ASCET V6.0

入門ガイド

著作権について

Document EC010010 R6.0.1 JP

目次

1 はじめに

ASCET は、組み込みソフトウェアシステムのファンクション開発とソフトウェア開発のための革新的なソリューションを提供します。ASCET は、新しい独自のアプローチによって、モデリング、コード生成、シミュレーション実験、といった開発工程の各段階を強力にサポートするので、品質向上や開発サイクルの短縮、さらにコスト低減を実現できます。

本書は、読者の方が ASCET について理解し、速やかに成果を得られるように支援するものです。システムについて順を追って紹介すると同時に、すべての情報を参考資料として利用しやすい形にまとめてあります。

1.1 ASCET ファミリ

ASCET 製品ファミリに含まれる各製品には、それぞれシミュレーションプロセッサとのインターフェース、サードパーティのソフトウェアパッケージとのインターフェース、さらに ASCET のリモートアクセスを行うためのインターフェース、といった機能が盛り込まれています。最新バージョンにおいては、ASCET 製品ファミリは以下の製品で構成されています。

- **ASCET-MD**
  モデルの開発とオフラインシミュレーションに使用します。
- **ASCET-RP**
  実験ターゲット（マイクロコントローラを搭載した ETAS シミュレーションボード）を使用した HiL（Hardware-in-the-Loop）シミュレーションやラピッドプロトタイピングをサポートします。また INTECRIO との接続機能も含まれます。
- **ASCET-SE**
  ECU の各種マイクロコントローラターゲットへのコード実装をサポートします。ターゲットとなるマイクロコントローラ用に最適化された実行コード（任意に設定されたオペレーティングシステムを含む）が生成されます。さまざまなタイプのコントローラをサポートし、2 種類の OS を統合できます。

また、以下のオプションモジュールも利用できます。

- **ASCET-DIFF**
  ASCET モデルの比較ツールです。
- **ASCET-SCM**
  外部のコンフィギュレーション管理ツールとのインターフェースを提供します。

これらの他にも、ご要望に応じてユーザー固有の製品を ASCET に組み込むことも可能です。詳しくは ETAS までお問い合わせください。

1.2 マニュアルについて

1.2.1 ユーザープロファイル

このマニュアルは、ECU（自動車制御ユニット）の開発や適合作業の経験がある方を対象としています。このマニュアルをお読みいただくには、信号測定や ECU に関する技術についての専門的な知識が必要です。

また、Windows（2000、XP、Vista のいずれか）の操作についての知識も必要です。メニューコマンドの実行、ボタン操作、さらには Windows のファイルシステム、特にファイルとディレクトリの関係についての知識が必要です。また Windows のファイルマネージャやエクスプローラ等の基本的な使い方を理解されていて、「ドラグアンドドロップ」操作に慣れていることも必要です。

Microsoft Windows の基本テクニックに慣れていない方は、まずそれらについて学習してから、ASCET をご使用ください。Windows オペレーティングシステムについての詳しい説明は、マイクロソフト社発行のマニュアルを参照してください。

またさらに ANSI C や JAVA 等のプログラミング言語に関する知識があれば、ASCET をより効率的にお使いいただくことができます。

1.2.2 マニュアルの構成

ASCET マニュアルは、以下の 3 つのドキュメントで構成されています。

1. 『ASCET V6.0 入門ガイド』
   ASCET で作業する際に必要な基本的な情報がまとめられています。
2. 『ASCET V6.0 インストールガイド』
   ASCET 製品のシステム要件やインストール方法についての情報がまとめられています。
3. ASCET オンラインヘルプ
   ASCET 製品の機能や操作方法が詳しく説明されています。

本ドキュメント『ASCET V6.0 入門ガイド』は、以下の章で構成されています。

- **第 1 章 「はじめに」**（本章）
  ASCET 製品とマニュアルについての概要、および操作時に役立つ一般情報がまとめられています。
- **第 2 章 「ASCET による自動車用組み込み制御ソフトウェアの開発」**
  ASCET 製品ファミリと、それによってサポートされる開発プロセスについての概要です。ASCET を使用する前に必ずお読みください。
- **第 3 章 「ASCET と AUTOSAR」**
  ASCET の AUTOSAR 対応についての概要を説明します。

- **「チュートリアル」**

  この章は、ASCET を初めて使うユーザーを対象としています。例題について実際に作業を進めながら、ASCET の使用方法を学習できます。
  チュートリアルは各工程ごとにいくつかのセクションに分かれていて、各セクションは互いに関連し合っています。なお、チュートリアルの演習を行う際は、あらかじめ「ASCET による自動車用組み込み制御ソフトウェアの開発」（13 ページ）をお読みください。

注記

ETAS では、ASCET のユーザートレーニングを実施しています。トレーニングにご参加いただくことにより、ASCET の基本的な機能と使用方法のほか、より詳しい知識を短時間で習得していただくことができます。

- **「用語集」**

  マニュアルで使用されている技術用語について解説されています。
- **「付録 A：ASCET に関するトラブルシューティング」**

  ASCET 使用時に ASCET 固有の問題が発生した際の対処法とエラーレポートの作成方法について説明されています。
- **「付録 B：一般的なトラブルシューティング」**

  ETAS ソフトウェア製品使用時に発生する可能性のある一般的な問題点とその解決方法がまとめられています。
- **「付録 C：AUTOSAR」**

  ASCET の AUTOSAR 対応機能についての詳しい情報がまとめられています。

ASCET の機能や操作方法についての詳しい情報は、オンラインヘルプに記載されています。オンラインヘルプの使用方法については 1.3.3 項を参照してください。

1.2.3 本書の使用法

表現について

ユーザーが実行するすべてのアクションは、いわゆる“Use-Case”形式で記述されています。つまり以下に示すように、操作を行う目標がタイトルとして最初に簡潔に定義され（例：「新しいコンポーネントを作成する」、「エレメントの名前を変更する」）、その下に、その目標を実現するために必要な操作手順が列挙され、必要に応じて ASCET のウィンドウやダイアログボックスのスクリーンショットが添付されています。

目標の定義：

前置き ...

- 手順 1

  手順 1 についての説明 ...
- 手順 2

  手順 2 についての説明 ...
- 手順 3

  手順 3 についての説明 ...

まとめ ...

具体例：

新しいファイルを作成する：

新しいファイルを作成する際は、他のファイルをすべて閉じておきます。

- **File → New** を選択します。

  “Create file”ダイアログボックスが開きます。
- 新しいファイルの名前を、“File name”フィールドに入力します。

  ファイル名は 8 文字以内でなければなりません。
- **OK** をクリックします。

新しいファイルが作成され、ユーザーが指定した名前で保存されます。このファイルを使用して以降の操作を行います。

表記上の規則

本書は以下の規則に従って表記されています。

表記例	説明
File → Exit を選択して、...	メニューコマンドは、**青の太字**で表記します。
OK をクリックして、...	ユーザーインターフェース上のボタン名は、**青の太字**で表記します。
<Ctrl> を押して、...	キーボードの各キーは、**< >** で囲んで表記します。
“Open File”ダイアログボックスが開きます。	プログラムウィンドウ、ダイアログボックス、入力フィールド等のタイトルは、“ ”で囲んで表記します。

表記例	説明
`setup.exe` ファイルを選択します。	リストボックス、プログラムコード、ファイル名、パス名等のテキスト文字列は、`Courier` フォントで表記します。
論理型のデータから算術型のデータへの変換は**できません**。	注意すべき個所、または新出の用語は**太字**、あるいは「 」で囲んで表記されます。
OSEK グループ（http://www.osek-vdx.org/ を参照してください）はさまざまな標準規格を策定しています。	インターネットへのリンクは、青い下線で表記されています。

特に重要な注意事項は、以下のように表記されています。

注記

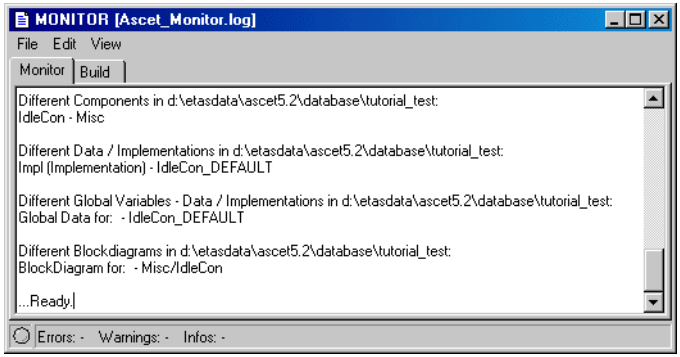
ユーザー向けの重要な注意事項

また PDF 文書において、索引、および他の部分を参照する個所（例：「xx を参照してください」の中の「xx」の部分）については、その参照先へのリンクが設けられているので、必要な参照個所を素早く見つけることができます。

1.3 ユーザーサポート機能

1.3.1 モニタウィンドウ

ASCET によって実行された処理に関する履歴は、モニタウィンドウ（詳しくは ASCET オンラインヘルプを参照してください）に出力されます。履歴データには、エラーや通知メッセージなども含まれます。何らかの情報が出力されると、直ちにモニタウィンドウが最前面に表示されます。



モニタウィンドウは、情報を表示するだけでなく、エディタ機能も持っています。

- モニタウィンドウの“Monitor”タブに出力された履歴データは、自由に編集できます。これによって、ユーザーのメモやコメントを ASCET の出力メッセージに付加できます。
- ASCET のメッセージを、付加したコメントと共にテキストファイルとして保存できます。
- 既に保存されている他の履歴データをロードして、記録された処理値内容を比較できます。

1.3.2 キーボードコマンドの一覧

<Ctrl> + <F1> を押すと、現在有効なキーボードコマンドの概要が表示されます。

1.3.3 マニュアルとオンラインヘルプ

デフォルト設定で ASCET をインストールすると、ASCET の入門ガイド（`ASCET V6.0 Getting Started.pdf`）とインストールガイド（`ASCET V6.0 Installation.pdf`）がマニュアルフォルダ（`ETAS¥ETASManuals`）にインストールされます。また日本語版マニュアルも用意されています。

Windows スタートメニューの ASCET 6.0 フォルダから **Online Manuals** を選択すると、このフォルダにアクセスできます。

PDF ファイルでは、索引やテキスト検索、または随所に設定されているハイパーリンクを使用して、必要な情報に素早くアクセスすることができます。

また、ASCET 使用中に **<F1>** キーを押すと、`ETAS¥ASCET6.0¥Help` ディレクトリにインストールされているオンラインヘルプから関連するトピックが開きます。

2 ASCET による自動車用組み込み制御ソフトウェアの開発

「自動車組み込みソフトウェア」の開発は、自動車の領域（インフォテイメント、シャシー、ボディ、パワートレイン）間や会社（自動車メーカーとサプライヤ）間の協力が要求される総合的な事業です。しかも、自動車組み込みソフトウェアは、個々のメカニカルサブシステムにおいて非常に重要な部分を占めています。その特徴は以下のような点です。

- センサからデータを読み取り、アクチュエータに送る制御値を計算する制御アルゴリズムを実現します。
- 一般的に、ECU（**E**lectric **C**ontrol **U**nit: 電子制御ユニット）内で、1 つまたは複数のマイクロコントローラとその他の電気部品や電子部品を用いて稼働します。
- 通常、自動車のライフタイム全体にわたって変更されることはありません。
- メカニカルサブシステムの安全性と信頼性に関するすべての要件に適合している必要があります。

このため、ソフトウェアで実装すべき機能についての「共通の理解」が、シームレスな統合、さらには機能自体には直接関係のない最適化（リソース消費など）のための基盤となります。特に後者は、ECU の大量生産を考えた場合、顕著な結果を生み出します。1 台の ECU について少しでもコストを減らすことができれば、製品ライン全体の莫大なコスト節減につながります。たとえば、メモリ使用量を削減することによって廉価なマイクロコントローラを使用できるようになれば、たとえ ECU1 台当たりのコストの違いはわずかでも全体のコスト削減は非常に大きなものとなります。

多くの場合、ECU ソフトウェアのファンクションをグラフィカルにモデリングすることが、上述の「共通理解」を実現するための要因となります。グラフィカルモデルは、組み込み C コードより抽象的である一方、テキスト記述に比べて解釈の余地がなく一義的であることから、より「形式的」であると言えます。グラフィカルモデルは PC 上でシミュレーション実行でき、また「ラピッドプロトタイピング」の手法により、開発工程内の早い段階から実車を用いた動作検証を行うこともできます。このようにファンクションのグラフィカルモデルは、「デジタル仕様書」としてさまざまな局面で利用できます。

ファンクションのグラフィカルモデルは、コード自動生成機能により自動車組み込みソフトウェアに変換することができます。この際には、機能性に直接関係のない製品特性（安全性、リソース消費など）を含む、各ターゲットに固有の設計情報をファンクションモデルに加味します。

ECU の動作環境は Hardware-in-the-Loop システム（HiL）によりシミュレートすることが可能であるため、ECU のテストを早期に実験室内で容易に行うことができます。ECU の HiL テストにおいては、実車テストの場合と比べて非常に柔軟なテストを実行することができます。

一般的に、自動車組み込みソフトウェアの「適合」は、開発工程の終わり近くにならないと完了させることができません。適合は多くの場合、実車においてシステム全体（つまりソフトウェアを組み込んだすべてのメカニカルシステム）を稼働させた状態で実行されるため、専用のツールやメソッドが必要となります。ソフトウェア生成時にはこのような条件についても考慮が必要です。

2.1 項では「モデルベース設計」の諸段階について詳しく説明し、ファンクションのグラフィカルモデル作成のために ASCET で採用している「抽象化メカニズム」についても説明します。2.2 項では ECU 製品開発環境における ASCET モデルの使用方法を紹介し、2.3 項で主要トピックについてまとめます。

2.1 モデルベース設計

自動車組み込み制御ソフトウェアの開発は、V モデルの形で表すことのできる複数の開発ステップからなります。最初に論理システムアーキテクチャの分析と設計、つまり、制御ファンクションの定義を行い、続いて物理的アーキテクチャ（ECU ネットワーク）を定義し、その後、ECU にソフトウェアを実装します。さらに各ソフトウェアコンポーネントの統合とテストを行ってから ECU を車両ネットワークに統合し、最後の仕上げの工程として、実装されたファンクションを実行するシステム全体を「適合」により微調整します。これは「トップダウン」式開発手法とは異なり、シミュレーションとラピッドプロトタイピングによる早期フィードバックが重要な要素となります。

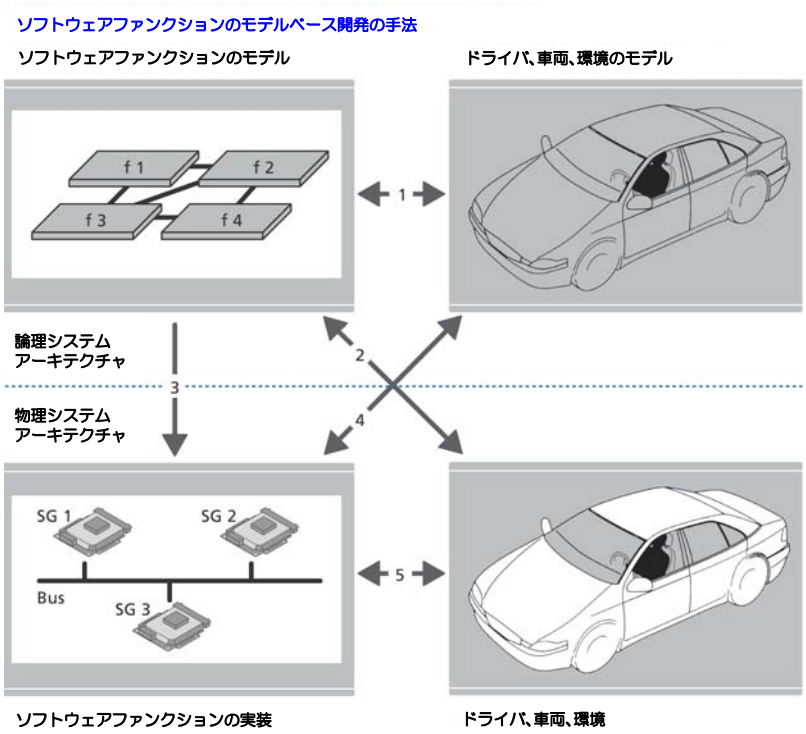


図 2-1　ソフトウェアファンクションのモデルベース開発

2.1.1　制御アルゴリズム開発

最初の工程は制御アルゴリズムの開発で、これは主に制御工学的な作業となります。まずは制御対象システム（「プラントモデル」と呼ばれます）を動的に分析することから始めます。プラントモデルは、車両（センサやアクチュエータなど）とその環境（道路状態など）、およびドライバーで構成されます。一般的には、パワートレインを含むエンジンとドライバー、あるいはシャシーと道路状態、といったさまざまなパターンに応じて、車両の各サブシステム（コンポーネント）について考察します。これらのモデルは、分析的に解決される微分方程式などを用いた「分析的モデル」と、数値的に解決される積分方程式による「シミュレーションモデル」のいずれかですが、実際には、大抵のプラントモデルは両者の混合したものとなります。

次に、一定の品質基準に基づいた「制御規則」が適用されます。制御規則はプラントのダイナミクスを補うものであり、数多くの法則に基づく優れた制御規則を利用することができます。自動車制御アルゴリズムは「閉ループ」の制御規則と「開ループ」の制御方針を一体化して実現されることが多く、後者は多くの場合オートマトンまたはスイッチング構造を採用したものです。このような制御アルゴリズムは、システム理論の観点から見ると「ハイブリッド」なシステムであると言えます。一般的に、制御規則は「制御機能」と「監視機能」を併せ持つ「セットポイント生成」ファンクションであり、それらの機能はすべてソフトウェアにより実現されます（「ソフトウェアによる制御アルゴリズムの実現」という項を参照してください）。ここでの第 1 ステップは、シミュレーションモデルにより表現される車両サブシステムを対象とする制御アルゴリズムを設計することです。ここでは制御アルゴリズムとプラントモデルはどちらもコンピュータ上で実行されます。一般にプラントは疑似連続系モデルとして設計され、制御アルゴリズムは離散系モデルとして設計されます。どちらのモデルも値の範囲は連続的なので、制御アルゴリズムとプラントが使用する状態変数とパラメータは、シミュレーションコード内では浮動小数点変数として実現されます。このモデルは 15 ページの図 2-1 の上の部分に示されています。ここに示される「論理システムアーキテクチャ」は、ドライバー、車両、および環境のモデルに結合される制御アルゴリズムを表すものです。1 の矢印は制御アルゴリズム設計の段階を表し、制御アルゴリズムのモデリングは「共有信号の使用」という観点において行われます。つまり、あるコンポーネントが信号の提供役となり、他のコンポーネントが同じ信号の要求役となることにより、信号を共有します。

ソフトウェアによる制御アルゴリズムの実現

制御アルゴリズムは「ハイブリッドシステム」、つまり、閉ループシステムと開ループシステムの混成体で、ほとんどの場合、開ループ部は有限ステートマシンのような非線形の離散システムです。制御アルゴリズムをマイクロコントローラ上で実行するには順次プログラミング言語（C 言語など）に変換する必要がありますが、これを最も容易に実現する方法は、割込みによりトリガされるメインループを作り、そのループの中で順次プログラムを内包する複数のサブルーチンを呼び出すというものです。サブルーチン間のデータ交換はグローバル変数により行います。割込みがメインループをトリガするたびに順次プログラムが繰り返し実行されるので、「タイミング割込み」を使用すれば、メインループで「サンプル値システム」を実現することができます。

しかしこのように制御アルゴリズムをソフトウェアでストレートに表現する方法は、「マルチレートシステム」（複数のサンプルレートを必要とするシステム）を実現するには限界があり、そのようなシステムは、1 つのメインループの代わりに複数のタスクを使用して実現する必要があります。このような「マルチタスキングシステム」においては、信号データがタスク間で適切に交換されなければなりませんが、状態変数、パラメータ、入力／出力信号などを C コードレベルで確実に扱うことは容易ではありません。そこで、制御アルゴリズムを ASCET で作成することにより、このような制御アルゴリズムの「制御工学的視点」と「実装的視点」のギャップを埋めることが可能となります。ASCET においては、C コードのように変数とサブルーチンを単純に使用するのではなく、以下のような「構造」を用いて制御アルゴリズムをモデリングします。

- モジュール

- クラス
- プロジェクト

これらの構造を組み合わせることにより、さまざまなターゲット上で複雑な制御アルゴリズムを構築し実行することができます。「ターゲット」とは制御アルゴリズムが実行されるシステム、つまり PC やラピッドプロトタイピングシステム、または ECU のマイクロコントローラを指します。実行する際は、まず ASCET モデルを汎用的な C コードに変換し、その C コードを各ターゲット上で実行可能なコードに変換します。その際に使用されるすべての「モデリング構造」は、データベース内で維持管理されます。

モジュール

「モジュール」は、（状態）変数、パラメータ、入力／出力信号に対する順次ステートメント（つまりそれらのアイテムに対して順に実行される命令文）を定義するためのものです。「順次ステートメント」は、ブロックダイアグラムエディタ（BDE）において、変数に割り当てられた「シーケンスコール」によって定義されます。つまり、順番が定義された各シーケンスコールにより式の結果が変数に代入されます。また複数の順次ステートメントを 1 つの「プロセス」（サブルーチンに相当します）にまとめることができます。なお ASCET では、ステートメントの記述方法として BDE の代わりに「ESDL プログラミング言語」を使用することもできます。

入力信号は「受信メッセージ」としてモデリングされ、受信メッセージの値を式に読み込んで計算することができます。出力信号は「送信メッセージ」としてモデリングされ、式の結果を送信メッセージに代入する（書き込む）ことができます。ブロックダイアグラムエディタでは、メッセージへの代入は、変数の場合と同様にシーケンスコールにより表されます。

「パラメータ」は独特な表現形式を持っています。パラメータの値を式に読み込むことはできますが、パラメータに値を代入することはできません。

以上をまとめると、モジュールは送信メッセージと受信メッセージを使用して他のモジュールとデータ交換を行います。また、モジュールは複数のプロセスを持ち、プロセスには任意の数の順次ステートメントが含まれます。モジュールにはメッセージ以外に変数やパラメータも内包されます。受信メッセージから読み込んだ値はモジュール内のプロセス間で共有できますが、送信メッセージへの書き込みには各プロセス間の排他的アクセスが必要です。1 つのモジュール内のプロセス間でしか交換されないメッセージが存在する可能性もあり、そのような専用メッセージは「送受信メッセージ」と呼ばれます。

クラス

プロセスが実行される際、内部変数（制御アルゴリズムの状態など）を処理するためにデータを保存する必要が生じる場合があります。コンピュータサイエンスの観点から、内部変数はいくつかの型に分類され、複数の型を組み合わせて「複合型」を作成することもできます。複合型のエレメントについてはステートメントを定義することができ、各処理を合わせて複数のサブファンクションまたはメソッドを形成することができます。このうち「メソッド」は引数を取ることができ、通常のデータ操作で複合型のデータエレメントにアクセスすることができま

す。このようなメソッドを伴う複合型を「クラス」と呼びます。クラスは「型」のひとつであるため、変数を定義する場合と同様に「インスタンス化」が可能です。ASCET では、変数およびクラスのインスタンスをクラスまたはモジュールの中に定義することができます。

あるクラスを別の（= 外側の）クラスのスコープ内のインスタンスとして定義することにより、インスタンス化されたクラスのメソッドを外側のクラスのメソッドから呼び出すことができます。このようにしてメソッド（たとえばフィルタリングアルゴリズムなどの計算を実現するメソッド）をインスタンス化することにより、その計算結果を呼び出し側のメソッドの計算の中で利用することができます。このメカニズムを利用すれば、制御アルゴリズムを「型定義されたオブジェクト階層」として表現することができます。トップレベルのクラス、つまり他のクラス内でインスタンス化されていない最も外側のクラスのメソッドを呼び出すと、「メソッドの出力値」という結果が得られますが、その結果を算出する際は、組み込まれているメソッドインスタンスが順次呼び出され、後続の計算に利用されます。このことから、「トップレベルメソッドの実行」はオブジェクト指向プログラムの「順次実行」に相当することがわかります。

パラメータ

「パラメータ」を情報工学的に説明すれば、「プログラムからは読み込みのみ可能で、書き込みできない特殊な変数」ですが、自動車制御工学の見地から説明すると「制御アルゴリズムを特定の車両に合わせて調整するために使用されるデータ」となります。パラメータの値は制御アルゴリズムの実行開始前に設定され、その値は制御アルゴリズムの実行中には変化しません[1]。しかしそれでもパラメータは「変数」の特殊な形であるため、変数と同様の方法で分類することができます。

クラスにはパラメータを定義することができ、その際、パラメータは複合型（クラス）に含まれる 1 つの要素として表現されます。クラスは複数回インスタンス化できるので、そこに定義されたパラメータも複数回存在することができます。しかし原則として、パラメータは読み取り専用メモリ内に存在し、起動時に専用メソッド（コンストラクタなど）により初期化されることはありません。値の初期設定は実行に先立って（設計時などに）行われ、その際に設定されたパラメータ値のセットを「データセット」と呼びます。設計時にパラメータ値を挙動クラスのインスタンスに割り当てる場合、データセットを特定のインスタンスに関連付けて設定する必要がありますが、ASCET では、クラス設計時に仮インスタンス用のデータセットを定義しておき、特定のインスタンスへのデータセットの関連付けは、各インスタンスの作成時に行います。

モジュール内でのクラスの使用

上述のように、制御アルゴリズムの順次実行はトップレベルクラスのメソッドを呼び出すことから始まります。この「メソッド呼び出し」はプロセスの実行により開始されます。一般に、メソッドの引数はプロセスの受信メッセージにより供給され、メソッドの戻り値は送信メッセージに供給されます（さらにこれらのメソッドにもモジュールの内部変数から値が供給される場合があります）。

---

1. 「アダプティブパラメータ」についてはここでは考慮しません。

リアルタイム処理の観点から見ると、トップレベルクラスのメソッドを呼び出すプロセスは、カプセル化されたインスタンスに属するメソッドの「呼び出しスタック」を形成します。「リーブインスタンス」（“leave instance”）のメソッドも、そのプロセスがマッピングされているタスクのコンテキストで実行されます。メソッドの呼び出しスタックを深くするとイベントの反応性が低下してしまうため、クラスを設計してリアルタイムコンポーネントに組み込む際には、オブジェクト指向の再利用性とタスクスケジュールの反応性のバランスを調整する必要があります。

プラントモデリングのための連続系ブロック

ASCET には連続系モデルのための専用ブロックが用意されています。これらの「連続系ブロック」（CT ブロック）には以下の 2 種類があります。

- エレメンタリブロックをグループ化した「構造ブロック」
- エレメンタリシステムのダイナミクスを定義した「基本ブロック」

基本ブロックは下に示す標準形の非線形システムを想定し、動的挙動をオブジェクト指向形式で定義したものです。

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

メソッドには、初期化メソッドと終了メソッド、f を実現するための入力／更新／デリバティブメソッド、さらに g を実現するための直接的／間接的な出力メソッドと間接出力メソッドがあります。さらに、状態イベント検知メソッドや、状態イベント発生時に実行する処理を記述するイベントメソッドもあります。それ以外にも重要なものとして、従属パラメータを解決するためのメソッドがあります。式は ESDL と C のどちらの構文でも記述できます。

閉ループシミュレーション用プロジェクト

ECU ソフトウェアは、相互通信を行う一連のモジュールとオペレーティングシステムとで構成されます。オペレーティングシステムの「コンフィギュレーション」にタスクとそのスケジュールを定義し、オペレーティングシステム自体はタスクとメッセージを実現します。「タスクスケジュール」にはプロセスのタスクへの割り当て情報が含まれます。PC で閉ループシミュレーションを実行する場合は、制御アルゴリズムのリアルタイムコンポーネントに CT ブロック（19 ページの「プラントモデリングのための連続系ブロック」という項を参照してください。）がアタッチされます。リアルタイムコンポーネントと CT ブロックのメッセージ間のバインディングは、「名前」によるものではなく明示的に、つまりポートをグラフィカルに接続することにより行われます。CT ブロックのメソッドは数値的な積分アルゴリズムから呼び出されます。積分アルゴリズムは、最終的なオペレーティングシステムコンフィギュレーション内の独立したタスクとして実行されます。

プロセスがタスクに割り当てられて適切な CT ブロックタスクの作成が終わると、OS コンフィギュレーションが実行形式のコードに変換されます。PC 上での閉ループシミュレーションの場合、適切なイベントキューと数値ソルバを備えたシミュレーション環境が生成されます。ただしこのシミュレーション環境は「リアルタイム実行環境」ではありません。

2.1.2 ラピッドプロトタイピング

一般的に、設計時に使用されるプラントモデルは、設計工程全体にわたって一環して参照できるほど詳細には定義されていないため、モデルではなく実車を使用しての制御アルゴリズムの検証は欠かせません。この検証工程において、制御アルゴリズムは初めて「リアルタイム」で実行されることになります。ソフトウェアコンポーネントのエントリ（実行開始）ポイントはオペレーティングシステムタスク内にマッピングされますが、それ以外に、ハードウェアアクセス専用のソフトウェアコンポーネントを作成して制御アルゴリズムのソフトウェアコンポーネントに接続する必要があります。この工程は、15 ページの図 2-1 では、実際の環境でドライバーが運転する自動車と論理システムアーキテクチャの結び付け（図の矢印 2）として表されていて、ここではさまざまな方法が考えられます。1 つめは、自動車とのインターフェース専用の I/O ボードを備えた専用の「ラピッドプロトタイピングシステム」を使用する方法です。ラピッドプロトタイピングシステム（「RP システム」とも呼ばれます）は高性能のプロセッサボードと I/O ボードとで構成され、各ボード間は内部バスシステム（VME など）により接続されています。一般に、これらのプロセッサボードは製品用 ECU とは異なり、浮動小数点演算ユニットを備え、ROM や RAM の容量も大きく、高性能なものです。バス接続された I/O ボードを介してセンサやアクチュエータとインターフェースを取るので、さまざまなユースケースに柔軟に対応できます。この工程においては、「ECU の生産コスト」よりも「制御アルゴリズムのラピッドプロトタイピング」に重点が置かれます。

上記のようなラピッドプロトタイピングシステムにおいては、インターフェース要件は、多くの場合、ボード上の専用回路で実現されます。しかしこれには柔軟性に限界があるため、代わりに従来の ECU とそのマイクロコントローラペリフェラル、およびその他の ECU エレクトロニクスを使用してセンサ／アクチュエータ

とのインターフェースを実現するという方法も考えられます。それにより、I/O ハードウェア抽象化レイヤのソフトウェアコンポーネントを後のシリーズ生産で再利用できるという副次的効果が得られます。図 2-2 は制御／監視ファンクションがバイパスシステム上で稼働し、そのバイパスシステムがセンサとアクチュエータを介して自動車に接続されているようすを示しています。

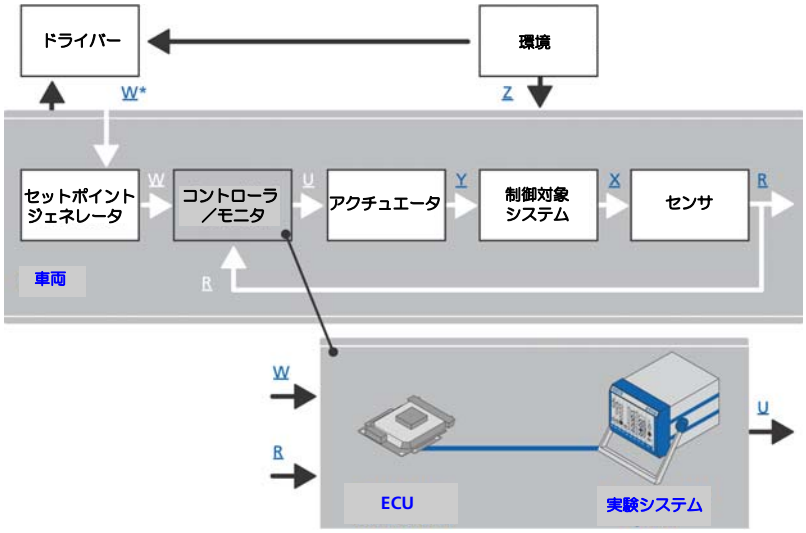


図 2-2　典型的なラピッドプロトタイピングシステム

図 2-2 に示すようなバイパス構成におけるラピッドプロトタイピングでは、ECU のマイクロコントローラのペリフェラルを使用してセンサとアクチュエータを駆動します。つまり、I/O ドライバは製品用 ECU 上で稼働し、制御アルゴリズムはラピッドプロトタイピングハードウェア上で実行されます。3 つの信号 W、R、U はデジタル値で、それぞれセットポイント、プラントの反応をサンプリングした値、デジタルアクチュエータ信号を表します。アクチュエータ信号は、ドライバーの希望 W* に合った状態で車両を動作させるための電気的または機械的信号 Y に変換されます。W はサンプリングされたデジタル信号です。機械的または電気的信号 X で表される車両の状態がサンプリングされ、デジタル信号 R として制御アルゴリズムに供給されます。さらに、道路条件などのノイズ信号 Z が存在し、これは測定される入力信号とは異なり、制御アルゴリズム内で直接考慮されるものではありませんが、車両の挙動に影響を与えます。RP システムは、プロセッサボードと専用の通信ボードのみを使用し、その他の車両信号を直接使用することはありません。センサ値 R、セットポイント値 W、アクチュエータ値 U はハイスピードリンクで送信されますが、一般的な ECU ハードウェアにはハイスピード通信リンクに対応する専用回路が装備されています。

ソフトウェア開発の観点から見ると、製品 ECU 上で稼働する既存のソフトウェアと、新たに開発された制御アルゴリズムにおける「構造化されたインターフェース」により、ラピッドプロトタイピングの効率は大幅に向上します。

リアルタイム I/O モジュール

「ラピッドプロトタイピング実験」には、高性能マイクロプロセッサに加えて通信と I/O のための専用のハードウェアが必要です。ETAS の ES1000 ファミリの場合、さまざまな機能を実現するための VME ボードを使用でき、システム内通信は VME バスにより行われます。

このようなラピッドプロトタイピングシステムは、構成を柔軟に変更できる「組み込みシステム」であるといえます。通信と I/O ハードウェアの機能を実現するにはハードウェアと制御アルゴリズムを結びつける「基本ソフトウェアモジュール」が必要ですが、ASCET においては、これらすべての基本ソフトモジュールは「リアルタイム I/O」と呼ばれる 1 つの ASCET モジュールで表現されます。たとえば、CAN バッファから信号を読み込んでそれを送信メッセージとして提供するプロセスがある場合、このプロセスはオペレーティングシステムタスクの中で「スケジュール」されますが、その際に使用する信号名と CAN フレーム ID を事前に専用エディタで設定しておくことができます。

制御アルゴリズムを ETAS ラピッドプロトタイピングシステム上でテストする際は、コンフィギュレーションパラメータに基づいてこの「リアルタイム I/O モジュール」を生成する必要があります。このモジュールは汎用的な ASCET の C モジュールとして表現されるののので、これが他のリアルタイムコンポーネント、つまり、一般的な ASCET モジュールにアタッチされることにより、実行可能なラピッドプロトタイピング制御アルゴリズムができあがります。

ラピッドプロトタイピング用のプロジェクト

ラピッドプロトタイピング用のプロジェクトには連続系ブロックで表現されるプラントモデルは内包されません。その代わりにリアルタイム I/O モジュールが内包され、ラピッドプロトタイピングプロジェクト用に設定されます。このモジュールは、モデルレベルにおいてはメッセージにより制御アルゴリズムモジュールと通信を行います。そのため、リアルタイム I/O モジュールのコンフィギュレーションに応じた複数のプロセスがオペレーティングシステムタスクにフックされます。

2.1.3 制御アルゴリズムの実装と ECU への統合

ラピッドプロトタイピングの工程を経た制御アルゴリズムは、実車 に対しても有効なものとなります。ただしラピッドプロトタイピングシステム用に生成されたコードは、RAM リソースや浮動小数点ユニットなど、専用プロセッサボードの強力な機能を使用することを前提としています。それをメモリ容量や演算能力が限られた条件下で実行できる制御アルゴリズムにするためには、制御アルゴリズムのモデルの再設計が必要となります。たとえば、計算式を浮動小数点から固定小数点の制御アルゴリズムに変換し、効率、スケーラビリティ、モジュール性といったさまざまな制限事項に対処する必要があります。このようにして再設計されたデザインを用いて、適切な製品用コードが自動生成されます。

浮動小数点から固定小数点への変換

自動車などの「物理プラントモデル」においては、車速と加速度、冷却液温度、ヨーレート、バッテリ電圧などの「物理量」が扱われますが、シミュレーションモデルにおいてこれらの物理量は 64 ビットまたは 32 ビットの float 型変数で表現されます。シミュレーションモデルは閉ループ制御システムであるため、車両モデルと制御アルゴリズムモデルはどちらも浮動小数点で表現される必要があります。しかし、実際には浮動小数点ユニットは高価であるため、一般の車載マイクロコントローラには使用されません。つまり、制御アルゴリズムを車載マイクロコントローラに実装するためには、浮動小数点から固定小数点への変換が必要となります。

例： 冷却液温度の範囲を -50 ℃～ 150 ℃とします。これらの値を直接 16 ビット整数に実装するのでは極めて非効率的です。これでは図 2-3(a) に示すように 16 ビットで表現できる範囲の 0.3%しか使用されず、温度の分解能は 1 ビット =1 ℃となってしまい、測定された温度が 83.4 ℃であっても制御ソフトウェア内では 80 ℃として表現されてしまいます。

これを、図 2-3(b) に示すように測定値に 217.78 を掛けることにより、分解能を 1 ビット当たり約 0.0046 ℃に変えることができます。しかしこの変換は単なる「浮動小数点の乗算」であるため、適切なものとはいえません。

別の方法として、分解能を 1 ビット当たり 0.0078125 ℃に設定する方法があります。この乗算は 7 ビットの左シフト演算により表現できます。この演算を温度範囲に当てはめると、-6400 から 19200 までのビットパターンが得られるので、16 ビット整数型の値の範囲の 39%が使用されることになります。このスケーリングを図 2-3(c) に示します。

またさらに、符号なし 16 ビット整数値を使用し、1 ビット当たり 0.00390625 ℃という分解能に加えて「オフセット」を併用することにより、使用率をさらに高めることができます。オフセットを -12800 に設定した場合、温度範囲は -12800 ～ 38400 となり、51200 個の値の範囲を使用できることになり、使用率は図 2-3(d) に示すように 78%を超えます。ただしここではオフセット計算のための減算処理が必要になります。

