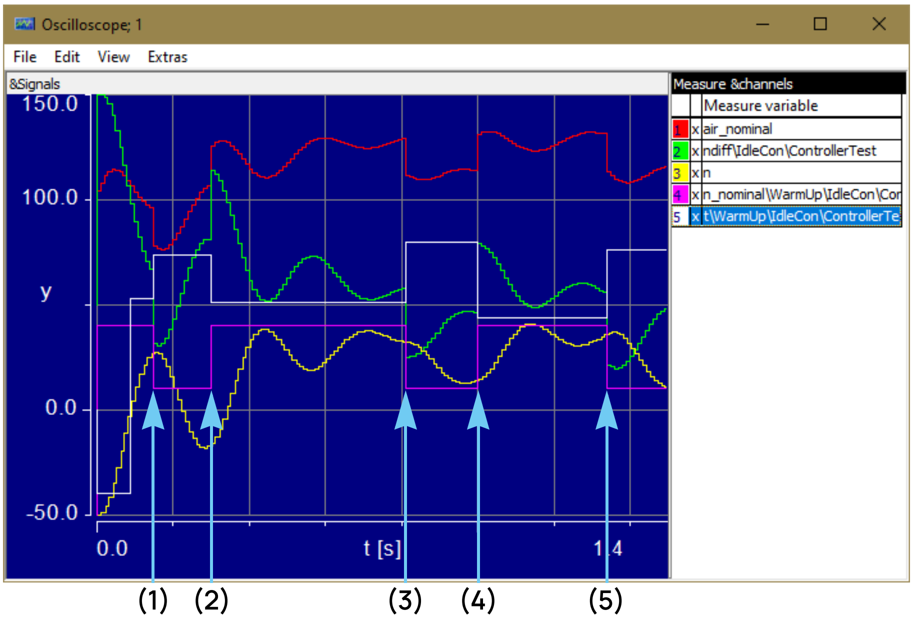
1. Select **Build** › **Experiment**.
2. Set up the experiment as described on Page 82.
3. Open an additional scalar calibration window for the value `U_t`.
4. Add `n_nominal` and `t` to the oscilloscope and set up the channels according to the following table.

	Min.	Max.	Extent
`n_nominal`	0	2000	1.5 s
`t`	-50	150	1.5 s

5. Start the experiment.

6. Adjust the value `U_t` and observe its effect.

   If the value of `t` exceeds the 70 degree limit (1, 3 and 5 in the figure below), the state machine sets `n_nominal` to the lower value of 600. If the temperature falls below 60 degrees (simulated by adjusting `U_t`; 2 and 4 in the figure below), `n_nominal` regains the original value of 900.



4.9.5 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating a state diagram
- Creating and assigning conditions, actions and triggers
- Experimenting with state machines
- Integrating state machines into other components

4.10 Lesson 9: Hierarchical State Machines

Now that you have familiarized yourself with the way state machines work in the preceding lesson, we shall look at creating a more complex system. This unit concentrates on hierarchical state machines. You will also learn how to use the system libraries and components supplied with ASCET, such as timers.

ASCET permits structuring of state machines in closed and open hierarchies. With closed hierarchies, the internal functionality is concealed, with open hierarchies the substates are also shown graphically.

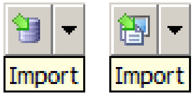
You will build a traffic light control system to run through the individual phases of a traffic light using parameterizable timing. The traffic light will also have an error status where it will flash.

4.10.1 Specifying the State Machine

First you will import the libraries you need and prepare for the task.

To import the system library

1. In the Component Manager, click **Import**.

   DB WS

   The "Select Import File" window opens.

2. In the "Import File" field, use the button to select the `ETAS_System_Library.*`[1] file from the Export directory of your ASCET installation (e.g. `C:\ETAS\ASCET6.4\export`).

   The **OK** button is now available.

3. Click **OK** to start the import.

   The "Items available for import" window opens. All objects contained in the file are selected.

4. Confirm the import of all files with **OK**.

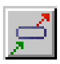
   The files are imported. This can take up to several minutes. When the import procedure is finished, all imported items are listed in the "Imported Items" window.

The second step is to specify the two main states possible for the traffic light (`NormalMode` and `ErrorMode`).

To create the state machine

1. In the Component Manager, create the folder `Tutorial\Lesson9`.
2. Create a new state machine (cf. Page 87) and call it `Light`.
3. Open `Light` in the state machine editor.

   You can start specifying the state machine that will control your traffic light.



4. Create the two states `ErrorMode` and `NormalMode`.
5. Add `Timer` from the `Counter_Timer` folder of the `ETAS_SystemLib` library to the state machine.

To specify the state diagram

1. Specify the necessary data elements as follows:
   - input `error` of type `Logic`
   - outputs `yellow`, `green`, `red` of type `Logic` to symbolize traffic light colors

1) `*` = `exp` (binary export format; can only be imported into a database) or `axl` (XML-based export format)

- continuous parameters `BlinkTime`, `YellowTime`, `GreenTime`, `RedTime` for the different traffic light phases

To get more practice with dependent parameters, you will configure the parameters so that only the green phase is specified; the other parameters are given values dependent on that:

```
RedTime = 2 * GreenTime
YellowTime = GreenTime/3
BlinkTime = YellowTime/10
```

2. Now specify calculations and dependencies of the individual parameters.

   To do this, activate the **Dependent** option under "Dependency" in the properties editor for the parameters `RedTime`, `YellowTime` and `BlinkTime`.

   The properties editor is started with a double-click the element name or via the **Edit** context menu.

