ASCET-SE V6.1

ユーザーズガイド

著作権について

Document EC014201 R6.1.0 JP

目次

1 はじめに

ASCET-SE（**ASCET S**oftware **E**ngineering）は、以下のような目的のために使用されます。

- マイクロコントローラターゲットに対応する C コードを生成
- コードにターゲット用オペレーティングシステムまたは実行環境（RTE: RunTime Environment）を統合
- オプションとして、ターゲット用コンパイラとリンカを呼び出して実行プログラムと適合用ディスクリプションファイルを生成（ETAS の INCA などに利用可能）

本書では以下の作業を行う方法について説明されています。

- ASCET-MDで開発されたモデルをASCET-SEでCコードに変換するための属性を定義する
- システムのリアルタイム要件、およびマイクロコントローラターゲット上でそれを実現する方法を定義する
- ASCET が生成したコードにサードパーティ製 C コードを統合する
- ASCET が生成するコードの内容を理解する
- 効率的な方法でモデルを構築する

1.1 本書について

1.1.1 対象ユーザー

本書『ASCET-SE ユーザーズガイド』は ASCET マニュアル（『ASCET 入門』およびオンラインヘルプ）を補足するものです。これからコード生成について理解しようする場合、あらかじめ ASCET の基本的機能と操作方法を理解されていることが前提条件となります。

本書は以下のような方を対象としています。

1. C プログラミング言語の基本を理解している
2. 組み込みコントローラ用 C プログラムのコンパイルやリンクを行った経験がある
3. マイクロコントローラに関する知識がある

1.1.2 本書の構成

本書の構成と内容は、以下のとおりです。

章番号	内容
2	ASCET を使用するうえでの安全上のヒント
3	ASCET-SE をこれから新たに使用するユーザーのための概要説明、およびインストールされるファイルの一覧
4	コード生成のためのモデルエレメントの実装
5	C コード生成に関する設定、コンパイル／ビルド処理の制御とカスタマイズ
6	ASCET-SE で使用する特性テーブル用補間ルーチンの作成
7	アプリケーションのリアルタイムスケジューリングを行うオペレーティングシステムの統合
8	ECU の適合に使用する ASAM-MCD-2MC ファイルの生成
9	ハンドコードされた C コードの統合、ランタイムにおける ASCET-SE の呼び出し、または ASCET-SE からの呼び出し、ASCET ビルド処理への C コードの統合
10	最適化されたコードを生成するためのヒント
11	既存のプロジェクトを別のターゲット用にコピーする方法
12	量子化（固定小数点）演算を使用する際の注意事項
13	ASCET-SE が生成するコードの特定と構成、モデルの各パートと C コードとの対応付けなど
14	ASCET-SE の動作原理
15	ASCET-SE のコード生成に関する制約事項
16	お問い合わせ先

1.1.3 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。

表記例	説明
“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。
`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。

本書で採用している記述および表記の規則は、『ASCET 入門ガイド』の「はじめに」という章に記載されています。

また、PDF ファイルでお読みいただいている場合、「... を参照してください」と書かれた部分の「...」の部分には、その参照先へのリンクが設定されていますので、その部分をクリックするだけで容易に参照先へジャンプできます。

1.2 インストール

ASCET-SE のインストールについては『ASCET インストールガイド』をお読みください。

他の ETAS 製品と同様、ASCET-SE を使用するには、有効なライセンスファイルが必要です。ライセンスを取得するポータルサイトの URL は、製品ご購入時のエンタイトルメントレターをご参照ください。ライセンスのインストールと管理は ETAS ライセンスマネージャで行います。

ASCET-SE のインストールはサイレントモードで行うことができます（詳細は『ASCET インストールガイド』の「コマンドラインからのインストール」の項を参照してください）。インストールするターゲットの選択は、環境変数を定義するか、または `install.ini` の `[SilentInstallation]` セクションを編集することによって行えます。

環境変数を使用する場合は、ASCET-SE をインストールする前に各変数を環境内にセットしておく必要があります。その際は、以下の例のようなバッチファイルを使用することをお勧めします。

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

上記の各変数は、それぞれ ASCET-SE の各ターゲットに対応しています。`true` にセットされたターゲットはインストールされ、`false` にセットされたターゲットはインストールされません。指定されていないターゲットについては、デフォルトとして `true` にセットされているものとみなされます。