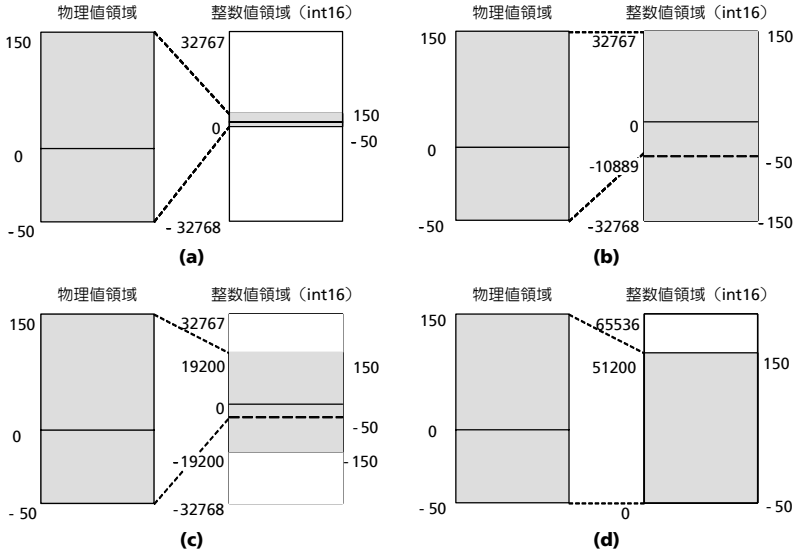


図 2-3　スケーリングなしのマッピング (a)、任意マッピング (b)、2^7 スケーリング (c)、2^8 スケーリングとオフセット (d)

この関係は下の線形関係で表現できます。

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

または、以下のように一般的な形で表現することもできます。

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

上の式の `scal` がスケーリングファクタで、x がオフセットです。分解能はスケーリングファクタの逆数になるので、物理値は以下の式の実装値で表されます。

```
phys_value = impl_value / scal - ofs
```

固定小数点数の演算

変数に実装変換式（“implmentation formula”）を定義すると、その変数を扱うメソッドやプロセスのステートメント、つまり式や代入文に影響を与えます。以下のような、物理値を表す 2 つの変数を用いた単純な代入文でさえも、実装情報（“implementation"）、つまり割り当てられた実装変換式が異ると、単純な演算ではなくなってしまいます。

```
a = b
```

a と b を符号なし 8 ビット変数（値の範囲 : 0 ～ 255）として、それぞれに以下の実装変換式を適用してみましょう。

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

すると、a の物理値の分解能は 2 になり、b の物理値の分解能は 3 になります。この実装情報に基づいて代入 a=b を表すと、以下のようになります。

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

これを以下のように単純に移項してみます。

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

このように、元のステートメント a=b から、`a_impl = (3/2) * b_impl` というステートメントが得られました。実装変換式[1] は、定数を用いた単純な算術演算です。しかし、最大精度を実現するための一連の演算においては注意が必要です。もし上の式のとおりに最初に除算を行ってしまうと、除算は整数で行われるため、結果に約 50%の誤差が生じ、変換式が効果的でなくなってしまいます。この変換には、以下のようなステートメントの方が適しています。

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

物理変数を扱うステートメントにおいて、このように調整された実装変換式を使用する場合でも、一般的に、さらにいくつかの要素を考慮する必要があります。

まずは「オーバーフロー」の問題です。上の式において最初に 3 をかけると、`b_impl` が 255/3=85 より大きい場合はオーバーフローが発生します。同様に、アンダーフローと丸め誤差についても注意が必要です。先に 2 で割ると、右シフト演算が行われ、最小桁のビットが失われてしまいます。これでは `b_impl` の値が 1 または 0 である場合には値の区別がなくなり、どちらの場合も `a_impl`（つまり a）には 0 が代入されてしまいます。また事実上、a=b という代入は物理範囲が同じ（ここでは 0 ～ 510）である場合に限り意味をなすことから、`b_impl` には 510/3=170 という最大値が想定されますが、これではオーバーフローが発生する可能性があるため、それを回避しなければなりません。この対策として、コード生成時にケースごとに処理を選択するという方法が考えられます。つまり、`b_impl` が 170 以下の場合は先に乗算を行い、`b_impl` が 170 より大きい場合は先に除算を行うようにするのです。しかし、そのためには余分なコードが必要になってしまいます。そのためここでは、値の範囲全体において最大精度 1.5 の許容誤差を受け入れる、ということが必要になってしまいます。一般的に、複雑な結合や式を用いる場合は言うまでもなく、オペランドの少ない通常の算術演算の場合でもこれ以上に困難な状況となる可能性は十分にあります。

C コードのクラスとモジュール

レガシーコードの再利用やマイクロコントローラのペリフェラルアクセスを行うような場合、内部挙動が C コードで記述されたメソッド（またはプロセス）を含むクラス（またはモジュール）を定義する必要が生じる場合があります。C コードクラスと C コードモジュールのコードには実装情報が含まれ、このコードをターゲット用実行プログラムに順に組み込んでいきます。つまり C コードクラスと C コードモジュールはターゲットに依存しているので、プロジェクトのターゲットを変更する場合は、変更後のターゲットに合った C コードを作成しなおす必要があります。

1. 少なくとも ASCET がサポートする変換式の場合

組み込みマイクロコントローラ用プロジェクト

上述のように、C コードクラスと C コードモジュールはマイクロコントローラのペリフェラルにアクセスすることができます。ASCET プロジェクトエディタではオペレーティングシステムの設定と生成を行えるため、制御アルゴリズムを記述したモジュールを含む組み込みマイクロコントローラ用プロジェクトを「統合プラットフォーム」として使用することができます。この際、コードジェネレータは、OS スケジュールやモジュール間メッセージ通信を調べてタスクやメッセージを生成し、さらにプロセスからメッセージへのアクセスコード[1] を生成します。こうしてでき上がったプロジェクトとそのすべての内包モジュールの C コードは、マイクロコントローラに書き込むための .hex ファイルに変換され、その際、すべての測定変数と適合変数（パラメータ）が定義された ASAP2 ファイルも生成されます。

ただし製品 ECU 用の生成時には専用のビルド環境と基本ソフトウェアモジュールが使用される場合も多く、そのような場合は、アプリケーションソフトウェア、つまり制御アルゴリズムのモデリングのみを ASCET[2] で行います。この場合は、プロセスのアクセスコードを含むメッセージとタスクボディ、つまり OS エディタで定義されたプロセスのシーケンスが生成されるので、このタスクボディを外部の OS コンフィギュレーションエディタにコピーします。

2.1.4 各種プロジェクトにおける制御アルゴリズムの再利用

ASCET のモデリングエレメントはすべてデータベース内で管理されますが、同じ制御アルゴリズムを使用するプロジェクトでも、ターゲットが異なればモジュールの数や種類が異なります。

- 閉ループシミュレーション用プロジェクト：制御アルゴリズム用のモジュールと CT ブロックを参照します。
- ラピッドプロトタイピング用プロジェクト：制御アルゴリズム用のモジュール（閉ループシミュレーションの場合と同じモジュール）とリアルタイム I/O モジュールを参照します。リアルタイム I/O モジュール用のコンフィギュレーションデータはプロジェクトに保存されます。
- 組み込みマイクロコントローラ用プロジェクト：制御アルゴリズム用のモジュールとペリフェラルアクセス用のモジュール（C コード）を参照します。固定小数点コードが必要な場合、モジュール、クラス、プロジェクトに実装変換式を定義する必要があります。またコードを生成する前に、プロジェクトに適した実装情報を選択する必要があります。

ASCET プロジェクトは PC、ラピッドプロトタイピングシステム、プロダクション ECU[3] などさまざまなターゲット上で実行できますが、PC またはラピッドプロトタイピングシステム上で実行される際は、ASCET の統合実験環境（EE）を使用して実験を行います。また ECU を使用した実験の場合は、実車で行う制御アルゴリズムの適合作業（ファインチューニング）[4] と同様、測定／適合システム INCA[5] を用いて行います。

---

1. 一般的にはマクロで実現されます。
2. このユースケースは「additional programmer」と呼ばれます。
3. または評価ボード

ソフトウェアの面から見ると、実験には以下の 4 種類のタイプがあります。

1. 物理実験
2. 量子化実験
3. 実装実験
4. オブジェクトベースのコントローラ実装実験

「物理実験」には実装情報は必要ありません。「量子化実験」には量子化が必要で、「実装実験」と「オブジェクトベースの実装実験」にはさらに整数の基本型と限界値についての情報が必要です。ASCET の制御アルゴリズムモデルを構成するステートメントから生成されるコードは、ターゲットと実験のタイプに応じて異なります。「物理実験」においては、物理ステートメントが real64 の変数で表現されるので、量子化の影響はありません。「量子化実験」でも real64 の変数が基礎に使用されますが、物理ステートメントに量子化の影響が付加されます。「実装実験」では実装情報が使用され、整数型がベースとなります。つまり実装実験用に生成されるコードの変数には、実装情報として指定されている基本型が使用され、物理ステートメントは実装変換式に基づく実装ステートメントに変換されます。

「オブジェクトベースのコントローラ実装実験」では、実装実験の型と実装ステートメントが使用されますが、モジュールとクラスの構造は別の形で表現されます。ASCET 内の各変数には基本型、限界値、実装変換式だけでなく、マイクロコントローラのメモリレイアウトが反映された「メモリクラス」も定義できます。しかし前述のように、オブジェクトベースのコントローラ実装実験は製品 ECU 用でしか行えないうえ、オンライン実験は INCA などの測定／適合ツールでしか行えません。

PC またはラピッドプロトタイピングターゲットを使用し、かつ基本型、限界値、オフセット、量子化に関するすべての実装情報がすべてのエレメントに定義されている場合は、実験のタイプを切り替えるだけで、物理環境における実装変換式や整数基本型の作用を観察することができます。

2.1.5 実験室における技術システムアーキテクチャのテスト

実装／統合工程の結果として、物理的システムアーキテクチャ、つまり「ECU ネットワーク」ができあがります。これらの ECU のテストはプラントモデルを使用してリアルタイムに行われます。プラントモデルは、センサ／アクチュエータのモデルや専用ボードを使用することにより、本来は ECU の電装部品から取得される電気信号をシミュレートできます。このようなシステムは Hardware-in-the-Loop システム（HiL）と呼ばれ、プロセッサボードと I/O ボードとで構成されます。プラントモデルには典型的な運転操作をシミュレートするさまざまな値が初期設定されていて、HiL 上で運転操作がシミュレートされることによって ECU に

---

4. 実験用の ECU リソースには限りがあるので、特別な対応が必要になります。これについては本書では説明しません。
5. ASCET プロジェクトが CT ブロックだけを含み、そのプロジェクトが PC またはラピッドプロトタイピングシステム上で実行される場合、LABCAR OPERATOR の EE を使用します。

対するセンサデータが出力され、さらに ECU からアクチュエータデータが入力されます。このようにして ECU が正しく統合されたかどうかを調べることができます。HiL テストは 15 ページの図 2-1 の矢印 4 で示されている部分です。

2.1.6 実車における技術システムアーキテクチャのテストと仕上げ

前述のように、一般的なプラントモデルは車両のダイナミクスを完全に表せるほど詳細化されていません。今日では適合作業の多くを HiL システムで行えますが、車両の制御アルゴリズムの最終仕上げは実車の製品 ECU に組み込まれた製品ソフトウェアで行う必要があります。このためには、物理システムアーキテクチャを実車に組み込み、性能試験場でテストを行う必要があります。この最終工程においては制御アルゴリズムのパラメータ値の微調整のみを行います。

2.2 生産環境における ASCET の利用

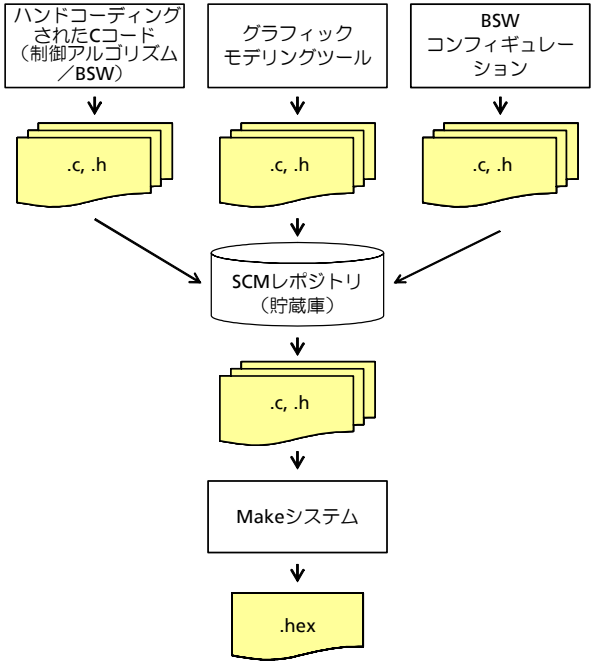


図 2-4 高度なソフトウェア生産環境

典型的な「ハンドコーディング環境」においては、複数のソフトウェア開発者によって、制御アルゴリズム用、およびオペレーティングシステムを含む基本ソフトウェアモジュール用の C コードが作成されます。さらに、「ECU インテグレータ」の役割を持つ担当者がすべてのソースコードファイルを集め、コンパイラとリンカを起動する「Make ツールチェーン」を実行します。C コードはソフト

ウェア開発者間でファイルシステムやソースコード管理（SCM）システム[1] を介して受け渡されます。「SCM」はさまざまなバージョンのソースコードファイルを保持しながらコンフィギュレーションの作成と維持管理を行うことのできるデータベースです。これは ECU ソフトウェアの生成／統合のベースラインとして使用されます。C コードファイルの 2 つのバージョン間の相違を調べる際は、「ディファレンスブラウザ」を用いてプログラムテキスト内の相違点を強調表示させることができます。この 10 年程の間に SCM システムとディファレンスブラウザは広く使用されるようになり、それによって自動車組み込みソフトウェアの品質が大幅に向上しました。

高度なソフトウェア生産環境においては、一部の制御アルゴリズム用 C ファイルは制御アルゴリズムモデル（実装情報が設定された ASCET モデルなど）から生成されますが、多くの基本ソフトウェアモジュール用 C ファイル（OS や COM スタックなど）はいわゆる「コンフィギュレータ」により生成されます。このような高度な生産環境の例（ASAM-MCD-2MC ファイルの生成工程を除く）を 28 ページの図 2-4 に示します。ここには C コード生成環境と、SCM データベースおよび Make システムが示されています。このような高度な生産環境をさらに詳しく検討し、ASCET による制御アルゴリズム用 C コードのモデルベース生成に注目すると、モデルは、ソースコードの基礎となるものでありながら、一方では制御アルゴリズム開発の過程でラピッドプロトタイピングの結果を反映して進化していくものであることがわかります。したがって、モデルも SCM データベースで維持管理することが必要となります。

ASCET コンポーネントはローカルデータベースに保存され、1 つのローカルデータベースにはモデルの 1 つのバージョンだけが保存されますが、「ASCET-SCM インターフェース」はローカルデータベースから SCM レポジトリ（= 貯蔵庫）へのリンクを確立し、バージョン間のモデルの交換を可能にします。このモデル交換は 30 ページの図 2-5 の (a) に示されています。ソースコード開発ではバージョン間のディファレンスブラウジングが不可欠ですが、モデルベース開発においてもそれは同じです。ASCET-SCM インターフェースを「Model-Diff-Browser」で強化すれば、たとえばモジュールのブロックダイアグラムエディタ内に追加されたメッセージなどが強調表示されるようにすることができます。

2.2.1 モデル変換

組み込みリアルタイムソフトウェアの開発は制御エンジニアとコンピュータエンジニアの両者により進められます。開発工程においては、制御ソフトウェアの開発を「挙動ドリブン」の観点から始める場合と、「構造ドリブン」の観点から始める場合があり、さらには両方の観点から始める場合もあります。ASCET（および AUTOSAR）の垂直アプローチにはこの両方のアプローチが統合されていますが、純粋に挙動的または構造的なアプローチによるモデルを使用する必要がある場合も考えられます。

「挙動的」な方法をとる場合、閉ループ制御アルゴリズムのモデリングには MATLAB®/Simulink® を使用するのが一般的です。それは、少なくとも PC シミュレーションを行う際に問題となる多くの構造体関連の詳細事項に煩わされること

1. 一般的な SCM システムとして、CVS や SubVersion があります。

がないためです。そして PC シミュレーションを実行した後は、制御アルゴリズム部はブロックダイアグラムの形式で ASCET のモジュールまたはクラスに引き継がれ、プラント部は ASCET CT ブロックとして表現されます。

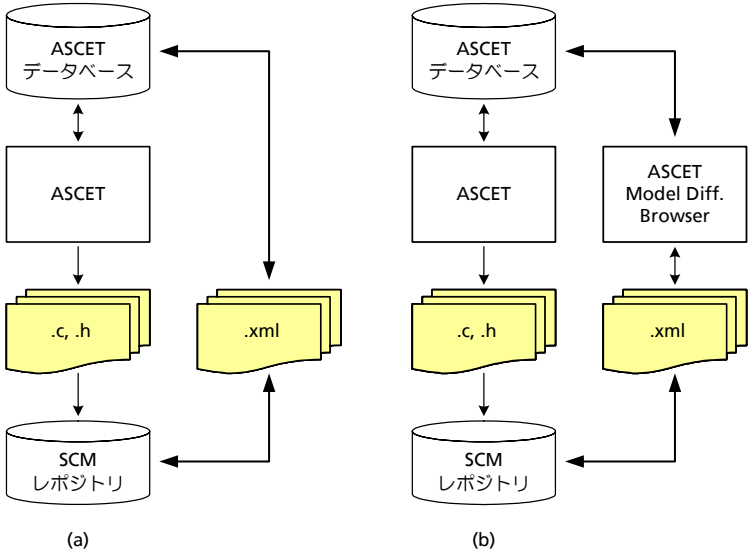


図 2-5 ASCET-SCM インターフェース。ディファレンスブラウジング機能がある場合（b）とない場合（a）

「構造的」な方法をとる場合は、UML の 1.4 または 2.0 が一般的に広く使用されています。非常にシンプルなクラスとオブジェクトのダイアグラムに加えて、インタラクションおよびコラボレーションのダイアグラムを使用して車両アクセスシステムや診断などの複雑なインタラクションを表記できます。ただしクラス、オブジェクト、およびそれらのインタラクションについては、ASCET モデルの中で改良されています。ASCET では、シーケンスコールによって、UML に相当するもの（つまりクラスとオブジェクト）にオブジェクトインタラクションが反映されます。これらのオブジェクトを ASCET リアルタイムオブジェクトの中に組み込めば、UML モデルにリアルタイム的側面が加わるだけでなく、モデルをそのまま実行することが可能になります。

ETAS のパートナーである Aquintos 社のモデルツーモデルコンバータ（M2M）を使用すると、このような挙動的モデルや構造的モデルを ASCET 用に変換することができます。このコンバータは UML 1.4 製品および MATLAB/Simulink に対応しています。

2.3 まとめ

ASCET は自動車組み込みソフトウェア開発のさまざまな段階において、制御アルゴリズムのモデルベース設計および実装をサポートします。抽象化の採用により、物理的な制御アルゴリズムモデルを後続の開発工程全体におけるすべての実装情報の基盤として使用でき、ターゲットを変更する際にブロックの交換などを行う必要がありません。さらにディファレンスブラウジング機能付きの SCM インターフェースを使用すれば、ECU 生産開発環境への ASCET の統合をさらにシームレスに実現できます。

3 ASCET と AUTOSAR

本章では ASCET の AUTOSAR 対応について説明します。3.1 項では AUTOSAR の趣旨を概説し、続いて 3.2 項では AUTOSAR のランタイム環境（RTE）について説明し、3.3 項で ASCET がサポートする AUTOSAR エレメントを紹介します。

本章は AUTOSAR 対応の概論についてのみ説明するもので、詳細については「付録 C：AUTOSAR」をご参照ください。AUTOSAR から発行されているドキュメントは、AUTOSAR のホームページ（http://www.autosar.org/）からダウンロードできます。

3.1 概要

一般的に、自動車組み込みソフトウェアの開発工程において、複数のベンダーから提供されるソフトウェアコンポーネントを複数のネットワークや ECU（電子制御ユニット）、各種ソフトウェアアーキテクチャで構成される車両プロジェクトに統合するには、非常な手間がかかります。また、ソフトウェアの「再利用性」に制限があれば、高機能を持った検証済みの高品質なソフトウェア を提供するためには多大な工数が必要となります。

このような状況の中で、車両メーカーとベンダーとのパートナーシップにより組織された「AUTOSAR」では、主に基本的なシステムファンクションとファンクションインターフェースを標準化することにより、車両エレクトロニクス用ソフトウェアの共同開発の簡素化、費用低減、工期短縮、品質向上などを図り、さらには安全関連システムの設計に必要なメカニズムの提供も目指しています。

これらの目標を達成するため、AUTOSAR では自動車組み込みソフトウェア用のアーキテクチャを定義しています。このアーキテクチャには、各アプリケーションの機能を実装する**「ソフトウェアコンポーネント」**（**SWC**：AUTOSAR **S**oft**w**are **C**omponent）が含まれていますが、この SWC は ECU に依存しないものであるため、再利用、交換、スケールの変更、統合が容易に行えます。

このように抽象化された SWC 環境は、**「仮想ファンクションバス」**（**VFB**：**V**irtual **F**unction **B**us）と呼ばれています。実際の AUTOSAR 対応の ECU 内においてこの VFB は、各 ECU に依存するプラットフォームソフトウェアにマッピングされます。AUTOSAR「プラットフォームソフトウェア」の機能は、「ランタイム環境」（**RTE**：**R**un **T**ime **E**nvironment）と**基本ソフトウェア**（**BSW**：**B**asic **S**oft**w**are）とに分けられます。BSW に含まれるものは、通信や I/O の機能のほか、各種ソフトウェアコンポーネントに必要な諸機能（診断やエラー報告、不揮発メモリ管理など）です。

3.1.1 RTA-RTE と RTA-OS

「ランタイム環境」は、ソフトウェアコンポーネント、BSW モジュール、オペレーティングシステム（OS）の間のインターフェースとなるものです。SWC の相互接続においては、RTE は通信の「スイッチボード」のような役割を果たします。この役割は、コンポーネントが単体の ECU に常駐する場合でも、車載バスによりネットワーク化された複数の ECU に常駐する場合でも同じです。

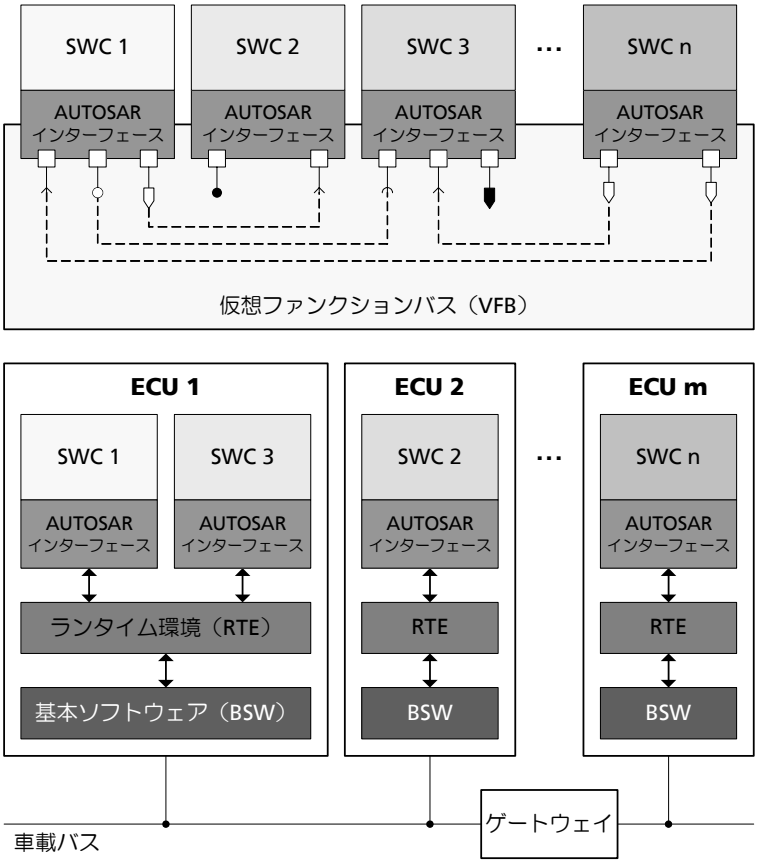


図 3-1 AUTOSAR ソフトウェアコンポーネント（SWC）の通信は、「ランタイム環境（RTE）および基本ソフトウェア（BSW）」によって実現される「仮想ファンクションバス（VFB）」経由で行われます。

AUTOSAR では、OS は RTE を通じて SWC のランナブルエンティティを呼び出します。アプリケーションソフトウェアの実行制御に関しては、基本ソフトウェアの中でも RTE と OS がキーモジュールとなります。

ETAS は十余年にわたり、「ERCOS^{EK}」、および現在の「RTA-OSEK」という 2 つの車両用オペレーティングシステムを自動車業界に供給してきました。AUTOSAR ランタイム環境「RTA-RTE」、および AUTOSAR オペレーティングシステム「RTA-OS」（2008 年第 3 四半期発売予定）は、AUTOSAR の主要ソフトウェアモジュールをサポートし、RTA の機能範囲を拡張します。現行の RTA-OSEK は、AUTOSAR OS Scalability Class 1 をサポートしています。

AUTOSAR インターフェースに準拠する RTA-RTE および RTA-OSEK は、サードパーティベンダーの基本ソフトウェアモジュールをシームレスに統合することが可能です。

3.1.2 AUTOSAR ソフトウェアコンポーネントの作成

システムアーキテクチャや AUTOSAR インターフェースのディスクリプションを作成するための AUTOSAR オーサリングツールに加え、ETAS のモデリングツール「ASCET」を活用することにより、AUTOSAR に適合する制御機能の挙動を記述し、実装することが可能となります。

ASCET モデルは、容易に AUTOSAR に適応させることができます。これは、AUTOSAR コンセプトの多くの部分を ASCET のインターフェース記述と同じような形式で定義することができるためです。実際に行う作業としては、各アプリケーションのインターフェースを AUTOSAR に適合するように修正するだけです。従来のモデルを実際に適合させた事例からわかるように、現行バージョンの ASCET バージョンでこの作業を行っても、それほど多くの所要時間は要しません。

なお ASCET V6.0 においては、AUTOSAR V2.1 に準拠する AUTOSAR SWC の記述、および AUTOSAR 準拠 SWC の量産コード生成がサポートされます。

3.2 ランタイム環境とは

VFB（仮想ファンクションバス）は、ソフトウェアコンポーネントの再利用化のための抽象概念です。**RTE（ランタイム環境）**は、この VFB の機能を実行するために必要なメカニズムを提供します。最もシンプルな RTE は、VFB を実装するだけでできあがりますが、実際には、各種インターフェースやインフラストラクチャを用意して、ソフトウェアコンポーネントを以下のように扱えるようにする必要があります。

1. ECU（VFB モデル）に依存せずに SWC を RTE に実装できること
2. アプリケーションソフトウェア自体を修正することなく、ECU および大規模な車両ネットワークに SWC を統合できること（Systems Integration モデル）。

具体的には、RTE には以下のような機能が必要となります。

- ソフトウェアコンポーネント用通信インフラストラクチャ

  ここでいう「通信」には、同一 ECU 上のソフトウェアコンポーネント間の通信（「ECU 内通信」）と、異なる ECU 上のソフトウェアコンポーネント間の通信（「ECU 間通信」）との両方が含まれます。

- ソフトウェアコンポーネントのリアルタイムスケジューリング

  AUTOSAR におけるリアルタイムスケジューリングとは、SWC のランナブルエンティティを、設計時に定義された時間的条件に基き、オペレーティングシステムが提供するタスクにマッピングすることを指します。

BSW（基本ソフトウェア）は、RTE により実装される「抽象概念」の下層に位置するため、アプリケーションソフトウェアコンポーネントから直接アクセスすることはできません。つまり、SWC がオペレーティングシステムや通信サービスなどに直接アクセスすることはできません。そのため RTE は、それらのサービスをカバーする「抽象概念」を提供する必要があります。この抽象概念は、ソフトウェアコンポーネントがどこに配置されていても常に不変でなければなりません。この条件に基づき、ソフトウェアコンポーネント間のすべての通信は、標準化された RTE インターフェースの呼出しにより行われることになります。

さらに RTE は、1 つまたは複数の ECU に配置された SWC で構成される既存の「アーキテクチャ」を実際に利用する際にも使用されます。RTE を効果的に実装するために、アーキテクチャに必要な RTE のコードはビルド時において各 ECU ごとに決定されます。標準化された RTE インターフェースの実装は、RTE 生成ツールにより自動的に行われますが、その際、指定されたコンポーネントインタラクションとコンポーネントアロケーションを実現するための適切なインターフェースが生成されます。

同じ ECU 上に配置された 2 つのソフトウェアコンポーネント間の通信には「ECU 内通信」を使用できますが、どちらか一方を別の ECU に移した場合、両者間の通信には「車両ネットワーク」が必要となります。

このようにして生成される RTE により、各 ECU 用の基本ソフトウェアの差異は、以下のようにアプリケーションソフトウェアコンポーネントから隠蔽されることになります。

- 一貫したインターフェースをソフトウェアコンポーネントに提供することにより、ソフトウェアコンポーネントの再利用が可能となります。つまり、一度設計して作成されたソフトウェアコンポーネントを複数の ECU に再利用できます。
- このインターフェースは、VFB 内に「抽象概念」としてに実装された AUTOSAR 基本ソフトウェアにバインドされます。

3.3 ASCET がサポートする AUTOSAR エレメント

ASCET は以下の AUTOSAR エレメントをサポートしています。

3.3.1 AUTOSAR ソフトウェアコンポーネント

AUTOSAR ソフトウェアコンポーネント（SWC: AUTOSAR **S**oft**w**are **C**omponent）はアプリケーションレベルの汎用コンポーネントで、CPU 内、および車両ネットワーク内の配置に依存しません。システム設計者が定義した制約条件に従い、システム設定時において任意の ECU 上に配置することができます。

ただし SWC は AUTOSAR システムを分散化する際の最小単位であるため、1 つのコンポーネントは必ずその全体を 1 つの ECU 上に配置する必要があります。

SWC を作成する際は、そのコンポーネントタイプ（「SWC タイプ」）を定義します。この「SWC タイプ」は、SWC の固定的特性（ポート名や、ポートがインターフェースによりどのように使用されるか、SWC の挙動、などの情報）を定義するものです。SWC タイプには、システム内で一意の名前を付ける必要があります。これらの点をふまえ、各 SWC は以下のもので構成されます。

- 形式に従って定義された SWC ディスクリプション ― コンポーネントのインフラストラクチャの設定方法を定義するもの
- SWC の実装コード ― C コードによる SWC の機能記述

SWC は、システム設定時において「インスタンス化」することにより、使用可能となります。ここでの「タイプ」と「インスタンス」は、従来のプログラミング言語における「型」と「変数」の関係と似ています。つまり、従来のアプリケーション内で一意の型名（つまり SWC タイプ）を定義し、その型を用いて、一意の名前を持つ変数（1 つまたは複数の SWC インスタンス）を宣言するのと同じ要領です。

3.3.2 ポートとインターフェース

VFB モデルにおいて SWC は、インターフェースによってアクセスされる「ポート」を通じて他のコンポーネントとの通信を行います。**「インターフェース」**は、通信によって伝達される情報や通信のセマンティックスを制御します。SWC はポートを通じてインターフェースにアクセスし、この「ポート」と「ポートインターフェース」を組み合わせたものを**「AUTOSAR インターフェース」**と呼びます。

ポートには以下の 2 つのクラスがあります。

- **提供ポート（P ポート：Provided port）** ― SWC はこのタイプのポートを使用してデータやサービスを他の SWC に提供します。P ポートには「センダポート」と「サーバーポート」とがあります。
- **要求ポート（R ポート：Required port）** ― SWC はこのタイプのポートを使用して他の SWC にデータやサービスを要求します。R ポートには「レシーバポート」と「クライアントポート」とがあります。

> **注記**
>
> 本書では、非 AUTOSAR ポートとの混同を避けるため、AUTOSAR ポートを「R ポート」および「P ポート」と呼びます。

ASCET V6.0 は以下のタイプのインターフェースのみサポートしています。

- センダ - レシーバ（シグナル渡し）

SWC の各 P ポートおよび R ポートには、その SWC が提供または要求するインターフェースタイプを定義します。

1 つのシステムが複数の SWC インスタンスにより構築されている場合、それらのインスタンスの R ポートと P ポートを接続します。この際、必ず 1 つのセンダを 1 つにレシーバに接続する必要があります。

センダ- レシーバ通信

センダ - レシーバ通信においては、最小単位のデータエレメントで構成されるシグナルを 1 つの SWC が送信し、1 つまたは複数の SWC がそのシグナルを受信します。

1 つの SWC タイプに複数の「センダ - レシーバポート」を持たせることができます。

各センダ - レシーバポートには、個別に送受信できる複数のデータエレメントを定義することができます。インターフェース内のデータエレメントとしては、単純データ型（integer や float など）や複合データ型（array や matrix など）を使用できます。

センダ - レシーバ通信は単方向通信なので、レシーバからの応答が必要な場合は、その応答を別のセンダ - レシーバ通信としてモデリングする必要があります。

センダ - レシーバインターフェースを要求する SWC の R ポートはインターフェースのデータエレメントを読み取ることができます。一方、センダ - レシーバインターフェースを提供する P ポートはデータエレメントを書き込むことができます。

センダ - レシーバ通信においては“1:n”（1 つのセンダ、複数のレシーバ）または“n:1”（n 個のセンダ、1 つのレシーバ）の形が可能です。後者の場合、センダ側での同期は行われません。

3.3.3 ランナブルエンティティとタスク

ランナブルエンティティ（「ランナブル」とも呼ばれます）は SWC に含まれる一連のコードを指し、ランタイムおいて RTE によりトリガされます（3.2 項参照）。これは ASCET における「プロセス」にほぼ相当します。

各ソフトウェアコンポーネントは 1 つまたは複数のランナブルエンティティで構成され、RTE は、ランタイムにおいて各ランナブルエンティティにアクセスします。ランナブルエンティティは、以下のイベントによりトリガされます。

- **タイミングイベント** ー 周期的スケジューリングイベント（周期的タイマティックなど）です。ランナブルエンティティは通常実行用のエントリポイントを提供します。
- R ポートのデータ受信によりトリガされるイベント（DataReceive イベント）。

AUTOSAR ランナブルエンティティはいくつかのカテゴリに分類されますが、ASCET はカテゴリ 1 のランナブルエンティティをサポートしています。

ランナブルエンティティを実行するためには、そのランナブルエンティティを AUTOSAR オペレーティングシステムのタスクに割り当てます。

3.3.4 ランタイム環境

RTE（ランタイム環境）については 3.2 項で説明しています。

4 チュートリアル

このチュートリアルは、主に、ASCET を初めて使うユーザーを対象として、実際のモデル記述作業を例に ASCET の基本的な使い方を説明するものです。チュートリアルは、互いに関連しあうコンポーネントを扱う複数のレッスンに分かれています。なお、チュートリアルを始める前に、あらかじめ「ASCET を理解する」（15 ページ）を読んで ASCET の基本概念を理解しておいてください。

4.1 単純なブロックダイアグラムを作成する

ASCET では、アプリケーションを構成するメインブロックとしてクラスやモジュール等のコンポーネントを使用します。製品に含まれている既成のコンポーネントを使ったり、独自のコンポーネントを新たに作成することもできます。

ASCET では、コンポーネントは、通常グラフィックを使用して記述します。すべてのコンポーネントの定義が終わったらそれらを組み合わせ、ASCET ソフトウェアシステムの基礎となるプロジェクトを構築します。最終的なソフトウェアシステムは、グラフィカルなモデル記述から生成された C コードで構成され、マイクロコントローラや実験ターゲットコンピュータ上で実行することができます。

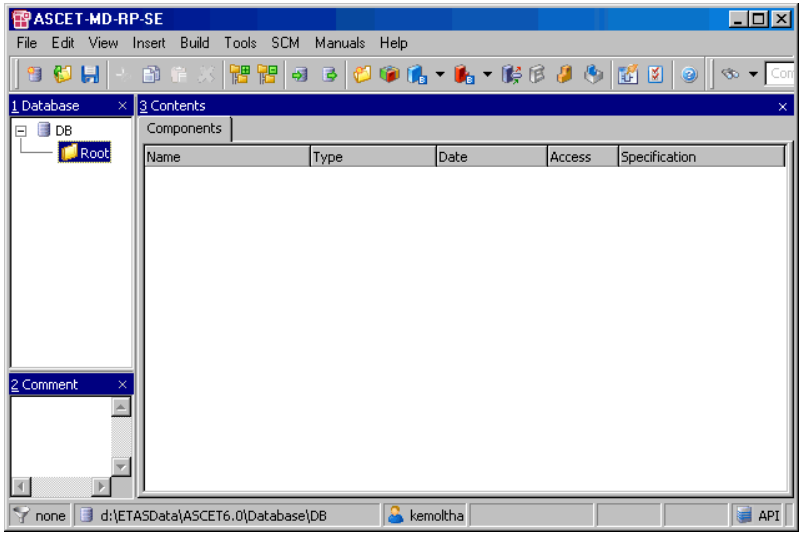
4.1.1 準備

最初に、まずチュートリアルのデータベースを開きます。このチュートリアルで作業するコンポーネントはすべてこのデータベースに格納されるので、すべての作業はこのデータベースを開いた状態で行います。

このチュートリアルで作成するコンポーネントとプロジェクトは、データベース `Tutorial` 内の `ETAS_Tutorial_Solutions` というフォルダ内にあらかじめ格納されているので、ここで説明するコンポーネントをすべてユーザー自分で定義する必要はありません。

ただし、少なくともレッスン 1、3、および 4 のコンポーネントだけは、練習のためにご自分で作成することをお勧めします。

ASCET を起動すると、以下のようなコンポーネントマネージャが開き、前回のセッションで開いていたデータベースがロードされます。



`Tutorial` データベースがない場合は、以下のようにして新しいデータベースを作成します。

新しいデータベースを作成する：

- コンポーネントマネージャから、**File → New Database** を選択します。

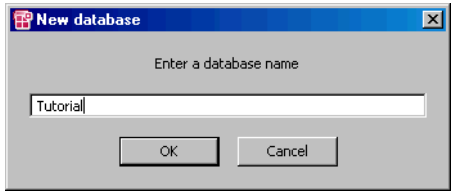
または

- **New** ボタンをクリックします。

または

- **<Ctrl> + <N>** を押します。

  “New database” ダイアログボックスが開きます。



- `Tutorial` という名前を入力します。

- **OK** をクリックします。

  新しいデータベースが作成されます。ここには、データベース名と Root というフォルダのみが含まれます。

Tutorial データベースすでに存在している場合は、以下のようにしてそのデータベースを開きます。

データベースを開く：

- コンポーネントマネージャから、**File → Open Database** を選択します。

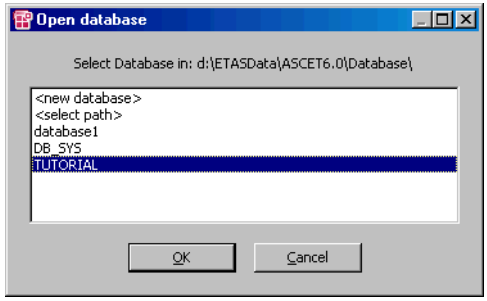
または

- **Open** ボタンをクリックします。

または

- **<Ctrl> + <O>** を押します。

  “Open database”ダイアログボックスが開き、現在設定されているデータベースパスに存在するデータベースの一覧が表示されます。



- 一覧の中に Tutorial データベースがある場合は、それを選択して **OK** をクリックします。

  コンポーネントマネージャに Tutorial データベースの内容が表示されます。

- Tutorialデータベースが他のパスに格納されている場合には、<select path> オプションを選択して **OK** をクリックします。
- “Select database path”ダイアログボックスが開くので、データベースを指定して **OK** をクリックします。