3. Click the **Formula** button to start the formula editor.
4. In the formula editor, specify the calculation for each of the dependent parameters.

```
Redtime : 2*x
YellowTime : x/3
BlinkTime : x/10
```

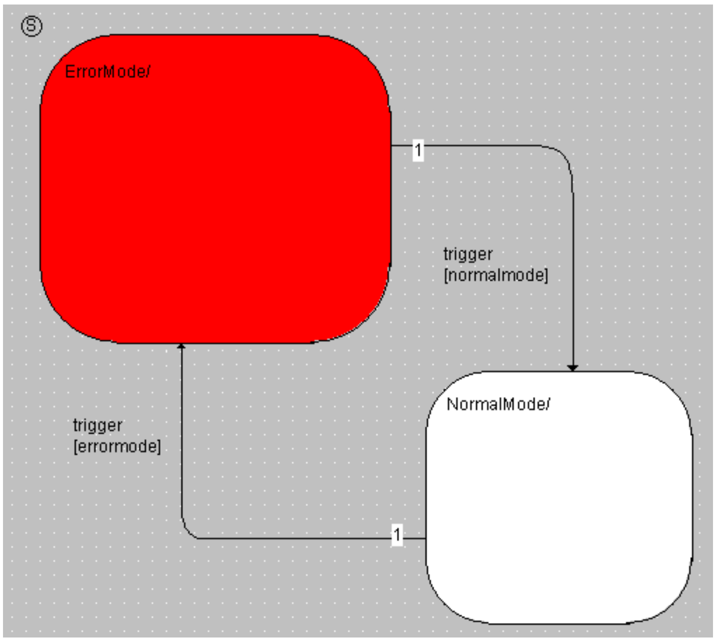
5. Close the formula editor and the properties editor.
6. Open the dependency editor via the context menu **Edit Data.**
7. Assign the corresponding model parameter to the formal parameter `x` for each of the dependent parameters.

```
RedTime : x = GreenTime
YellowTime : x = GreenTime
BlinkTime : x = YellowTime
```

8. Give the data elements meaningful values (e.g. `GreenTime = 30`).
9. Open the state editor for the `ErrorMode` state.
10. Define this state as the start state and color it red.
11. Enlarge both states so that the hierarchies can be inserted.

12. Create and specify the transitions between the two states.

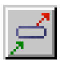
    The normal state `NormalMode` is activated when the input `error` is `false` (i.e. there has not been an error), and `ErrorMode` is activated when there is an error.



13. Save the state machine.
14. You might like to experiment with the main states.

The next step towards creating the traffic light control system is to specify the substates. First specify the performance in the error mode (state `ErrorMode`). In this state, a yellow flashing light will be output. To do this, introduce two substates `YellowOff` and `YellowOn`; with the timer as switch between them. In the `YellowOn` state, the output `yellow` will be set to `true`, while the `YellowOff` state sets it back to `false`.

To specify the substates for the error mode



1. Create the states `YellowOff` and `YellowOn` and place them inside the state `ErrorMode`.
2. Define `YellowOff` as start state, and color `YellowOn` yellow.
3. Define the response of the state `YellowOff` in the state editor.
   i. For the entry action, select `ESDL` in the combo box for the "Entry" tab and enter the following code:

```
green = false;
red = false;
yellow = false;
Timer.start(BlinkTime);
```

   ii. For the static action, enter the following code on the "Static" tab:

```
Timer.compute();
```

4. Now define and describe the `YellowOn` state.

   Entry action:

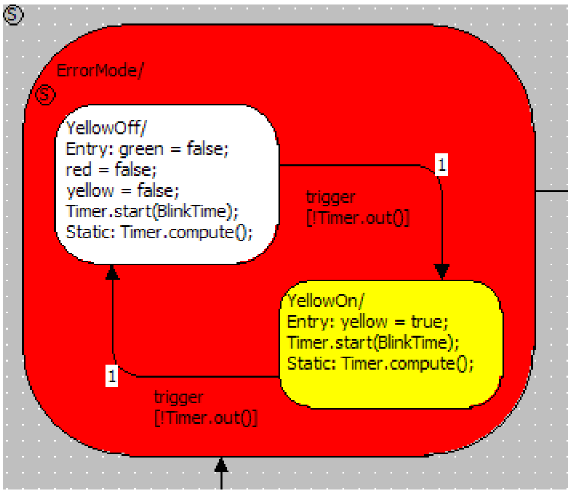
   ```
   yellow = true;
   Timer.start(BlinkTime);
   ```

   Static action:

   ```
   Timer.compute();
   ```

5. Now define the transitions between the two substates.

   The condition for a state transition is that the timer has run out.



This means that the `ErrorMode` state is started in the `YellowOff` state. As well as switching off the color signals, the entry action starts the timer with the parameterizable flashing time. The static action of the `YellowOff` state calls the timer function `compute()` each time, which decrements the timer counter. When this counter is 0, the timer function `out()` returns the code `false`, thus fulfilling the transition condition `!Timer.out()`. The state `YellowOn` works in a similar way, however, in the entry action, the `Yellow` color signal is switched on.

The next step is to specify the performance in normal operation. To do this, create a start state, `AllOff`, and place it within the `NormalMode` state. Use the exit action to set all the color signals to a defined state. Now think about a suitable response for the traffic light control system.