`TRG_SELF_CONTAINED_MODE` は、複数のターゲットがファイルを共有するかどうかを指定するものです。これが `true` にセットされていると、インストールされたすべてのターゲットディレクトリ（`trg_*`）に共通ターゲットファイルがコピーされるので、このファイルをターゲットごとに個別に変更することができます。

`false` の場合は、共通ターゲットファイルは `common-se` という共有ディレクトリにインストールされます。このファイルを変更すると、その内容が全ターゲットに適用されます。

環境変数を設定する代わりに `install.ini` ファイル内のインストールパラメータを編集してターゲットを選択することもできます。そのためには `[SilentInstallation]` セクション内のエントリを以下のように編集します。

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

`install.ini` の内容は、環境変数をオーバーライドします。

1.3 略語と用語

ASAM-MCD

Association for **S**tandard of **A**utomation and **M**easuring 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（Measuring）、適合（Calibration）、診断（Diagnosis）についてのワークグループ

ASAM-MCD-2MC ファイル

適合作業に使用されるプログラムデスクリプションの標準フォーマット

ASCET

ETAS の 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／設計ツール

ASCET-SE

ASCET Software **E**ngineering - ASCET マイクロコントローラターゲット組み込みパッケージ。各種マイクロコントローラ で実行可能なアプリケーションコードを生成します。

AUTOSAR

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

BLOB

Binary **L**arge **Ob**ject — ASAM-MCD-2MC 内に記述される、インターフェース固有の情報

CPR

Code **P**roduction **R**ules（コード生成ルール）

ECCO

Embedded **C**ode **C**reator and **O**ptimizer（組み込みコードクリエータおよびオプティマイザ）

ECU

Electic **C**ontrol **U**nit（電子制御ユニット）

ESDL

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

ETK

エミュレータテストプローブ（ドイツ語：**E**mulator-**T**est**k**opf）

INCA

INtegrated **C**alibration and **A**cquisiton System — ETAS の適合・測定ツール

OIL

OSEK **I**mplementation **L**anguage（OSEK 実装言語）

OS

Operating **S**ystem（オペレーティングシステム）

OSEK

自動車エレクトロニクス向けオープンシステムについてのワークグループ（ドイツ語：Arbeitskreis **O**ffene **S**ysteme für die **E**lektronik im **K**raftfahrzeug）

OSEK オペレーティングシステム

OSEK 規格に準拠したオペレーティングシステム

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）— ランタイムに 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-OSEK

ETAS の OSEK 互換のリアルタイムオペレーティングシステム

RTA-RTE

ETAS が実装する AUTOSAR **R**un**T**ime **E**nvironment（ETAS の AUTOSAR 実行環境）

RTE

AUTOSAR RTE（AUTOSAR **R**un**T**ime **E**nvironment）― 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含む実行環境

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）

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規則を記述したもので、変換式（線形変換）、値の上下限値を指定するインターバル、および必要に応じてその他の詳細情報（メモリ割り当てなど）で構成されています。

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いくつか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。

クラス

ASCET のコンポーネントの型の 1 つです。ASCET におけるクラスは、オブジェクト指向のクラスに相当します。クラスの機能は、メソッドとして記述されます。

コード生成

物理モデルから実行コードへの変換の第 1 ステップです。ここでは物理モデルがターゲットごとに異なる ANSI C コードに変換されます。

コンポーネント

ASCET モデルにおける、再利用可能なファンクションの基本単位です。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の形式で記述できます。各コンポーネントは、エレメントを演算子で結合して構築された機能単位です。

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

プロセスをタスクに割り当てて、オペレーティングシステムによるタスクの起動条件を定義することです。

スコープ

各エレメントのスコープは、ローカル（コンポーネント内でのみ使用できます）またはグローバル（プロジェクト全体で使用できます）のいずれかです。

ターゲット

プログラムや実験が実行されるハードウェアです。ASCET-SE では各種マイクロコントローラが「ターゲット」となり、コンパイラの種類も選択できます。

タスク

オペレーティングシステムによってスケージューリングされる一連の機能のエントリポイントです。各タスクには優先度、スケジューリングモード、動作モードなどの属性があります。ASCET-SE においてはタスクの機能は複数のプロセスの集合として定義されます。タスクが実行されると、そのタスクのプロセスが指定された順に実行されます。

トリガ

オペレーティングシステムの場合はタスクを、ステートマシンの場合はアクションをアクティブにする事象を指します。

パラメータ

パラメータ（特性値、カーブ、マップ）は、ASCET モデル内で実行される計算では値を変えることのできないエレメントですが、実験を行ってこれらの値を適合することができます。