新しいコンポーネントを作成するために、まず Tutorial という名前の新しいトップレベルのフォルダと LessonN という各レッスン用のサブフォルダを作成します。

新しいトップレベルフォルダを作成する：

- “1 Database”ペインでデータベース名を選択します。
- メニューアイテム**Insert** → **Folder**を選択します。

または

- **Insert Folder** ボタンをクリックします。

または

- **<Insert>** を押します。

  “1 Database”ペインに `Root` という名前の新しいトップレベルのフォルダが表示されます。
- トップレベルフォルダの名前を`Tutorial`に変更します。強調表示されている名前に上書き入力してから **<Enter>** を押します。
- フォルダ `Tutorial` を選択します。
- もう一度 **Insert** → **Folder** を選択します。

  “1 Database”ペインに `Folder` という名前の新しいフォルダが表示されます。
- 新しいフォルダの名前を`Lesson1`に変更します。

このチュートリアルで作成するコンポーネントはすべて、いずれかの `LessonN` フォルダに格納します。各レッスンごとに新しいフォルダを作成してください。データベースにはトップレベルフォルダが必ず 1 つ以上あり、その中に任意の数のサブフォルダを作成することができます。

注記

すべてのフォルダ名やアイテム名、およびそれらの中の変数やメソッドの名前は、ANSI C 標準に準拠している必要があります。

次に、最初のコンポーネントを `Lesson1` フォルダに作成します。

コンポーネントを作成する：

- “1 Database”ペインのフォルダ `Lesson1` をクリックします。
- **Insert** → **Class** → **Block Diagram** を選択します。

  “1 Database”ペインの `Lesson1` フォルダの下に、`Class_Blockdiagram` という新しいコンポーネントが表示されます。このコンポーネントは *class* タイプです。class タイプは ASCET で頻繁に使用されます。

- このコンポーネントの名前を`Addition`に変更します。

4.1.2 クラスを定義する

`Tutorial/Lesson1` フォルダに新しいコンポーネントを作成したので、次にその機能を定義します。まず最初にコンポーネントのインターフェース、つまり、メソッド、引数、および戻り値を定義し、次にそのコンポーネントの機能を定義するブロックダイアグラムを作成します。

コンポーネントの機能を定義する：

- “1 Database”ペインでコンポーネント `Addition` を選択します。
- メニューアイテム **Edit → Open Component** を選択します。

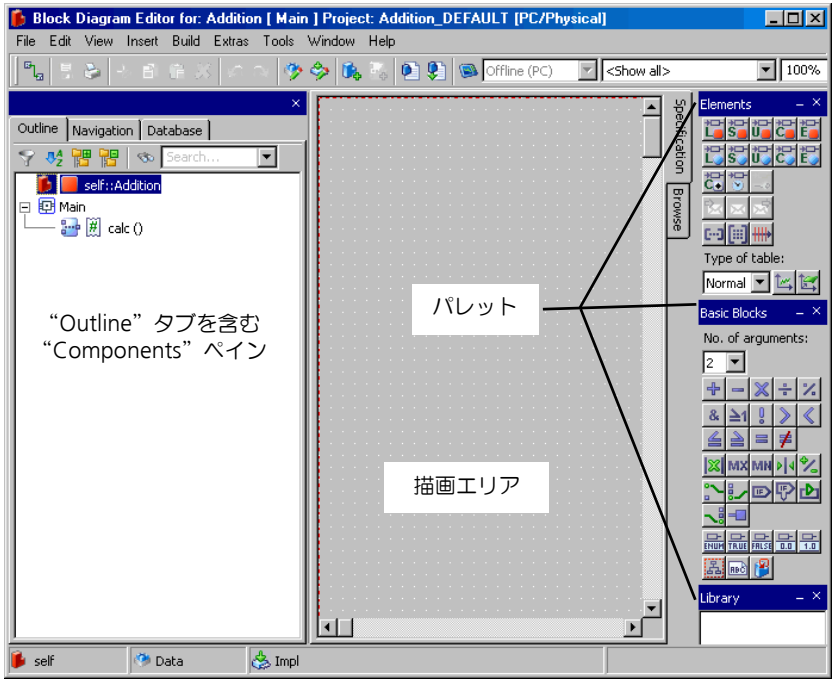
または

- コンポーネントをダブルクリックします。

または

- **<Enter>** を押します。

  ブロックダイアグラムエディタが開きます。これは、コンポーネントの機能を定義するためのメインウィンドウです。



- “Outline”タブに表示されているメソッド `calc` を選択します。このメソッドは、デフォルトメソッドとして自動的に作成されたものです。
- **Edit → Rename** を選択します。

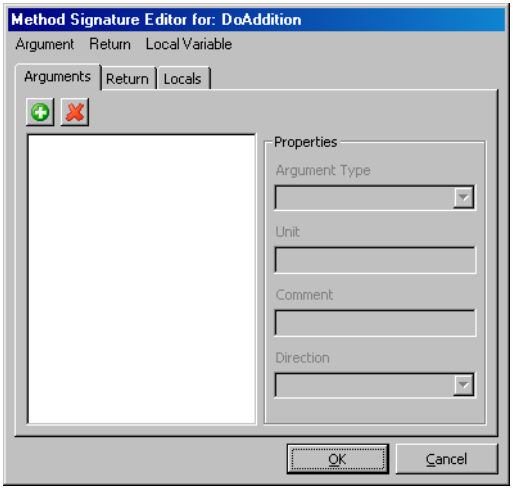
または

- **<F2>** を押します。

  メソッド名 `calc` が強調表示されます。
- このメソッドの名前を `DoAddition` に変更します。
- **Edit → Properties** を選択します。

または

- メソッド名をダブルクリックします。
  このメソッドのシグネチャエディタが開きます。



クラスには、少なくとも 1 つのメソッドを定義する必要があります。ASCET におけるメソッドは、オブジェクト指向プログラミング言語で用いられる「メソッド」や、手続き型プログラミング言語で用いられる「関数」に似ています。メソッドは任意の数の引数を入力し、1 つの戻り値を出力することができます（これらはすべて任意指定です）。引数は、コンポーネントにデータを渡すために使用され、戻り値はコンポーネント内で行われた計算の結果を「外側」に返すために使用されます。

これらの「メソッドシグネチャ」は、シグネチャエディタを使用して定義します。ここでは `continuous` 型の引数を 2 つと戻り値を 1 つ定義します。

メソッドシグネチャを定義する：

- シグネチャエディタで **Argument → Add** を選択します。
  “Arguments” ペインに、`arg` という新しい引数が表示されます。
- この引数の名前を `input1` にします。
- もう 1 つの引数を作成し、`input2` という名前を付けます。
  デフォルトでは引数の型は *continuous*（または略して *cont* ）になっています。ここでは、この型のまま使用します。
- シグネチャエディタの "Return" タブを選択します。

- **Return Value** オプションをオンにします。

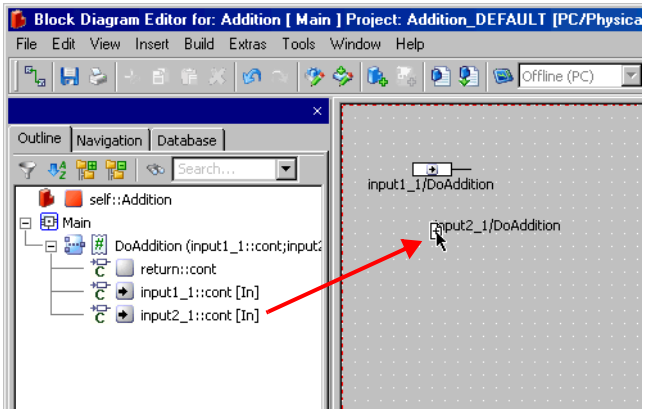
  戻り値の型も、デフォルトで *cont* になります。
- **OK** をクリックして、シグネチャエディタを閉じます。

メソッド `DoAddition` の引数と戻り値の名前が、ブロックダイアグラムエディタ左側の“Outline”タブ内のメソッドの下に表示されます。これで、ブロックダイアグラムを作成してこのコンポーネントの機能を定義する準備が整いました。

コンポーネント Addition の機能を定義する：

- “Outline”タブの第 1 引数をドラッグして、ブロックダイアグラムエディタの描画エリアにドロップします。

  この引数のシンボルが、描画エリアに表示されます。



- 同じドラッグアンドドロップ操作により、もう 1 つの引数と戻り値を追加します。

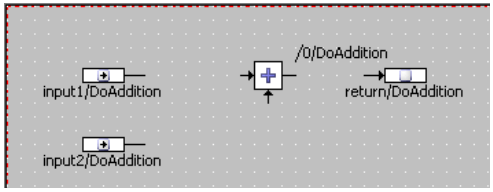
- “Basic Blocks”パレット内の **Addition** ボタンをクリックします。

  マウスカーソルに加算演算子がロードされました。
- 描画エリア内の、引数のシンボルと戻り値のシンボルの間をクリックします。

  加算シンボルが挿入されます。デフォルトでは、これには 2 つの入力ピン（矢印で示されます）と、1 つの出力ピンが付いています。出力ピンは右側に付きます。

エレメントと演算子は、描画エリアの任意の位置にドラッグして配置することができます。

次に、エレメント同士を接続して、情報の流れを定義します。



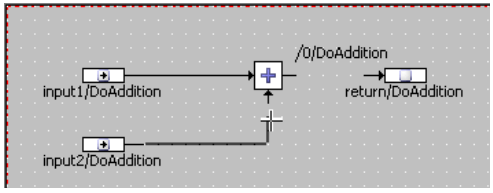
ダイアグラムエレメントを接続する：

- “General”ツールバー内の **Connect** ボタンをクリックします。
- または、描画エリア内の、エレメントが配置されていない部分を右クリックします。

  カーソルは、描画エリア内で十字カーソルに変わります。
- 第 1 引数を表すシンボルの出力ピンをクリックして、接続を開始します。

  マウスカーソルを動かすと、それに従って線が引かれます。描画エリア内でクリックするたびに、その時点で引かれていた線が固定されます。このようにして、接続線の道筋を確定していきます。
- 加算シンボルの左側にある入力をクリックします。

  これで、第 1 引数のシンボルが加算シンボルの入力に接続されます。



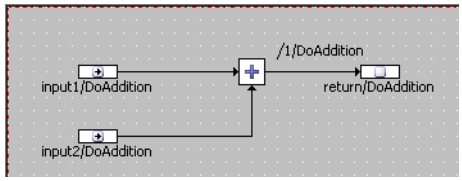
- 第 2 引数のシンボルを、加算シンボルのもう 1 つの入力に接続します。

- 戻り値のシンボルを、加算シンボルの出力に接続します。

  加算演算子と戻り値との接続が、緑色の線で表示されます。緑色の線は、この演算のシーケンスがまだ決定されていないときうことを表わしています。

- **Tools → Sequence Calls → Sequencing - Ignore Current** を選択して、加算のシーケンスを自動的に決定します。

  加算演算子と戻り値の間の接続が黒色の線で表示されます。



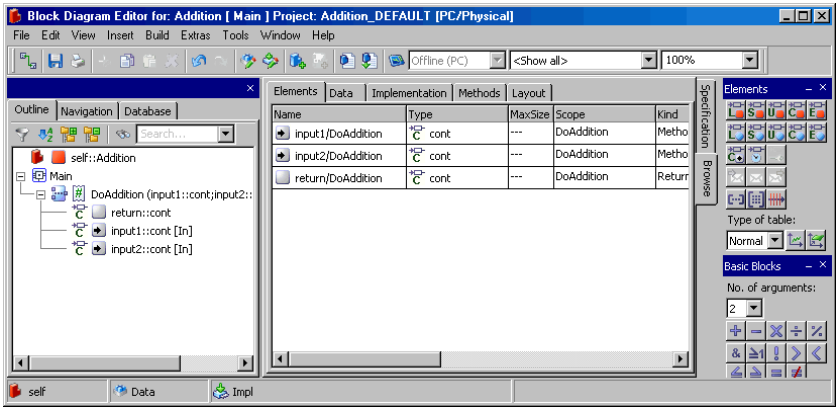
これで、最初のコンポーネントの定義、つまり機能記述は完了です。

最後に、コンポーネントのレイアウト、つまり、このコンポーネントが他のコンポーネント内で使用される際にどのように表示されるようにするかを定義します。

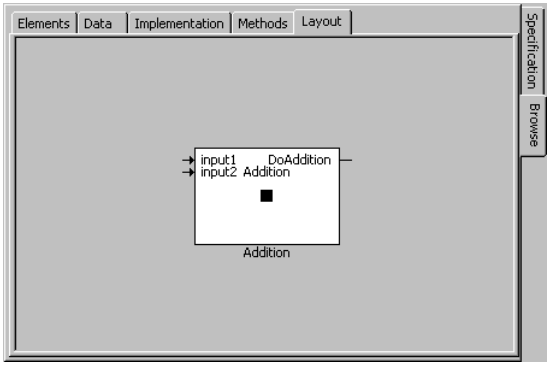
レイアウトの編集方法は 2 通りあります。

コンポーネントのレイアウトを編集する：

- ウィンドウ右端の“Browse”タブをクリックして“Browse”ビューに切り替えます。

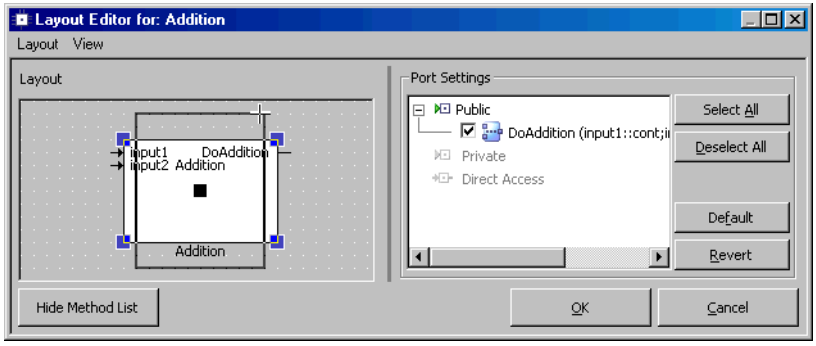


- “Layout”タブを選択し、レイアウトが表示されたエリアをダブルクリックしてレイアウトエディタを開きます。



- または、**Edit → Component → Layout** を選択します。

  レイアウトエディタ（“Layout Editor”ダイアログ）が開きます。



- ブロックをクリックするとそのブロックのハンドルが表示されるので、そのハンドルをドラッグしてブロックを任意のサイズに変更します。
- 引数と戻り値のピンをドラッグして、配置を調整します。
- **OK** をクリックします。

これで、コンポーネントは完成しました。このレッスンの最後の作業として、コンポーネントをデータベースに保存します。

コンポーネント `Addition` を保存する：

- **File → Save** を選択し、ブロックダイアグラムエディタを閉じます。

- コンポーネントマネージャで **Save** ボタンをクリックします。

  作業内容は、この操作をするまではディスクに書き込まれません。

  ブロックダイアグラムエディタ内で **Save** を選択した際は、変更部分はキャッシュメモリに格納されるだけです。そこで、作業中は定期的にコンポーネントマネージャの **Save** ボタンで保存を行うことをお勧めします。

  ユーザーオプションを設定して、変更内容が自動的に保存されるようにすることもできます（オンラインヘルプを参照してください）。

ここで、練習のため、同じ機能を ESDL（**E**mbedded **S**oftware **D**escription **L**anguage）でモデリングしてみましょう。そうすれば、ESDL エディタや、外部ソースコードエディタの使用法を習得することができます。

まず初めに、モジュールインターフェースを ESDL タイプの新しいモジュールにコピーしてから、そのモジュールの名前を変更します。それから、そのモジュールで実現したい機能を、ASCET の ESDL エディタ、または外部テキストエディタを使用して記述します。

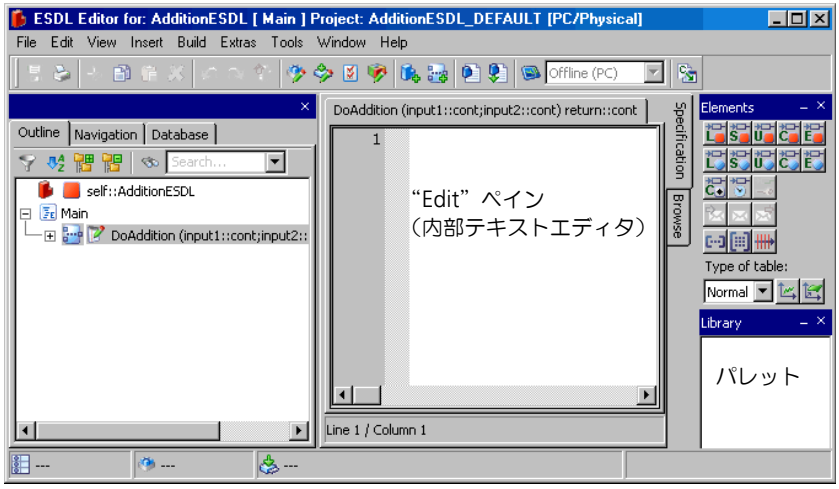
コンポーネント `Addition` をコピーして定義する：

- コンポーネントマネージャの“1 Database”ペインでコンポーネント `Addition` を選択します。
- ショートカットメニューから **Edit → Reproduce As → ESDL** を選択します。

  コンポーネントのコピーが作成され、`Addition1` という名前が自動的に付けられます。
- 新しいコンポーネントの名前を `AdditionESDL` に変更します。

- “1 Database”ペインで、新しいコンポーネントの名前をダブルクリックします。

  `AdditionESDL` 用のエディタが開きます。このエディタにもさまざまな編集機能が備わっています。



- 内部テキストエディタの“Edit”ペインに、以下のような機能を入力します。

  ```
  return input1 + input2;
  ```

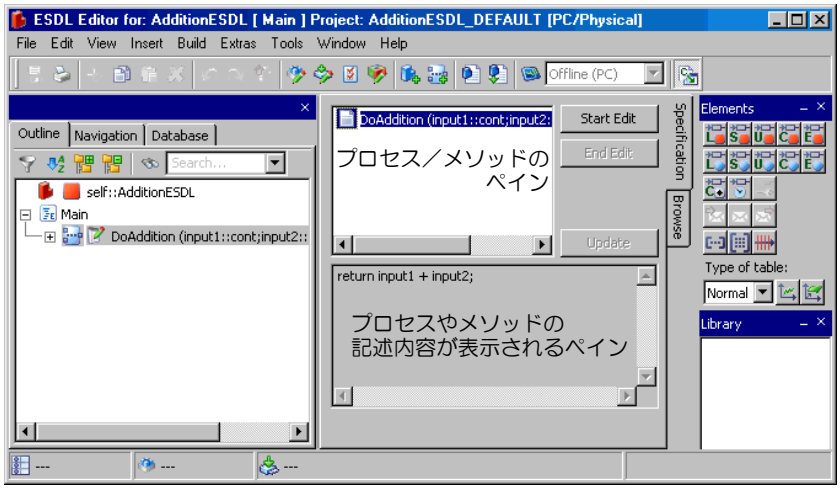
- 今度は **Activate External Editor** ボタンをクリックして、外部エディタモードに切り替えます。

  変更内容を保存するかどうか、尋ねられます。

- **Yes** をクリックして確定します。

  変更内容が保存され、ESDL エディタは外部エディタモードに切り替わります。

  外部エディタモードに切り替わると、以下のような画面になります。



- プロセス／メソッドのペインから、定義したいメソッドまたはプロセスを選択します。

  すでに入力されている機能が定義フィールドに表示され、**Start Edit** ボタンが有効になります。

- **Start Edit** をクリックして外部エディタを起動します。

注記

外部エディタが起動されると、Windows に登録されている、*.c と *.h というファイルに割り当てられたアプリケーションが呼び出されます。この外部エディタと ASCET 間のデータ転送は一時ファイルを介して行われるので、外部エディタを閉じる前や ESDL エディタにデータを転送する前には、ファイルが保存されていることを確認してください。

Start Edit ボタンが無効になり、**End Edit** および **Update** ボタンが有効になります。

- 外部エディタで機能を記述します。
- 記述した機能を保存します。
- ESDL エディタで **Update** をクリックし、外部エディタからデータを転送します。

- 最後に **End Edit** をクリックして、外部エディタとの連結を解除します。

  End Edit をクリックしても外部エディタは開いたままですが、それ以降、外部エディタで変更した内容を ESDL エディタに転送することはできません。
- 再度 **Activate External Editor** をクリックして、外部エディタモードを終了します。
- **Build → Analyze Diagram** を選択して、入力したコードを検証します。

  エラーがあった場合は、ASCET モニタウィンドウに表示されます。

4.1.3 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- データベースを開く
- フォルダを作成して名前を付ける
- コンポーネントを作成して名前を付ける
- メソッドのインターフェースを定義する
- 描画エリアにダイアグラムエレメントを配置する
- ダイアグラムエレメントを接続する
- コンポーネントのレイアウトを編集する
- 機能記述用のビュー（“Specification”ビュー）とブラウザビュー（“Browse”ビュー）とを切り替える
- コンポーネントを保存する
- コンポーネントのインターフェースをコピーする
- ESDL エディタを使用する
- 外部エディタを使用する

4.2 コンポーネントの実験を行う

`Addition` および `AdditionESDL` というコンポーネントができあがったので、それを用いて実験を行います。実験環境では、コンポーネントがどのように機能するかを、シミュレーションによって運用時と同様に見ることができます。実験環境には各種ツールが用意されていて、コンポーネント内の入力、出力、パラメータ（適合変数）、および変数（測定変数）の値をモニタすることが可能です。

4.2.1 実験環境（Experiment Environment）を起動する

実験環境を、ブロックダイアグラムエディタまたは ESDL エディタから起動します。起動するには、実験したいコンポーネントを指定して実験環境を開きます。

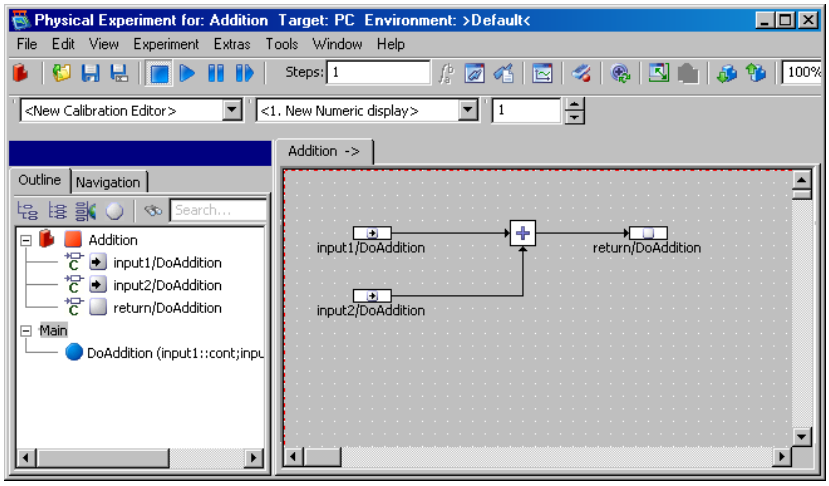
実験環境を起動する：

- ASCET コンポーネントマネージャから、`¥Tutorial¥Lesson1` フォルダに入っている `Addition` のブロックダイアグラムを開きます。
- ブロックダイアグラムエディタで **Build → Experiment** を選択します。

  実験用のコード生成が行われます。ASCET はユーザーが定義したモデルを分析し、そのモデルで記述されている機能を実現する C コードを生成します。さまざまなプラットフォーム用のコードを生成することができます。

  このレッスンでは、デフォルト設定を使用して PC 用のコードを生成します。

コードの生成とコンパイルが終わると、実験環境が開きます。



4.2.2 実験をセットアップする

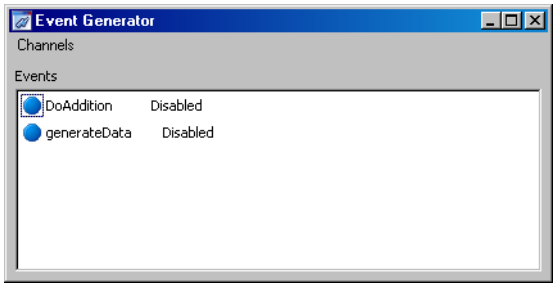
実際に実験を開始する前に、環境の設定が必要です。つまり、実験用に生成する入力値を指定し、さらに実験結果をどのように表示するかを指定します。このために、イベントジェネレータ、データジェネレータ、そして最後に測定システムを順にセットアップしていきます。

イベントジェネレータをセットアップする：

- **Open Event Generator** ボタンをクリックします。

  “Event Generator”ウィンドウが開きます。シミュレートする各メソッドごとに、イベントを 1 つと `generateData` イベントを作成する必要

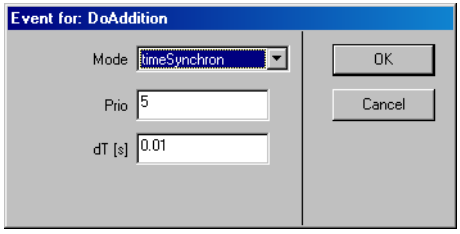
があります。運用時にはオペレーティングシステムが行うスケジューリングが、これらのイベントによってシミュレートされます。



- `DoAddition` イベントを選択します。
- **Channels → Enable** を選択します。
- もう一度、イベント `DoAddition` を選択します。
- **Channels → Edit** を選択します。

  これら 2 つの操作は、"Elements" フィールドのショートカットメニューからも行えます。

  “Event” ダイアログボックスが開きます。

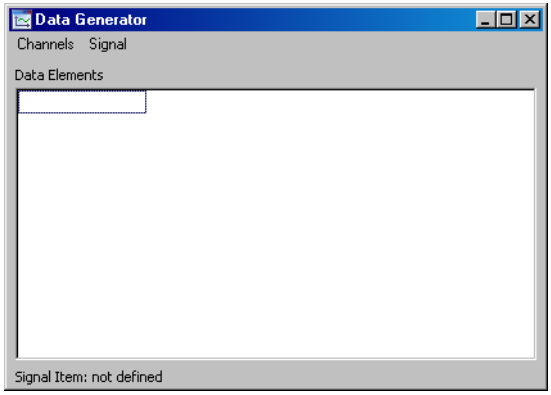


- `dT` の値を 0.001 にします。
- **OK** をクリックします。
- イベントジェネレータ内で `generateData` イベントを選択し、その `dT` 値を 0.001 にします。
- “Event Generator” ウィンドウを閉じます。

データジェネレータをセットアップする：

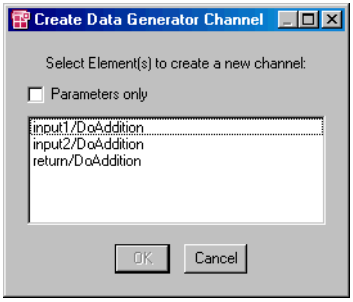
- **Open Data Generator** ボタンをクリックします。

  “Data Generator”ウィンドウが開きます。



- **Channels → Create** を選択します。

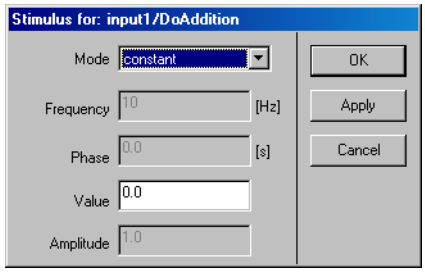
  “Create Data Generator Channel”ダイアログボックスが開きます。



- エレメントの一覧から、`input1/DoAddition` および `input2/DoAddition` という変数を選択します。
- **OK** をクリックします。

  2 つの変数が“Data Generator”ウィンドウの“Data Elements”ペインにリストアップされます。
- “Data Elements”ペインの `input1/DoAddition` を選択します。

- **Channels → Edit** を選択します。
  “Stimulus” ダイアログボックスが開きます。



- 以下の値を設定します。

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

- **OK** をクリックして “Stimulus” ダイアログボックスを閉じます。
- 同様にして `input2` の値を以下のように設定します。

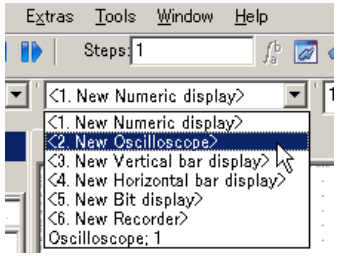
  Mode : sine
  Frequency : 2.0Hz
  Phase : 0.0s
  Offset : 0.0
  Amplitude : 2.0

- “Data Generator” ウィンドウを閉じます。

上記のように設定すると、周波数と振幅が異なる 2 つの正弦波が得られます。`Addition` コンポーネントはこの 2 つの正弦波を合成し、その結果として得られるカーブを出力します。

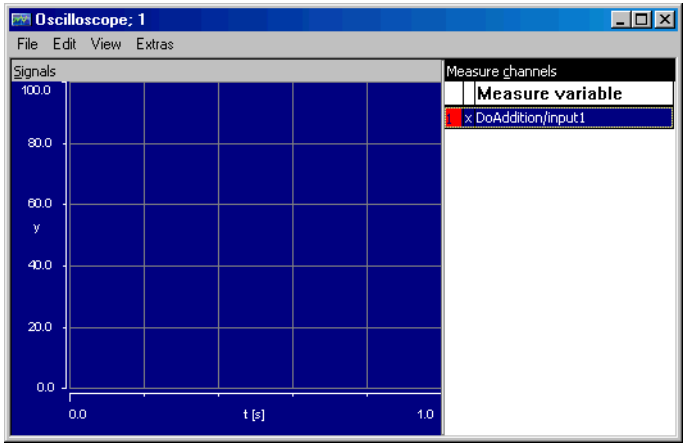
これらのカーブをオシロスコープに表示して波形を確認できるようにするため、測定システムをセットアップします。

- “Physical Experiment”ウィンドウの“Measure View”コンボボックスで、データ表示タイプとして `<2. New Oscilloscope>` を選択します。



- “Outline”タブ内のエレメントリストを開きます。
- `input1/DoAddition` を選択します。
- **Extras** → **Measure** を選択します。

  `input1` という測定チャンネルが割り当てられたオシロスコープウィンドウが開きます。実験環境の“Measure View”リストが更新され、そこにこの測定ウィンドウのタイトルが追加されます。



- “Physical Experiment”ウィンドウの“Outline”タブ内の `input2/DoAddition` を選択します。
- **Extras** → **Measure** を選択します。

  `input2` という測定チャンネルがオシロスコープに追加されます。

- 同様にして`return/DoAddition`を測定ウィンドウに追加します。
- 実験環境で、**File → Save Environment** を選択します。

これで実験のセットアップが終わり、実験を開始することができるようになりました。環境設定を保存したので、次回、このコンポーネント用の実験環境を起動すると、同じ環境が再びロードされます。

4.2.3 実験環境を使用する

実験環境には、コンポーネント内の変数の値を表示したり、実験実行中にセットアップ内容を変更するための機能が用意されています。また、値の表示方法を選択したり調整することもできます。

実験を開始する：

- “Physical Experiment”ウィンドウの **Start Offline Experiment** ボタンをクリックして実験を開始します。

  シミュレーションが実行され、その結果がオシロスコープに表示されます。

- **Stop Offline Experiment** ボタンをクリックして、実験を停止します。

この時点では、オシロスコープには、カーブのごく一部しか表示されていません。オシロスコープにカーブの他の部分を表示するには、値軸（信号値を示すY座標軸）のスケールを変更する必要があります。

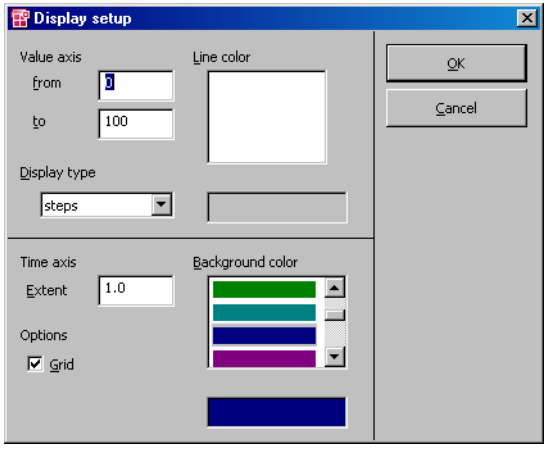
オシロスコープのスケールを変更する：

- オシロスコープウィンドウの“Measure Channels”リストから、3つのチャンネルをすべて選択します。

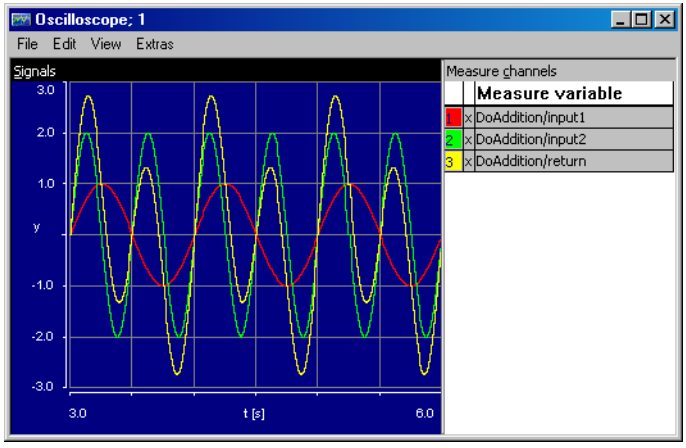
  複数のチャンネルを選択するには、**<Ctrl>** キーを押し下げたまま個々のチャンネルをクリックします。

  すべてのデータエレメントが強調表示され、これから行う変更は3つのチャンネルすべてに反映されます。
- **Extras → Setup** を選択します。

“Display Setup”ダイアログボックスが開きます。



- “Value Axis”の範囲を -3 ～ 3 に変更します。
- “Time Axis Extent”を 3 にします。
- “Backgroud color”リストから背景色を選択します。
- **<Enter>** を押します。



これで、値が適切なスケーリングで表示されるようになりました。入力された 2 つの正弦波と、その 2 つを合成した出力波形が表示されます。入力値を調整して、出力がどのように影響を受けるかを調べてみましょう。

実験の入力値を変更する：

- “Physical Experiment”ウィンドウで **Tools** → **Data Generator** を選択して、“Data Generator”ウィンドウを開きます。
- データジェネレータ上で、変更したい変数を選択します。
- **Channels** → **Edit** を選択します。
  “Stimulus”ダイアログボックスが開きます。
- 値を適宜調整します。
- **Apply** をクリックします。

オシロスコープのカーブが、新しい設定に応じて変化します。実験の実行中に、すべての設定を変更することができます。

4.2.4 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

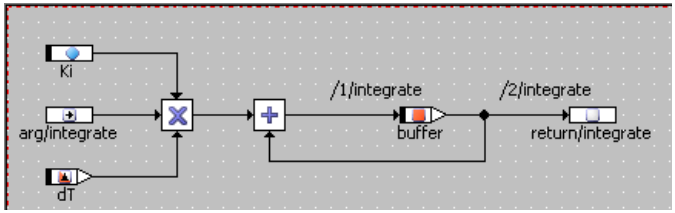
- 実験環境を呼び出す
- イベントジェネレータをセットアップする
- データジェネレータをセットアップする
- 測定システムをセットアップする
- 実験を開始し、停止する
- 実験の環境設定を保存する
- 実験実行中にスティミュレーションの内容を変更する

4.3 再利用可能なコンポーネントを定義する

このレッスンでは、マイクロコントローラソフトウェアでよく用いられる機能単位である「積分器」のクラスを作成します。このダイアグラムはやや複雑ですが、ここまでに学習した方法で作成し、実験することができます。

この例では、与えられる時間と速度から走行距離を算出する積分器を定義します。速度はメートル／秒の単位で与えられ、それを dT（秒）の周期で積算します。各周期ごとに値がアキュムレータに累算され、アキュムレータの値は、所定の時間経過後の走行距離（メートル）となります。

ASCET では、アキュムレータなどの標準ブロックは、シンプルに図示されます。



4.3.1 ダイアグラムを作成する

ダイアグラムを記述する前に、`addition` コンポーネントの場合と同様の準備作業が必要です。まず `Tutorial` フォルダ内に新しいフォルダを作成してから、新しいクラスを追加します。これで、メソッドのインターフェース、さらにブロックダイアグラムとレイアウトを定義することができます。

まず、フォルダと新しいクラスを作成することから始めます。

積分器のクラスを作成する：

- コンポーネントマネージャで、`Tutorial` フォルダを開きます。
- 新しいフォルダを作成し、その名前を `Lesson3` にします。
- `Lesson3` フォルダ内に新しいクラスを作成し、その名前を `Integrator` にします。

積分器のインターフェースを定義する：

- “1 Database” ペインで、エレメント `Integrator` を選択します。
- エレメントをダブルクリックするか、または **Edit → Open Component** を選択します。

  ブロックダイアグラムエディタが開きます。
- メソッド `calc` の名前を `integrate` に変更します。

- メソッド `integrate` を編集して、1 つの引数（`cont` 型）と戻り値（`cont` 型）を追加します。
- `integrate` の引数と戻り値を描画エリアに配置します。

この積分器は、ここまでのレッスンでは使用されなかった「変数」および「パラメータ」という 2 つのタイプのエレメントを使用します。

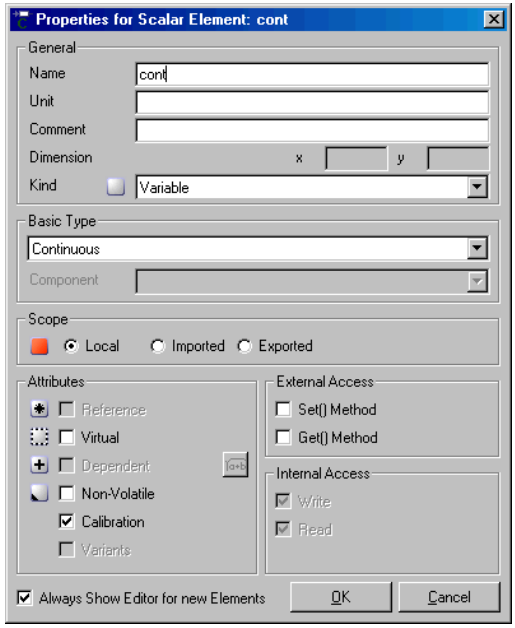
「変数」は、プログラミング言語の変数と同じように用いられます。変数には値を格納することができ、以降の計算ではその値を読みとることができます。これとは対照的に、「パラメータ」は読みとり専用です。パラメータの値は、たとえば新しい実験環境での適合作業によってなど、コンポーネントの外部でしか変更できず、コンポーネント内での計算で上書きすることはできません。なお ETAS の測定・適合ツールでは、「変数」は「測定変数」、「パラメータ」は「適合変数」と呼ばれています。

さらにこの例では、「依存パラメータ 」を定義します。しかし、これは積分器の機能とは無関係です。依存パラメータ は 1 つまたは複数のパラメータに依存するもので、その値は別のパラメータの値から算出されます。この計算は値が定義された時や適合された時にだけ行われます。依存パラメータは、ターゲットコード内においては通常のパラメータと全く同じように機能します。

変数を作成する：

- “Elements”パレット内の **Continuous Variables** ボタンをクリックします。

  プロパティエディタが開きます。



- “Name”フィールドに `buffer` という名前を入力します。
- **OK** をクリックします。

  変数の名前が `buffer` になりました。マウスカーソルにこの continuous 型の変数がロードされ、カーソルが十字カーソルに変わります。
- 描画エリア内をクリックして、変数を配置します。