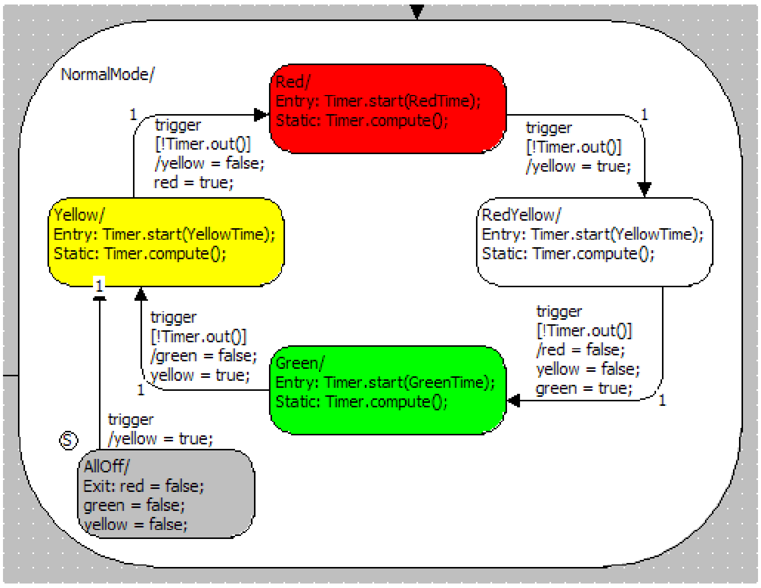
In this example, you should describe the activation or deactivation of the individual color signals in the transition actions, not in the entry actions of the states.

<u>To specify the substates in normal operation</u>

1. Create the states `AllOff` (start state), `Yellow`, `Red`, `RedYellow`, and `Green`, and place them inside the `NormalMode` state.
2. Specify the response for the states by starting the appropriate timer for each color (entry action) and initiating timer processing in the static action (`Timer.compute()`).

3. Define the state transitions and describe the response of the states within the transition actions.

   The transition from `AllOff` to `Yellow` should generally occur, all other transitions should happen after the relevant timer has run out.



4. Enter the actions for each color signal in the "Action" tab of the transition editor.
5. Close the transition editor and save the state machine.

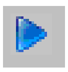
That completes the specification of your traffic light control system. Before you can experiment with it, you should enter meaningful values for the parameters in the various color timers.

4.10.2 Experimenting with the Hierarchical State Machine

You can experiment with the hierarchical state machine in the same way as with the basic state machine. Please do not forget to activate the animation in the experiment.

Experimenting with the State Machine

1. Open the experimentation environment.
2. Right-click one of the states and select **Animate States** from the context menu.
3. Enable the `trigger` event.



4. Start the experiment with the state machine.
5. Change the `GreenTime` parameter and thus the dependent parameters as well.
6. Occasionally, set the `error` input to `true`.

4.10.3 How Hierarchical State Machines Work

Hierarchical state machines work in the same way as normal state machines. In principle, hierarchical state machines only represent a graphic structure of the total set of responses. As an extra task, consider or demonstrate how the response described could be achieved without a hierarchy.

The traffic light example is constructed with two hierarchical states. The system switches between the two states `ErrorMode` and `NormalMode` using the logical input variable `error`. The sub-responses are defined within these states.

To understand this, look at the processing in the `ErrorMode` hierarchy state. Each time the trigger is called, the condition for the transition from the hierarchy state `ErrorMode` to the hierarchy state `NormalMode` is checked (condition: `!error`). If no transition is necessary, the transition from substate `YellowOff` to `YellowOn` or vice versa is checked, and the necessary actions are performed.

If you now look at `NormalMode`, this means that, again, for each trigger call it is first checked whether the input `error` is `true`, and therefore a transition to `ErrorMode` is necessary. Only if this is not the case, the transitions from the sub-states (`AllOff`, `Yellow`, `Red`, `RedYellow`, and `Green`) are checked. In the traffic light example, it is checked whether the timer has run out.

You can have a look at the code generated from the state diagram to clarify this process.

Displaying generated code

1. In the state machine editor, select **Build > View Generated Code** to display the code generated.

   The code from the components is written to a temporary file and then opened with the default C code editor on your computer.

4.10.4 Summary

After completing this lesson you should be able to perform the following tasks in ASCET:

- Creating hierarchical state diagrams
- Describing the way the states behave in actions and also in the transition actions
- Importing modules, classes or components
- Importing system components from ASCET libraries
- Using the Timer system component
- Using dependent parameters
- Displaying generated code

5 Support Function for Feedback to ETAS in Case of Errors

While developing ASCET, the functional safety of the program was of utmost importance. Should an error occur nevertheless, please forward the following information to ETAS:

- Which step were you about to perform with ASCET when the error occurred?
- What kind of error occurred (wrong function, system error or system crash)?
- Which model element or model was edited at the time of the error?

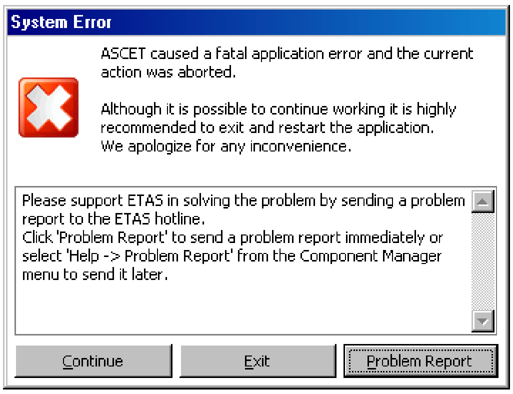
> **NOTE**
>
> To allow ASCET to be updated and developed further, it is important that you report any errors which have occurred with an application to ETAS. You can use the "Problem Report" method for this purpose.

When you use the support function, ASCET compresses the entire contents of the "log" directory (all `*.log` files) including a textual description into an archive file named `EtasLogFiles00.zip` in the `...\ETAS\LogFiles\` subdirectory. For additional archive files, the file name is incremented automatically (up to `19`) to avoid that older archive files are immediately overwritten.

This automatically created zip file contains the following:

- product-related log files created at installation time (necessary for uninstall action)
- ETAS log files stored in the ETAS log files directory matching the file name pattern `*.log`
- recursive registry export of ETAS (32bit)-key (and sub keys): `HKEY_CURRENT_USER\Software\ETAS`
- registry export of ETAS (32bit)-key (and sub keys): `HKEY_LOCAL_MACHINE\Software\ETAS`

If a critical system error occurs, the following window is displayed:



What to do in case of an error

> **NOTE**
>
> It is generally advisable to close the program (without saving) and to restart it. Thus, the risk of possible subsequent errors is omitted.