プロジェクト

組み込みソフトウェアシステム全体を記述したものです。ファンクション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する設定、および通信を定義するバインディングシステムが含まれます。

プロセス

オペレーティングシステムにより実行されるタスクから呼び出される機能単位です。プロセスは ASCET モジュール内で定義され、引数と戻り値を持ちません。プロセスの入力と出力にはメッセージが使用されます。

変数

ASCET モデルの実行時に読み書きできるエレメントです。変数の値は、適合実験において測定することも可能です。

メソッド

オブジェクト指向プログラミングのコンセプトに基づくもので、クラスの機能を記述する単位です。メソッドは引数と 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におけるデータの妥当性を保証するための、ASCET のリアルタイム言語構成体です。

メモリクラス

抽象メモリ領域の名前を指し、これらが後に ECU メモリ上の実際のアドレス空間に割り当てられます。

モジュール

ASCET のコンポーネントの型の 1 つです。モジュールには、オペレーティングシステムから実行される複数のプロセスが含まれます。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` はブール値（`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。

優先度

OS で扱われる各タスクには数値で表される優先度が割り当てられていて、数値が大きくなるほど優先度は高くなります。レディタスクは優先度に従って順に実行されます。

リソース

組み込みシステム内で相互排他的に使用されるパーツ（タイマなど）のモデリングに使用されます。リソースはアクセスする前に必ず予約する必要があり、それを使用するタスクの処理が終わるとそのリソースは再び解放されます。

リテラル

ブロックダイアグラム内で使用されます。リテラルには 1 つの値（連続変数や論理変数など）としてそのまま扱われる文字列が格納されます。

2 安全なアプリケーションソフトウェア設計のためのヒント

安全で再現性のあるマイクロコントローラコードの生成を実現するため、ASCET と ASCET-SE にはさまざまなメカニズムが備わっていますが、細部においては、コードジェネレータによってチェックできない部分もあります。原因は、技術的な理由であったり、コード実装時に正確な判断が行われないことによる場合もあります（たとえばモデルの正確さはその使用方法により左右されるため）。

このような条件を前提として、この章では、ASCET でアプリケーションソフトウェアを設計する際に注意するべき点を挙げて解説します。

2.1 補間ルーチンについて

現バージョンの ASCET-SE に含まれている補間ルーチンは、「サンプルオブジェクトライブラリ」として提供されているものです。これらのルーチンはユーザーが補間ルーチンの扱いを学習するためのサンプルとして作成されたもので、実際の ECU ソフトウェア用に作成されたものではありません。

危険

ETAS グループ各社およびその子会社、代理店、支社は、上記の補間ルーチンを原因とするすべての損害や負傷について、その責任を負いません。

2.2 FPU の使用

ASCET-SE には浮動小数点コードを生成する機能があり、これは、浮動小数点演算ユニット（FPU）を備えたマイクロコントローラを使用する場合に利用できます。

しかし浮動小数点を使用しないアプリケーションの場合は、タスクの切り替え時に FPU の浮動小数点レジスタを保存しないことによって実行時間やスタック消費量を削減することができます。RTA-OSEK にはこのような最適化の機能があり、あるタスクのプロセスやメソッドが FPU をまったく使用しない場合、ASCET-SE は OS コンフィギュレーションの最適化を自動的に有効にします。

メソッドやプロセスが FPU を使用するかどうかという情報は、実装情報内のフラグで定義します。デフォルト状態ではこのフラグは ON（FPU を使用）になっていますが、FPU を使用しないプロセス／メソッドについてはこのフラグを OFF にすることができます。

ただしこのフラグを OFF にする場合は、必ずユーザーの責任においてそのプロセス／メソッドで浮動小数点が使用されないことを確認してください。

もし浮動小数点を使用するプロセス／メソッドについてこのフラグが OFF になっていると、タスク切り替え時に浮動小数点に関する情報は保存されず、無効となってしまい、アプリケーションが予期しない挙動を示す恐れがあります。

FPU を使用するかどうかが明確でない場合、このフラグは ON にしておいてください。

2.3 Non-Volatile エレメント

ASCET-SE は、5.3 項 「`memorySections.xml` ファイル」に説明されているように、複数のメモリクラスを扱うことができます。各メモリエリアは、Volatile（揮発性メモリ）または Non-Volatile（不揮発性メモリ）のいずれかであるため、ASCET-SE は各メモリクラスに両方の型のエレメントが混在していないかどうかをチェックし、混在が認められた場合はエラーメッセージが出力されます。