  変数が描画エリア内に配置され、“Outline”タブ内にその変数の名前が反転表示の状態で追加されます。

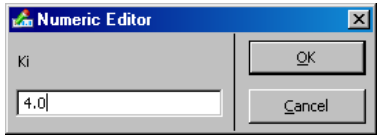
上記の操作でプロパティエディタが自動的に開かない場合は、そのまま変数を描画エリアに配置し、その後、“Outline”タブ内のその変数をダブルクリックしてプロパティエディタを開いてください。プロパティエディタの **Always show dialog for new elements** オプションをオンにしておけば、次回のエレメント作成時から、自動的にプロパティエディタが開くようになります。

パラメータを作成する：

- **Continuous Parameter** ボタンをクリックします。

  プロパティエディタが開きます。
- “Name” フィールドに `Ki` という名前を入力します。
- **OK** をクリックします。
- 描画エリア内をクリックして、パラメータを配置します。
- “Outline” タブでこのパラメータを右クリックし、ショートカットメニューから **Data** を選択します。

  データコンフィギュレーションウィンドウ（数値エディタ）が開きます。



- 入力フィールドに `4.0` と入力してから **<Enter>** を押します。

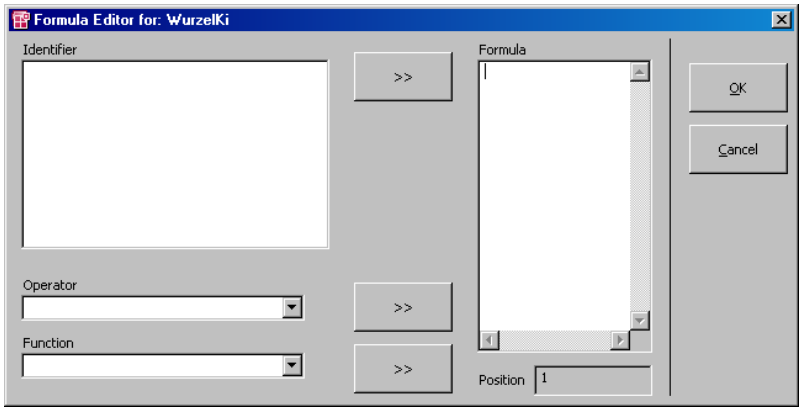
  この値がパラメータのデフォルト値になります。ダイアグラム内のすべてのパラメータや変数について、このようにしてデフォルト値を割り当てることができます。

依存パラメータを作成する：

- **Continuous Parameter** ボタンをクリックします。

  プロパティエディタが開きます。
- “Name” フィールドに `WurzelKi` という名前を入力します。
- “Attribute” フィールドの **Dependent** オプションを選択します。

- **Formula** ボタンでフォーミュラエディタを開きます。



- “Identifier” フィールドを右クリックしてショートカットメニューを開き、**Add** を選択します。

  仮パラメータが作成されます。
- “Identifier” フィールドに新しいパラメータの名前 x を入力します。
- “Formula” フィールドに変換規則を定義します。

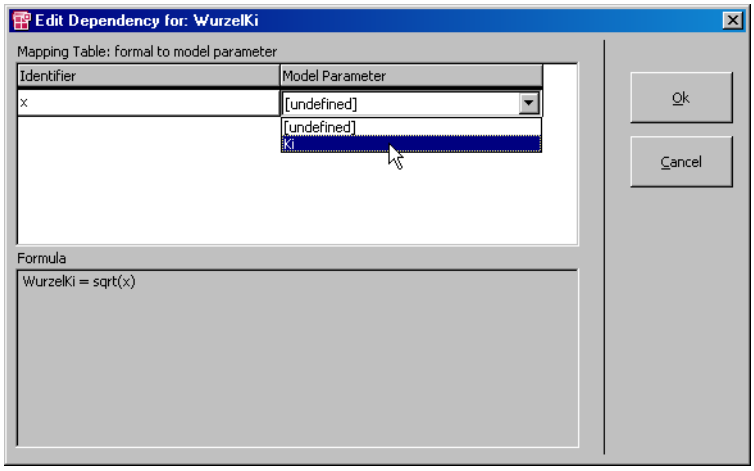
  “Operator” コンボボックスから演算子を選択して、複雑な式を定義することができます。演算子は、“Operator” フィールドのとなりの >> ボタンを押すたびに 1 つずつ “Formula” フィールドに挿入されます。

  同様に、“Function” コンボボックスで関数を選択し、“Function” コンボボックスの隣の >> ボタンを押して “Formula” フィールドに挿入することもできます。
- ここでは例として、仮パラメータの平方根を求める計算を定義します。

  Identifier：x
  Formula：`sqrt(x)`
- **OK** でプロパティエディタを閉じます。

  カーソルが十字カーソルに変わります。
- 描画エリア内をクリックして、パラメータを配置します。

- ブロックダイアグラムエディタで、“Outline”タブの `WurzelKi` を右クリックしてショートカットメニューを開き、**Data** を選択します。
- “Dependency Editor”ウィンドウで、コンボボックス内のモデルパラメータ（この例では `Ki`）を仮パラメータに割り当てます。



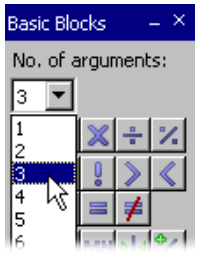
- **OK** をクリックして、データ入力を完了します。

  これで、パラメータ `Ki` に依存し、適合時に `Ki` に基づいて自動的に算出される依存パラメータが定義されました。後で実験を行う時に、この依存関係を確認することができます。

すべてのエレメントを作成したので、次に積分器を定義します。ダイアグラムの残りの部分を以下のようにして完成させてください。

ダイアグラムを作成する：

- “Basic Blocks”パレット内“No. of arguments"コンボボックスの値を 3 にし、乗算演算子の入力の数を指定します。



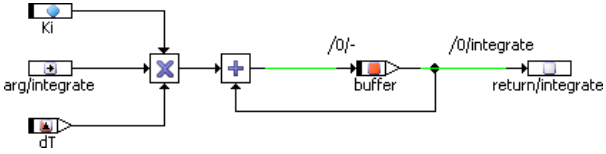
- 乗算演算子を作成し、描画エリアに配置します。

- **dT** ボタンをクリックして dT エレメントを作成します。

  プロパティエディタが開きます。この演算子を作成する前に、“Argument Size”の値を 2 に戻してください。
- dT エレメントを描画エリアに配置します。
- 2 つの入力を持つ加算演算子を作成し、描画エリアに配置します。

  この演算子を作成する前に、“Argument Size”の値を 2 に戻してください。
- エレメントを下図のように接続します。

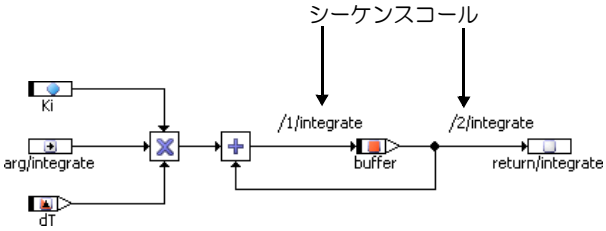
  バッファと戻り値の入力を示す線は、緑色で表示されます。



ここまでで、ダイアグラムのすべてのエレメントの設定が終わりました。次に、シーケンスコールを定義して、計算の順序を決定します。

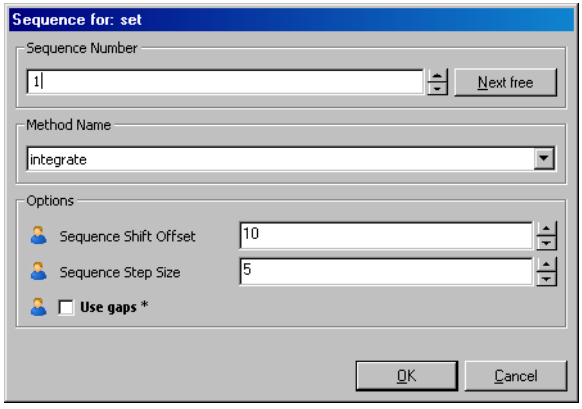
シーケンスコールに値を割り当てる：

- 変数 `buffer` の上に表示されているシーケンスコールを右クリックします。



- ショートカットメニューから **Edit** を選択します。

  シーケンスエディタが開きます。



- **OK** をクリックして、デフォルト設定を確定します。

  この設定により、この代入処理は、積分器のアルゴリズムの冒頭部分で行われます。

シーケンスコール内のシーケンス番号を調整する：

- `integrate` の戻り値の上に表示されているシーケンスコールを右クリックします。
- ショートカットメニューから **Edit** を選択します。
- シーケンスエディタで、“Sequence Number”の値を 2 にします。
- **OK** をクリックします。

  この設定により、変数 `buffer` が更新された後に戻り値への代入が行われます。

レイアウトを調整する：

- **Edit → Component → Layout** を選択します。
  レイアウトエディタが開きます。
- または、“Information/Browse” ビューから “Layout” タブ内のレイアウトをダブルクリックして、レイアウトエディタを開くこともできます。
- 引数を `integrate` からブロックの左側の中程にドラッグします。
- 戻り値をブロックの右側の中程にドラッグします。
- **OK** をクリックします。

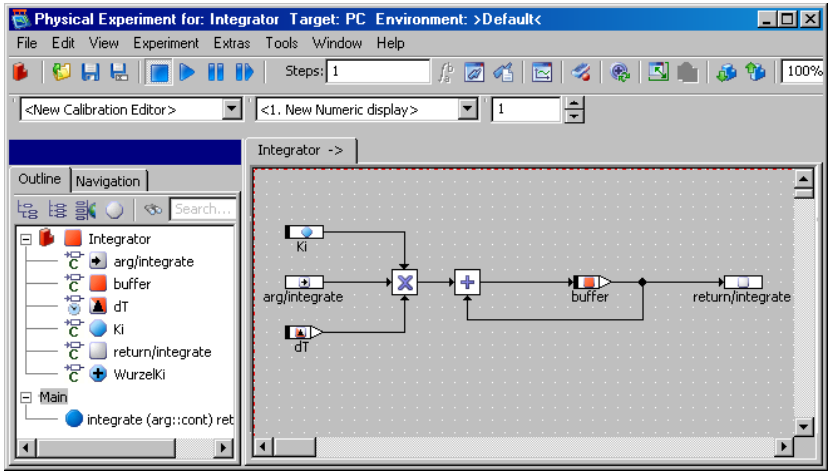
これで、積分器クラスのダイアグラムは完成です。今度は **File → Save** を選択して、ダイアグラムの変更内容を保存します。ダイアグラム自体に影響を与えない部分の変更内容は、自動的に保存されます。次に、コンポーネントマネージャウィンドウで **File → Save Database** を選択して、変更内容をデータベースに保存します。

4.3.2 積分器の実験を行う

ここでも、初めにイベントジェネレータ、次にデータジェネレータ、最後に測定システムをセットアップします。

積分器用の実験をセットアップする：

- **Build → Experiment** を選択して、実験環境を開きます。



- **Event Generator** ボタンをクリックします。
- イベント `integrate` をアクティブにして、`dT` はデフォルト値 0.01 のままにします。
- “Event generator” ウィンドウを閉じます。

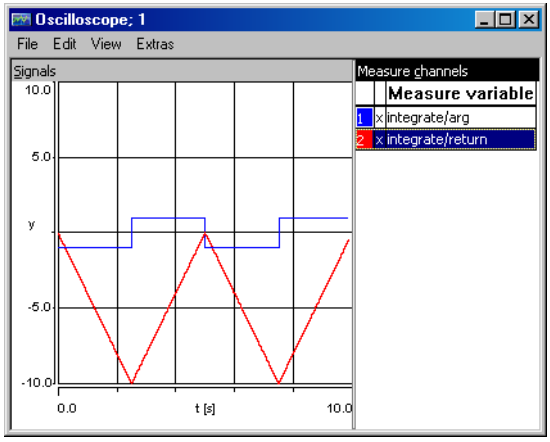
- **Data Generator** ボタンをクリックします。
- **Channels → Create** を選択して `integrate` から引数を選択し、`integrate` メソッド用のデータチャンネルを作成します。
- 以下の値を設定します。

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

- データジェネレータを閉じます。
- `integrate` メソッドの `arg` と `return` をオシロスコープウィンドウに割り当てます。
- 値軸の範囲を -10～10 にし、時間軸の範囲を 10 秒にします。

- **Start Offline Experiment** をクリックして、実験を開始します。

`integrate` メソッドの出力値は、引数が正なら増加し、負なら減少します。入力カーブの正と負の部分の時間は等しいので、安定範囲内の値が出力され続けます。



実験を停止してから再開する際、すべての変数の値は停止された時に保存された値で続行されますが、場合によっては変数を初期値にリセットする必要が生じる場合もあります。

実験をリセットする：

- **Extras → Reinitialize → Variables**（または **Parameters** 、**Both**）を選択します。

  選択したコマンドに応じて、すべての変数またはすべてのパラメータ、またはその両方が初期値にリセットされます。

次に、実験の条件を変更して、積分器ファンクションのカーブを表示してみましょう。`Ki` パラメータを調整、つまり「適合」し、入力値を変更します。

積分器の実験を行う：

- “Outline”タブの `Integrator` エレメントを展開します。
- パラメータ `Ki` を選択します。
- **Extras → Calibrate** を選択します。

  このパラメータ用の数値エディタが開きます。
- 5 を入力します。

  オシロスコープの出力カーブの勾配が急になります。
- 値を 3 にします。

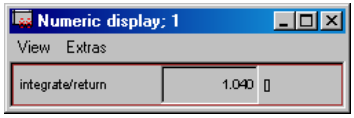
  出力カーブの傾斜が緩くなります。
- パラメータの値を 4 に戻してから数値エディタを閉じます。
- “Data Generator”ウィンドウを開きます。
- 入力パルスのオフセットを `-0.5` にします。
- **OK** をクリックします。

これで正の部分の方が大きくなるので、出力値が増加し、ある時点でオシロスコープの表示範囲を超えてしまいます。このような場合、個々の値について、オシロスコープのスケールを変更することができます。また、以下のようにして数値表示ウィンドウを開いて出力値を表示することもできます。

値を数値で表示する：

- 実験環境の“Measure View”コンボボックスから `<1. Numeric Display>` を選択します。
- “Outline”タブで、`integrate` メソッドの戻り値 (`return`) を選択します。

- **Extras → Measure** を選択します。
  “Numeric display” ウィンドウに、現在の戻り値が表示されます。



- 依存パラメータ `WurzelKi` も表示します。

- `Ki` の値を変え、**Update Dependent Paramters** ボタンで `WurzelKi` を更新し、実験を行います。

4.3.3 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- パラメータを作成する
- 依存パラメータを作成し定義する
- 変数を作成する
- 複数の入力を伴う演算子を作成する
- シーケンスコールのシーケンス番号を設定する
- デフォルト値を割り当てる
- 実験中に値を適合する
- “Numeric display” ウィンドウに値を表示する

4.4 実際的な例

このレッスンでは、標準の PI フィルタを若干拡張したものをベースにして、コントローラを作成します。このコントローラは、自動車のアイドリング時のエンジン回転速度を一定に保つために用いられます。

エンジンのアイドリング速度をコントロールする場合、実際の回転数 n がアイドリング時の目標値 `n_nominal` に近い状態を確実に維持する必要があります。`n_nominal` から値 n を引いて、コントロールする偏差を決定します。

この実際の回転数の偏差が、`air_nominal` の値を算出するための基礎になります。`air_nominal` はスロットル位置、つまりエンジンの吸気量を決定します。

4.4.1 コントローラを定義する

コントローラのダイアグラムを作成する手順は、これまでのレッスンで用いた手順と同じです。

- コンポーネントマネージャで新しいフォルダを追加し、コンポーネントを作成します。

- インターフェースを定義し、ブロックダイアグラムを作成します。

これまでのレッスンと大きく異なるのは、コントローラをモジュールとして実装する点です。モジュールはプロジェクトの最上レベルのコンポーネントとして用いられます。モジュール内では、プロジェクトを構成する「プロセス」が定義されます。

コントローラコンポーネントを作成する：

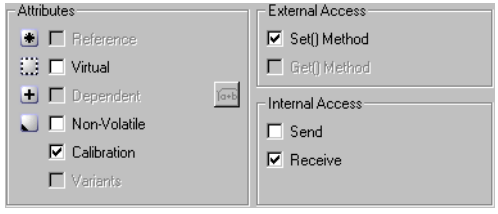
- コンポーネントマネージャで、新しいサブフォルダを `Tutorial` フォルダに追加し、その名前を `Lesson4` にします。
- `Lesson4` フォルダを選択し、**Insert → Module → Block diagram** を選択して新しいモジュールを追加します。
- 新しいモジュールの名前を `IdleCon` にし、ブロックダイアグラムエディタを開きます。
- “Outline” タブで、ダイアグラム `process` の名前を `p_idle` に変更します。

モジュールの機能は「プロセス」という単位で定義されます。モジュールにおけるプロセスは、クラスにおけるメソッドに相当します。プロセスはメソッドとは異なり、引数や戻り値を伴いません。ASCET におけるプロセス間の通信、つまりデータ交換は、「受信メッセージ」（入力）および「送信メッセージ」（出力）と呼ばれる方向性のあるメッセージを使用して行われます。

ここで作成するコントローラは、実際の回転数を受信メッセージ n として受け取り、それを使用して算出されたスロットル位置を `air_nominal` という送信メッセージに代入して出力します。

コントローラのインターフェースを定義する：

- **Receive Message** ボタンをクリックして受信メッセージを作成し、その名前を n にします。
- メッセージ n のプロパティエディタを開き、**Set() Method** オプションをオンにします。



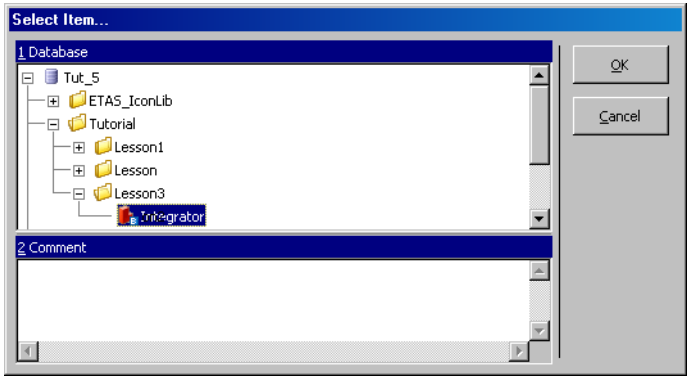
- **Send Message**ボタンをクリックしてから描画エリア内をクリックして、送信メッセージを作成します。

- 送信メッセージの名前を `air_nominal` にします。
- メッセージ`air_nominal`のプロパティエディタを開き、**Get() Method** オプションをオンにします。

このコントローラには、レッスン 3 で作成した積分器 Integrator を使用します。

コントローラに Integrator を追加する：

- **Insert → Component** を選択して、“Select item”ダイアログボックスを開きます。



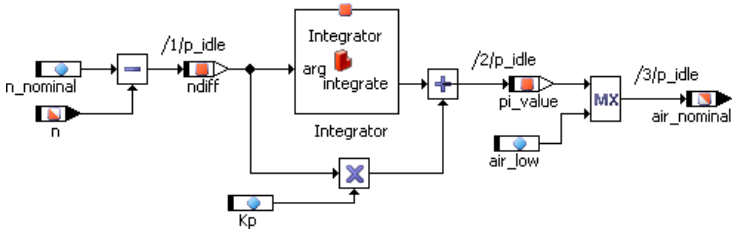
- “1 Database”ペインの `Tutorial¥Lesson3` フォルダから `Integrator` を選択し、**OK** をクリックします。
- 積分器がコンポーネント `IdleCon` に内包されます。コンポーネントは参照により内包されるため、この積分器の元の定義を変更すると、他のコンポーネントに内包されるこの積分器のコンポーネントにも、その変更が反映されます。

これまでに追加したエレメント以外に、以下のエレメントをコントローラに追加する必要があります。

- `ndiff` と `pi_value` という、continuous 型の 2 つの変数
- `n_nominal`、`Kp`、および `air_low` という、continuous 型の 3 つのパラメータ

コントローラの残りの部分を定義する：

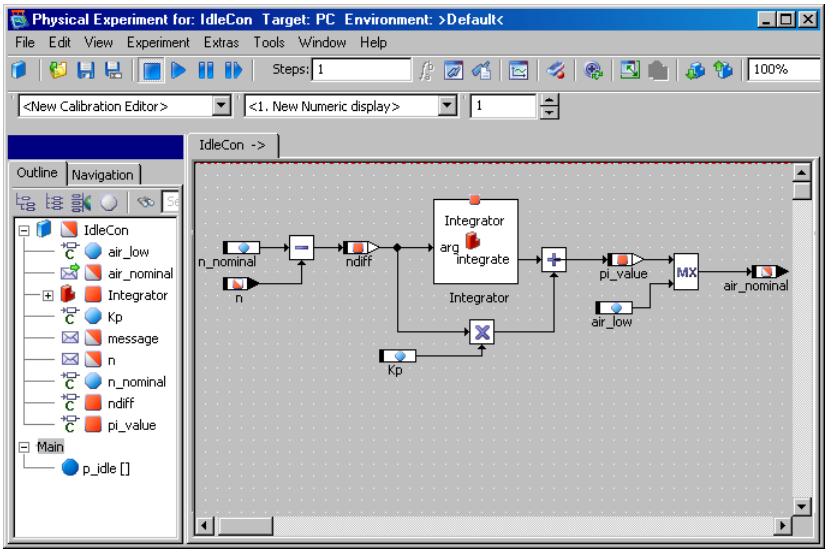
- 必要な演算子や他のエレメントを作成してから、それらを下のブロックダイアグラムのように接続します。



- “Outline”タブから、`n_nominal` パラメータを選択して、**Edit** → **Data** を選択します。
- `n_nominal` の値を 900 にします。
- `Kp` の値を 0.5 にします。
- ダイアグラム内の定義を保存し、変更をデータベースに適用します。

4.4.2 コントローラの実験を行う

モジュールの実験は、コンポーネントの実験と同様の手順で行います。まず、データジェネレータとイベントジェネレータをセットアップし、次に測定システムをセットアップします。



実験をセットアップする：

- **Build → Experiment** を選択して、実験環境を開きます。
- “Event Generator”ウィンドウを開き、プロセス `p_idle` 用のイベントをイネーブルにして、`dT` はデフォルト値の 0.01 のままにしておきます。

  プロセス用のイベントも、メソッド用のイベントと同じように機能します。
- “Data Generator”ウィンドウを開き、受信メッセージ n 用のチャンネルに以下の値を設定します。

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

- 変数 `n_diff` および `air_nominal` をオシロスコープに割り当てます。
- オシロスコープの値軸の範囲を -500 ～ 500 に、時間軸の範囲を 2 にします。

- **Save Environment** ボタンをクリックします。

これで実験のセットアップが終わり、回転数の偏差とスロットル位置の関係を表示できるようになりました。

コントローラの実験を行う：

- **Start Offline Experiment** ボタンをクリックして、実験を開始します。
- 変数KiおよびKp用の適合ウィンドウを開きます。そこから、値 `Ki` および `Kp` を調整し、その出力に対する影響を調べることができます。

  有意義な値に戻すために、実験中にモデルを再初期化する必要があるかもしれません。

4.4.3 プロジェクト

プロジェクトは、完結したソフトウェアシステムとして機能する ASCET ソフトウェアの単位です。プロジェクトは、実験ターゲットやマイクロコントローラターゲットを使用して、オンラインでリアルタイムな実験を行うことができます。個々のコンポーネントの実験はオフライン（PC ベース）でしか行えません。

実験はプロジェクト単位で実行されます。プロジェクト用のコードが生成されると、オペレーティングシステムコードも必ず生成されます。ASCET で作成されたソフトウェアシステムをリアルタイムに実行するには、オペレーティングシステムのセットアップが必要です。ソフトウェアシステムをリアルタイムに実行する実験を「オンライン実験」と呼びます。これまでのレッスンで行った実験はすべてオフライン実験であり、リアルタイムなシミュレーションは行われませんでした。

注記

オンライン／オフラインにかかわらず、ASCET の実験は実際にはすべてプロジェクトの単位で行われます。このことはオフライン実験でデフォルトプロジェクトを使用する（このプロジェクトはユーザーからは見えない場合もあります）ことでも明らかです。オペレーティングシステムを定義する目ために明示的にプロジェクトを作成してセットアップしなければならないのは、オンライン実験の場合だけです。ただし、ユーザー独自のアプリケーション用にデフォルトプロジェクトを設定することもできます。

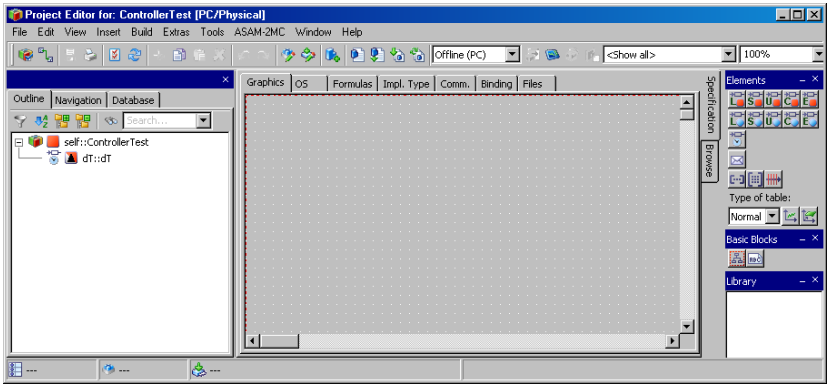
4.4.4 プロジェクトをセットアップする

コンポーネントマネージャで、`IdleCon` モジュールと同じフォルダにプロジェクトを作成します。

プロジェクトを作成する：

- コンポーネントマネージャで、**Insert → Project** を選択するか、または **Insert Project** ボタンをクリックして、新しいプロジェクトを追加します。
- プロジェクトの名前を`ControllerTest`にします。
- **Edit → Open Component** を選択するか、またはプロジェクトエレメントをダブルクリックします。

  プロジェクト用のプロジェクトエディタが開きます。



次に、プロジェクトの“Outline”タブに `IdleCon` コントローラを追加します。

プロジェクトにコンポーネントを追加する：

- プロジェクトエディタで **Insert → Component** を選択して、“Select Item”ダイアログボックスを開きます。
- “1 Database”リストから、`Tutorial/Lesson4` フォルダのコンポーネント、`IdleCon` を選択します。
- **OK** をクリックして、このコンポーネントを追加します。

  コンポーネントの名前が、プロジェクトエディタの“Outline”タブに表示されます。

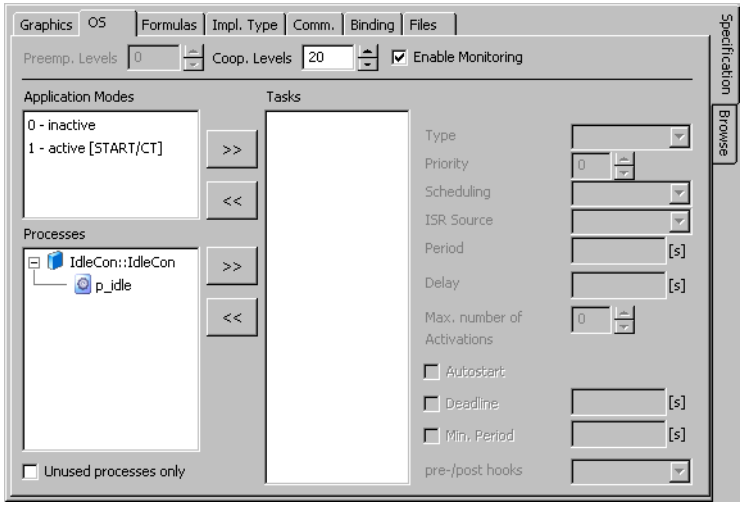
参照されるコンポーネントは、非参照コンポーネントに内包されます。つまり、内包されたコンポーネントのダイアグラムを変更すると、その変更がプロジェクト全体に反映されます。

プロジェクトに含まれるタスクとプロセスのスケジューリングは、オペレーティングシステムによって行われます。プロジェクト用のコードを生成する前に、タスクをいくつか作成してそれらにプロセスを割り当てておく必要があります。

オペレーティングシステムのスケジューリング条件は、プロジェクトエディタの“OS”タブに定義します。ここで、`p_idle` プロセスが 10ms ごとに起動されるように、オペレーティングシステムのスケジュールを定義しましょう。

プロジェクト用のオペレーティングシステムのスケジュールをセットアップする：

- “OS”タブをクリックします。



- **Task → Add** を選択して、新しいタスクを作成します。
- 作成したタスクの名前を `Task10ms` にします。

  作成されるタスクは、デフォルトでは Alarm タスク、つまり、オペレーティングシステムにより周期的に起動されるタスクです。
- “Period”フィールドで、このタスクの周期を 0.01 秒にします。

  この周期は、タスクが起動される頻度を規定します。この例では、10ms ごとに起動されます。

- “Processes” リストの `IdleControl` というアイテムを展開します。
- プロセス `p_idle` を選択し、**Process → Assign** を選択します。

  このプロセスが `Task10ms` タスクに割り当てられ、“Tasks” リストのこのタスク名の下に表示されます。

プロジェクト内においては、インポートエレメントやエクスポートエレメントを用いてプロセス間通信を行います。これらのエレメントはグローバルエレメントで、モジュール間通信の送信メッセージと受信メッセージに相当します。グローバルエレメントはプロジェクト単位で宣言され、プロジェクト内の各モジュールに含まれる同名のエレメントに結び付けられていなくてはなりません。

グローバルエレメントを定義する：

- プロジェクトエディタで、**Extras → Global Elements → Resolve Globals** を選択します。

  グローバルエレメントが作成され、それぞれ対応するエレメントに結び付けられます。同じ名前のエレメント同士が自動的に結び付けられます。

4.4.5 プロジェクトの実験を行う

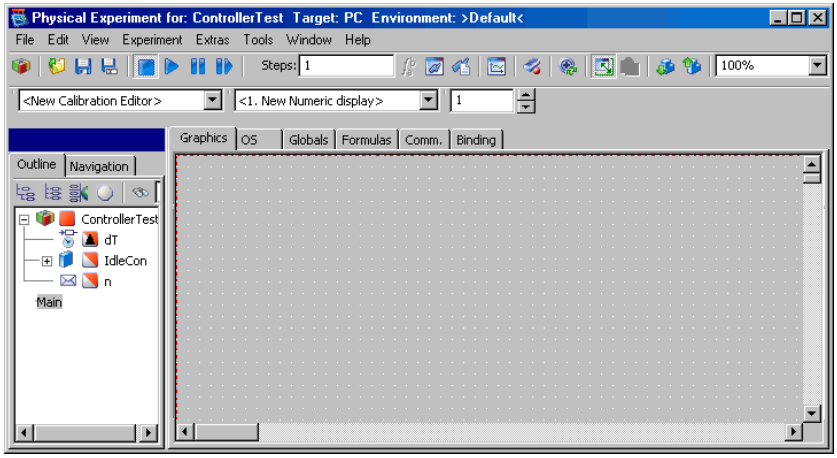
まず、このプロジェクトのオフライン実験を行います。オフライン実験は、ハードウェアを接続せずに PC 上で行うことができます。プロジェクトのデフォルト設定では PC 上で稼働するようになっているので、この設定を変更する必要はありません。プロジェクトのオフライン実験は、コンポーネントのオフライン実験と同様の手順で行います。

実験をセットアップする：

- コンポーネントマネージャで、**File → Save Database** を選択します。

  実験環境を起動する前に、変更内容を必ずデータベースに保存することをお勧めします。
- プロジェクトエディタで **Build → Experiment** を選択します。

プロジェクトのコードが生成され、オフライン実験が開きます。



- **Open Event Generator** ボタンをクリックします。

  プロジェクト用のイベントジェネレータには、コンポーネントの実験の場合のように各メソッドや各プロセス用のイベントではなく、実験で使用する各タスク用のイベントが表示されます。
- ダイアログボックスからタスク `generateData` をイネーブルにし、`dT` 値にはデフォルトの 0.01 秒を採用します。

  タスク `Task10ms` はデフォルト状態ですでにイネーブルになっているので、これでタスク generateData および 10ms のイベントの `dT` 値はともに 0.01 秒になります。これ以上変更の必要はありません。
- イベントジェネレータを閉じます。
- データジェネレータと測定システムを、前回の実験と同じ値でセットアップします（80 ページの「コントローラの実験を行う」を参照してください）。
- **File → Save Environment** を選択して、環境設定を保存します。

実験を行う：

- **Start Offline Experiment** ボタンをクリックします。

- 前項と同様にKiおよびKpパラメータを調整して、出力への影響を確認します。

4.4.6 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- モジュールを作成する
- モジュール内のメッセージを作成する
- コンポーネントマネージャで作成したコンポーネントをブロックダイアグラムに組み込む
- プロジェクトを作成する
- プロジェクトにコンポーネントを内包する
- タスクを作成し、それらにプロセスを割り当てる
- プロジェクトの実験を行う

4.5 プロジェクトを拡張する

このレッスンでは、コントローラを少し改良して一層実用的なものにします。センサで読みとった値を実際の値に変換する、シグナルコンバータを作成します。たとえば自動車制御アプリケーションで用いられるような多くのセンサは、温度、位置、毎分の回転数などの測定値に対応する電圧を返します。この電圧と測定値の関係は、必ずしも一次関数で表せるとは限らないので、ASCET では、この種の対応関係を効率的にモデリングできる特性テーブルを使用できます。

4.5.1 シグナルコンバータを定義する

シグナルコンバータをモデリングするために、まずフォルダとモジュールを作成して、機能を定義します。シグナルコンバータは 2 つの特性カーブを使用して、入力値に対する出力値を求めます。

モジュールを作成する：

- コンポーネントマネージャで、新しいフォルダ `Tutorial¥Lesson5` を作成します。
- 新しいモジュールを作成し、その名前を `SignalConv` にします。
- エレメントをダブルクリックしてブロックダイアグラムエディタを開きます。
- ブロックダイアグラムエディタで **Insert → Process** を選択して 2 つめのプロセスを作成します。
- 2 つのプロセスの名前を `n_sampling` および `t_sampling` にします。

- “Outline”タブに、2 つの受信メッセージ U_n および U_t と、2 つの送信メッセージ t および n を作成します。

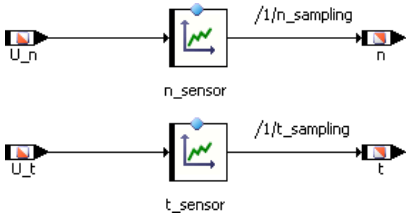
- **One-D Table** ボタンをクリックして、特性カーブエレメントを作成します。

  プロパティエディタが開きます。
- t_sensor という名前を入力します。
- “Dimension”フィールドの“x”の部分に値 13 を入力します。

  これにより、この特性カーブを最大 13 列まで広げることができるようになりました。

  特性カーブは 1 次元なので、“Dimension”フィールドの“y”の部分は無効になっています。
- **OK** をクリックしてプロパティエディタを閉じます。
- 描画エリア内をクリックして、テーブルを配置します。

  このテーブルが“Outline”タブに追加されます。
- 最大 2 列の第 2 のテーブルを作成し、n_sensor という名前にします。
- 下図のようにエレメントを接続し、シーケンシングを編集して、この処理を行うプロセスを割り当てます。



次に、2 つの特性カーブのデータを編集します。ASCET ではテーブルエディタを使用して各種テーブルデータ（配列、特性カーブ、および特性マップ）を編集します。

テーブルを編集する：

- テーブル t_sensor を右クリックし、ショートカットメニューから **Data** を選択します。

  テーブルエディタが開きます。

- テーブルのサイズを以下のように調整します。

テーブルが 13 列に拡張され、z 値はすべてデフォルトで 0 になります。

- 以下の値を入力します。上の行が X 行、下の行が Z 行の値です。

0.00 0.08 0.30 0.67 1.17 2.50 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

まずサンプルポイント（X 値）を左から右の順に入力して、テーブルを編集します。

- 編集しようとする X 値をクリックしてから、ダイアログボックスに新しい値を入力します。

  新しい値は両隣のサンプルポイントの間の値でなければなりません。

- 次に、Z 値（出力値）をクリックし、強調表示された値の上に適切な値を入力します。
- 同じ方法で、以下のデータを使用して、第 2 のテーブルを編集します。

  0.0 10.0
  0.0 6000.0

- ブロックダイアグラムエディタで、**File → Save** を選択します。
- コンポーネントマネージャで、**Save** ボタンをクリックして、変更内容を保存します。

この例では、第 2 のテーブルは出力が入力の変化に従って直線的に変化する関係を表せばよいので、必要なサンプルポイントは 2 つだけです。値の補間モードとして線形補間を指定したので、これで充分に機能します。

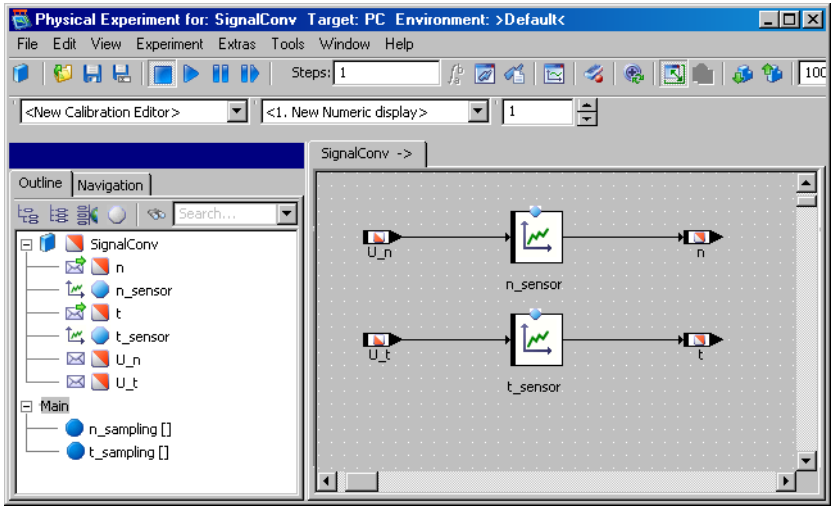
線形補間では、2 つのサンプルポイント間の入力値に対応する出力値は、直線から求められます。この場合、入力値が 0 なら 0 が返され、10 なら 6000 が返されます。入力値が 5 の場合、戻り値は補間により 3000 になります。

4.5.2 シグナルコンバータの実験を行う

新しいコンポーネントの実験を行い、テーブルによる変換処理の結果を調べてみましょう。2 つのテーブルの値の範囲は互いに異なるので、それぞれに専用のオシロスコープウィンドウを使用します。

実験をセットアップする：

- コンポーネントマネージャから **Build → Experiment** を選択して、実験環境を開きます。



- コンポーネント内の各プロセス（`n_sampling`、`t_sampling`、`generateData`）用にイベントを作成し、各イベントの `dT` 値を 4ms にします。
- データジェネレータで、メッセージ `U_n` 用と `U_t` 用にチャンネルを 1 つずつ作成し、両チャンネルに以下の値を設定します。

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

- メッセージ `n` および `U_n` のオシロスコープウィンドウと、メッセージ `t` および `U_t` のオシロスコープウィンドウを作成します。

  後者のオシロスコープを作成する際には、“Select Measure View”コンボボックスで必ず `<2. New Oscilloscope>` を選択しておいてください。