1. Click the **Problem Report** button.

   The support function is started.

2. Describe the error and forward the information – together with the model – to ETAS.

Or

1. Click the **Exit** button.

   ASCET is closed; all modifications that have not been saved will be lost.

2. Close any message boxes prompting you to save data without saving any data.
3. Restart ASCET.

Or

1. Click the **Continue** button.

   > **NOTE**
   >
   > When you continue using ASCET after a system error, subsequent errors or incorrect configurations cannot be excluded.
   >
   > Use the **Continue** button only if you have to save important configuration data.

   The application continues to run; the program jumps back to the location where it was before the error occurred.

2. Save your data.
3. Exit ASCET.
4. Restart ASCET.

6 Troubleshooting General Problems

This chapter gives some information of what you can do when problems arise that are not specific to an individual software or hardware product.

6.1 Network Adapter Cannot be Selected via Network Manager

Cause: APIPA is Disabled

The alternative mechanism for IP addressing (APIPA) is usually enabled on all Windows systems. Network security policies, however, may request the APIPA mechanism to be disabled. In this case, you cannot use a network adapter which is configured for DHCP to access ETAS hardware. The ETAS Network Manager displays a warning message.

The APIPA mechanism can be enabled by editing the Windows registry. This is permitted only to users who have administrator privileges. It should be done only in coordination with your network administrator.

To enable the APIPA mechanism

1. Open the Registry Editor:
   i. Press <WINDOWS LOGO> + <R>.
   ii. Enter `regedit` and click **OK**.
2. Open the folder `Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpic\Parameters\`.
3. Select **Edit > Find** to search for the key `IPAutoconfigurationEnabled`.

   If you cannot find any instances of the registry key mentioned, the APIPA mechanism has not been disabled on your system, i.e. there is no need to enable it. Otherwise, proceed with the following steps.
4. Set the value of the key `IPAutoconfigurationEnabled` to `1` to enable the APIPA mechanism.

   You may find several instances of this key in the Windows registry which either apply to the TCP/IP service in general or to a specific network adapter. You only need to change the value for the corresponding network adapter.
5. Close the registry editor.
6. Restart your workstation in order to make your changes take effect.

6.2 Search for Ethernet Hardware Fails

There is a number of different causes which might lead to connection problems, most of them being based on inappropriate Windows or hardware settings. Usually, these can easily be modified, once they have been identified.

The following list of causes shall help you in finding the root cause of the problem and fixing it.

Cause: Versions of Hardware and ETAS Software Not Compatible

If you are using ETAS hardware with ETAS MC software, you can use the ETAS HSP Update Tool to check the firmware version of your hardware:

- Make sure you use the ETAS HSP Update Tool with the latest HSP (Hardware Service Pack) version.
- Also use the HSP Update Tool to check whether the hardware is compatible with the MC[1] software used.
- Make sure any additional drivers for that hardware are installed correctly.

You can get the required HSP from the ETAS internet pages at www.etas.com.

If you still cannot find the hardware using the HSP Update Tool, check whether the hardware offers a Web interface and whether you can find using this interface. Otherwise check whether one of the following causes and solutions might apply.

Cause: Personal Firewall Blocks Communication

Personal firewalls may interfere with access to ETAS Ethernet hardware. The automatic search for hardware typically cannot find any Ethernet hardware at all, although the configuration parameters are correct.

Certain actions in ETAS products may lead to some trouble if the firewall is not properly parameterized, e.g. upon opening an experiment in ASCET or searching for hardware from within INCA or HSP.

If a firewall is blocking communication to ETAS hardware, you must either disable the firewall software while working with ETAS software, or the firewall must be configured to give the following permissions:

Permissions given through the firewall block ETAS hardware:

- Outgoing limited IP broadcasts via UDP (destination address 255.255.255.255) for destination ports 17099 or 18001
- Incoming limited IP broadcasts via UDP (destination IP 255.255.255.255, originating from source IP 0.0.0.0) for destination port 18001
- Directed IP broadcasts via UDP to the network configured for the ETAS application, destination ports 17099 or 18001
- Outgoing IP unicasts via UDP to any IP in network configured for the ETAS application, destination ports 17099 through 18020
- Incoming IP unicasts via UDP originating from any IP in the network configured for the ETAS application, source ports 17099 through 18020, destination ports 17099 through 18020
- Outgoing TCP/IP connections to the network configured for the ETAS application, destination ports 18001 through 18020

> **NOTE**
>
> The ports that have to be used in concrete use cases depend on the hardware used. For more precise information on the port numbers that can be used please refer to your hardware documentation.

1) **m**easurement and **c**alibration

Permissions given through the firewall block XCP on Ethernet:

- Outgoing IP multicasts for XCP Slave Detection via UDP to any IP in network, destination IP 239.255.0.0, port 5556.
- Incoming IP multicasts for XCP Slave Detection via UDP from any IP in network, destination IP 239.255.37.45, port 3745.

Contact your IT responsible to clarify whether the required permissions are, or can be, given by the firewall.

> ***NOTICE***
>
> **Unsecured computer system**
>
> Changes to your firewall configuration can make your system unsecured.
>
> Always consult your IT responsible and/or check the IT security policies of your company before changing your firewall configuration and reconnecting the computer to the network.

Cause: Client Software for Remote Access Blocks Communication

PCs or notebooks which are used outside the ETAS hardware network sometimes use a client software for remote access which might block communication to the ETAS hardware. This can have the following causes:

- A firewall which is blocking Ethernet messages is being used (see “Cause: Personal Firewall Blocks Communication” on page 105).
- By mistake, the VPN client software used for tunneling filters messages. As an example, Cisco VPN clients with versions before V4.0.x in some cases erroneously filtered certain UDP broadcasts.