Non-Volatile 変数として指定されたものは、ECU がリブートされても ECU メモリ上で消去されずに保存されます。このため、仮にデータエディタで初期値を指定したとしても、実際には初期化されません。

初期化が必要な場合は、ユーザーの責任において、機能記述の一部として明示的に行ってください。

2.4 ユーザー定義のデータ型の使用

ユーザー定義されたデータ型を使用する場合、`a_user_def.h` というファイル内で宣言されているその型が、対応する ASCET のデータ型がカバーする値の範囲に合っている必要があります。たとえば `sint8` に対応するユーザー定義のデータ型は、値の範囲 -128 ～ 127 を保持できるサイズでなければなりません。

ASCET はユーザー定義されたデータ型のサイズの妥当性をチェックできないため、開発プロセス内の他の工程（コードレビューなど）において必ず宣言内容がチェックされるようにしてください。

3 コード生成入門

ASCET-SE のコンフィギュレーションには、以下の 2 つの重要な要素が含まれます。

1. 物理モデル
   ターゲットに依存しない「機能」を記述したものです。ASCET では、ブロックダイアグラム、ステートマシン、テキスト（ESDL ディスクリプション）など、いくつかの方法でモデルを表現できます。一般的に、このモデルの開発とテストは ASCET-MD で行います。
2. インプリメンテーション
   物理モデルからターゲット上で実行されるコードへのマッピング情報です。

コード生成においては、インプリメンテーションの情報に基づき、物理モデルがマイクロコントローラ用の C コードに変換されます。この変換過程で重要なのは、浮動小数点演算を使用するアルゴリズムから固定小数点演算アルゴリズムへの変換です。

また、コード生成において、オペレーティングシステム（OS）またはランタイム環境（RTE）を物理モデルのリアルタイム要件（サンプリングレートやモデル間通信など）に対応させるためのコンフィギュレーションファイルも生成されます。つまりこれらのファイルは、ASCET から OS または RTE への「要求事項」が定義されたものです。

ASCET-SE は以下の 2 種類のリアルタイムターゲット向けのコード生成をサポートしています。

1. OSEK オペレーティングシステム（OSEK OS）
2. AUTOSAR ランタイム環境（AUTOSAR RTE）

ASCET-SE に含まれる「OSEK OS」のサポート機能は、ETAS の RTE-OSEK を対象としていますが、他の OSEK OS や、OSEK OS と同等のスケジューリングモデルを採用しているその他の OS 用のコードも生成できます。

3.1 ASCET-SE のコンポーネント

ASCET-SE は以下のコンポーネントで構成されています。

- ASCET-SE コードジェネレータ：ASCET エクスパンダおよび ECCO（**E**mbedded **C**ode **C**reator and **O**ptimizer）で構成されます。
- コンフィギュレーションファイル：特定のターゲット用にコード生成機能をカスタマイズするためのもので、C コードファイル、OS コンフィギュレーションファイル、測定・適合用 ASAM-MCD-2MC（A2L）ファイルなどが含まれます。
- Make ファイル: ターゲット固有の OS 生成や C コンパイラツールチェーンを呼び出すためのファイルです。
- HEX ファイルリーダ

これらのコンポーネントには以下の機能があります。

- ASCET

  サポートされているターゲットとコンパイラごとに固有の設定を提供し、これによって各ターゲット用のプロジェクトを定義することができます。OS 設定（タスクやタイミング情報など）の設定には ASCET の OS エディタを使用します。

  注記

  ASCET のモデリング機能は ASCET-SE 製品には**含まれません**。モデリングを行うには別製品の ASCET-MD が必要です。

- エクスパンダ

  各ターゲット用の中間コードを作成します。これには変数とパラメータの完全な定義や初期設定が含まれます。

- ECCO コードジェネレータ

  エクスパンダが出力した中間コードから最終的な C コードを作成します。ここでさまざまな「グローバルな最適化」が行われます。

  ここでは `temp.oil` という OIL ファイルが作成されます。このファイルは RTA-OSEK への入力として必要です。さらに、RTA-OSEK に必要な一般的な OIL オブジェクトが含まれるテンプレート OIL ファイルがユーザーから提供される必要があります。