2 つのテーブルのサンプリングポイントの分解能とそれに対応する補間値は大きく異なるので、各チャンネルの表示設定は、それぞれのオシロスコープ内で個別に行う必要があります。

オシロスコープを測定値にあわせて調整する：

- プロセス `n_sampling` 用のオシロスコープ（チャンネル `U_n` および `n`）を選択します。
- “Measure Channels” リストから、メッセージ `n` を選択し、**Extras → Setup** を選択します。

  メッセージ `n` 用の “Display Setup” ダイアログボックスが開きます。
- 値軸の範囲を 0 ～ 6000 にし、時間軸の範囲を 0.5 にします。
- メッセージ `U_n` 用の “Display Setup” ダイアログボックスを開きます。
- その値軸の範囲を -1 ～ 11 にします。

  時間軸の範囲は、1 つのオシロスコープウィンドウ内のすべての変数について同じでなければならないので、時間軸の範囲を変更する必要はありません。
- プロセス `t_sampling`（チャンネル `U_t` および `t`）用のオシロスコープを選択し、そのチャンネルを以下のようにセットアップします。

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

- **File → Save Environment** ボタンをクリックして、環境設定を保存します。

これで、実験を実行してシグナルコンバータの機能を調べることができるようになりました。2 つの変換処理の相違を調べてみましょう。

実験を行う：

- **Start Offline Experiment** ボタンをクリックします。

  n_sensor テーブルでは、入力される正弦波の振幅だけが変化します。ここでの入力は 0 ～ 10 ボルトの電圧信号です。これが 0 ～ 6000rpm の回転速度にマッピングされます。

  テーブル t_sensor では、入力される電圧と出力される温度との関係は一次関数では表せません。この関係は自動車制御用に一般に用いられている温度センサの応答特性カーブと同じです。

- データジェネレータからの入力をさまざまな波形に変更して、両方の出力カーブに表れる影響を調べます。

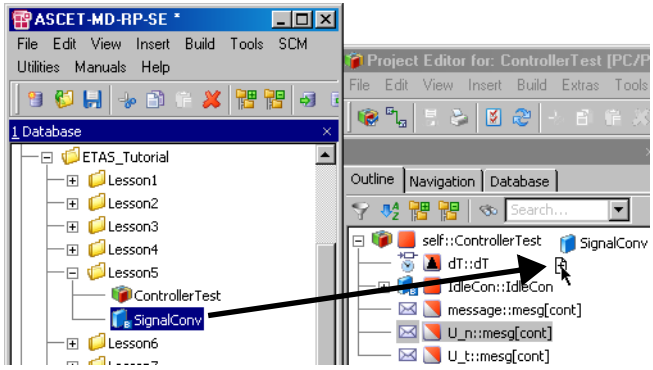
4.5.3 シグナルコンバータをプロジェクトに統合する

シグナルコンバータの定義が終わったので、これを、レッスン 4 で作成したプロジェクトに統合します。シグナルコンバータからの出力信号が、エンジンコントローラへの入力信号として使用されます。

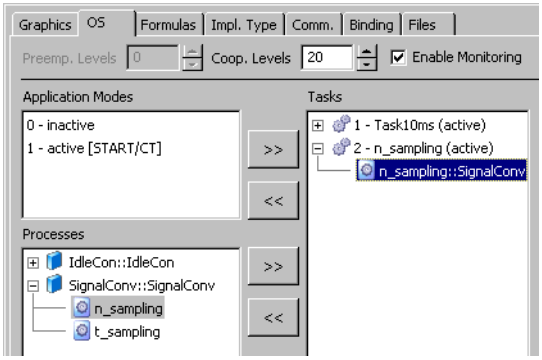
プロジェクトにシグナルコンバータを統合するために、新しいプロセス用のタスクをセットアップし、プロセス間通信に必要なグローバルエレメントを宣言してその結び付けを行います。

シグナルコンバータをプロジェクトに追加する：

- プロジェクト`ControllerTest`をプロジェクトエディタで開きます。
- モジュール `SignalConv` を、コンポーネントマネージャの“1 Database”リストからプロジェクトの“Outline”タブにドラッグします。



- “OS”タブをクリックして、オペレーティングシステムエディタを開きます。
- 新しいタスク `n_sampling` を作成します。
- 新しいタスクの周期を 0.004 秒にします。
- プロセス`n_sampling`をタスク`n_sampling`に割り当てます。



これで、プロジェクトに 2 つのタスクが定義されました。第 1 のタスクは 10 ミリ秒ごと、第 2 のタスクは 4 ミリ秒ごとに起動されます。1 つのタスクに割り当てられているすべてのプロセスは、そのタスクに定義されているインターバルで実行されます。この例ではそれぞれのタスクにプロセスが 1 つずつしかありませんが、必要に応じて任意の数のプロセスを割り当てることができます。

シグナルコンバータを統合するためには、次にモジュール間の通信を解決します。プロセス間通信はグローバルエレメントを介して行われます。プロジェクト内で用いられるすべてのグローバルエレメントは、対応するモジュール内でメッセージとして定義されていなければなりません。

デフォルトでは、送信メッセージはモジュール内に定義されますが、受信メッセージは、通常モジュールにインポートされるものであるため、プロジェクト内で受信メッセージを定義する必要があります。

グローバルエレメントをセットアップする：

- **Extras → Global Elements → Resolve Globals** を選択して、結び付けを自動で行います。
- **Extras → Global Elements → Delete Unused Globals** を選択して、前のレッスンで使用した結び付けを削除します。

必要なグローバルエレメントはすべて、自動的に作成され、同じ名前のエレメントに結び付けられます。たとえばグローバルメッセージ `U_n` は、`SignalConv` 内のメッセージ `U_n` に自動的に結び付けられます。

レッスン 4 ではメッセージ n をプロジェクトのグローバルメッセージとして定義しましたが、このレッスンではメッセージ n をモジュール SignalConv に定義し、モジュールのプロセス間通信に使用します。そのため、使用しないグローバルメッセージ n を削除する必要があります。

プロジェクトの実験を行う：

- **Build → Experiment** ボタンをクリックして実験環境を開きます。
- イベントジェネレータを開き、タスク n_sampling をイネーブルにします。
- このタスクの dT 値を 4 ミリ秒にします。

  プロジェクトのオフライン実験では、オンライン実験時にはオペレーティングシステムが行うスケジューリングを、イベントジェネレータがシミュレートします。
- データジェネレータを開き、既存のデータチャンネルを削除します。
- メッセージ U_n 用に新しいチャンネルを設定します。
- チャンネル U_n を以下のように設定します。

Mode：	pulse
Frequency：	1.0Hz
Phase：	0.0
Offset：	4/3
Amplitude：	1/3

- 回転速度センサの出力電圧 U_n をアクティブにします。

  シグナルコンバータは、特性テーブル n_sensor を使用してこの電圧値を回転数 n に変換します。

  上記の値により、n が前の実験（信号処理なし）と同じ範囲で出力されます。

- **Save Environment** ボタンをクリックします。
- 実験を開始します。

出力されるカーブは、信号処理を行わない例で出力されたカーブと同じはずです。データジェネレータによりスティミュレートされる値は異なりますが、テーブルで処理され、前の実験と同じ出力値になります。

4.5.4 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- 特性カーブエレメントを作成して使用する
- コンポーネントをプロジェクトに追加する
- プロジェクト内のコンポーネント間の通信を定義する

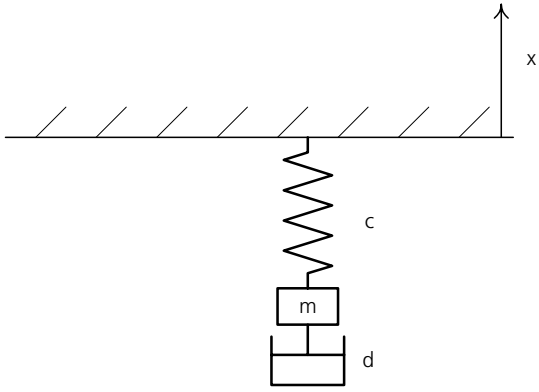
4.6 連続系をモデリングする

物理、機械、電子、およびメカトロニクスに関する処理を現実的にモデリングするには微分方程式を用いることが多く、連続系メソッドが要求されます。このようなメソッドをこれまでの章で作成したプロジェクトに統合する前に、本章では詳しい例を用いて、連続系のモデリングについて説明します。

ASCET は、いわゆる「CT ブロック」による連続系のモデリングとシミュレーションをサポートしています。CT は“Continuous Time”（連続時間）の略で、疑似連続的な時間ステップで計算処理が行われることを意味します。ASCET における連続系モデリングは、連続系の設計に用いられる標準的な記述形式である、状態空間表現をベースとしています。この形式では、CT 基本ブロックを非線形 1 次常微分方程式と非線形出力方程式で定義することができます。ASCET は、これらの微分方程式の最適な解を見つけるための、いくつかのリアルタイム積分メソッドを提供します（詳細は ASCET オンラインヘルプを参照してください）。

以下に、ばね－質量系の、地球の重力による減衰を伴う動きを例にして、連続系のモデリングの手順を説明します。

4.6.1 運動方程式



上図の質量 m には、以下の力が働いています。

- 重力：$F_g = -mg$
  （g = 重力の加速度）

- ばねの力：$F_F = -c\ (x + l_0)$
  （c = ばね定数、l_0 = 静止時のばねの長さ、x = 質量 m の位置）
- 減衰 $F_D = -\ d\ x'$
  （d = 減衰定数、x' = 質量の速度）

これにより、以下の運動方程式が得られます。

$$mx'' = -mg + F \text{ あるいは } x'' = -g + F/m\ (\text{ ただし } F = F_F + F_D)$$

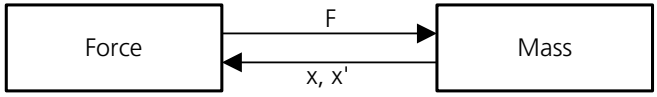
この 2 次微分方程式を（$x = x$、$v = x'$ により）2 つの 1 次微分方程式にすると、以下のようになります。

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

以下のモデル設計では、これらの微分方程式を使用します。

4.6.2 モデル設計

ばね－質量系のモデルは、CT ブロックを 1 つだけ用いてシンプルに設計することもできますが、ここでは 2 つの CT ブロックを用いてこのモデルをモデリングします。その過程で「直接通過」や「間接通過」のプロパティについて説明し、これらのプロパティを適切に設定して算術ループを避ける方法を紹介します。



- Force ブロックでは、質量 m の位置と、速度 x′ から求められる摩擦力に基づいて、ばねの力 F を算出します。
- Mass ブロックでは、ばねの力 F から加速度 x′′ を算出します。x′′ を積分して、速度 x′ と位置 x を算出します。

一見して、このシステムでは、どちらのブロックも相手のブロックから要求される出力を算出するために相手のブロックからの入力が必要なので、算術ループになってしまうように見えます。

このループは、「直接通過 」または「間接通過」のプロパティを適切に設定することによって回避することができます。

- Force ブロックでは、下の方程式により算出される出力変数 F は、入力変数 x および x′ に直接依存しています。したがって、このブロックは直接通過として定義されます。

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

- 一方、Mass ブロックでは、出力変数 x および x′ は入力変数 F に直接依存しているわけではなく、ブロックの内部ステート変数に依存しています。これらは、少なくとも初めは初期値になっているので、入力変数 F の値がわ

からなくても、初期値に基づいて出力変数 x および x′ を算出することができます。F の値がわかっている場合には、出力変数は微分方程式を用いて算出されます。

```
x′ = v
v′ = -g + F/m
```

したがって、このブロックは、間接通過として定義します。

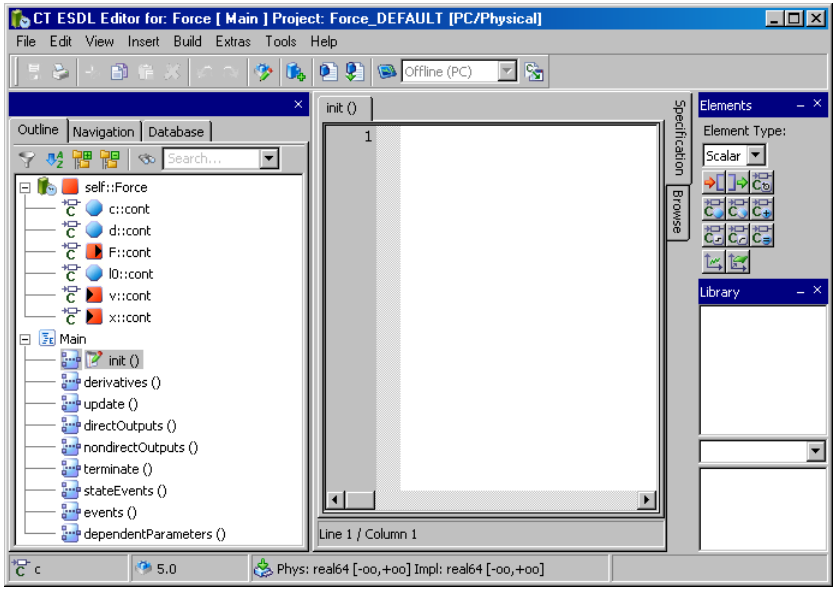
モデルを作成する：

- コンポーネントマネージャでフォルダを作成し、その名前を Lesson6 にします。
- このフォルダ内で **Insert → Continuous Time Block → ESDL** を選択して、Force および Mass というブロックを作成します。
- Force ブロックをダブルクリックして、ESDL エディタを開きます。

- **Input** ボタンをクリックして、x および v という 2 つの入力（continuous 型）を作成します。

- **Output** ボタンをクリックして、出力 F（continuous 型）を作成します。

- **Parameter** ボタンをクリックして、定数 c（ばね定数）、d（減衰定数）、l_0（静止時のばねの長さ）を作成します。



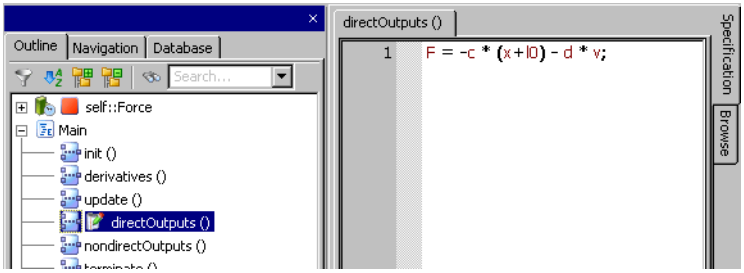
“Outline”タブ内のメソッドは自動的に作成されます。

- “Outline”タブ内の各定数を右クリックして、ショートカットメニューから **Data** を選択します。

  “Numeric Editor”ダイアログボックスが開きます。

- 定数に現実的な値を割り当てます（例、ばね定数 c には 5.0、減衰定数 d には 1.0、静止時のばねの長さ l_0 には 2.0）。
- “Outline”タブ内のメソッド `directOutputs[]` をクリックします。
- “Edit”フィールドに、力を算出する以下の式を定義します。

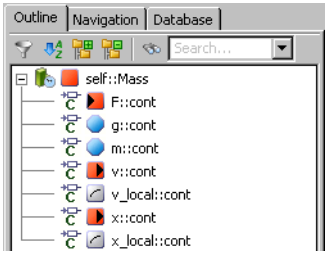
```
F = -c * (x + l0) - d*v;
```



- **Generate Code** ボタンをクリックします。

  CT ブロック `Force` がコンパイルされます。
- `Mass` ブロックをダブルクリックして、ESDL エディタを開きます。
- Force ブロックの場合と同様にして、入力 F、2 つの出力 x および v、パラメータ m（質量）、および定数 g（重力加速度）を作成します。
- Force ブロックの場合と同様にして、g および m に値を割り当てます（g は 9.81、質量 m はたとえば 2.0）。

- **Continuous State** ボタンをクリックして、出力を内部で計算するためのステート変数 `x_local` および `v_local` を作成します。



- `derivatives[]` メソッドに、計算に必要な微分方程式を定義します。

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

- `nondirectOutputs[]` で、ステート変数 `x_local` および `v_local` を出力 x および v に渡します。

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

- `init[]` メソッドでは、ユーザーが `resetContinuousState()` 関数を使用して、x と v に現実的な初期値を与えることができます。

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

- **Generate Code** ボタンをクリックします。

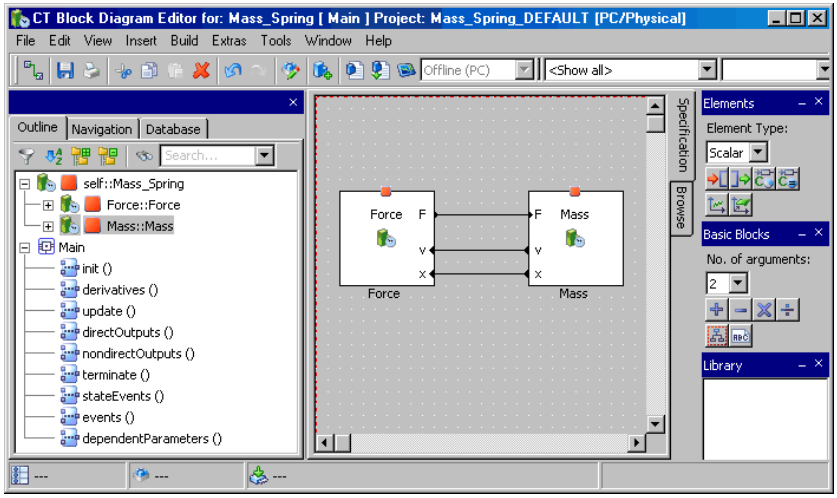
  CT ブロック `Mass` がコンパイルされます。

ブロックダイアグラムエディタ（BDE：**B**lock **D**iagram **E**ditor）を使用して、2 つの基本 CT ブロックを結合して 1 つの CT 構造ブロックにします。

2 つの基本 CT ブロックを結合する：

- コンポーネントマネージャの `Lesson6` フォルダを選択し、**Insert → Continuous Time Block → Block Diagram** を選択して新しいブロック `Mass_Spring` を作成します。
- 新しいブロックをダブルクリックしてブロックダイアグラムエディタ（BDE）を開きます。
- コンポーネントマネージャから `Mass` および `Force` ブロックを 1 つずつドラッグして BDE ウィンドウにドロップし、`Mass_Spring` 内に組み込みます。

- 対応する入力と出力とを接続します。



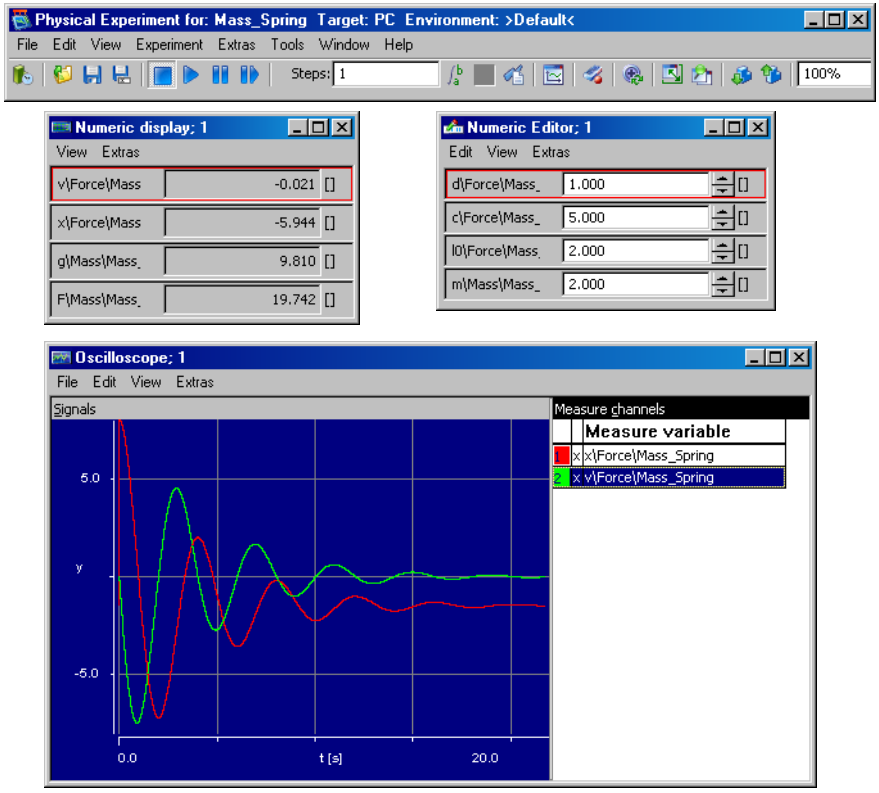
注記

CT 基本ブロックの 1 つをダブルクリックすると、そのブロックを編集できるようになります。ブロックに修正を加えるとライブラリ全体、つまり、その基本ブロックを使用するすべての構造ブロックに影響します。

- **Build → Experiment** を選択します。

  CT ブロックがコンパイルされ、実験環境が起動されます。

- 以下のように数値エディタとオシロスコープを設定します。



- オシロスコープ内のチャンネルのスケールを、x は -10 ～ 0、v は -8 ～ +8 に調整します。

4.6.3 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- モデルを作成してプロセスをシミュレートする
- ESDL エディタを使用して、直接および間接通過の CT ブロックを作成する
- ブロックダイアグラムエディタを使用して、複数の CT ブロックを結合する
- 物理実験を行う

4.7 プロセスモデル

前のレッスンでの CT ブロックの紹介に続いて、それらの CT ブロックを使用したコントローラのテストを行います。ASCET では、制御対象となる物理プロセスのモデルを開発し、閉制御ループによってコントローラモデルの実験を行うことができます。これにより、実際の車両を使用する前にコントローラのテストを入念に行っておくことができます。

ここでは、モータの物理プロセスを扱います。この物理プロセスモデルはエンジン回転速度センサの値 `U_n` を返します。そしてコントローラはこの値を処理し、値 `air_nominal` を返します。コントローラの出力値によりエンジンのスロットル位置が決まり、さらにこのスロットル位置が回転速度に影響します。

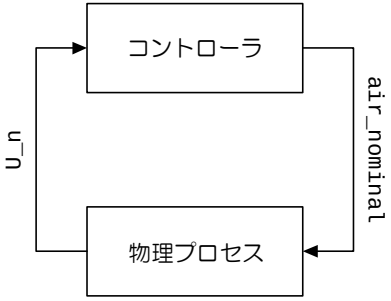


図 4-1 閉ループ実験

このプロセスモデルには CT ブロックを使用します。連続系コンポーネントはプロセスモデルに特に適していて、モデルのベースとなるのは、PT2 系をモデリングする以下の微分方程式です。

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

式 4-1 PT2 系

この方程式においては、パラメータ T、D、および K に適切な値を設定する必要があります。

4.7.1 プロセスモデルを定義する

連続系コンポーネントの作成方法は、他のコンポーネントの作成方法とは異なります。連続系コンポーネントには入力と出力があり、これらは引数と戻り値に相当します。主な違いは、連続系ブロックは複数の入力と出力を伴うことができ、それらが特定のメソッドと結びついていないことです。各連続系ブロック内には、あらかじめ一連のメソッドが固定的に定義されていて、これをユーザーが変更することはできません。

ここでは、ESDL コードを使用します。ESDL コードの構文は C++ や Java に似ています。オブジェクトのメソッドは、「オブジェクト名 . メソッド名 (引数); 」の形式で呼び出されます。微分に用いるメソッドは `ddt()` と呼ばれます。たとえば、方程式 $sp = \dot{s}$ は、ESDL では `s.ddt(sp);` という文で表記されます。

連続系コンポーネントを作成する：

- コンポーネントマネージャでフォルダ `Tutorial¥Lesson7` を作成します。
- 連続系ブロックを追加するために、**Insert → Continuous Time Block → ESDL** を選択します。
- 新しいコンポーネントの名前を `ProcModel` にします。
- **Edit → Open Component** を選択して、ESDL エディタを開きます。

  ここではもちろん外部テキストエディタも使用できます。使用方法は、チュートリアルの最初の部分に説明されています。

プロセスモデルを編集するには、まず必要なエレメントを追加してから、メソッド `derivatives()` および `nondirectOutputs()` を編集します。

プロセスモデルを編集する：

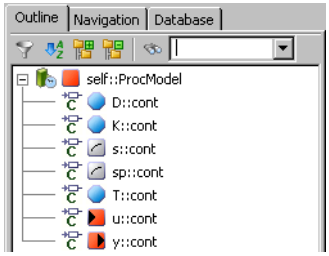
- ESDLエディタの**Continuous State**ボタンを使用して、2 つの連続ステートを作成します。
- これらのステートの名前を s および sp にします。

- 入力 u を作成します。
- 出力 y を作成します。

  どのエレメントも `cont` 型です。

- パラメータ D を作成します。
- パラメータ K および T を作成します。

  このプロセスモデルの“Outline”タブは、以下のようになります。



- 各パラメータを以下のように設定します。

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

- “Online” タブで `derivatives` メソッドを選択し、そのコードを以下のように編集します。

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

この図は内部テキストエディタを使用した例です。

注記

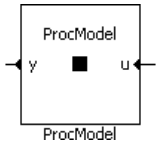
微分方程式の解法については、ASCET オンラインヘルプの連続系ブロックの記述法についてのトピックを参照してください。

- “Outline” タブから `nondirectOutputs` メソッドを選択し、以下のテキストを入力します。

```
y = s;
```

- レイアウトエディタでレイアウトを調整します。

プロセスモデルの場合、出力を左側、入力を右側に配置するのがよいでしょう。



- Edit → Save を選択します。

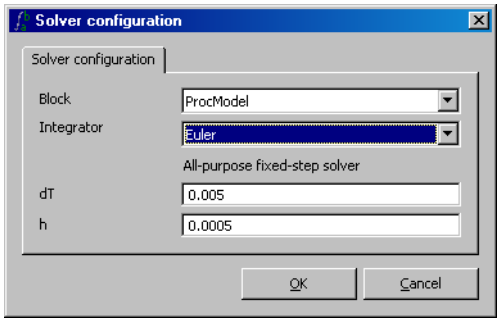
- コンポーネントマネージャの **Save** ボタンをクリックして、プロセスモデルを保存します。

これで、新しいモデルを用いて実験を開始する準備ができました。

モデルの実験を行う：

- ESDL エディタで **Build** → **Experiment** を選択して、実験環境を開きます。

- **Open CT Solver** ボタンをクリックして、“Solver Configuration” ダイアログボックスを開きます。
  現在のコンフィギュレーションが表示されます。



- **OK** をクリックしてデフォルト設定を有効にします。
- データジェネレータを開き、入力 u のチャンネルを作成します。
- 以下の値をチャンネル u に設定します。

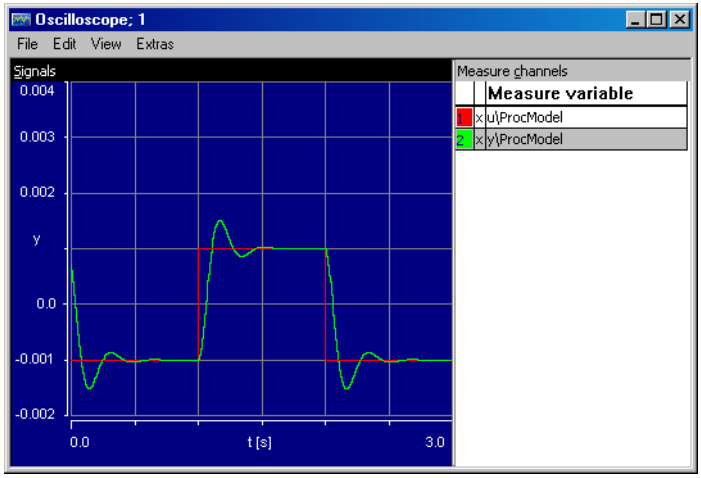
Mode :	pulse
Frequency :	0.5Hz
Phase :	0.0s
Offset :	-0.5
Amplitude :	1.0

- チャンネル u および y のオシロスコープウィンドウを開きます。
- オシロスコープの各測定チャンネルを、以下のように設定します。

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

- **Save Environment** ボタンをクリックします。

- 実験を開始します。
  出力は下図のようになるはずです。



4.7.2 プロセスモデルを統合する

閉制御ループを作成するために、前に作成したコントローラプロジェクトにプロセスモデルを統合します。これまでのレッスンと同様に、モジュールを内包させ、オペレーティングシステムをセットアップしてグローバルエレメントを結び付けるという作業が必要です。

注記

作業を単純にするために、プロセスモデルを同じプロジェクトに追加します。この方法は、閉ループシミュレーションの早い段階でのテストにしばしば役立ちますが、プロセスモデルは組み込みシステムの構成要素ではないため、通常は、ネットワーク経由で各プロジェクト向けに配布されます。

プロセスモデルを内包させる：

- コンポーネントマネージャから `ControllerTest` 用のプロジェクトエディタを開きます。
  プロジェクトエディタで、コンポーネント `ProcModel` を“Outline”タブに追加します。
- プロジェクトエディタの“OS”タブを選択し、CT タスクのスケジューリングを定義します。

- タスク `simulate_CT1` を選択し、“Period” フィールドの値を 0.01（秒）に設定します。

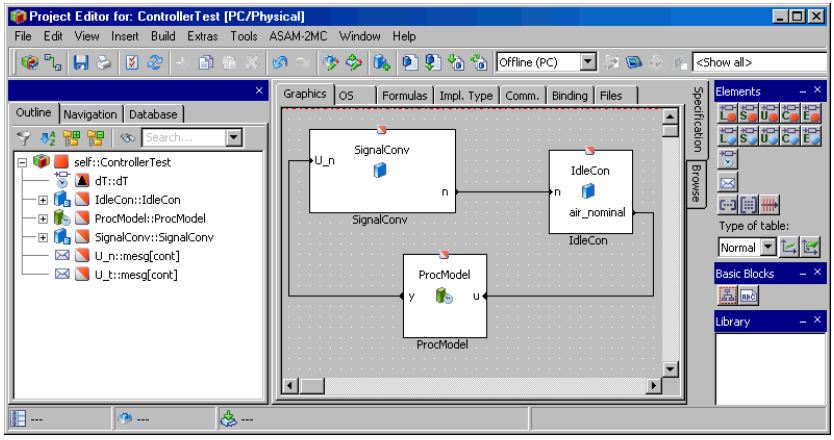
  コントローラとプロセスモデルが、共に同じ時間間隔で実行されます。

連続系ブロックとモジュールの結び付けを自動的に行うことはできないので、ブロックダイアグラムで明示的に接続する必要があります。

モジュールと連続系ブロックの結び付きを調整する：

- “Graphics” タブをクリックします。
- “Outline” タブから、3 つのコンポーネントをドラッグし、描画エリアにドロップします。
- モジュールのメッセージを、CT ブロックの対応する入出力に接続します。

  この例では、`ProcModel` の出力 `y` にグローバルメッセージ `U_n` を接続し、`ProcModel` の入力 `u` にグローバルメッセージ `air_nominal` をそれぞれ接続します。
- 各コンポーネントを右クリックして **Unconnected Ports** を選択し、未接続のポートがダイアグラムに表示されないようにします。



モジュール間通信のためのメッセージの結び付けは、自動的に行われます。同じ名前のメッセージ同士が結びつきます。

これでプロジェクトが完成し、実験を行えるようになりました。今度はオンライン実験を行うので、ASCET-RP がインストールされていることと、リアルタイムターゲット（ES1000 など）が接続されていることが必要となります。これらの条件が揃っていない場合は、これまでどおりオフライン実験を行ってください。

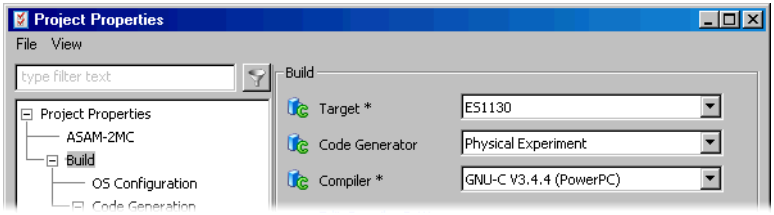
注記

オフライン実験を行う場合には、グローバルメッセージ U_n をデータジェネレータから必ず削除してください。

プロジェクトをオンライン実験用にセットアップする：

- **Project Properties** ボタンをクリックします。
- “Project Properties” ダイアログボックスの “Build” ページで、ターゲットに ES1130、コンパイラに GNU-C V3.4.4 (PowerPC) を選択します。

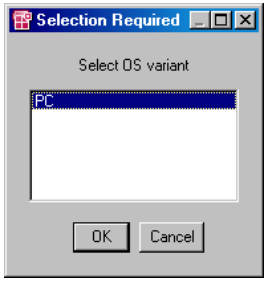
  これらのオプション設定により、ハードウェアとそれに対応するコード生成用コンパイラが指定されます。



- **OK** をクリックして、ダイアログボックスを閉じます。

  Reconnect to Experiment of selected Experiment Target および **Select Hardware** ボタンが有効になります。
- “OS” タブをクリックして、オペレーティングシステムエディタを開きます。

- 前に作成したスケジューリング設定をコピーするために、**Operating System → Copy From Target** を選択します。



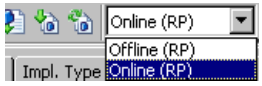
- “Selection Required” ダイアログから **PC** を選択して **OK** をクリックします。

  これで、新しいターゲット用のプロジェクトに、以前の PC でのオフラインシミュレーション用に定義したものと同じスケジューリング設定が定義されました。

オンライン実験には、オフライン実験とは異なる点がいくつかあります。オンライン実験ではイベントジェネレータやデータジェネレータは用いません。イベントジェネレータは、オンライン実験用に生成されるオペレーティングシステムタスクのスケジューリングをオフライン実験でシミュレートするためのものです。

オンライン実験では、実験コードの実行（つまり OS の起動）と測定とを別々に開始します。このためツールバーにもそれぞれ専用のボタンが用意されています。これは、測定によって実験のリアルタイム動作が影響を受ける場合があるので、場合によっては測定を行わずに実験を行う必要があるためです。

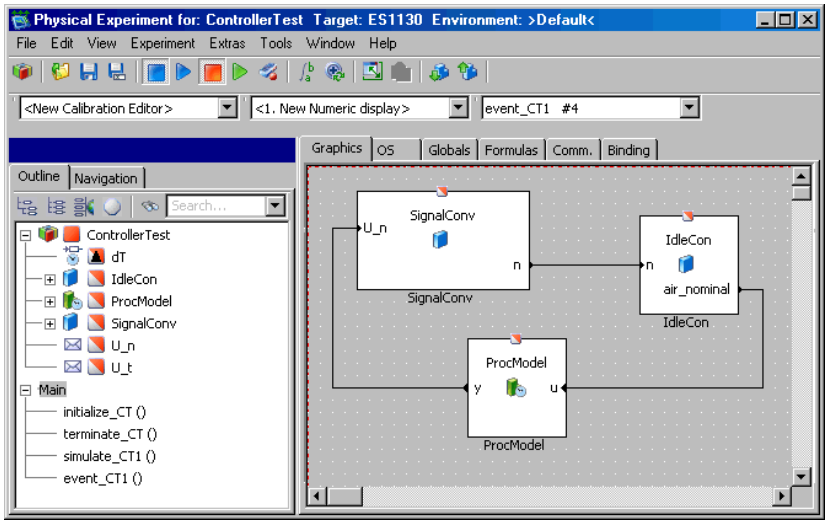
プロジェクトの実験をオンラインで行う：



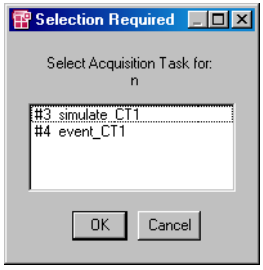
- “Experiment Target” コンボボックスから `Online (RP)` を選択します。

  `Offline (RP)` はターゲット上でオフライン実験を行うためのものです。

- **Build → Experiment** ボタンをクリックします。
  実験用のコードが生成され、前に定義した環境と同じ環境で実験が開きます。



プロジェクトに複数のタスクが含まれている場合、各測定変数の値をどのタスクで取得するかを尋ねるダイアログボックスが開きます。



- “Selection Required” ダイアログボックスで、#3 simulate_CT1 というタスクを選択して **OK** をクリックします。
- `n` および `n_nominal` を既存のオシロスコープに設定し、それらの値の範囲を 0 ～ 2000 にします。
- 変数 `n_nominal`、`Ki` `Kp` 用の数値エディタを開きます。

- **Start OS** ボタンをクリックしてから、**Start Measurement** ボタンをクリックします。

  実験が開始され、結果がオシロスコープに表示されます。`n` の値が急速に `n_nominal` に近づき、その状態が維持されます。
- `n_nominal` を数値エディタで修正します。

  `n_nominal` の値を変えると、それに応じて `n` の値が変わるはずです。
- パラメータ `Ki` および `Kp` を調整して、制御ループの動作を最適化することができます。

4.7.3 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- 連続系ブロックを作成し、定義する
- 連続系ブロックの実験を行う
- 連続系ブロックをプロジェクトに統合する
- 変数の結びつきを定義する
- 異なるターゲットに切り替える
- プロジェクトの実験をオンラインで行う

4.8 ステートマシン

ステートマシンは、有限数の明確なステート間を遷移するシステムをモデリングする際に有効な手段です。ASCET は、コンポーネントをステートマシンとして定義するための強力な機能を備えています。このレッスンでは、温度によるアイドリングエンジンの目標回転数の変化を表す、単純なステートマシンを定義しテストします。それから、そのステートマシンをプロジェクトに統合します。そして次のレッスンで階層ステートマシンを構築します。

エンジンが冷えている時には、高速アイドリングさせて回転を維持する必要があります。エンジンが暖まったら、燃料消費を減らすためにアイドリングの回転速度を下げます。このためのステートマシンは「エンジンが冷えている」および「エンジンが暖まっている」という 2 つのステートを持つ 2 段階制御を行います。

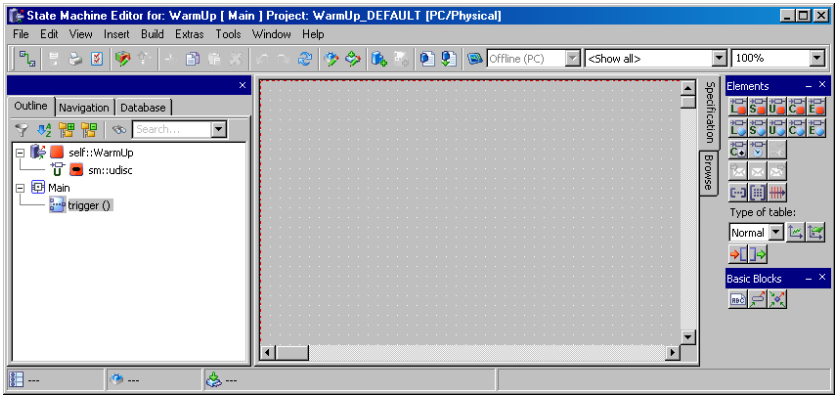
4.8.1 ステートマシンを定義する

ステートマシンはステート図と、アクションやコンディションの定義から構成されます。アクションやコンディションの定義により、さまざまなステートで、およびステートからステートへの遷移時に何を行うかを指定できます。これらはブロックダイアグラムと ESDL コードのどちらででも定義できます。