  If this might be the case, please update the software of your VPN client.

Cause: ETAS Hardware Hangs

Occasionally the ETAS hardware might hang. In this case switch the hardware off, then switch it on again to reinitialize it.

Cause: ETAS Hardware Went Into Sleep Mode

In order to save power, some ETAS devices will go to sleep mode if they do not see that they are connected to another device/computer.

To solve that, connect your Ethernet cable from your computer to the "HOST"/ "Sync In" port on the device. After the device turns on, connect to the device using the web interface and change the settings so that the device stays always on. Consult the device's manual for details on how to do that.

Cause: Network Adapter Temporarily Has No IP Address

Whenever you switch from a DHCP company LAN to the ETAS hardware network, it takes at least 60 seconds until ETAS hardware can be found. This is caused by the operating system’s switching from the DHCP protocol to APIPA, which is being used by the ETAS hardware.

Cause: ETAS Hardware Connected to Another Logical Network

If you use more than one PC or notebook for accessing the same ETAS hardware, the network adapters used must be configured to use the same logical network. If this is not possible, it is necessary to switch the ETAS hardware off and on again between different sessions (repowering).

Cause: Device Driver for Network Card Not in Operation

It is possible that the device driver of a network card is not running. In this case you will have to deactivate and then reactivate the network card.

To deactivate and reactivate the network card

1. To deactivate the network card, open the Control Panel.
2. Go to the "Network and Sharing Center", then click the "Change adapter settings" link.
3. In the "Network Connections" window, right-click the used network adapter and select **Disable** in the context menu.
4. In order to reactivate the network adapter right-click it again and select **Enable**.

Cause: Laptop Power Management Deactivates Network Card

The power management of a laptop computer can deactivate the network card. Therefore you should turn off power monitoring on the laptop.

To switch off power monitoring on the laptop

1. Press <WINDOWS LOGO> + <X> to open the power user menu, then select **Device Manager** from that menu.
2. In the Device Manager open the tree structure of the entry **Network Adapters**.
3. Right-click the used network adapter and select **Properties** in the context menu.
4. Select the **Power Management** tab and deactivate the **Allow the computer to turn off this device to save power** option.
5. Select the **Advanced** tab. If the property **Autosense** is included, deactivate it also.
6. Click **OK** to apply the settings.

Cause: Automatic Disruption of Network Connection

It is possible after a certain period of time without data traffic that the network card automatically interrupts the Ethernet connection. This can be prevented by setting the registry key `autodisconnect`.

To set the registry key autodisconnect

1. Open the Registry Editor.
2. Go to `HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\lanmanserver\parameters`, select the Registry Key `autodisconnect` and change its value to `0xffffffff`.

7 Tool Classification for ISO26262

The ISO26262 standard for safety-critical software in automotive systems (ISO26262:2011) requires software development tools to be analyzed to determine what tool qualification measures are required.

Analysis is an assessment of likelihood that a tool introduces errors into the system under development and that those errors go unchecked. It follows that analysis is valid only in the context in which the tool operates, i.e. it can only be assessed in the context of your development process.

This appendix provides some guidance on how to satisfy the requirements on tools arising from ISO26262. References have the form ‹Part›§‹Section›, for example 8§11 means Part 8, Section 11 of the standard.

The key requirements are described in 8§11.4.4, in particular 8§11.4.4.1 regarding planning of qualification and 8§11.4.4.2 regarding the availability of information. Note that some of these requirements have both a user and a supplier obligation. For example, users shall determine the environment in which the tool is used (8§11.4.4.1c), and the supplier shall describe the environment for operation (8§11.4.4.2c).

The following table outlines the input requirements for tool classification according to ISO26262 for which information about ASCET is required and explains where to find supporting evidence.

Requirement synopsis	**ISO26262 Reference**	**ASCET Evidence**
Unique identification number	8§11.4.4.1.a	Versions of ASCET are referred to by their major.minor version number, for example ASCET V6.4. Version strings in ASCET include a sub-minor version number that indicates the refresh number. The initial release is assigned zero, for example the initial release of ASCET V6.4 is version V6.4.0. Each refresh of an ASCET version increments the refresh number. Basic version information is available by selecting **Help** > **About**. Additional information about which parts of the core product are installed is available by selecting **Help** > **Loaded Packages** in the ASCET database browser. This displays version information in the Monitor window. Information about which ASCET-SE targets are installed is accessed through **Help** > **Loaded Targets**. When ASCET is installed, a file called inst.ref is generated in the root of the installation directory. This file contains the fully qualified path names of all files installed and a calculated checksum of the installed files.
Configuration of software tool	8§11.4.4.1.b	The configuration of ASCET is defined by: - The ASCET model itself, either as a workspace or database - Configuration held in the `*.mk` and `*.ini`, `*.a2l`, `*.template` and `*.xml` files in the target directory
Use cases	8§11.4.4.1.c	N/A. This is a property of your use of ASCET in your development process. However, you should read chapter 2 and chapter 3 to understand the basic scope of application of ASCET.
Execution environment	8§11.4.4.1.d	N/A. This is a property of your development process.
Maximum ASIL that may be violated	8§11.4.4.1.e	N/A. This is a property of the system being developed.
Methods for qualification	8§11.4.4.1.f	N/A. This is the output from the analysis process.