  RTA-OSEK により、ASCET で生成した `temp.oil` とユーザーから提供されたテンプレート OIL ファイルとがマージされ（7.2 項を参照してください）、OS の `*.c` および `*.h` ファイルが生成されて Make 処理に渡されます。

  注記

  RTA-OSEK OS のコンフィギュレーションツールおよびターゲットプラグインは ASCET-SE 製品には**含まれません**。詳しくは ETAS の営業窓口までお問い合わせください。

- Make 処理

  Make 処理により、ボタンをクリックするだけで実行ファイルを自動生成できます。この処理では各種ファイル（コンパイラファイル、リンカファイル、ロケータファイル、ログファイル）が解析され、実験ターゲットの

コード生成の場合と同様に、エラーが検出されるとエラーメッセージが出力されます。サードパーティ製の OSEK OS または ANSI C ターゲットを使用する場合は、さらに他のツールや C コードの統合が必要になります。

注記

ターゲット用コンパイラおよびリンカは ASCET-SE 製品には**含まれません**。これらはコンパイラの販売元に別注してください。サポートされるコンパイラとリンカのバージョンは、ASCET-SE のリリースノートに記載されています。

- HEX ファイルリーダ

  リンカにより生成された実行ファイルを読み取るパーサーです。アドレス情報が抽出され、標準化された内部構造体として表現されます。このアドレス情報は ASAM-MCD-2MC ファイルの生成に必要です。

注記

ここでいう「アドレス」とは、ASCET エレメントとして宣言されたエレメントのアドレスのみを指します。

3.2 コード生成プロセスの概要

次の図は、ASCET-SE ワークフローの概要を示しています。

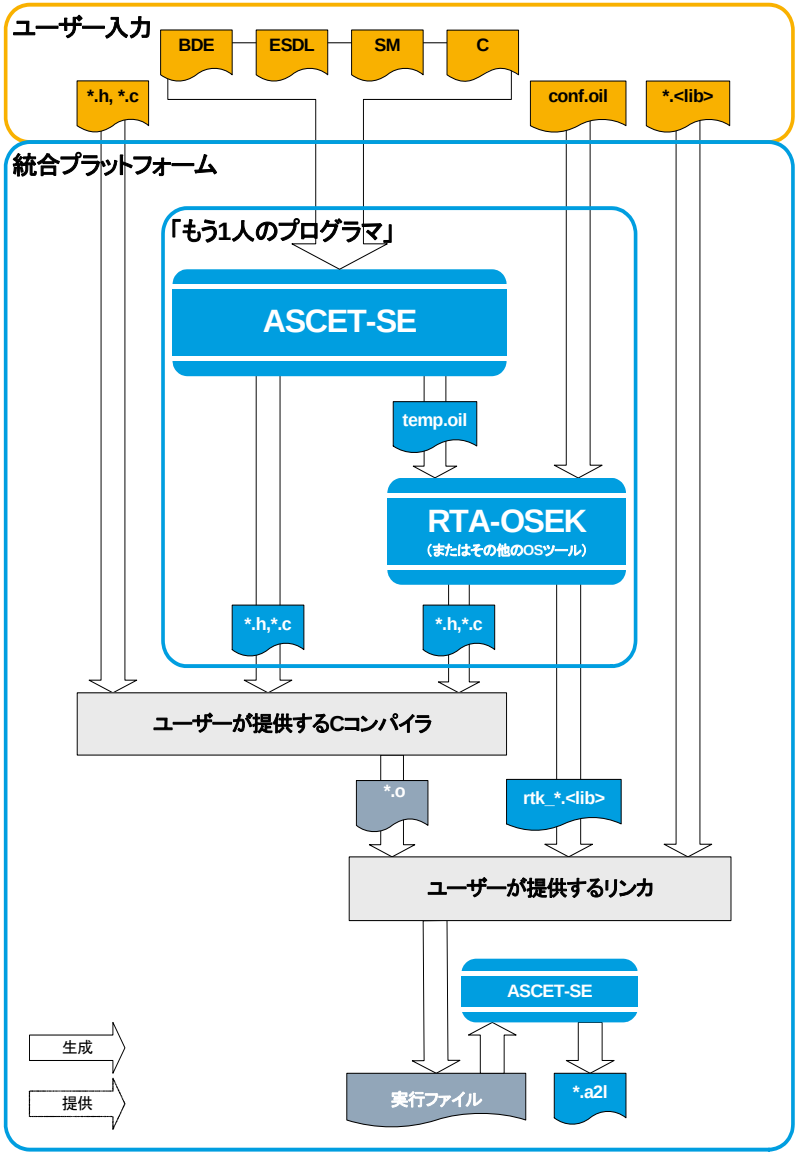


図 3-1 ASCET-SE の基本ワークフロー

3.2.1 組み込み制御ソフトウェアのコード生成

C コードは「完全なモデル」（つまり 1 つの ASCET プロジェクト）から特定のターゲットを対象として生成されます。この際、以下に対応する C ソースコードファイルが生成されます。