ステートマシンはブロックダイアグラムエディタで定義します。これ以外の方法としては、それぞれのステートやトランジションごとに開くテキストエディタにESDL コードを直接書き込む方法があります（注記：ここでは ESDL エディタは開きません）。ステートマシンには、他のコンポーネントとのデータ交換に使用される入力と出力があります。

ステートマシンを作成する：

- コンポーネントマネージャで、フォルダ `Tutorial¥Lesson8` を作成します。
- **Insert → State Machine** を選択するか、または **Statemachine** ボタンをクリックして新しいステートマシンを作成します。
- 作成したステートマシンの名前を `WarmUp` にします。
- “1_Database”リスト内の新しいステートマシンの名前をダブルクリックして、ステートマシンエディタを開きます。



ステートマシンを作成する場合には、まずステートダイアグラムを定義してから、ステートやステートトランジションに関連付けるさまざまなアクションやコンディションを定義します。

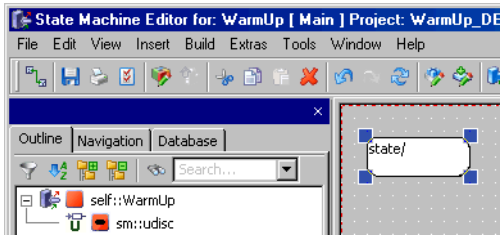
エンジンをコントロールするこのステートマシンには、「エンジンが冷えている」および「エンジンが暖まっている」という 2 つのステートがあります。

ステートダイアグラムを定義する：

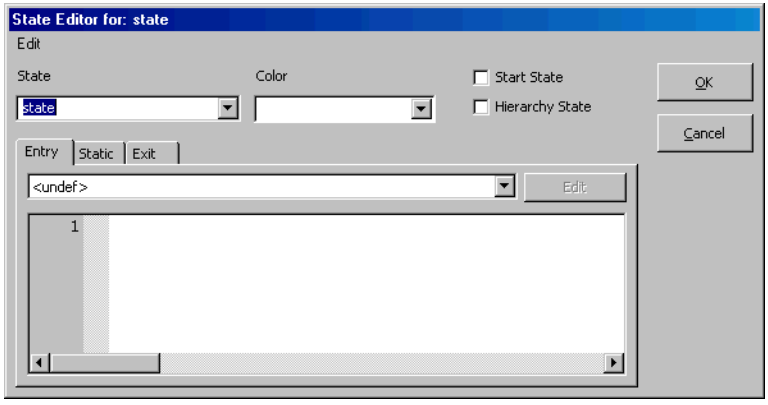
- **State** ボタンをクリックして、カーソルにステートアイテムをロードします。

- 描画エリア内の、ステートを配置したい位置をクリックします。

  クリックした位置にステートシンボルが表示されます。



- ステートシンボルをもう 1 つ作成し、描画エリア内の最初のステートの下に配置します。
- 先に作成したステートシンボルを右クリックし、ショートカットメニューから **Edit State** を選択して、ステートエディタを開きます。



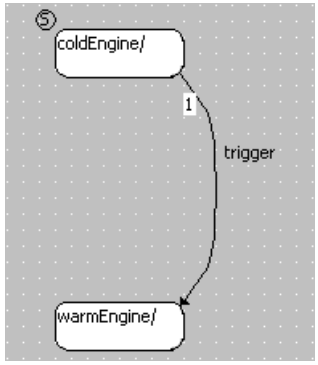
- “State” フィールドに `coldEngine` というステート名を入力します。
- **Start State** オプションをオンにして、このステートを、最初にステートマシンが起動されたときのステート（「開始ステート」）にします。

  ステートマシンには必ず開始ステートを 1 つ定義する必要があります。

- **OK** をクリックして、ステートエディタを閉じます。

  ステートの名前が、ステートシンボル内に表示されます。
- もう 1 つのステートシンボルの名前を `warmEngine` にします。
- 描画エリア内のシンボルが表示されていない位置を右クリックして、接続モードに切り替えます。
- `coldEngine` ステートシンボルの右半分をクリックして接続を開始し、次に `warmEngine` ステートシンボルの右半分をクリックして、両ステートを接続します。

  両ステートシンボルを結ぶ線が引かれます。この線は下向きの矢印になっていて、ステート間で起こり得るトランジションを表します。



- 下のステートシンボルの左半分から上のステートシンボルの左半分に向けて、もう 1 つのトランジションを作成します。
- **File → Save** を選択し、作成したステートマシンを保存します。
- コンポーネントマネージャで **File → Save Database** を選択してデータベースを保存します。

ステートマシン構築の際に次に行うのは、そのインターフェースの定義です。温度値用の入力と、回転数用の出力が必要です。さらに、高温と低温、および回転数（rpm）を定義するパラメータが必要です。

ステートマシンのインターフェースを定義する：

- **Input** ボタンをクリックして、入力を作成します。
- その入力の名前を `t` にします。

- **Output** ボタンをクリックして、出力を作成します。
- その出力の名前を `n_nominal` にします。

- **Continuous Parameter** ボタンで4つのパラメータを作成します。
- 各パラメータの名前と値を以下のように設定します。

  `t_up` = 70
  `t_down` = 60
  `n_cold` = 900
  `n_warm` = 600

次に、両ステートと、その間のトランジションについて、アクションとコンディションを定義します。各ステートに定義できるアクションは以下の 3 種類です。

- そのステートに入るたびに実行される Entry アクション
- そのステートを離れるたびに実行される Exit アクション
- そのステート内に留まって、状態が変化しない時に実行される Static アクション

同様に、各トランジションには、トリガイベント、コンディション、優先度およびアクションを定義することができます。トリガ名とコンディション名は、トランジションの隣に表示されます。ステートマシンが作成されるとトリガが 1 つ、自動的に作成されます。

アクションとコンディションは、普通のダイアグラムか ESDL コードで定義されます。この例では、ESDL コードを使用します。

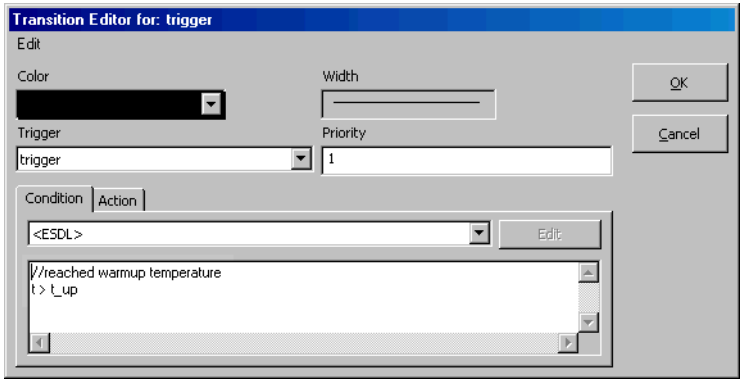
トリガのアクションとコンディションを定義する：

- `coldEngine` ステートから `warmEngine` に向かって接続されているトランジションを右クリックします。
- ショートカットメニューから **Edit Transition** を選択して、トランジションエディタを開きます。

  coldEngine から warmEngine へ遷移するための条件は、「実際の温度値 `t` が `t_up` より大きい」ということです。

- “Condition” タブのコンボボックスから <ESDL> を選択します。

  このコンボボックスではあらかじめ定義されているオプション設定を変更することができます。
- コンディションのコードペインに、下図のコードを入力します。



注記

トランジションエディタでは、コンディションの終わりにセミコロンを付けません。これは、通常の ESDL コードにおいてコンディションを括弧で囲んで記述する場合も同じです。

  このコンディションの評価結果が `true` なら、エンジンのアイドリング速度が `n_warm` になります。

  このコードはステートマシンダイアグラムに表示されます。この例では、トランジションのコンディションに別名（1 行目のコメント）を付けたので、それがダイアグラムに表示されます。
- アクション用にも <ESDL> を選択し、以下のコードを入力します。

```
n_nominal = n_warm;
```
- **OK** をクリックして、トランジションエディタを閉じます。

  ダイアグラムには、ステートマシンのコンディションとアクションが表示されます。

- `warmEngine` から `coldEngine` へのトランジションを編集するために、別のエディタを開きます。
- コンディションに <ESDL> を選択し、以下のコードを入力します。

  ```
  t < t_down
  ```

  このコンディションには別名（コメント）を付けなかったので、ダイアグラムにはこのコンディションのコード全体が表示されます。
- アクションに <ESDL> を選択し、下図のコードを入力します。

  ```
  n_nominal = n_cold;
  ```

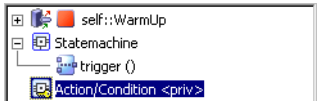
- エディタを閉じ、**File → Save** を選択します。

アクションとコンディションは、ESDL の代わりに BDE で定義することもできます。ただしその際は、先にアクションとコンディション用のブロックダイアグラムを作成しておく必要があります。

アクションとコンディションのブロックダイアグラムを作成する：

- ステートマシンエディタで **Insert → Diagram → Actions/Conditions BDE** を選択します。

  “Outline” タブに `ActionCondition_BDE` という名前のブロックダイアグラムが作成されます。
- このデフォルト名を確定します。

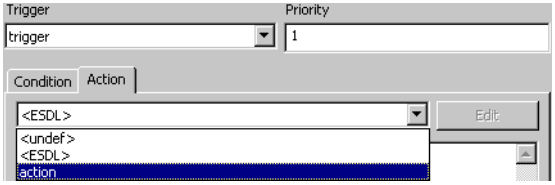


これでアクションとコンディションを定義することができるようになりました。

アクションとコンディションをブロックダイアグラムで定義する：

- “Diagrams” フィールドの `ActionCondition` ダイアグラムをクリックします。
- **Insert → Action** と **Insert → Condition** で、新しいアクションとコンディションを作成します。

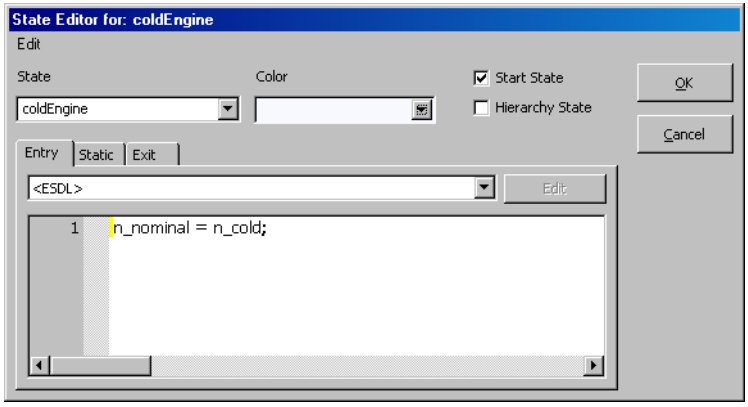
作成したアクションとコンディションは、トランジションエディタ内の各タブのコンボボックスで選択することができます。**Edit** ボタンをクリックすれば、直接 BDE を開いてグラフィック編集を行えます。



この時点で、出力 `n_nominal` の初期値はまだ定義されていません。パラメータとは異なり、この値を設定することはできないので、代わりに開始ステートである `coldEngine` のアクションを定義します。ただし開始ステートの Entry アクションは最初にステートマシンが起動された時には実行されないため、初期値の定義は Static アクション内で行う必要があります。

Entry アクションを定義する：

- 開始ステート `coldEngine` を右クリックします。
- ショートカットメニューから **Edit State** を選択して、ステートエディタを開きます。



- “Entry” タブのコンボボックスから <ESDL> を選択して、Entry アクションを定義します。

  ここでの設定は、ASCET オプションダイアログボックスの “Defaults” ページで設定されているデフォルトの記述形式よりも優先されます。
- コードペインに `n_nominal = n_cold;` と入力して、`n_nominal` の初期値を 900 にします。

- **OK** をクリックして、ステートエディタを閉じます。

これで、ステートマシンの定義が終わりました。この実験を開始する前に、ここであらかじめステートマシンの動作を理解しておいてください。

4.8.2 ステートマシンの動作

一般的に、標準コンポーネントの動作（機能）はそのグラフィック記述を見れば簡単に理解できますが、ステートマシンの場合はその動作が一見ではわかりにくい場合があります。本項では、前項の例を用いて、ステートマシンの原理について説明します。ステートマシンとその機能についての詳細は、ASCET オンラインヘルプのステートマシンエディタに関するトピックを参照してください。

ステートマシンの各ステートにはステート名、Entry アクション、Static アクション、および Exit アクションが定義されています。さらに、各ステートは他のステートからとの間にトランジション（状態遷移を表す接続線）を持ち、各トランジションには優先度、トリガ、アクションおよびコンディションが定義されています。

各ステートマシンにはそれぞれ 1 つの開始ステートが必要です。ステートマシンが初めて起動されると、開始ステートが有効になり、開始ステートのアクション（つまりステートマシンの初期処理）が実行された後、開始ステートから他のステートへのトランジションを発生させるコンディションが調べられます。この例では、このようなトランジションは 1 つしかなく、そのコンディションは `t > t_up` です。ここでは、入力値が `t_up` パラメータの値より大きいかどうかが調べられます。もしそうならコンディションは真（true）なので、トランジションが発生します。

パラメータ `t_up` および `t_down` は温度を規定します。エンジンがこれらの温度にならないと、目標回転速度を変更することができません。今の例では、エンジン温度が 70 度を超えたら、回転速度を 600rpm に下げることができます。その後、エンジン温度が 60 度を下回ると、目標速度を 900rpm にリセットしなければなりません。

トランジションが発生すると必ず、そのトランジションに定義されているトランジションアクションが実行されます。今の例では、`coldEngine` から `warmEngine` へのトランジションが発生すると実行されるトランジションアクション `n_nominal = n_warm` により、変数 `n_nominal` の値が 600 になります。反対に、トランジションアクション `n_nominal = n_cold` は `n_nominal` の値を 900 にします。トランジションが発生すると、離れようとするステートの Exit アクションと、入ろうとするステートの Entry アクションも実行されます。この例では、これらのアクションは定義されていないので、何も行われません。

ステートマシンが第 2 のステートに入ると、第 2 のステートから第 1 のステートへのコンディションが満たされるまでは、第 2 のステートに留まります。ステートマシンが 1 つのステートに留まっている間は、ステートマシンが起動されるたびに Static アクションが実行されます。ステートマシンを起動させるのは必ず外部のイベントで、1 回の起動によりステートマシンの 1 回のパスが開始されます。

ステートマシンの 1 回のパスは、まず現在のステートから他のステートへのトランジションについてのすべてのコンディションを調べることから始まります。コンディションは、その優先度の順に調べられます。あるコンディションが true なら、それに対応するトランジションが発生し、Exit アクション、トランジションアクションおよび Entry アクションが実行されます。最初に調べたコンディションが true だった場合、同じステートから他のステートへの他のトランジション（最初に調べたコンディションよりも優先度が低いコンディション）は調べられません。どのコンディションも true でない場合には、現在のステートのままになり、そのステートに留まるパスが実行されるたびに Static アクションが 1 回実行されます。

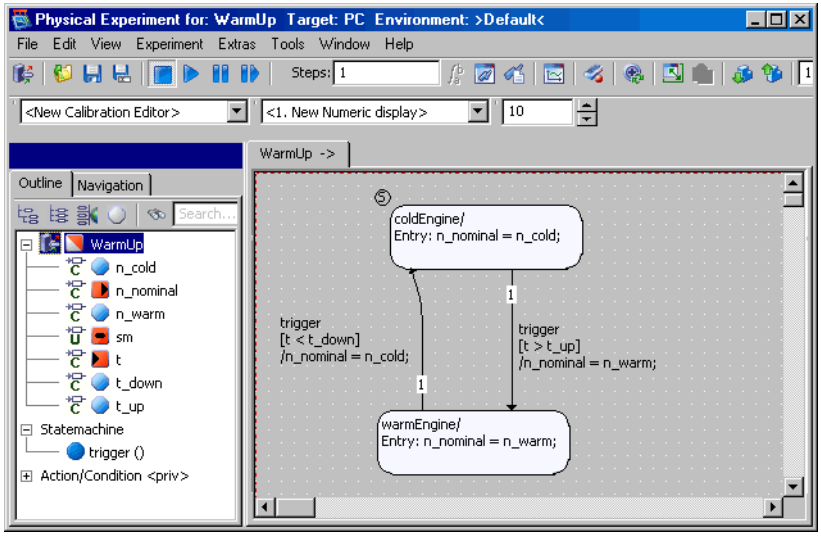
第 2 のトランジションのコンディションが true になると、つまり、入力値がしきい値よりも小さくなると、ステートマシンは第 1 のステートに戻ります。それから、入力値が再びしきい値より大きくなるまでは、そのステートのままになります（第 1 のステートには Static アクションが定義されていないので、何も行われません）。

4.8.3 ステートマシンの実験を行う

実験には、ステートマシンの場合も他のタイプのコンポーネントの場合と同様の機能がありますが、それ以外に、ステートマシンの実験を行う場合にだけ使用できるアニメーションという機能があります。これは、実験実行中にステートマシンダイアグラム内のカレントステート（現在アクティブになっているステート）を強調表示する機能です。

ステートマシンの実験を行う：

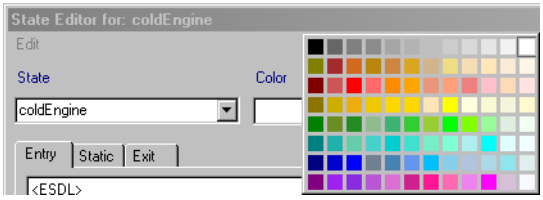
- ステートマシンエディタで **Build → Experiment** を選択して、実験環境を開きます。



- ステートの 1 つを右クリックし、ショートカットメニューから **Animate States** を選択します。
- `trigger` イベントをイネーブルにします。
- データジェネレータで、変数 `t` のチャンネルを作成します。
- このチャンネルに周波数 1Hz、オフセット 70、振幅 20 の正弦波を割り当てます。
- `t` および `n_nominal` 用のオシロスコープウィンドウを開きます。

- **Start Offline Experiment** ボタンをクリックして、ステートマシンの実験を行います。

- 個々のステートの色を変えて表示を見やすくするために、**Exit to Component** ボタンで実験環境を終了し、ステートエディタを開きます。
- “Color” コンボボックスで色を選択します。



- 再度実験を行います。

実験が実行されると、`n_nominal` の値は、正弦波が対応する温度しきい値を上回ったり下回ったりするたびに変化します。適合システムを使用してしきい値を変更し、それによる出力の変化を調べることができます。また、ステートダイアグラムにおいては、カレントステートが強調表示されます。

4.8.4 ステートマシンをコントローラに統合する

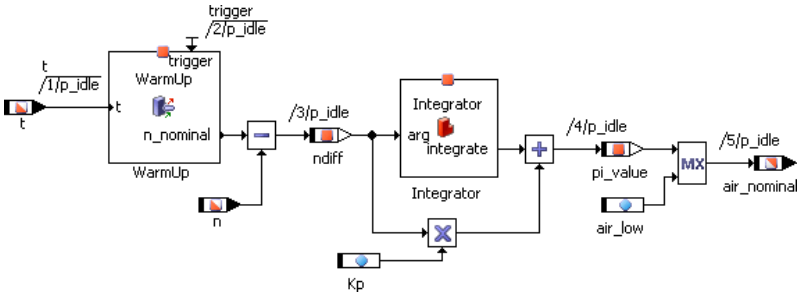
ASCET の他のすべてのコンポーネントと同様に、ステートマシンもやはり、別の任意のタイプのコンポーネントを構成する要素として用いることができます。では、ステートマシンをコントローラモジュールに統合し、回転速度をエンジン温度に合わせて調整できるようにしましょう。

ステートマシンを統合する：

- コンポーネントマネージャから、ブロックダイアグラムエディタでモジュール `Lesson4¥IdleCon` を開きます。

- ダイアグラムと“Outline”タブから、パラメータ `n_nominal` を削除します。

  ブロックダイアグラムにおいて、このパラメータの代わりにステートマシンを使用します。
- **Insert → Component** を選択し、コントローラの“Outline”タブにステートマシンを追加します。
- 受信メッセージを作成し、その名前を `t` にします。
- 削除した変数の代わりにコンポーネント `WarmUp` の出力を減算の演算子に接続し、同コンポーネントの入力を受信メッセージ `t` に接続します。
- ダイアグラムを下図のように調整します。すべてのアイテムが正しい順序で接続されるように、シーケンシングを調整します。



- ダイアグラムを保存し、コンポーネントマネージャで **Save** ボタンをクリックします。

修正後のコントローラをプロジェクト内で機能させるために、プロジェクトにも変更を加える必要があります。今度は、まだ使用されていなかった温度センサも統合します。

プロジェクトを変更する：

- プロジェクト`ControllerTest`用のプロジェクトエディタを開きます。
- “OS”タブに切り替えます。
- プロセス`t_sampling`をタスク`Task10ms`に割り当てます。
- **Task → Move Up** を使用して、プロセス `t_sampling` をタスクの先頭に配置します。
- **Build → Experiment** を選択します。

- 値 U_t 用に、もう 1 つのスカラ適合ウィンドウを開きます。
- 変数 t をオシロスコープに追加します。
- **Start OS** ボタンをクリックします。
- **Start Measurement** ボタンをクリックします。
- 値 U_t を調整し、その影響を調べます。

  t の値が限界値（70 度）を超えると、ステートマシンは n の目標値を低い方の値 600 に切り替えます。温度が 60 度を下回ると（U_t を調整することによりシミュレートされます）、n の目標値は元の値 900 に戻ります。

4.8.5 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- ステートダイアグラムを作成する
- コンディション、アクション、およびトリガを作成し、ステートマシンに割り当てる
- ステートマシンの実験を行う
- ステートマシンを他のコンポーネントに統合する

4.9 階層ステートマシン

前のレッスンでステートマシンの機能について理解できたので、今度はやや複雑なシステムを構築してみます。このレッスンでは階層ステートマシンについて集中的に学習し、また、タイマなど ASCET に付属しているシステムライブラリとコンポーネントの使用法を学びます。

ASCET では、閉階層と開階層を用いてステートマシンを構築することができます。開階層の場合はその中に含まれるサブステートも図示され、閉階層の場合それらは省略されます。

交通信号制御システムを構築して、パラメータで指定できるタイミングを使用して交通信号の各フェーズへの移り変わりを実現してみましょう。この交通信号にはエラーステータスもあり、そのときには信号が点滅します。

4.9.1 ステートマシンを定義する

まず、必要なライブラリをインポートして準備します。

システムライブラリをインポートする：

- コンポーネントマネージャで **Import** ボタンをクリックします。

  “Select Import File” ダイアログボックスが開きます。

- “Import” フィールドの  ボタンで、ASCET のインストールディレクトリ下のエクスポートディレクトリ（例：`C:¥etas¥ASCET6.0¥export`）に格納されている `SystemLibETAS.exp` というファイルを選択します。

  OK ボタンが有効になります。
- **OK** をクリックしてインポートを開始します。

  “Import” ダイアログボックスが開き、選択されたファイルに含まれるすべてのオブジェクトが一覧表示されます。
- **OK** ボタンをクリックします。

  ファイルがインポートされます。この処理には数分かかります。インポートが終了すると、インポートされたアイテムの一覧が “Import Items” ウィンドウに表示されます。

次に、交通信号を制御するために必要な 2 つのメインステート（`NormalMode` と `ErrorMode`）を定義します。

ステートマシンを作成する：

- コンポーネントマネージャで、フォルダ `Tutorial¥Lesson9` を作成します。
- **Insert → State Machine** を選択して、新しいステートマシンを作成し、その名前を `Light` にします。
- **Edit → Open Component** ボタンをクリックして、ステートマシンエディタを開きます。

  ここから、交通信号を制御するステートマシンを定義します。

- 2 つの主要ステート `ErrorMode` および `NormalMode` を作成します。

次に、システムライブラリのタイマをプロジェクトに追加します。

タイマオブジェクトを追加する：

- **Insert → Component** を選択します。
- “Select item” ダイアログボックスで、`SystemLib_ETAS` ライブラリの `Counter_Timer` フォルダにあるタイマオブジェクト、`Timer` を選択します。

- **OK** で確定します。

  ステートマシンの“Outline”タブにオブジェクト `Timer` が追加されます。

ステートダイアグラムを定義する：

- 必要なデータエレメントを以下のように定義します。
  - `Logic` 型の入力 `error`
  - 信号の 3 色を表す、`Logic` 型の 3 つの出力（`yellow`、`green`、`red`）
  - 信号のさまざまなフェーズを表す、`Continuous` 型の 4 つのパラメータ（`BlinkTime`、`YellowTime`、`GreenTime`、`RedTime`）

  依存パラメータについて理解を深めるために、緑色のフェーズの値だけを定義し、他のパラメータには緑色のフェーズの値に依存する値が設定されるようにします。

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

- 次に、個々のパラメータの計算式と依存関係を定義します。
- このためには、パラメータ `RedTime`、`YellowTime` および `BlinkTime` について、プロパティエディタの“Dependency”の下にある **Dependent** オプションをオンを選択します。

  プロパティエディタを起動するには、パラメータをダブルクリックするか、または **Edit** ショートカットメニューを使用します。

- **Formula** ボタンをクリックしてフォーミュラエディタを開きます。
- フォーミュラエディタを使用して、それぞれの依存パラメータについての計算式を定義します。最初に仮パラメータ x を作成し、それからフォーミュラペインに計算式を入力します。

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

- フォーミュラエディタとプロパティエディタを閉じます。

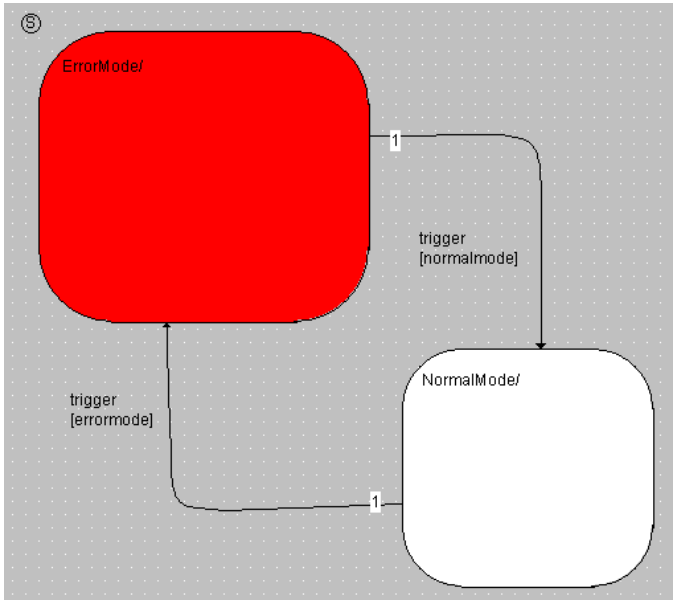
- ショートカットメニュー **Edit Data** でディペンデンシーエディタを開きます。
- それぞれの依存パラメータの仮パラメータ x に対応するモデルパラメータを割り当てます。

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

- データエレメントに有意義な値を与えます（例、`GreenTime = 5`）。
- ステート `ErrorMode` ステートエディタで開きます。

  ステートエディタを開くには、編集するステートをダブルクリックするか、**Edit State** ショートカットメニューを使用します。
- このステートを開始ステートとして定義し、色を赤にします。
- 両方のステートを拡大表示し、階層を挿入できるようにします。
- 2 つのステートの間にトランジションを作成します。
- ショートカットメニュー **Edit Transition** で、またはトランジションを示す曲線をダブルクリックして、トランジションエディタを開きます。
- トランジションダイアログボックスにコンディションを入力して、2 つのステートの間のトランジションを定義します。入力 `error` が `false`（つまり、エラーが発生していない）なら正常ス

テート NormalMode になり、エラーの場合には ErrorMode になるように、ESDL でコンディションを記述します。



- **File → Save** を選択し、作業内容を保存します。
- コンポーネントマネージャで **File → Save Database** を選択し、データベースを保存します。
- これらのメインステートについて、実験を行ってみてください。

次に、交通信号制御システムに必要なサブステートを定義します。ここではまずエラーモード（ErrorMode ステート）における処理を定義します。このステートでは、黄色の点滅光が出力されます。このためには、YellowOff および YellowOn という 2 つのサブステートを定義し、この両者間の切り替えをタイマで行うようにします。YellowOn ステートでは出力 Yellow が true になり、YellowOff ステートでは false になります。

エラーモード用のサブステートを定義する：

- YellowOff および YellowOn というステートを作成し、それらをステート ErrorMode 内に配置します。
- YellowOff を開始ステートとして定義し、YellowOn の色を黄色にします。

- ステート `YellowOff` の対応をステートエディタで定義します。

  ステートエディタを開くには、**Edit State** ショートカットメニューを使用するか、このステートをダブルクリックします。

- Entry アクションを入力します。“Entry” タブのコンボボックスで ESDL を選択し、以下ののコードを入力します。

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

- Static アクションを入力します。“Static” タブに以下のコードを入力します。

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

- 次に、同じ方法で、`YellowOn` ステートについて定義します。

  Entry アクション：

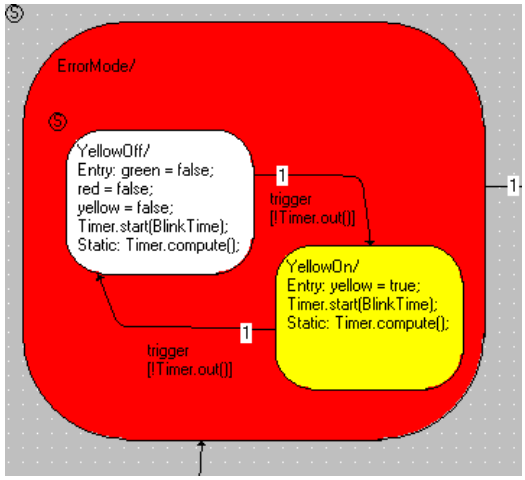
```
yellow = true;
Timer.start (BlinkTime);
```

  Static アクション：

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

- 今度は、2 つのサブステートの間のトランジションを定義します。

  状態遷移の条件は、タイマがタイムアウトすること（`Timer.out() == false`）です。



つまり、`ErrorMode` ステートは `YellowOff` ステートで開始されます。このステートの Entry アクションで、黄色の点灯信号をオフにし、パラメータで指定される点滅時間を計測するタイマを起動します。このステートの Static アクションではタイマ関数 `compute()` が毎回起動され、タイマカウンタをデクリメントします。このカウンタ値が 0 になると、タイマ関数 `out()` がコード `false` を返し、トランジションのコンディションが真になります。そして、ステート `YellowOn` では、Entry アクションにより、黄色の点灯信号 `Yellow` がオンになります。

次に、正常動作時の動作を定義します。このためには、開始ステート `AllOff` を作成して `NormalMode` ステート内に配置します。そしてその Exit アクションで各色の点灯信号を定義されたステートに設定します。では、交通信号制御システムとして、どのように処理すればよいかを考えてみましょう。

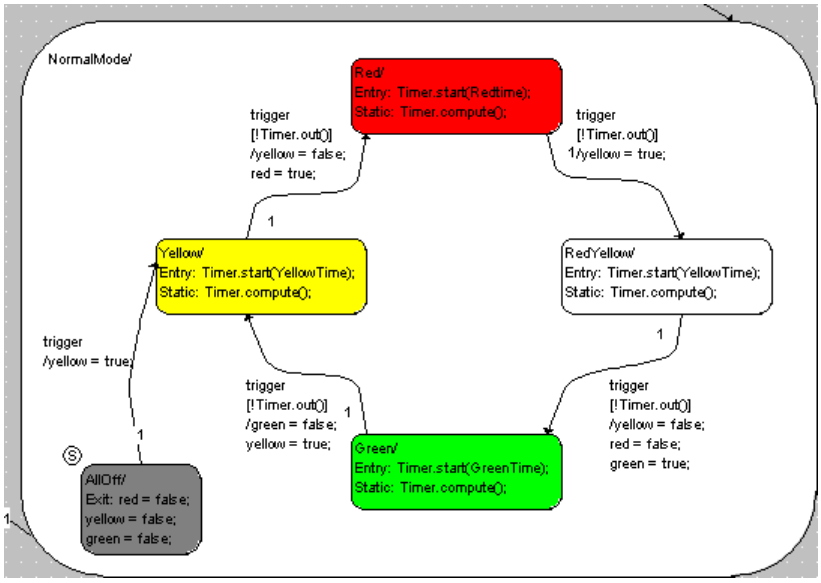
この例では、各色の点灯信号のオン／オフ切り替えを、今までのようにステートの Entry アクションにではなく、トランジションアクションに定義します。

正常動作時のサブステートを定義する：

- ステート `AllOff`（開始ステート）、`Yellow`、`Red`、`RedYellow`、`Green` を作成して、`NormalMode` ステート内に配置します。
- これらのステートの動作を定義します。これには、Entry アクションで各色用のタイマを起動し、Static アクションでタイマ処理（`Timer.compute()`）を行うようにします。

- ステートトランジションを定義し、トランジションアクション内にステートの動作を定義します。

  正常時は `AllOff` ステートから `Yellow` ステートに遷移し、他のトランジションは、対応するタイマのカウントダウンが終了した時に発生します。



- 各色の点灯信号用のアクションを、トランジションエディタの“Action”タブに入力します。
- トランジションエディタを閉じて、**File → Save** を選択します。

これで、交通信号制御システムの定義が終わりました。この実験を始めるには、各色用タイマのパラメータに適切な値を入力する必要があります。

4.9.2 階層ステートマシンの実験を行う

階層ステートマシンの実験も、基本的なステートマシンの実験と同じ方法で行うことができます。実験では、アニメーション機能を起動することを忘れないでください。

ステートマシンの実験を行う：

- ステートマシンエディタで **Build → Experiment** を選択して、実験環境を開きます。

- ステートの 1 つを右クリックし、ショートカットメニューから **Animate States** を選択します。
- `trigger` イベントをイネーブルにします。

- **Start Offline Experiment** ボタンをクリックして、ステートマシンの実験を行います。

- `GreenTime` パラメータの値を変更し、さらに **Update Dependent Parameter**を使用して依存パラメータの値を更新して、実験を行ってください。
- `error` 入力を `true` にして、動作の変化を確認してください。

4.9.3 階層ステートマシンの動作

階層ステートマシンは、通常のステートマシンと同じように機能します。基本的に、階層ステートマシンはさまざまな動作すべてをグラフィック構造で表しているにすぎません。余裕のある方は、階層ステートマシンで定義した動作を、どのようにすれば階層なしで実現できるか考えてみましょう。

この交通信号の例では、2 つの階層ステートを使いました。システムは論理入力変数 `error` の値に応じて、2 つのステート `ErrorMode` と `NormalMode` を切り替えます。それぞれの動作は、これらのステートの中で定義されています。

これを理解するために、`ErrorMode` ステート内の処理に注目してみましょう。トリガが起動されるたびに、階層ステート `ErrorMode` から `NormalMode` へのコンディション（コンディション：`!error`）が調べられます。主要ステートのトランジションが必要ない場合には、サブステートのトランジション `YellowOff` から `YellowOn` へ、またはその逆のトランジションのコンディションが調べられ、必要なアクションが行われます。

`NormalMode` について考えてみると、トリガが呼び出されるたびに、まず `error` 入力が `true` かどうか、つまり遷移が必要かどうかが調べられ、この条件に該当しなかった場合にのみ、各サブステート（`AllOff`、`Yellow`、`Red`、`RedYellow`、`Green`）間のコンディションが調べられます。交通信号の例では、タイマがタイムアウトしたかどうかも調べられます。

以上の処理を明らかにするために、ステートダイアグラムから生成されるコードを見てみましょう。

生成されたコードを表示する：

- ステートマシンエディタで **Build → View Generated Code** を選択して、生成されたコードを表示します。

  コンポーネントのコードはテンポラリファイルに書き込まれ、Windows に登録されているアプリケーションで開かれます。

注記

生成されたコードを表示する際、拡張子 `*.c` および `*.h` のファイルに関連付けられているアプリケーションが、Windows のレジストリ内から検索されて開きます。

4.9.4 まとめ

このレッスンでは、ASCET で以下の作業を行いました。

- 階層ステートダイアグラムを作成する
- ステートアクションとトランジションアクションの動作を定義する
- モジュール、クラス、コンポーネントをインポートする
- ASCET ライブラリのシステムコンポーネントをインポートする
- 既存の Timer コンポーネントを使用する
- 依存パラメータを使用する
- 生成されたコードを表示する

5 用語集

この用語集では、ASCET のドキュメントで使用されている用語について解説します。各用語の一般的な意味よりも、ここでは、ASCET について使用される場合の意味について説明します。

用語はアルファベット順、あいうえお順に並んでいます。

5.1 略語集

ASAM-MCD

Assosiation for the **S**tandardization of **A**utomation and **M**easurement systems（オートメーションおよび測定システム標準化委員会）の、測定（**M**easurement）、適合（**C**alibration）および診断（**D**iagnosis）についてのワークグループ

ASCET

ECU ソフトウェア開発ツール

ASCET-MD

ASCET Modeling and **D**esign - ASCET モデリング／デザインツール

ASCET-RP

ASCET Rapid **P**rototyping - ASCET ラピッドプロトタイピングツール

ASCET-SE

ASCET Software **E**ngineering - ASCET ソフトウェアエンジニアリングツール

AUTOSAR

Automotive **O**pen **S**ystem **A**rchitecture（http://www.autosar.org/ 参照）

BDE

Block **D**iagram **E**ditor（ブロックダイアグラムエディタ）

CPU

Central **P**rocesing **U**nit（中央演算処理装置）

ECU

Embedded **C**ontrol **U**nit（組み込み制御ユニット）

ERCOS^{EK}

OSEK 準拠の ETAS リアルタイムオペレーティングシステム

ESDL

Embedded **S**oftare **D**escription **L**anguage（組み込みソフトウェア記述言語）

ETK

独語：**E**mulator**t**ast**k**opf（Emulator Test Probe：エミュレータ・テストプローブ）

FPU

Floating **P**oint **U**nit（浮動小数点演算ユニット）

HTML

Hypertext **M**arkup **L**anguage（ハイパーテキストマークアップ言語）

INCA

Integrated **C**alibration and **A**cquisition Systems（統合計測・適合システム）

INTECRIO

ETAS の新しい製品ファミリ。さまざまなビヘイビアモデリングツールで作成されたコードを統合してコンフィギュレーション設定や実行コードの生成を行い、ラピッドプロトタイピング実験を行うための実験環境を提供します。

OS

Operating **S**ystem（オペレーティングシステム）

OSEK

独語：Arbeitskreis **O**ffene **S**ysteme fur die **E**lektronik im **K**raftfahrzeug（Open Systems and the Corresponding Interfaces for Automotive Electronics：自動車エレクトロニクス用オープンシステムおよびインターフェース）

RAM

Random **A**ccess **M**emory（ランダムアクセスメモリ）

RE

Runnable **e**ntity（ランナブルエンティティ：実行時に RTE によってトリガされる、SWC 内の一連のコードで、ASCET の「プロセス」にほぼ相当するもの）

ROM

Read **O**nly **M**emory（読み取り専用メモリ）

RTA-RTE

ETAS の AUTOSAR **r**un**t**ime **e**nvironment（ETAS の AUTOSAR 実行環境）

RTE

AUTOSAR **r**un**t**ime **e**nvironment（AUTOSAR 実行環境：ソフトウェアコンポーネント、基本ソフトウェア、オペレーティングシステム間のインターフェースを提供するもの）

SWC

Atomic AUTOSAR **s**oft**w**are **c**omponent（アトミックな AUTOSAR ソフトウェアコンポーネント：AUTOSAR における分割不可能な最小単位のソフトウェアコンポーネント）