Requirement synopsis	ISO26262 Reference	ASCET Evidence
Description of product features	8§11.4.4.2.a	An overview of the product features is provided in chapter 2 of this document. Individual features themselves are described in the online help and the other user documentation.
Provision of user manual	8§11.4.4.2.b	User manuals and online help are supplied with each version of ASCET and for each ASCET add-on. Manuals can be found in *<install dir>*\..\ETASManuals\ASCET Vx.y. Online help is accessed by pressing <F1> or selecting **Help** > **Contents...**
Valid operating environment	8§11.4.4.2.c	The valid operating environment is described in the ASCET Release Notes for each product and add-on. Release notes can be found in *<install dir>*\..\ETASManuals\ASCET Vx.y
Behavior under anomalous operating conditions	8§11.4.4.2.d	Errors in the ASCET configuration (e.g. syntactically or semantically incorrect model, incompatible options etc.) are checked by the tool itself at code generation/build time and reported to in the "Messages" tab of the Monitor window. ASCET cannot be installed on a host PC running an unsupported OS.
Known issues and workarounds	8§11.4.4.2.e	Known issues at the point of release are documented in the ASCET Release Notes. Known issues identified after release, together with workarounds where appropriate, are informed of new KIRs by email. All KIRs are publicly available from www.etas.com/kir. Critical issues identified in a released product are patched with "Hot Fixes". Users are informed of new hot fixes by email. All hotfixes are available for download from the ETAS download center at www.etas.com/en/products/download_center.php.
Detection of erroneous output	8§11.4.4.2.f	ASCET generates warnings to draw your attention to aspects of the model that do not prevent code generation from completing successfully, but may not reflect the intention of design. An additional degree of safety can be achieved by promoting warnings to errors, thereby stopping ASCET from generating code if any promoted warning occurs. ASCET is tested extensively before release in an effort to minimize code generation errors. However, it is recommended that the development process in which ASCET is used includes sufficient measures to ensure that no potential error goes unchecked. A development process complying with the verification requirements in ISO26262 8§9 should be sufficient.

8 Contact Information

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:

www.etas.com/en/hotlines.php

ETAS offers trainings for its products:

www.etas.com/academy

ETAS Headquarters

ETAS GmbH

Borsigstraße 24	Phone:	+49 711 3423-0
70469 Stuttgart	Fax:	+49 711 3423-2106
Germany	Internet:	www.etas.com

Glossary

In this glossary the technical terms and abbreviations used in the ASCET documentation are explained. Many terms are also used in a more general sense, but only the meaning specific to ASCET is explained here.

The terms are listed in alphabetic order.

Abbreviations

ASAM-MCD

Association for **S**tandardisation of **A**utomation- and **M**easuring Systems, with the working groups **M**easuring, **C**alibration, **D**iagnosis
(German: **A**rbeitskreis zur **S**tandardisierung von **A**utomations- und **M**esssystemen, mit den Arbeitsgruppen **M**essen, **C**alibrieren und **D**iagnose)

ASCET

Development tool for control unit software

ASCET-MD

ASCET Modeling and Design

ASCET-RP

ASCET Rapid Prototyping

ASCET-SE

ASCET Software Engineering

AUTOSAR

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

BD

Block **D**iagram

BDE

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

CPU

Central **P**rocessing **U**nit

ECU

Embedded **C**ontrol **U**nit

ESDL

Embedded **S**oftware **D**escription **L**anguage; a textual modeling language

ETK

emulator test probe (German: **E**mulator**t**ast**k**opf)

FPU

Floating **P**oint **U**nit

HTML

Hyper**t**ext **M**arkup **L**anguage

INCA

Integrated **C**alibration and **A**cquisition Systems

INTECRIO

An ETAS product family. INTECRIO integrates code from various behavioral modeling tools, facilitates all necessary configurations, allows the generation of executable code, and provides an experiment environment for the execution of the Rapid Prototyping experiment.

NV

non-**v**olatile

NVRAM

non-**v**olatile RAM

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)

RAM

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

RE

Runnable **e**ntity; a a piece of code in an SWC that is triggered by the RTE at runtime. It corresponds largely to the processes known in ASCET.

ROM

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

RTA-RTE

AUTOSAR runtime environment by ETAS

RTE

AUTOSAR runtime environment; provides the interface between software components, basic software, and operating systems.

SCM

source-**c**ode **m**anagement

SM

state **m**achine

SWC

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

UML

Unified **M**odeling **L**anguage

XML

E**x**tensible **M**arkup **L**anguage

Terms

Action

An action is part of a state machine and associated with states or transitions of the state machine. An action is a piece of functionality, whose execution is triggered by the state machine.

Application mode

An application mode is part of the operating system of ASCET. An operating mode describes different conditions a system can be in, e.g. EEPROM-programming mode, warm-up, or normal mode.

Argument

An argument is the input to a method of a class. Arguments can only be used in the specification of the method they belong to, and not in other methods of the class.

Arithmetic services

User-defined C functions to optimize elementary operations, such as addition operations, and to extend such operations with special properties, such as value limits.

Array

An array is a one dimensional static list of elements of the basic scalar type `continuous` or `discrete`, indexed by the basic scalar type `discrete`.

ASAM-MCD-2MC file

Default exchange format used for projects in ASCII format for the description of measurement and calibration values. The files have the extension `*.a2l`.

Basic model types

Basic model types are used to model physical behavior. There are three types: `continuous`, `discrete` and `logical`. A number of operations, such as addition or comparison, are defined for the basic model types. The implementation is used to transform the model types to implementation types.

Block diagram

A block diagram is a graphical description for a component in which the various elements, operators and inputs/arguments and outputs/return values are connected by directed lines. A block diagram consists of one or more diagrams. The description in terms of block diagrams is a physical description, in contrast to the description with C code.

Bypass experiment

In a bypass experiment, ASCET is directly connected to a microcontroller, and parts of the microcontroller software are simulated by ASCET.

Calibration

Calibration is the manipulation of the values (physical / implementation) of elements during the execution of an ASCET model (experiment).

Calibration window

ASCET working window which can be used to modify parameters.

Characteristic

General term used for characteristic map, curve and value (see also "Parameter" .

Characteristic line

Two-dimensional parameter.

Characteristic map

Three-dimensional parameter.

Characteristic value

One-dimensional parameter (constant).

Class

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

Code

The executable code is the "actual" program with the exception of the data (contains the actual algorithms). The code is the program part which can be executed by the CPU.

Code generation

Code generation is the first step in the transformation of a physical model to executable code. The physical model is transformed into ANSI C-Code. Since the C code is 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.

Component Manager

Working environment in which the user can set up ASCET and manage the data he created and which are stored in the database or workspace.

Condition

A condition is used to describe the control flow in a state machine. It returns a logical value which determines, whether a transition from one state to another takes place.

Constant

A constant is an element that cannot be changed during execution of an ASCET model.

Container

Containers serve as containers for projects, classes and modules. Their purpose is to structure models and databases/workspaces and place different database/workspace items under a common version control.

Data generator

The data generator is part of the experimentation environment. It is used to stimulate the inputs or variables in the model under experimentation.

Data logger

With the data logger measurement data can be read from an experiment and stored to disk for further analysis.

Data set

A data set contains/references the initial data for all elements of a component or project.

Database

A way to store all information specified or produced with ASCET. In ASCET, a database is structured into folders. On the Windows file system, a database is stored in a binary format.

Description file

Contains the physical description of the characteristics and measured values in the control unit (names, addresses, conversion formulas, functional assignments, etc.).

Diagram

A diagram is used for the graphical specification of components as block diagrams or state machines.

Dimension

The dimension is used to describe the 'size' of basic elements. The dimension can either be scalar (zero dimensional), array (one dimensional) or characteristic line/table.

Distribution

A distribution contains the sample points for one or more group characteristic lines/maps.

Editor

See **Calibration window**.

Element

An element is a part of a component which reads or writes data, for instance a variable, parameter or other component used within a component.

Event

An event is an (external) trigger that starts an action of the operating system, e.g., a task.

Event generator

The event generator is part of the experimentation environment. It is used to describe the order and the timing in which events are generated for the activation of tasks (methods/processes/time frames) in the case of an offline experiment.

Experiment

An experiment defines the settings in the experiment environment that are used to test the proper functioning of components or projects. It contains information about the size, position and content of the measurement and calibration windows, as well as the settings of the event generator, data generator and the data logger. An experiment can be executed either offline (non real-time) or online (real-time) and can be used to control a technical process in a bypass or fullpass application. In all cases, instrumented code generated from an ASCET specification is used for experiment execution.

Experiment environment

Main working environment in which the user performs his experiments.

Fixed-point code

From the physical specification, fixed-point code can be generated which can be executed on processors without a floating point unit.

Folder

A folder is a management unit for structuring an ASCET database or workspace. A folder contains items of any kind.

Formula

A formula is part of an implementation describing the transformation from the model types to the implementation (data) types.

Fullpass experiment

In a fullpass experiment, ASCET is directly connected with an experimental microcontroller, and the entire application is simulated by ASCET.

Group characteristic line/map

Group characteristic lines/maps are characteristic lines/maps that share the same distribution of axis points but have different return values. The distribution of axis points and the individual group tables are specified as separate elements.

HEX file

Exchange format of a program version as Intel Hex or Motorola S Record file.

Hierarchy

A hierarchy block is used to structure the graphical specification of a block diagram.

Icon

Icons can be used to illustrate the function of ASCET components.

Implementation

An implementation describes the transformation of the physical specification (model) to executable fixed point code. An implementation consists of a (linear) transformation formula and a bounding interval for the model values.

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 templates. Implementation types contain the main specifications of an implementation; they are defined in the project editor and can be assigned to individual elements in the implementation editors.

Intel hex

Exchange format used for program versions.

Interface

An interface of a component describes how the component exchanges data with other components. It can be compared to the `.h` file in C.

Kind

There are three kinds of elements: variables, parameters, and constants. Variables can be read and written. Parameters can only be read but can calibrated during experimentation. Constants can only be read and not written to during experiments.

L1

The message format for exchanging data between the host and the target, where the experiment is run. Data is transferred, e.g. for displaying values in measure windows.

Layout

A component has a graphical representation that shows pins for the inputs/ arguments, outputs/return values and time frames/methods/processes. Additionally, the layout contains an icon that graphically represents the component when used within other components.

Literal

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

Measuring

Recording of data which is either displayed or stored, or both displayed and stored.

Measure window

ASCET working window which displays measured signals during a measurement.

Measured signal

A variable to be measured.

Measurement

A measurement is the representation of values (physical / implementation) of variables/parameter during an experiment. The values can be displayed with various different measurement windows like oscilloscopes, numeric displays, etc.

Measuring channel parameters

Parameters which can be set for the individual channels of a measuring module.

Message

A message is a real-time language construct of 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 can have one or more arguments and no or one return value.

Model type

Each element of an ASCET component specification is either a component of its own or is of a model type In contrast to implementation types, model types represent physical values.

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.

Monitor

With a monitor the data value of an element can be displayed in a diagram during an experiment.

Motorola-S-Record

Exchange format used for program versions.

Offline experiment

During offline experimentation the code generated by ASCET can be run on the PC or an experimental target, but it does not run in real-time. Offline experimentation focuses on testing the functional specification of a system.

Online experiment

In the online experiment the projects are executed in real-time with the behavior defined in the real-time operating system. The code always runs on an experimental target in real-time. The online experiment focuses on the operating system schedule and the corresponding real-time behavior of the control system.

Operating system

The operating system is used to schedule the execution/activation of an ASCET software system. The operating system also provides services for communication (messages) and access to reserved parts of the hardware (resources).

OSEK operating system

Operating system conforming to OSEK.

Parameter

A parameter (characteristic value, curve and 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

Every task has a priority in the form of a number. The higher the number, the higher the priority. The priority determines the order in which tasks are scheduled.