UML

Unified **M**odeling **L**anguage（統一モデリング言語）

XML

Extensible **M**arkup **L**anguage（拡張マークアップ言語）

5.2 その他の用語

ASAM-MCD-2MC ファイル

プロジェクト用のデータ交換に使用される ASCII 形式の標準フォーマットファイルで、測定および適合に必要なディスクリプションが含まれています。`*.a2l` という拡張子が付きます。

C コード

コンポーネントを記述する形式の 1 つで、内容はインプリメンテーションに依存します。

HEX ファイル

プログラムの 1 つのバージョンをハードウェア間で交換するためのファイル形式です。フォーマットは Intel HEX と Motorola S Record のいずれかです。

Intel Hex

プログラムの 1 つのバージョンをハードウェア間で交換するためのファイル形式です。

L1

ホストと、実験が行われているターゲットとの間でデータ転送を行うためのメッセージフォーマットです。データ転送は、たとえば測定ウィンドウでの値の表示のために行われます

Motorola-S-Record

プログラムの 1 つのバージョンをハードウェア間で交換するためのファイル形式です。

OSEK オペレーティングシステム

OSEK 準拠のオペレーティングシステムです。

アイコン

ASCET コンポーネントの機能を視覚的に表すために使用されます。

アクション

ステートマシンの構成要素で、ステートマシンのステートまたはトランジションに関連付けて定義されます。一連の機能からなり、その実行はステートマシンによりトリガされます。

アプリケーションモード

ASCET のオペレーティングシステムの動作モードです。EEPROM プログラミングや暖機運転モード、通常モードなどシステムに想定されるさまざまな状態を表します。

イベント

オペレーティングシステムのアクション（タスクの起動など）の外部トリガとなるものです。

イベントジェネレータ

実験環境の一部を成すもので、オフライン実験において、タスク（メソッド／プロセス／タイムフレーム）起動のためのイベント生成の順序とタイミングを定義するために使用されます。

インターフェース

コンポーネントのインターフェースは、そのコンポーネントが他のコンポーネントとどのようにデータ交換を行うかを定義します。これは C 言語環境における .h ファイルにたとえられるものです。

インプリメンテーション

物理記述（モデル）から実行形式の固定小数点コードへの変換方法を定義するものです。モデルデータの変換に使用される線形変換式と上下限値とで構成されます。

インプリメンテーションキャスト

このエレメントを使用すると、連続する算術演算の中間値のインプリメンテーションを、そのエレメントの物理表記を変えることなく任意に変更することができます。

インプリメンテーション型

プロジェクト全体で使用できる複数のインプリメンテーションを「インプリメンテーション型」として定義しておき、それを必要に応じて各エレメントに個別に割り当てることができます。

ウィンドウエレメント

実験で使用される適合エレメントと測定エレメントを指す一般的な用語です。

エディタ

→「**適合ウィンドウ**」参照

エレメント

コンポーネントを構成する要素で、データ（変数やパラメータ、またはコンポーネント内で使用される他のコンポーネント）の読み取りや書き込みを行います。

エレメントタイプ

エレメントのタイプには、変数、パラメータ、定数の 3 種類があります。変数の値は読み取りも書き込みも可能です。パラメータの値は読み取ることしかできませんが、実験中に適合（書き込みによる値の調整）を行うことができます。定数の値は常に読み取ることしかできず、実験中でも書き込みは行えません。

オシロスコープ

測定ウィンドウの一種で、実験中にデータの値をグラフ表示します。

オフライン実験

ASCET で生成されたコードが PC または実験ターゲット上で実行される環境ですが、リアルタイムな実行はサポートされません。ここでは主にシステムの機能記述についてのテストに重点が置かれます。

オペレーティングシステム

ASCET ソフトウェアシステムの起動や実行をスケジューリングするものです。また内部通信のためのサービス（メッセージ）やハードウェアの予約部分（リソース）へのアクセスもサポートします。ASCET のオペレーティングシステムは、ERCOS^{EK} というリアルタイムオペレーティングシステムをベースとしています。

オンライン実験

ASCET で生成されたコードが実験ターゲット上で、リアルタイムオペレーションシステムに設定された挙動に基づいてリアルタイムに実行されます。ここでは、スケジューリング等のリアルタイムな挙動 すが、リアルタイムな実行はサポートされません。ここでは主にシステムの機能記述についてのテストに重点が置かれます。

階層

階層ブロックは、ブロックダイアグラムによるグラフィック定義を階層化（構造化）するために使用します。

型 （またはモデルデータ型）

ASCET モデルでは、`cont (continuous)`、`udisc (unsigned discrete)`、`sdisc (signed discrete)`、`log (logic)` といったさまざまな型の変数やパラメータを使用できます。`cont` は任意の値を持つことのできる物理量に使用され、`udisc` は正の整数値、`sdisc` は負の整数値に使用されます。また `log` は論理値（`true` または `false`）に使用されます。これらの型は、生成されるコードで使用される実装データ型とは異なります。加算や比較などさまざまな基本演算がそれぞれの型用に用意されています。実装情報によって、モデルデータ型が実装データ型に変換されます。

クラス

ASCET のコンポーネントの 1 つで、オブジェクト指向言語におけるクラスと同じものです。クラスの機能はメソッドにより定義されます。

グループ適合カーブ／マップ

1 つの座標ポイントディストリビューションを共有する複数の特性カーブ／マップを指し、総称して「グループテーブル」とも呼ばれます。各グループテーブルの出力値はそれぞれ異なります。座標ポイントのディストリビューションと個々のグループテーブルは、それぞれ独立したエレメントとして定義されます。

コード

コード（実行形式のコード）は、実際のアルゴリズムを含むプログラムで、データは含まれません。コードは、CPU によって実行されるプログラムの部分です。

コード生成

物理モデルから実行形式コードへの変換の第 1 ステップです。物理モデルが ANSI C コードに変換されます。C コードはコンパイラ（つまりターゲット）に依存するので、ターゲットごとに異なるコードが生成されます。

固定小数点コード

物理記述から生成されるコードの一種で、浮動小数点演算ユニットを持たないマイクロプロセッサ上で実行するためのものです。

コンディション

ステートマシンの制御フローを定義するためのものです。あるステートから別のステートへのトランジションを発生させるかどうかを決める論理値を返します。

コンテナ

プロジェクト、クラス、またはモジュールを保管しておくためのものです。モデルやデータベースの構築や、異なるデータベースアイテムを共通のバージョン管理下に置くために使用されます。

コンフィギュレーションダイアログボックス

測定／適合ウィンドウ、または各ウィンドウに表示される変数のコンフィギュレーションを行うために使用されるダイアログボックスです。

コンポーネント

ASCET における再利用可能な機能の基本単位です。コンポーネントはクラス、モジュール、またはステートマシンによって定義されます。各コンポーネントは、演算子で接続されて機能を形成する数々のエレメントで構成されます。

コンポーネントマネージャ

ASCET の作業環境設定を行ったり、ユーザーによって作成されデータベースに格納されたデータの管理を行うためのユーザーインターフェースです。

算術演算サービス

オンライン実験の環境設定が格納されます。測定／適合ウィンドウに関する情報（ウィンドウのサイズや位置、およびウィンドウに含まれるデータチャンネルの割り当てやサンプリングレート、表示形式等）や、イベントジェネレータ、データジェネレータ、データロガー等の設定が含まれます。

実験

コンポーネントやプロジェクトの機能をテストするために使用される実験環境の設定が定義されたものです。ここには、測定ウィンドウや適合ウィンドウに関する情報（ウィンドウのサイズや位置、およびその内容）や、イベントジェネレータ、データジェネレータ、データロガーについての設定が含まれます。実験は、オフライン（非リアルタイム）またはオンライン（リアルタイム）で行われ、バイパスまたはフルパス環境において物理プロセスを制御するシミュレーションが行われます。どの条件においても、ASCET で記述されたモデルから生成されたコードが使用されます。

実験環境

測定／適合作業を行うための操作環境です。

実装データ型

C 言語の基礎であるデータ型（`unsigned byte`（`uint8`）、`signed word`（`sint16`）、`float`）を指します。

スケジューリング設定

プロセスへのタスクの割り当て、およびオペレーティングシステムによるタスク起動に関する定義です。

スコープ

各エレメントは、ローカル（コンポーネント内でのみ使用可能）またはグローバル（プロジェクト内で定義されている）のスコープを持ちます。

ステート

ステートマシンを構成する要素です。ステートマシンは常にそのステートのうちの 1 つの状態（つまり、いずれか 1 つのステートがアクティブな状態）になっています。いずれか 1 つのステートが開始ステートとしてマークされていて、これがステートマシンの初期ステートになります。ステート同士はトランジション（遷移）を示す曲線で接続されています。各ステートには Entry アクション（そのステートに入ったときに実行されるアクション）、Static アクション（そのステートのまま変わらないときに実行されるアクション）、および Exit アクション（そのステートから出るときに実行されるアクション）が定義されます。

ステートマシン

ASCET のコンポーネントの 1 つです。その挙動は、トランジションで接続される複数のステートによって構成されるステートダイアグラムで記述されます。

測定値

実験中に画面上に表示される変数やパラメータの値です。値は、さまざまな測定ウィンドウ（オシロスコープや数値ディスプレイ）に表示することができます。

測定ウィンドウ

実験中に測定信号を表示する、ASCET の作業ウィンドウです。

測定チャンネルパラメータ

測定モジュールの個々のチャンネル用に設定されるパラメータです。

ターゲット

実験の対象となるハードウェアを指します。ターゲットには実験ターゲット（PC、トランスピュータ、PowerPC）とマイクロコントローラターゲット（ECU）があります。

ダイアグラム

コンポーネントをブロックダイアグラムまたはステートマシンによって定義するために使用される記述形式です。

タスク

オペレーティングシステムから起動される単位で、その中に複数のプロセスとその実行順が割り当てられます。タスクは、アプリケーションモード、起動用のトリガ、優先度、スケジューリングモード等の属性を持ちます。タスクが起動されて実行されると、そのタスクに割り当てられたプロセスが指定された順序で実行されます。

定数

ASCET モデルの実行中に値を変更することのできないエレメントです。

ディスクリプションファイル

ECU 内の特性値と測定変数の物理情報（名前、アドレス、変換式、ファンクション割り当てなど）を記述するファイルです。

ディストリビューション

1 つまたは複数の特性カーブ／マップのブレイクポイント（「サンプルポイント」とも呼ばれます）が定義されたものです。

ディメンション

基本エレメントのサイズを定義するものです。ディメンションの型は、スカラ（0 次元）、配列（1 次元）、特性カーブ／テーブルのいずれかです。

データ

プログラム用の変数の集合で、適合時に使用されます。

データジェネレータ

実験環境の一部を成すものです。実験対象のモデルの入力または変数をスティミュレートするために使用されます。

データセット

コンポーネントまたはプロジェクトのすべてのエレメントの初期データ、あるいはその参照が設定されています。

データベース

ASCET で生成されるすべての情報が格納されます。フォルダによる階層構造になっています。

コンポーネントマネージャ

データベースを作成し、格納されるデータを管理するための作業環境です。

データロガー

実験環境で測定されるデータを読み取り、後で分析できるようにディスクに保存する機能を持ちます。

適合

ASCET モデルの実行（実験）中に、パラメータの値（物理／インプリメンテーション）を調整することです。

適合ウィンドウ

パラメータの修正に使用する ASCET の作業ウィンドウです。

特性カーブ

2 次元のパラメータ（適合変数）です。

特性データ

マップ、カーブ、および特性値を表す一般的な用語です。（「パラメータ」も参照してください。）

特性値

1 次元のパラメータ（適合変数）です。

特性マップ

3 次元のパラメータ（適合変数）です。

トランジション

ステート間の遷移を表し、起こり得るステートの遷移について記述されたものです。各トランジションにはステートマシンのトリガの 1 つが割り当てられ、優先度、コンディション、アクションを持ちます。

コンディション

ステートマシンのフロー制御を定義するために使用されます。あるステートから別のステートへのトランジションを起こすかどうかを決める論理値を返します。

トリガ

タスクの実行、またはステートマシンの実行をトリガ（起動）するものです。

バイパス実験

ASCET がマイクロコントローラに直接接続され、マイクロコントローラソフトウェアの一部をシミュレートする実験です。

配列

基本スカラ型 `continuous` または `discrete` の静的な 1 次元リストで、基本スカラ型 `discrete` のインデックスを持ちます。

パラメータ

ASCET のモデル内での計算では値を変更できないエレメント（特性値、特性カーブ、および特性マップ）です。ただし実験中に適合させることができます。

引数

クラスメソッドへの入力です。その引数が属するメソッドの記述においてのみ使用でき、クラス内の他のメソッドでは使用できません。

フォルダ

ASCET データベースの階層構造を形成する単位で、この中に各アイテムが格納されます。

フルパス実験

ASCET が実験用マイクロコントローラに直接接続され、アプリケーション全体が ASCET によってシミュレートされる環境です。

プログラム

コードとデータから構成され、ECU の CPU により実行される 1 つの単位です。

プログラムバージョン

ECU プログラムが格納された HEX ファイルを指します。プログラムバージョンの一部が、データバージョンと共に 1 つのファイルとなります。

プロジェクト

組み込みソフトウェアシステム全体についての記述です。機能を定義するコンポーネント、オペレーティングシステムに関連した設定、および内部通信を定義するバインディングのしくみが記述されます。

プロセス

並行して実行される機能の単位で、オペレーティングシステムにより起動されます。プロセスはモジュール内に記述され、引数／入力や戻り値／出力を持ちません。

ブロックダイアグラム

コンポーネントをグラフィカルに記述するものです。さまざまなエレメント、演算子、および入力／引数や出力／戻り値が、方向性のある線で接続されます。1 つのブロックダイアグラムは複数のダイアグラムで構成されます。ブロックダイアグラムによる記述は、C コードによる記述とは異なり、物理記述です。

変換式

インプリメンテーション（実装情報）の一部で、データのモデルデータ型から実装データ型への変換方法を定義するものです。

変数

ASCET のモデルの実行中にモデルから値の読み取りや書き込みができるエレメントです。また適合作業において値を変更することもできます。

また広義では、パラメータ（特性データ）と測定信号を示す一般的な用語としても使用されます。

メソッド

オブジェクト指向プログラミングで使用されるクラスにおいて、そのクラスの機能の一部を記述するものです。メソッドは任意の数の引数と 1 つの戻り値を持ちます。

メッセージ

並行して実行されるプロセス間のデータ交換の整合性を保証するための、ASCET のリアルタイム言語構造体です。

モジュール

ASCET のコンポーネントの 1 つです。オペレーティングシステムによって起動される複数のプロセスが記述されています。モジュールを他のコンポーネント内のサブコンポーネントとして使用することはできません。

モニタ

実験中にエレメントの値をダイアグラム上に表示する機能です。

ユーザープロファイル

ユーザー固有のオプション設定です。

優先度

各タスクに対して数値で与えられる優先度で、この数値が大きいほど優先度が高くなり、優先的にスケジューリングされます。

ランナブルエンティティ

→「**RE**」参照

ランタイム環境

→「**RTE**」参照

リソース

組み込みシステムにおいて、排他的に使用される部分（タイマなど）をモデリングする際に使用されます。このような部分をアクセスするには、使用前にそれを予約（確保）して使用後に解放する、という処理が必要ですが、この一連の処理がリソースの概念によって行われます。

リテラル

コンポーネントの記述に使用されるもので、連続値や論理値として解釈される文字列を定義します。

レイアウト

コンポーネントは、入力／引数や出力／戻り値を表すピンや、タイムフレーム、メソッド、プロセスを表すグラフィック表現（レイアウト）を持ち、さらにそのレイアウトには、そのコンポーネントが他のコンポーネント内で使用される時のグラフィック表示に用いられるアイコンも含まれています。

6 付録 A：ASCET に関するトラブルシューティング

この章では、ASCET 使用時に ASCET 固有の問題（ネットワーク等に関する一般的な問題以外）が発生した際の対処方法を説明します。

6.1 トラブルシューティングと ETAS へのお問い合わせ

ASCET の製品開発においては、プログラムの機能上の安全性が最も重要視されています。しかし万一ご使用中にエラーが発生した場合には、以下の情報を ETAS までお送りください。

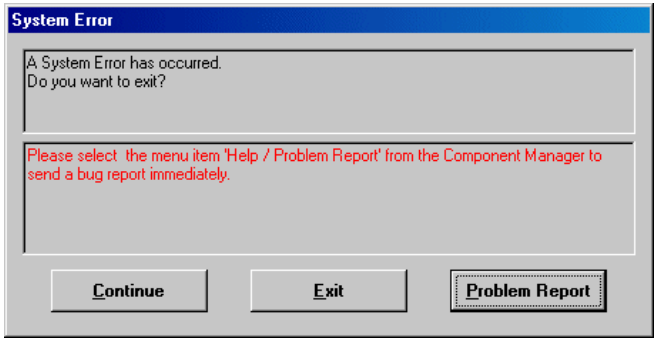
- エラーが発生したときにどのような作業をしていましたか？
- 発生したエラーはどのようなものでしたか？（関数エラー、システムエラー、システムのクラッシュなど）
- エラーが発生した時に、どのようなモデル（またはモデルエレメント）を編集していましたか？

注記

ASCET の品質をより高めるためには、ASCET の操作中にエラーが発生した際にユーザーの皆様からお送りいただくレポートは、非常に重要です。お手数ですが、できる限り以下の手順でレポートをお送りください。

サポート機能を使用すると、ASCET は自動的にログディレクトリの内容（すべての `*.log` ファイル）とテキスト情報を、`...¥ETAS¥LogFiles¥` サブディレクトリの `EtasLogFiles00.zip` というファイルに圧縮します。2 回目以降の ZIP ファイルには番号が付き、この番号は 1 から順に 19 までインクリメントされます。

重大なシステムエラーが発生すると、以下のようなウィンドウが開きます。



エラー発生時の操作：

1. **Problem Report** ボタン
   - **Problem Report** ボタンをクリックします。

     サポート機能が起動されます。
   - エラー内容を記述し、作成された ZIP ファイルを使用していたモデルと共に ETAS の LabCar ホットラインに送信してください。
1. **Exit** ボタン
   - **Exit** ボタンをクリックします。

     ASCET が終了します。保存されていなかった変更はすべて失われます。

     データを保存することを勧めるメッセージボックスが開いた場合、データの保存は行わずにメッセージボックスを閉じてください。
   - ASCET を再起動します。
2. **Continue** ボタン

注記

Continue ボタンは、重要なコンフィギュレーションを保存する必要があるような場合に限り使用してください。そのまま作業を続行すると、さらにエラーが発生したり、不正なコンフィギュレーション設定が行われてしまう可能性があります。

- **Continue** ボタンをクリックします。

  ASCET は稼働し続け、エラー発生直前の状態に戻ります。
- データを保存します。
- ASCET を終了します。
- ASCET を再起動します。

一般には、エラーが発生したらデータの保存を行わずにプログラムを終了し、再起動することをお勧めします。それにより、引き続いてより重大なエラーが発生することを回避できます。

7 付録 B：一般的なトラブルシューティング

この章では、個々のソフトウェアやハードウェアに依存しない一般的な問題が発生した場合の対処法について説明します。

7.1 問題と解決法

7.1.1 ETAS ネットワーク用のネットワークアダプタを選択できない

原因：APIPA が無効になっている

IP アドレッシングの代替メカニズムである APIPA は、Windows 2000 ／ XP ／ Vista 環境では、通常は有効に設定されていますが、時にはネットワークセキュリティポリシーによって無効となっている場合もあります。そのような場合、DHCP 設定のネットワークアダプタ（ネットワークカード）を ETAS ハードウェアのアクセスに使用することはできず、そのアダプタを選択すると ETAS ネットワークマネージャは警告メッセージを表示します。

無効になっている APIPA メカニズムを有効にするには、Windows のレジストリを編集する必要がありますが、これを行うには管理者の権限が必要です。ネットワーク管理者の方にご相談のうえ行ってください。

APIPA メカニズムを有効にする：

- Windows の **スタート** メニューから **ファイル名を指定して実行** を選択します。
- `regedit` と入力して **OK** をクリックします。

  レジストリエディタが開きます。
- 以下のフォルダを開きます。

  `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/Topic/Parameters/`
- **編集 → 検索** を選択して以下のキーを検索します。

  `IPAutoconfigurationEnabled`
- APIPA メカニズムが有効になるように、見つかったキーの値をすべて `1` に変更します。

  Windows のレジストリ内には、この名前のキーがいくつか含まれている場合があります。これらは一般的な TCIP/IP サービス用のものと、個別のネットワークアダプタ用のものです。ETAS ネットワーク用に使用するアダプタについてのみ値を変更してください。
- レジストリエディタを閉じます。

- 変更されたレジストリの内容を有効にするため、PC を再起動します。

APIPA メカニズムが無効になっていない場合、レジストリ内にこのキーは存在しません。

7.1.2 PC に接続されたイーサネットハードウェアが検索されない

原因：パーソナルファイアウォールによる通信のブロック

パーソナルファイアウォールによって発生する可能性のある問題とその解決法は、151 ページの「パーソナルファイアウォールによる通信のブロック」を参照してください。

原因：リモートアクセス用クライアントソフトウェアによる通信のブロック

ETAS ハードウェアネットワーク外で使用されている PC には、リモートアクセス用クライアントソフトウェアがインストールされているものがあり、それによって ETAS ハードウェアへのアクセスが妨害される場合があります。それには以下のような状況が考えられます。

- イーサネットメッセージをブロックするファイアウォールが使用されている（148 ページの「原因：パーソナルファイアウォールによる通信のブロック」を参照してください）
- トンネリングに使用されている VPN クライアントソフトウェアが誤ってメッセージをフィルタリングしてしまうことがあります。たとえば Cisco VPN クライアントの V4.0.x より前のバージョンでは、特定の UDP ブロードキャストを不正にフィルタしてしまう、というケースがありました。

  このケースに該当する場合は、VPN クライアントソフトウェアのバージョンをアップデートしてください。

原因：ETAS ハードウェアのフリーズ

ETAS ハードウェアが何らかの理由でフリーズしてしまった可能性もあります。この場合は、ハードウェアの電源を切ってから再投入してください。これによってハードウェアは再初期化されるので、多くの場合、正常に戻ります。

原因：ネットワークアダプタへの IP アドレス割り当てが一時的に失われた

PC の接続を、DHCP が使用されている社内 LAN から ETAS ハードウェアに切り替える際、PC が ETAS ハードウェアを検知できるようになるまでに約 60 秒かかります。これはオペレーティングシステムが DHCP プロトコルから ETAS ハードウェア用の APIPA に切り替わる処理に要する時間です。

原因：ETAS ハードウェアが他の論理ネットワークに接続されている

1 つの ETAS ハードウェアに対して複数の PC からアクセスする場合、各 PC で使用されるネットワークアダプタは、同じ論理ネットワークを使用するように設定しておく必要があります。このように設定することが不可能である場合、他の PC を切り替える前に ETAS ハードウェアの電源を切って再投入してください。

原因：ネットワークカード用のデバイスドライバが起動していない

ネットワークカード用のデバイスドライバが起動していない可能性があります。その場合は、ネットワークカードを一旦無効にしてから再度有効にしてください。

ネットワークカードを無効にして、再度有効にする：

- ネットワークカードを無効にするには、Windows の **スタート** メニューから以下のように操作します。
  - Windows 2000 の場合：**設定 → ネットワークとダイアルアップ接続** を選択します。
  - Windows XP の場合：**設定 → ネットワーク接続** を選択します。
  - Windows Vista の場合：**設定 → コントロールパネル**（クラシックビューの場合）**→ ネットワークと共有センター → ネットワーク接続の管理** を選択します。
- ETASネットワーク用に使用されているデバイスを右クリックし、ショートカットメニューから **無効にする** を選択します。
- 続いて同じショートカットメニューから **有効にする** を選択し、カードを有効にします。

原因：ラップトップPC の電源管理システムによってネットワークカードが無効になっている

ラップトップ PC の電源管理システムにより、ネットワークカードが無効になっている場合があります。この場合、ラップトップ PC の電源管理を無効にしてください。

ラップトップ PC の電源管理を無効にする：

- Windows の **スタート** メニュー以下のように操作します。
  - Windows 2000 の場合：**設定 → コントロールパネル → システム** を選択します。
  - Windows XP の場合：**設定 → コントロールパネル → システム** を選択します。

  - Windows Vista の場合：**設定 → コントロールパネル**（クラシックビューの場合）**→ システム → 詳細な電源設定の変更**（メニューコマンド）を選択します。
- **ハードウェア** タブを選択し、**デバイスマネージャ** ボタンをクリックします。
- **デバイスマネージャ** ウィンドウで **ネットワークアダプタ** のツリーを展開します。
- 使用するネットワークアダプタを右クリックし、ショートカットメニューから **プロパティ** を選択します。
- 以下のようにして電源管理を無効にします。
  - Windows 2000 の場合：**電源の管理** タブを選択し、**コンピュータでこのデバイスの電源をオフできるようにする** オプションをオフにします。
  - Windows XP の場合：**電源の管理** タブを選択し、**コンピュータでこのデバイスの電源をオフできるようにする** オプションをオフにします。
  - Windows Vista の場合：**電源マネージャ** タブを選択し、**コンピュータでこのデバイスの電源をオフできるようにする** オプションをオフにします。
- **詳細設定** タブを選択し、**プロパティ** に **Autosense** が含まれている場合、これを無効にします。
- **OK** をクリックして設定を有効にします。

原因：ネットワークの自動切断

ネットワークカードのデータトラフィックが一定の時間途絶えると、ネットワークカードが自動的にイーサネット接続を切断する場合があります。これは、レジストリの `autodisconnect` キーを設定することによって避けることができます。

レジストリキー `autodisconnect` を設定する：

- レジストリエディタを開きます。
- `HKEY_LOCAL_MACHINE/SYSTEM/ControlSet001/Services/lanmanserver/parameters` というフォル

ダに含まれるレジストリキー `autodisconnect` の値を `0xffffffff` に変更します。

7.1.3 パーソナルファイアウォールによる通信のブロック

パーソナルファイアウォールは、ETAS のイーサネットハードウェアへのアクセスを妨害する場合があります。そのような場合、ハードウェアの自動検索時に、コンフィギュレーションが正しく設定されているのにもかかわらずイーサネットハードウェアがまったく検出されない、という状態が発生する可能性があります。

また、ファイアウォールが適切に設定されていないと、ETAS ソフトウェアにおける特定の操作（例：ASCET で実験を開く、INCA や HSP でハードウェアを検索する、など）を実行する際に不具合が発生する場合があります。

ファイアウォールによって ETAS ハードウェアとの通信がブロックされる場合は、ETAS ソフトウェア使用中はファイアウォールソフトウェアを無効にするか、またはファイアウォールの詳細設定を行って以下のアクセスを許可するようにしてください。

- UDP を経由する、デスティネーションアドレス 255.255.255.255 への出力方向の IP ブロードキャスト（デスティネーションポート：17099 または 18001）
- UDP を経由する、ソース IP アドレス 0.0.0.0 からデスティネーション IP アドレス 255.255.255.255 への入力方向の IP ブロードキャスト（デスティネーションポート：18001）
- UDP を経由する、ETAS ネットワークへの直接 IP ブロードキャスト（デスティネーションポート：17099 または 18001）
- UDP を経由する、ETAS ネットワーク内のすべての IP アドレスへの出力方向の IP ユニキャスト（デスティネーションポート：17099 ～ 18001）
- UDP を経由する、ETAS ネットワーク内のすべての IP アドレスからの入力方向の IP ユニキャスト（ソースポート：17099 ～ 18020、デスティネーションポート：17099 ～ 18020）
- ETAS ネットワーク内への出力方向の TCP/IP 接続（デスティネーションポート：18001 ～ 18020）

注記

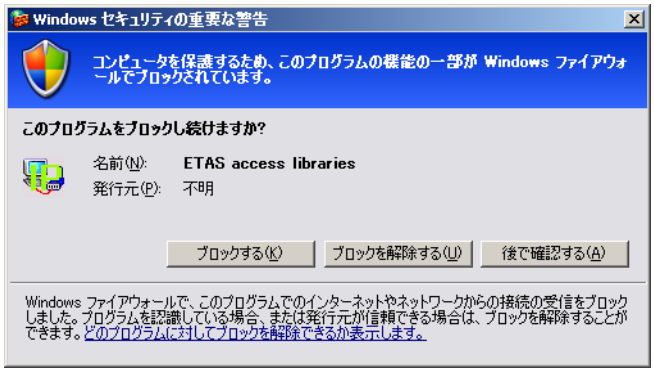
実際のポート番号は、使用するハードウェアに応じて異なります。ポート番号についての詳しい情報は、ハードウェアのドキュメントを参照してください。

Windows XP および Windows Vista に組み込まれているパーソナルファイアウォール以外に、サードパーティ（Symantec、McAfee、Blackice）の各種パーソナルファイアウォールも一般的によく使用されています。これらの各ファイアウォールではそれぞれ対処方法も異なる場合がありますので、お使いのパーソナルファイアウォールの説明書をよくお読みください。

以下に一例として、一般的に使用されている Windows XP（SP2）／ Vista のファイアウォールの設定方法をご紹介します。

Windows XP ／ Vista ファイアウォールの設定：管理者権限を持つユーザーの場合

PC の管理者権限を持っているユーザーの場合、ETAS ソフトウェアがファイアウォールによってブロックされると、以下のようなダイアログボックスが開きます。



製品のブロックを解除する：

- “Windows セキュリティの重要な警告”ダイアログボックスで、**ブロックを解除する** をクリックします。

  以降、該当する ETAS ソフトウェアはファイアウォールによってブロックされなくなります。この設定は、プログラムや PC の再起動後も維持されます。

上記の“Windows セキュリティの重要な警告”ダイアログボックスが開く前に、前もって ETAS ソフトウェアのブロックを解除しておくこともできます。

ファイアウォールの設定を変更して製品のブロックを解除する：

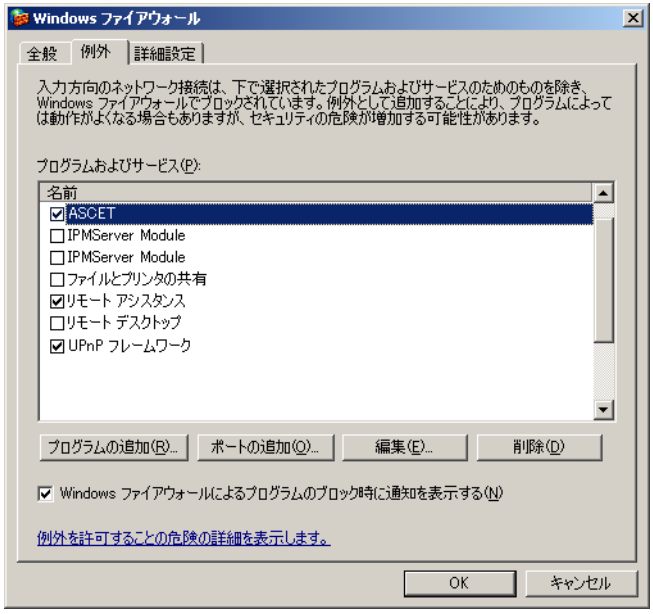
1. Windows XP
   - Windows のスタートメニューから **設定 → コントロールパネル** を選択します。
   - コントロールパネルから **Windows ファイアウォール** を選択します。
2. Windows Vista

- Windows のスタートメニューから **コントロール パネル → Windows ファイアウォール** を選択します。
- “Windows ファイアウォール”ダイアログボックスで、**Windows ファイアウォールによるプログラムの許可** をクリックします。

3. XP ／ Vista 共通

“Windows ファイアウォール”ダイアログボックス（XP の場合）または“Windows ファイアウォールの設定”ダイアログボックス（Vista の場合）が開きます。

- “例外”タブを選択します（下図は XP の場合です）。

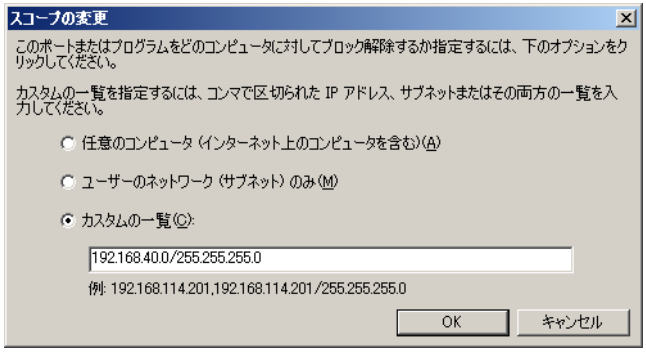


このタブには、ファイアウォールによるブロックから除外されているプログラムが一覧表示されます。

- **プログラムの追加** または **編集** ボタンをクリックして、新しいアイテム（プログラム）を追加するか、既存のアイテムを編集します。
- 以下のようにして、使用する ETAS ソフトウェアとそれに関連するサービスについて、正しく例外設定されていることを確認します。

– 例外リストから該当するアイテムを選択して **編集**（Vista の場合は **プロパティ**）をクリックし、“プログラムの編集”ダイアログボックスを開きます。

– **スコープの変更** をクリックし、“スコープの変更”ダイアログボックスを開きます。



– ETAS ハードウェアへのアクセスが行えるように、`192.168.40.xxx` という IP アドレスがブロック解除されていることを確認してください。

– **OK** をクリックして“スコープの変更”ダイアログボックスを閉じ、さらにプログラムの編集”ダイアログボックスも **OK** をクリックして閉じます。

- **OK** をクリックして“Windows ファイアウォール”ダイアログボックスを閉じます。

  以降、該当する ETAS ソフトウェアはファイアウォールによってブロックされなくなります。この設定は、プログラムや PC の再起動後も維持されます。

Windows XP ／ Vista ファイアウォールの設定：管理者権限を持たないユーザーの場合

権限（システム変更、書き込み、ローカルログオンなど）が制限されているユーザーの場合は、以下のように操作してください。

ETAS ソフトウェアを使用するユーザーは、所定のディレクトリ（`ETAS`、`ETASData`、ETAS の一時ディレクトリ）への“書き込み”の権利が必要です。それらの権利がない場合状態で ETAS ソフトウェアを起動すると、エラーメッセージが表示され、その後データベースが開きますが、正しい操作は行えません。これは、ETAS ソフトウェアの操作時にはデータベースファイルや `*.ini` ファイルの書き換えが必要なためです。

ETAS ソフトウェアは、必ず管理者権限のあるユーザーがインストールを行い、その後、Windows XP ／ Vista ファイアウォールの例外リストにそのプログラムを正しく登録してください。これが行われていないと、以下の事柄が生じます。

- ファイアウォールにより制限されているアクションを実行すると、“Windows セキュリティの重要な警告”ダイアログボックスが開きます。

製品のブロックを解除する（管理者権限のないユーザーの場合）：

- “Windows セキュリティの重要な警告”ダイアログボックスで、**このプログラムについてはこのメッセージを表示しない** をオンにします。
- **OK** をクリックしてダイアログボックスを閉じます。

  この後、管理者権限のあるユーザーが“Windows ファイアウォール”ダイアログボックス（XP の場合）または“Windows ファイアウォール設定”ダイアログボックス（Vista の場合）の“例外”タブで適切な設定を行い、ETAS ソフトウェアがハードウェアアクセスを行えるようにする必要があります。

8 付録 C：AUTOSAR

8.1 基本原理

AUTOSAR における SWC、つまり「アトミックソフトウェアコンポーネント」（実行されるソフトウェアの最小単位）の作成は、2 つのステップに分かれます。第 1 ステップでは、コンポーネントの「インターフェース」を定義します。インターフェースの定義は「オーサリングツール」を用いて行い、XML フォーマットで交換されます。XML ファイルはコンポーネント API ジェネレータに渡され、そこでインターフェース定義がヘッダファイルに変換されます。このヘッダファイルには、アトミックソフトウェアコンポーネントの内部挙動が使用するすべてのアクセスマクロが含まれます。一般に、コンポーネント API ジェネレータは RTE[1] ジェネレータの「コントラクトフェーズ部」に相当します。

第 2 ステップでは、アプリケーションソフトウェアコンポーネント開発者が C ファイルで内部挙動を記述しますが、その際、前出のヘッダファイル内に定義されているインターフェースを遵守する必要があります。このようにしてアトミックソフトウェアコンポーネントの `*.h` ファイルと `*.c` ファイルが作成され、コンパイルチェックが行える状態になります。アクセスマクロの解決は、RTE ジェネレータが、OS スケジュール、および RTE の作成対象となる ECU にマッピングされるすべてのコンポーネント（インスタンス）を認識できる「RTE 生成フェーズ」においてのみ行われます。

ASCET は、この「アトミックソフトウェアコンポーネントのオーサリングツール」および「挙動モデリングツール」の両方の機能を備えています。

「オーサリングツール」として使用される際は、ASCET 6.0 でサポートされている AUTOSAR モデリングエレメント（「モードグループ」、「インターフェース」、「アトミックソフトウェアコンポーネント」）が ASCET データベース内に作成され、維持管理されます。これらのエレメントは ASCET の OID（オブジェクト ID）を持ち、さまざまなインプリメンテーションを割り当てることができるため、複数のプロジェクトで再利用することができます。

AUTOSAR エレメントは XML ファイル内で互いに参照し合います。たとえば、「データエレメントプロトタイプ」は型を持ち、「センダ - レシーバ」インターフェース型内で使用されます。同様に、「インターフェース型」は「アトミックソフトウェアコンポーネント型」内で「ポートプロトタイプ」として使用されます。ASCET 6.0 では「アトミックソフトウェアコンポーネント型」を作成しますが、アトミックソフトウェアコンポーネントにはいわゆる「内部挙動」が定義され、内部挙動は「ランナブルエンティティ」で構成されます。「ランナブルエンティティ」は、AUTOSAR ECU の統合においてはオペレーティングシステムタスクにマッピングされ、コンポーネントの「部分的」挙動を表します。これは ASCET の「プロセス」とよく似ています。

---

1. **R**un **T**ime **E**nvironment：ランタイム環境

また ASCET は「挙動モデリングツール」としても使用できます。ASCET 6.0 において、アトミックソフトウェアコンポーネントの内部挙動はブロックダイアグラムエディタで定義されます。内部挙動は「変数」、「クラスインスタンス」、「パラメータ」、「シーケンスコール」で構成されます。マイグレーションの目的で「メッセージ」と「モジュール」もサポートされています。

AUTOSAR の XML およびコードの生成は ASCET プロジェクトにより行われます。つまり、AUTOSAR ユースケースにおける ASCET プロジェクトには 1 つのアトミックソフトウェアコンポーネント型が内包されます。AUTOSAR コード生成は、製品コードのターゲットである RTE-AUTOSAR ターゲット専用に機能します。コード生成時、アクセスマクロを含む C コードは ASCET の cgen フォルダに格納されます。生成された `*.c` と `*.arxml` の両方のファイルを取得するには、ASCET の「生成コードのエクスポート」機能を使用します。

8.2 ASCET における AUTOSAR モデリングエレメント

8.2.1 モード

ASCET は、AUTOSAR リリース 2.1 の「モード管理」をサポートしています。各モード型には「モード宣言」と呼ばれるものが必要で、これは ASCET の列挙型に似ています。典型的なモード宣言は init、run、diagnostics などです。ただし列挙型とは異なり、モードを表す整数値を明示的に設定することはできません。モードはデータベース内で維持管理され、エディタ内の最初のモード宣言が init モードになります。