Process

A process is a concurrently executable piece of functionality that is activated by the operating system. Processes are specified in modules and do not have any arguments/inputs or return values/outputs.

Program

A program consists of code and data and is executed as a unit by the CPU of the control unit.

Project

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

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 and then released again, which is done using resources.

Runnable entity

see RE

Runtime environment

see RTE

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).

State

A state is a part of a state machine. A state machine is always in a one of its states. One of the states is marked as the start state which is the initial state of the state machine. Each state is connected to other states by arcs. A state has an entry action (that is executed upon entry of a state), an static action (that is executed the state remains unchanged) and an exit action (that is executed upon exit of the state).

State machine

A state machine is one of the component types in ASCET. The behavior is described with a state graph consisting of states connected by transitions.

Target

A target is the hardware an experiment runs on. A target can either be an experimental target (PC, Transputer, PowerPC) or a microcontroller target.

Task

A task is an ordered collection of processes that can be activated by the operating system. Attributes of a task are its operating modes, its activation trigger, its priority, the mode of scheduling. On activation the processes of the task are executed in the given order.

Trigger

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

Transition

A transition is a connection between states. Transitions describe possible state changes. Each transition is assigned to a trigger of the state machine, has a priority, a condition, and an action.

Type

Variables and parameters are of type `cont` (continuous), `udisc` (unsigned discrete), `sdisc` (signed discrete) or `log` (logic). `Cont` is used for physical quantities that can assume any value; `udisc` for positive integer values, `sdisc` for negative integer values, and `log` is used for Boolean values (true or false).

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 changed with the calibration system.

Also: General term used for parameters (characteristics) and measured signals.

Workspace

A way to store all information specified or produced with ASCET. In ASCET, a workspace is structured into folders. On the Windows file system, a workspace is stored in form of several XML files.

Figures

Fig. 3-1 Model-Based Development of a Software Function 16

Fig. 3-2 typical rapid prototyping system 21

Fig. 3-3 Unscaled Mapping (a), Arbitrary Mapping (b), 2^7 Scaling (c), 2^8 Scaling with Offset (d) 24

Fig. 3-4 Advanced Software Production Environment 29

Fig. 3-5 ASCET-SCM interface without (a) and with (b) ASCET-DIFF 30

Fig. 4-1 ASCET Component Manager (A: with empty database, B: with empty workspace) 34

Fig. 4-2 ASCET block diagram editor (Specification view) 36

Fig. 4-3 ASCET ESDL editor (Specification view) 41

Fig. 4-4 ASCET offline experimentation environment 44

Fig. 4-5 ASCET project editor (Specification view) 61

Fig. 4-7 ASCET ESDL editor for CT blocks (Specification view) 75

Fig. 4-8 Closed-loop experiment 78

Fig. 4-9 ASCET online experimentation environment 85

Fig. 4-10 ASCET state machine editor (Specification view) 87

Index

A

application mode 115
arithmetic
    fixed-point 24
ASAM-MCD-2MC file 115
ASCET
    in production environment 29

B

block diagram 35

C

C code
    class 25
    module 25
class 18, 116
    C code 25
    tutorial 49
closed-loop simulation 20
component 117
Component Manager 117
condition 117
constant 117
contact information 112
container 117
continuous time blocks 20
    tutorial 73
control algorithm
    classes 18
    classes in modules 19
    continuous time blocks 20
    development 17
    ECU integration 22
    Implementation 22
    modules 18
    parameters 19
    projects 20
    rapid prototyping 20
    reuse 26
    software realization 17
conversion
    floating-point to fixed-point 23

D

data generator 117
data logger 117
data set 117
database 117, 123
description file 117
diagram 117
distribution 118

E

error
    continue 103
    exit 103
    problem report 103
    support function "Problem Report" . 102
    System Error window 102
    what to do in case of ~ 103
ETAS contact information 112
ETAS Safety Advice 8
event generator 118
experiment 118
    implementation 27
    object-based controller
        implementation 27
    object-based controller physical 27
    physical 27
    quantized 27
experiment environment 118
experimenting 43

F

features 12
fixed point code 118
fixed-point arithmetic 24
floating-point to fixed-point
        conversion 23
folder 118
formula 118
fullpass experiment 119

G

general operation
    monitor window 13
    supporting functions 13
glossary 113–123

I

implementation 119
implementation experiment 27
interface 119

K

kind 120

L

layout 120

M

measure 120
measurement 120
measuring channel parameters 120
model type 121
model-based design 16–28
    control algorithm development 17
module 18, 121
    C code 25
    tutorial 58
monitor 121

O

object-based controller implementation experiment 27
object-based controller physical experiment 27

P

parameter 19, 121
physical experiment 27
priority 121
Problem Report 102
process 122
process model 78
Product liability disclaimer 8
production environment 29
program 122
program description 122
project 20, 22, 122
- for embedded microcontrollers 26
- reuse control algorithm 26
- tutorial 61

Q

quantized experiment 27

R

rapid prototyping 20
- hardware configuration component 22
- projects 22

resource 122

S

Safety
- intended use 6

Safety information 7
scheduling 122
scope 122
state 122
state machine 86, 91, 122
- hierarchical 95

support function "Problem Report" 102

T

target 122
task 123
technical system architecture
- test in lab 28
- test in vehicle 28

tool classification for ISO26262 109–111
transition 123
troubleshooting 104–108
- network adapter cannot be selected 104
- search for Ethernet HW fails 104

tutorial 32–101
- continuous time system 73
- controller 57
- CT block 73, 78
- experimenting 43
- extend project 64
- hierarchical state machines 95
- module 58
- process model 78
- project 61
- reusable component 49
- simple block diagram 35
- state machines 86

type 123

V

variables 123