8.2.2 インターフェース

ASCET 6.0 でサポートされているインターフェースは「センダ - レシーバ」インターフェースのみで、これをエディタで見ると、メソッドを持たないレイアウトクラスによく似ています。センダ - レシーバインターフェース内には「データエレメントプロトタイプ」が含まれます。データエレメントプロトタイプは ASCET のモデル型のひとつであるため、1 つのインターフェース型に複数の「ASCET インプリメンテーション」（実装情報）を持たせることができます。センダ - レシーバインターフェースの ASCET インプリメンテーションは、そのデータエレメントプロトタイプのインプリメンテーションにより定義されます。データエレメントプロトタイプの ASCET インプリメンテーションは、ASCET のモデル型（`cont` など）に ASCET の実装型（`uint16` など）を割り当てるもので、ASCET 実装型の範囲は、モデル型 cont の物理範囲と実装変換式により定義されます。この ASCET 実装型の範囲によってデータエレメントプロトタイプのプリミティブな AUTOSAR の型が決定されます。AUTOSAR リリース 2.1 では、RTE が AUTOSAR の型を C のデータ型に変換し、この変換については RTE 仕様書[1] の 5.2.4.2 項に記載されています。これらのデータ型は ASCET の実装型と一致します。

---

[1] AUTOSAR_SWS_RTE リリース 2.1（www.autosar.org を参照してください）

データエレメントプロトタイプ用のデータ型は、すべての ASCET ビルトインモデル型、およびユーザー定義の列挙型およびレコード型から選択することができます。「配列」は直接使用することはできないので、レコード内にカプセル化する必要があります。「モード」は、モード以外の型のデータエレメントプロトタイプが他にない場合に限りセンダ - レシーバインターフェース内で使用できます。つまり、センダ - レシーバインターフェースにモードがデータエレメントプロトタイプとして含まれている場合は、すべてのデータエレメントプロトタイプは「モード型」でなければなりません。このような「モードインターフェース」の場合、データエレメントプロトタイプは「モード宣言グループプロトタイプ」と呼ばれます。

8.2.3 アトミックソフトウェアコンポーネント型

「アトミックソフトウェアコンポーネント」は階層的な ASCET クラスとモジュールの混成体です。アトミックソフトウェアコンポーネント（SWC）は、インターフェース型を「ポートプロトタイプ」としてインスタンス化することができます。つまり、1 つの SWC の中に同じインターフェース型のポートプロトタイプが複数存在する可能性があります。ポートプロトタイプはそれを提供したり要求したりすることによってその機能が実現されます。センダ - レシーバインターフェースを持つポートプロトタイプは、そのデータエレメントプロトタイプがコンポーネントにより受信される場合は「要求ポート」となり、データエレメントがコンポーネントにより送信される場合は「提供ポート」になります。

ポートプロトタイプは SWC のエレメント（部品）であり、その役割はプロパティエディタで選択できます。役割が適切に設定されたデータエレメントプロトタイプは ASCET のメッセージに似ています。ASCET モジュール内のメッセージ送受信は「メッセージ」により行われますが、AUTOSAR SWC はランナブルエンティティを使用してポートプロトタイプのデータエレメントプロトタイプにアクセスします。

したがって、SWC の内部挙動は一連のランナブルエンティティにより定義されることになります。ランナブルエンティティはデータエレメントプロトタイプを読み取るたびに、データ読み取りアクセスを実行します。データエレメントプロトタイプの送信も同様で、ランナブルエンティティはデータ書き込みアクセスを行います。ここでは ASCET のメッセージとは異なり、各種のデータエレメントアクセスが用意されています。最も一般的な種類は implicit（暗黙的）と explicit（明示的）です。

ASCET 内の SWC のエレメントリストにおいて「ポートプロトタイプ」は、変数、パラメータ、クラスインスタンス、モジュール、メッセージなど他のすべての内包エレメントと同様に表示されます。SWC にもダイアグラムがあり、ランナブルエンティティはダイアグラム内に定義されます。ASCET SWC はランナブルエンティティだけでなくメソッドもサポートしています。

ランナブルエンティティにおいては、シーケンスコールを使用してグラフィカルに「代入」をモデリングします。たとえば、データエレメントプロトタイプを暗黙的に読み取るためには、ポートプロトタイプをダイアグラムに引き込みます。そうすると、データエレメントプロトタイプのピンが式の演算に接続され、これが変数、または提供ポート内のデータエレメントプロトタイプへの書き込みアクセスになります。この代入はシーケンスコールにより行われ、シーケンスコール

はランナブルエンティティ（またはメソッド）によりトリガされます。「明示的データ読み取りアクセス」（「データ受信ポイント」とも呼ばれます）には専用のアクセス演算子と、データを格納するための変数またはメッセージが必要です。その変数またはメッセージのシーケンスコールには、トリガとしてランナブルエンティティまたはメソッドが割り当てられます。

8.3 ASCET インプリメンテーションの AUTOSAR への適用

前述のように、ASCET のセンダ - レシーバインターフェースのデータエレメントプロトタイプは ASCET のモデル型をサポートしていますが、AUTOSAR の型は、ASCET モデル型にインプリメンテーションが定義されてからでないと決定できません。つまり、ASCET データベース内の 1 つのセンダ - レシーバインターフェースエントリは、その ASCET インターフェースのインプリメンテーションの数と同じ数の AUTOSAR インターフェース型を表すことになります。ASCET 内の SWC はポートプロトタイプ内で ASCET センダ - レシーバインターフェースを使用するので、異なる複数のインプリメンテーションを持つことができます。その結果、1 つの ASCET アトミックソフトウェアコンポーネントタイプには、定義されている ASCET インプリメンテーションの数と同じ数の AUTOSAR アトミックソフトウェアコンポーネントタイプが定義されることになります。

8.3.1 サブパッケージ

AUTOSAR のリリース 2.1 においては、XML ディスクリプション内では 32 文字までのショート名しか使用できません。そのため生成されたコード内の異なるインプリメンテーションを区別するためにインプリメンテーション名を ASCET モデル名に連結する、という ASCET の古典的なアプローチを適用するには限界がありますが、これは XML ファイル内にサブパッケージを使用することにより解決できます。XML ファイル内にはアトミックソフトウェアコンポーネントとそのインターフェースのために 1 つのパッケージが生成されますが、エレメントを処理する前に各インプリメンテーション用のサブパッケージが生成されます。ASCET 6.0 はアトミックソフトウェアコンポーネントの 1 つのインプリメンテーション用の C コードと XML ディスクリプションしか生成しないので、サブパッケージの数は 1 つだけです。インプリメンテーションは、プロジェクトのコード生成時に前もって選択され、ASCET XML ジェネレータは、すべての参照（サブパッケージ `impl-1` 内のポートプロトタイプからサブパッケージ `impl-2` 内のインターフェースへの型参照など）を適切に設定します。ここで、アトミック SWC とインターフェースは異なる AUTOSAR パッケージであることに注意してください。

8.4 例

ここで非常に単純な例を紹介します（以下の説明には ASCET 6.0 の `AUTOSAR Tutorial` というサンプルデータベースを使用しています）。このアトミックソフトウェアコンポーネントをアトミックソフトウェアコンポーネントエディタで開くと、コンポーネントに含まれるエレメントが図 8-1 のように表示されます。

ここには非常に単純なアトミックソフトウェアコンポーネント（AR_VerySimpleExample）が含まれていて、そこにデータ通信用の 2 つのポートプロトタイプ（インターフェース型 IF_a の port2 とインターフェース型 IF_AB の port4）が含まれています。また各ポートの上にはモードポートが表示されています。このソフトウェアコンポーネントには 2 つのランナブルエンティティ（runnable_expl および runnable_impl）が含まれています。また AUTOSAR エレメントの他に、ランナブルエンティティが明示的データアクセスを行って調べる error という変数と、RTE アクセス用に作成されている戻り値型 enumeration とがあります。

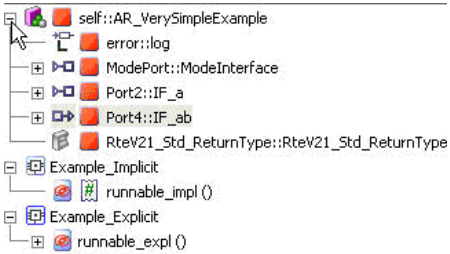


図 8-1　単純なアトミックソフトウェアコンポーネントのエレメント

インターフェース IF_a および IF_ab は、ASCET の cont モデル型であるデータエレメントプロトタイプをまとめたものです。インターフェース型 a の内容は図 8-2 のとおりで、左側のエレメントリストにデータエレメントプロトタイプが表示されています。

ここで“Implementation”タブを開いて AR_16bit というインプリメンテーションを選択してください。するとそのインプリメンテーションに定義されている変換式により、実装型が符号付き 16 ビット整数に変わります（図 8-3）。この設定に従ってコードと AUTOSAR XML ファイルが生成されます。

生成された XML ファイルの抜粋を図 8-4 に紹介します。これを見ると、ASCET interfaces という AR パッケージの下に AR_16bit というサブパッケージが生成されています。データエレメントプロトタイプ a は AR_16bit サブパッケージ内に組み込まれていて、autosar_types.arxml ファイル内の sint16 という実装型を参照（「型参照」）しています。

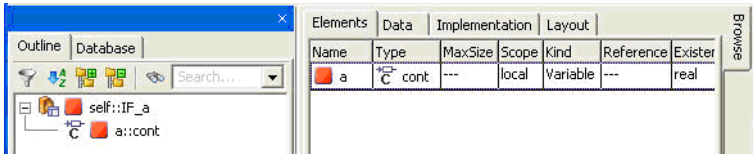


図 8-2　インターフェース IF_a

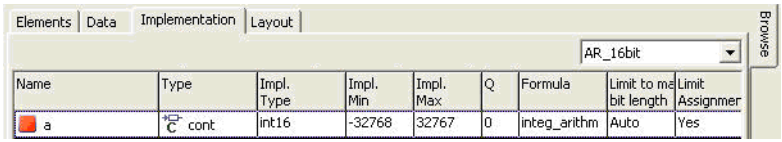


図 8-3　インターフェース IF_a（整数インプリメンテーション）

```
<AR-PACKAGE>
   <SHORT-NAME>ASCET_interfaces</SHORT-NAME>
   <DESC></DESC>
   <!--
Interfaces (RTE configuration)
-->
   <SUB-PACKAGES>
      <AR-PACKAGE>
         <SHORT-NAME>AR_16bit</SHORT-NAME>
         <DESC></DESC>
         <!--
   implementation name and component name define path
   -->
         <ELEMENTS>

         <SENDER-RECEIVER-INTERFACE>
            <SHORT-NAME>IF_a</SHORT-NAME>
            <DATA-ELEMENTS>
               <DATA-ELEMENT-PROTOTYPE>
                  <SHORT-NAME>a</SHORT-NAME>
                  <TYPE-TREF DEST="INTEGER-TYPE">/AUTOSAR_types/SInt16</TYPE-TREF>
                  <IS-QUEUED>false</IS-QUEUED>
               </DATA-ELEMENT-PROTOTYPE>
            </DATA-ELEMENTS>
         </SENDER-RECEIVER-INTERFACE>

         <SENDER-RECEIVER-INTERFACE>

         </ELEMENTS>
      </AR-PACKAGE>
      <AR-PACKAGE>
   </SUB-PACKAGES>
</AR-PACKAGE>
```

図 8-4　AR_16bit サブパッケージ内に生成されたインターフェース

前述のとおり（また図 8-1 にも示すとおり）、port2 というポートプロトタイプのインターフェース型は IF_a です。すべての ASCET モデリングエレメントと同様、製品コード生成に使用されるアトミックソフトウェアコンポーネントには複数のインプリメンテーションが定義されています。この例の AR_VerySimpleExample コンポーネントには 2 つのインプリメンテーションがあり、その 1 つは連続モデル用の 64 ビット real のデフォルトインプリメン

テーションである impl で、もう 1 つは符号付き int16 型を使用する AR_16bit です。ここでは図 8-5 に示すように、上記の操作によって整数インプリメンテーションが選択されているため、整数コードが生成されます。

図 8-5 16 ビット整数インプリメンテーションの選択

生成された AUTOSAR XML ファイルの抜粋を図 8-6 に紹介します。アトミックソフトウェアコンポーネント AR_VerySimpleExample は、ASCET インプリメンテーションと同じ名前のサブパッケージに組み込まれています。

```
<AR-PACKAGE>
  <SHORT-NAME>ASCET_swcomponents</SHORT-NAME>
  <DESC></DESC>
  <!--
 AR_VerySimpleExample>>AR_16bit (RTE configuration)
-->
  <SUB-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>AR_16bit</SHORT-NAME>
      <DESC></DESC>
      <!--
   implementation name and component name define path
  -->
      <ELEMENTS>

      <ATOMIC-SOFTWARE-COMPONENT-TYPE>
        <SHORT-NAME>AR_VerySimpleExample</SHORT-NAME>
        <PORTS>
          <R-PORT-PROTOTYPE>
          <R-PORT-PROTOTYPE>
          <P-PORT-PROTOTYPE>
        </PORTS>
      </ATOMIC-SOFTWARE-COMPONENT-TYPE>

      <INTERNAL-BEHAVIOR>

      <IMPLEMENTATION>

      </ELEMENTS>
    </AR-PACKAGE>
  </SUB-PACKAGES>
</AR-PACKAGE>
```

図 8-6 AR_16bit サブパッケージ内に生成されたアトミックソフトウェアコンポーネント

AUTOSAR パッケージの観点から見ると、この種の ASCET インプリメンテーション表現を使用するためには、すべての集合エレメント用の型参照にそれぞれ適切なパッケージが必要です。この種の「サブパッケージ参照」の例を図 8-7 に示します。インプリメンテーション AR_16bit のインターフェース IF_a を見つけるために、アトミックソフトウェアコンポーネント AR_VerySimpleExample 内の port2 の型参照は ASCET_interfaces パッケージだけでなく、そのサブパッケージである AR_16bit も参照します。

このためオーサリングツールと ECU 統合ツールは、サブパッケージを扱えることが必須条件となります。このサブパッケージ構造はコード生成や RTE アクセスにはまったく影響しません。

このようなサブパッケージ表現が使用されているのは、AUTOSAR リリース 2.1 に 32 文字制限があり、「エレメント名とインプリメンテーション名を連結する」という ASCET の古典的な方法は非常に限られた状況でしか使用できないためです。

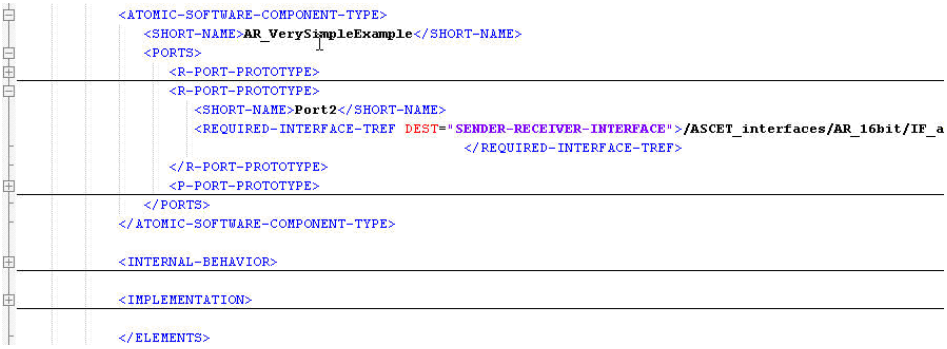


図 8-7　AUTOSAR XML ファイル内のサブパッケージ参照

ASCET ソフトウェアコンポーネントの内部挙動は、ブロックダイアグラムエディタで記述されます。この例では、それぞれ 1 つのランナブルエンティティを持つ 2 つのダイアグラムが使用されています。第 1 のダイアグラム内のランナブルエンティティはポートプロトタイプ port2 および port4 への暗黙的アクセスのみを実現しています。このブロックダイアグラムを図 8-8 に示します。要求ポートプロトタイプ port2 は左側に位置し、そのデータエレメントプロトタイプ a はランナブルエンティティ runnable_impl のシーケンスコール 5 番によりアクセスされ、グラフィカルな式の中で使用されます。この式の結果はシーケンスコール 5 番によりポートプロトタイプ port4 のデータエレメントプロトタイプ a に書き込まれます。ポートプロトタイプ port2 のデータエレメントプロトタイ

プ a は、さらにシーケンスコール 10 番によってもアクセスされます。シーケンスコール 10 番の式は a に 1（物理モデル）を加算し、その結果をポートプロトタイプ port4 のデータエレメントプロトタイプ b に暗黙的に書き込みます。

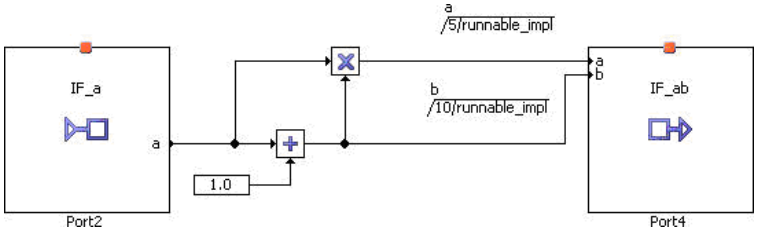


図 8-8　「暗黙的」ランナブルエンティティの内部挙動

生成されたコードとその暗黙的 RTE アクセスマクロを図 8-9 に紹介します。暗黙的通信の場合、RTE_IREAD アクセスマクロを式の中に使用でき、式は整数演算で実現されることがわかります。グラフィカルに見ると、暗黙的データエレメントアクセスはポートプロトタイプへの直接接続により示され、これがデフォルトアクセスと見なされます。

```
void AR_VERYSIMPLEEXAMPLE_AR_16BIT_runnable_impl (void)
{
   /* no messages have to be received */

   /* runnable_impl: sequence call #5 */
   Rte_IWrite_runnable_impl_Port4_a((sint16)(Rte_IRead_runnable_impl_Port2_a()
                                    * (Rte_IRead_runnable_impl_Port2_a() + (sint32)64) >> 6));
   /* runnable_impl: sequence call #10 */
   Rte_IWrite_runnable_impl_Port4_b((sint16)(Rte_IRead_runnable_impl_Port2_a() + (sint32)64));

   /* no messages have to be sent */
}
```

図 8-9　暗黙的ランナブルエンティティについて生成されたコード

明示的データアクセスは、図 8-10 に示すようにダイアグラム Example_Explicit で実現されています。このダイアグラムでは「RTE アクセスブロック」が使用されています。このブロック内で、ポートプロトタイプのデータエレメントプロトタイプに対するアクセスを、「暗黙的」(implicit)、「明示的」(explicit)、および「ステータス付き明示的」(explicit-with-status) アクセスのうちから選択することができます。このアクセスブロックがない場合はデフォルトとして暗黙的通信が行われますが、明示的通信を行う場合は必ずこのアクセスブロックが必要です。こうして、RTE アクセスブロック内においては explicit メソッドがデフォルトアクセスメソッドになります。このアクセスメソッドは port2 および port4 のデータエレメント a に使用され、port4 のデータエレメント b には explicit-with-status アクセスメソッドが使用されます。この場合、さらに、ASCET BDE 言語の if-then-else ブロックの else 部と同じ意味を持つ出力ピンを使用できます。この else 部を使用して、エラーフラグのセットやエラーファンクションの呼び出しを行うなどのデータフローをトリガできます。

明示的なデータ読み取りアクセスを使用するランナブルエンティティは、RTE からフェッチしたデータを格納するメモリを備えている必要があり、この例ではそのために「ランナブルローカル変数」が使用されています。他の方法としては、アトミックソフトウェアコンポーネント型、クラス、あるいはメッセージ内の変数を使用することもできます。メッセージはマイグレーションプロジェクト、つまり、モジュールがアトミックソフトウェアコンポーネントにインポートされる場合に使用できますが、OSEK プロジェクトの場合とは意味が少し異なります。

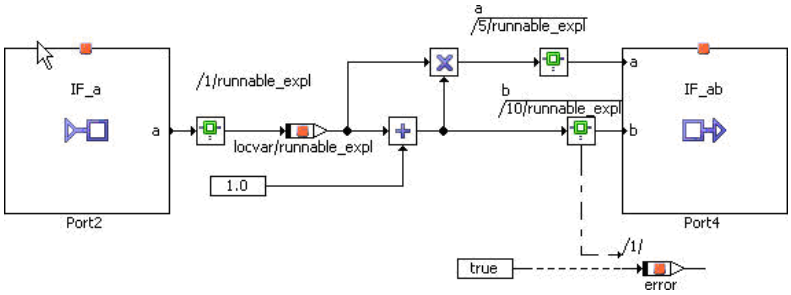


図 8-10 明示的ランナブルエンティティの内部挙動

生成されるコードを図 8-11 に示します。データエレメント a が RTE_Read マクロにより port2 から読み取られ、ランナブルエンティティのローカル変数 locvar のメモリロケーションに書き込まれています。また、169 ページの図 8-9 の暗黙的ランナブルエンティティの例では 2 つのアクセスが必要であったのに対し、明示的ランナブルエンティティでは必要な RTE アクセスは 1 つだけです。式への「データ配布」は runnable_expl の例では RTE ではなくローカル変数により行われています。

```
void AR_VERYSIMPLEEXAMPLE_AR_16BIT_runnable_expl (void)
{
   /* user defined local variables */
   uint8 _ASCET_RteStatus;
   sint16 locvar;

   /* no messages have to be received */

   Rte_Read_Port2_a(&(locvar));
   /* runnable_expl: sequence call #5 */
   Rte_Write_Port4_a((sint16)(locvar  *  (locvar  +  (sint32)64) >> 6));
   /* runnable_expl: sequence call #10 */
   if ((_ASCET_RteStatus = Rte_Write_Port4_b((sint16)(locvar  +  (sint32)64))) != RTE_E_OK)
   {
      /* RTE_ExplicitWithStatus-block: sequence call #Explicit write error/ Status #1 */
      _error = (uint8)true;
   } /* end if */

   /* no messages have to be sent */
}
```

図 8-11 暗黙的ランナブルエンティティのコード

明示的通信の場合、RTE 仕様書[1] の 5.4.1 項の規定に従い、RTE アクセスマクロの戻り値を表す列挙型を作成する必要があります。このような列挙型を図 8-12 に紹介します。

Value	Label
0	RTE_E_OK
1	RTE_E_INVALID
2	RTE_E_COMMS_ERROR
3	RTE_E_TIMEOUT
4	RTE_E_LIMIT
5	RTE_E_NO_DATA
6	RTE_E_TRANSMIT_ACK

図 8-12 RTE の標準的な戻り値

RTE にランナブルエンティティを実行させるためには、そのランナブルエンティティをイベントによりトリガする必要があります。各ランナブルエンティティについて 1 つ以上のイベントを指定する必要があり、RTE ジェネレータはこの情報を取得して適切なタスクにランナブルエンティティを割り当てます。ASCET では、イベントはアトミックソフトウェアコンポーネントの“Event Specification”タブに指定されます。この例では、下の 2 つのタイミングイベントが指定されています。

- `Ev_Timing_Explicit_10ms`
- `Ev_Timing_Implict_10ms`

イベント名は AUTOSAR の命名規則に従う必要があります。ランナブルエンティティもイベントもパッケージの内部挙動の一環であり、1 つのパッケージ内のエレメント名はすべて一意でなければなりません。たとえば、ランナブルランナブル A に対応するイベントの名前は、A ではなく ev_A という名前にする必要があります。

注記

ASCET V6.0 では、イベント名とランナブルエンティティ名が異なるかどうかのチェックは行われません。

1. AUTOSAR_SWS_RTE リリース 2.1（www.autosar.org を参照してください）

明示的タイミングイベントと明示的ランナブルエンティティの関連付けを図 8-13 に示します。

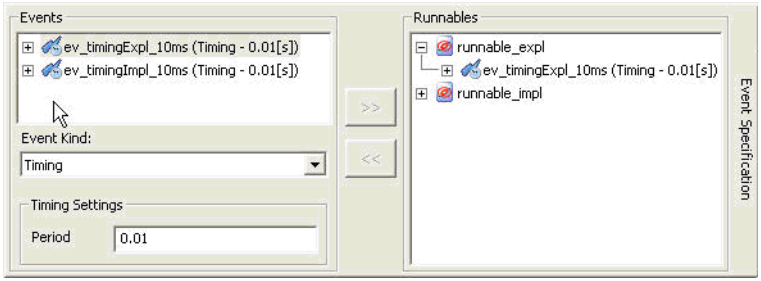


図 8-13 イベントをランナブルエンティティにマッピングする

AUTOSAR のランタイム実行のコンセプトには「モード」という表記法があります。アプリケーションソフトウェアコンポーネントの場合、ランナブルエンティティの実行を「モード無効化に関する依存性」（"mode-disabling dependency"）によって制御することができます。これは、所定のモードの時に RTE が所定のランナブルエンティティを実行しないように定義するものです。現在のモードはポートプロトタイプによりアトミックソフトウェアコンポーネントに伝えられます。

この例では、2 つのモードが定義されています。モードの定義は `OnOffMode` という名前のモード宣言グループ（または略して「モードグループ」）を作成することにより行われます。図 8-14（コンポーネントマネージャ）を参照してください。

図 8-14 モード宣言グループ

図 8-15 に示すように、このモードグループは `ModeInterface` という名前のセンダ - レシーバインターフェース内にモード宣言グループプロトタイプとしてインスタンス化されています。そしてこの `ModeInterface` はこの例のアト

ミックソフトウェアコンポーネント AR_VerySimpleExample 内の ModePort というポートプロトタイプ内にインスタンス化されています。これは 165 ページの図 8-1 のエレメントリストに含まれています。

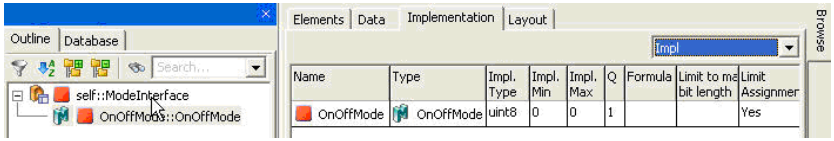


図 8-15 センダ - レシーバインターフェース内のモード宣言グループプロトタイプ

モードと RTE イベントの関連付けは自動的に行われるので、アトミックソフトウェアコンポーネント型によって「モードインターフェース」が 1 つのポートプロトタイプに統合されると、このコンポーネントのすべてのランナブルエンティティはモード無効化依存性に関連付けられていると想定されることになります。図 8-16 を見ると、イベント ev_timingExpl_10ms は OnMode の時に有効になり、イベント ev_timingImpl_10ms は OffMode の時に有効になることがわかります。

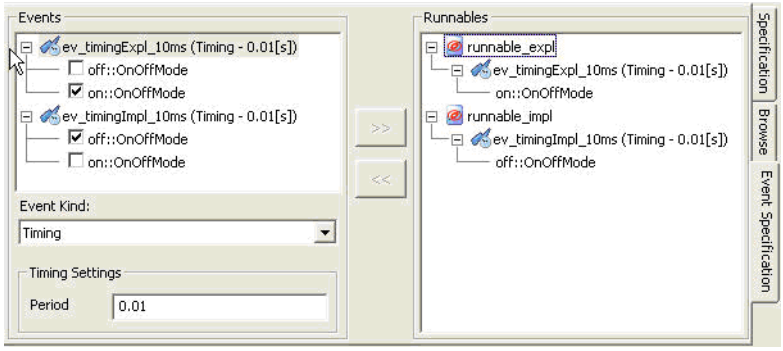


図 8-16 イベント定義エディタ

8.5 レガシープロジェクトの移植

ASCET の古典的なモデリング手法では「モジュール」と「クラス」が用いられますが、アトミックソフトウェアコンポーネント内でこれらのものを使用することができます。図 8-17 は、AR_VerySimpleMigration というアトミックソフトウェアコンポーネント内においてモジュールが再利用されているようすを示しています。このモジュールは SimpleMigrationModule という名前で、そのメッセージ receive_a は set メソッドが有効になっていて、メッセージ send_a および send_b は get メソッドが有効になっているため、各メッセージはアトミックソフトウェアコンポーネント内で受信メッセージまたは送信メッセージとして「表示」されます。その結果、モジュール内の受信メッセージを使

用して、ランナブルエンティティ migration_runnable_module_10ms のデータ受信ポイントから port2 のデータエレメントプロトタイプ a にデータを格納することができます。このモジュールの送信メッセージは、データ送信ポイントの引数として使用できます。

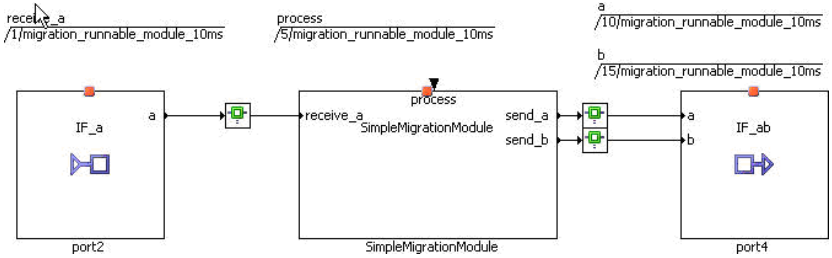


図 8-17 モジュールをアトミックソフトウェアコンポーネントに組み込む

シーケンスコール 5 番からわかるように、アトミックソフトウェアコンポーネントのブロックダイアグラムエディタでは、プロセスをメソッドのように呼び出すことができます。生成されたコードを図 8-18 に示します。モジュールについて見てみると、すべての受信メッセージが RTE から読み取られてからプロセスが呼び出され、送信メッセージが RTE に書き込まれます。

```
void AR_VERYSIMPLEMIGRATION_AR_16BIT_migration_runnable_module_10ms (void)
{
    /* no messages have to be received */

    Rte_Read_port2_a(&receive_a_AR_VERYSIMPLEMIGRATION_AR_16BIT_migration_runnable_module_10ms);
    /* migration_runnable_module_10ms: sequence call #5 */
    SIMPLEMIGRATIONMODULE_AR_16BIT_process();
    /* migration_runnable_module_10ms: sequence call #10 */
    Rte_Write_port4_a(send_a_AR_VERYSIMPLEMIGRATION_AR_16BIT_migration_runnable_module_10ms);
    /* migration_runnable_module_10ms: sequence call #15 */
    Rte_Write_port4_b(send_b_AR_VERYSIMPLEMIGRATION_AR_16BIT_migration_runnable_module_10ms);

    /* no messages have to be sent */
}
```

図 8-18 アトミックソフトウェアコンポーネント内のモジュールについて生成されたコード

図 8-19 からわかるように、プロセス内のアルゴリズムはアトミックソフトウェアコンポーネント型の中で使用されていることを認識していません。つまり、既存のモジュール、さらにはプロジェクトでさえも、1 つのアトミックソフトウェアコンポーネント型に容易に移植することができます。2 つのランナブルエンティティが共通メッセージを共有する 2 つのプロセスを呼び出す場合、ASCET コードジェネレータはそのメッセージの回りに排他領域を生成してメッセージを保護します。「ランナブルエンティティ間の相互変数」という概念は ASCET 6.0

ではサポートされていないので、モジュールをまったく使用しないアトミックソフトウェアコンポーネント型の中でもメッセージを使用できるようになっています。また、代わりに ASCET リソースを使用する方法もあります。

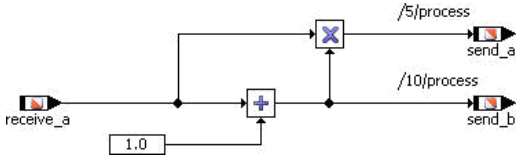


図 8-19 ASCET モジュール内のシンプルなアルゴリズム

アトミックソフトウェアコンポーネント型の内部挙動を階層構造にするために、モジュールだけでなくクラスも使用できます。これを図 8-20 に示します。シンプルなアルゴリズムは、図 8-21 に示すように、戻り値を返さない 1 つのメソッド calc と戻り値 a および b を返す 2 つのメソッドを使用して、ASCET クラスとしてモデリングされます。`SimpleMigrationClass` には入力引数を格納するためのメンバ変数が必要で、そこに格納された入力引数はそれぞれ 1 つの戻り値を提供するメソッドのステートメントの中で使用されます。ランナブルエンティティ内で暗黙的アクセスを使用する場合、データ読み取りアクセスマクロの戻り値を calc メソッド内で引数として直接使用でき、他の 2 つのメソッドの戻り値をデータ書き込みアクセスマクロ内で直接使用できます。生成されるコードを図 8-22 に示します。

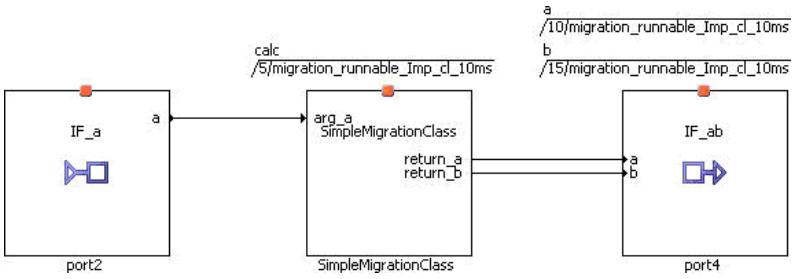


図 8-20 アトミックソフトウェアコンポーネント内で複数のメソッドを持つ ASCET クラスインスタンスを使用する

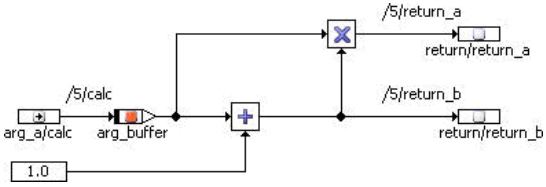


図 8-21 シンプルなアルゴリズムを複数のメソッドに分割したもの

```
void AR_VERYSIMPLEMIGRATION_AR_16BIT_migration_runnable_Imp_cl_10ms (void)
{
    /* no messages have to be received */

    /* migration_runnable_Imp_cl_10ms: sequence call #5 */
    SIMPLEMIGRATIONCLASS_AR_16BIT_calc(Rte_IRead_migration_runnable_Imp_cl_10ms_port2_a());
    /* migration_runnable_Imp_cl_10ms: sequence call #10 */
    Rte_IWrite_migration_runnable_Imp_cl_10ms_port4_a(SIMPLEMIGRATIONCLASS_AR_16BIT_return_a());
    /* migration_runnable_Imp_cl_10ms: sequence call #15 */
    Rte_IWrite_migration_runnable_Imp_cl_10ms_port4_b(SIMPLEMIGRATIONCLASS_AR_16BIT_return_b());

    /* no messages have to be sent */
}
```

図 8-22 古典的 ASCET メソッドを含むクラスを使用する際に生成されるコード

クラスの古典的メソッドモデリングに加え、ASCET 6.0 ではメソッド内で out パラメータを使用することができます。この機能は図 8-23 で使用されています。ここでは、図 8-20 の 3 つの古典的 ASCET メソッドの代わりに、1 つの入力引数と 2 つの出力引数を扱うメソッド `calc_return` が使用されています。

C などのプログラミング言語と同様、出力引数はメソッドに「参照渡し」で渡されます。つまり、メソッドを呼び出す前にメモリをアロケートする必要があります。この例では、`migration_runnable_Expl_class_10ms` に 2 つのローカル変数（`return_a` および `return_b`）があり、アルゴリズムの処理結

果はここに書き込まれます。アルゴリズムは図 8-24 に示すとおりです。これは引数の値に応じて排他的に機能するので、176 ページの図 8-21 に紹介したアルゴリズムのようなメンバ変数の追加は必要ありません。

ただし、少なくともランナブルローカル変数として多少の変数の追加が必要となるため、RTE アクセス用のソリューションとしては「明示的通信」が適しています。そこで、この例では変数（ランナブルローカル）`arg_a` を追加して使用しています。生成されるコードを図 8-25 に紹介します。

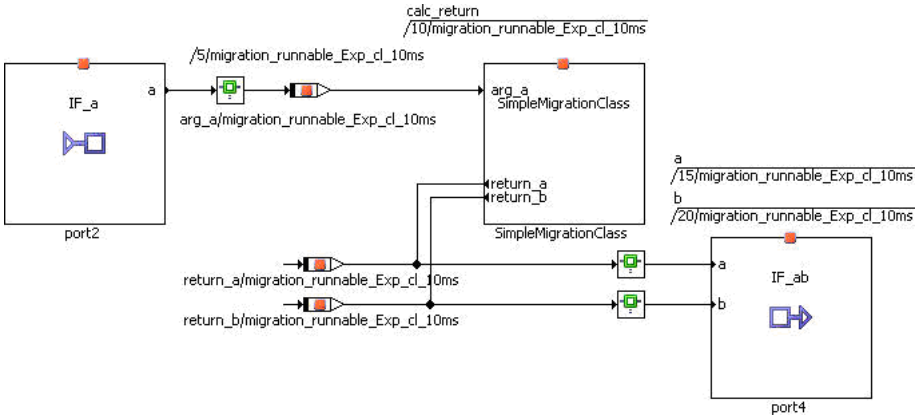


図 8-23 1 つのメソッドを持つ ASCET クラスインスタンスの使用

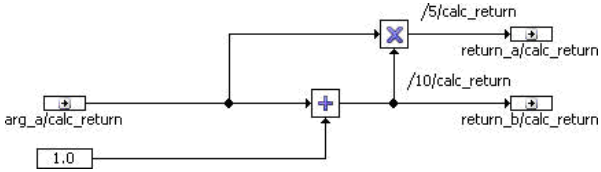


図 8-24 1 つのメソッドの中のシンプルなアルゴリズム

```
void AR_VERYSIMPLEMIGRATION_AR_16BIT_migration_runnable_Exp_cl_10ms (void)
{
    /* user defined local variables */
    sint16 arg_a;
    sint16 return_a;
    sint16 return_b;

    /* no messages have to be received */

    Rte_Read_port2_a(&(arg_a));
    /* migration_runnable_Exp_cl_10ms: sequence call #10 */
    SIMPLEMIGRATIONCLASS_AR_16BIT_calc_return(arg_a, &(return_a), &(return_b));
    /* migration_runnable_Exp_cl_10ms: sequence call #15 */
    Rte_Write_port4_a(return_a);
    /* migration_runnable_Exp_cl_10ms: sequence call #20 */
    Rte_Write_port4_b(return_b);

    /* no messages have to be sent */
}
```

図 8-25 1 つのメソッドを持つ ASCET クラスインスタンスを使用するアトミックソフトウェアコンポーネント用に生成されるコード

9 お問い合わせ先

製品に関するご質問等は、各地域の ETAS 支社までお問い合わせください。

ETAS 本社

ETAS GmbH

Borsigstrasse 14
70469 Stuttgart
Germany

Phone: +49 711 8 96 61-0
Fax: +49 711 8 96 61-105
E-mail: sales@etas.de
WWW: http://www.etas.com/

日本支社

イータス株式会社

〒 220-6217
神奈川県横浜市西区
みなとみらい 2-3-5
クイーンズタワー C 17F

Phone: (045) 222-0900
Fax: (045) 222-0956
E-mail: sales.jp@etas.com
WWW: http://www.etas.com/

その他の支社

上記以外の各国支社につきましては、ETAS ホームページをご覧ください。

各国支社	WWW:	www.etas.com/ja/contact.php
技術サポート	WWW:	www.etas.com/ja/contact.php

索引