- プロジェクト自体
- 各モジュール
- 各クラス
- 各 OS タスクボディ

このコードを所定のコンパイラおよびリンカによりコンパイルすれば、所定のマイクロコントローラを搭載した ECU で実行することができます。このコードには OS に必要なすべての変数とデータの定義や構造体が含まれています。ソフトウェアアーキテクチャ、つまりコード内のモデル構造のマッピングは、すべての ASCET-SE ターゲットについて共通です。

このようにして生成されたコードから、ビルド処理によって最終的な実行コードを作成します。

ASCET-SE は、実行コード作成について以下の 2 通りのユースケースに対応しています。

1. 「もう 1 人のプログラマ（“additional programmer”）」
   生成された C コードが外部ファイルにエクスポートされ、外部のビルド処理によってビルドされます。
2. 「統合プラットフォーム（“integration platform”）
   ASCET-SE はユーザーのコンパイラツールチェーンを使用して実行コードをビルドします。こちらについては以下の項で説明します。

3.2.2 コンパイラ、リンカ、ロケータ

「統合プラットフォーム」としてのユースケースにおいては、コンパイラ、リンカ、ロケータからなるターゲットツールチェーンが ASCET により駆動されるので、統合開発環境（IDE：**I**ntegrated **D**evelopment **E**nvironment）でのソフトウェア開発と同様の方法でプロジェクトをビルドすることができます。

ASCET では“Make”システムを使用してビルド処理を制御しますが、操作方法は実験ターゲット用のビルドと同じです。所定のメニューコマンドを選択するとビルドが開始され、正常にビルドが完了するとプロジェクトの実行プログラムが生成されるので、それを ECU に書き込むことができます。

3.2.3 ASAM-MCD-2MC の生成

ビルド処理の最後に、HEX ファイルリーダにより、生成された HEX ファイルから ASCET モデル内で宣言されているすべての変数とパラメータのアドレスが抽出されます。

所定のメニューコマンドを使用して ASAM-MCD-2MC ディスクリプション（一般に「A2L ファイル」と呼ばれます）を生成し、このファイルによりシステムの情報を適合システム（ETAS の INCA など）に供給します。

3.3 コード生成の手順

ASCET-SE で行うコード生成は一般的なコンパイル処理に似ていて、以下の 2 つのフェーズからなります。

第 1 フェーズでは、エクスパンダと呼ばれる「フロントエンド」がブロックダイアグラム、ESDL エディタ、C コードエディタなどで定義された ASCET モデルを中間コードに変換します。ここでは物理モデルが量子化モデルに変換されます。各モジュールと各クラスが個別に処理され、最適化もローカルに行われます。

第 2 フェーズでは、ECCO（**E**mbedded **C**ode **C**reator and **O**ptimizer）と呼ばれる「バックエンド」が、ASCET プロジェクトに対して「グローバルビュー」によるグローバルな最適化を行います。その後、ECCO はターゲットコンパイラの「イントリンシック」（コードをメモリセクションに配置するプラグマ命令など）を追加して、中間コードを C コードに変換します。ECCO は一連のコード生成ルール（CPR）に従ってこの変換を行います。このルールは所定の制約内でユーザーが変更することができるので、要件にあわせたコード生成を行うことができます。

この第 2 フェーズの成果として得られた一連の C ファイルを、ターゲット用コンパイラでコンパイルします。

3.3.1 ターゲットの選択

ASCET-SE のインストール時に、インストールするターゲットを選択します。ASCET-SE はインストールされたすべてのターゲット用にコードを生成できます。

各ターゲットは、以下の例のように、ターゲットのマイクロコントローラファミリの名前が割り当てられたディレクトリ `<install_dir>¥target¥trg_<targetname>` にインストールされます。

```
<install_dir>¥target¥trg_c16x
<install_dir>¥target¥trg_mpc55xx
```

移植可能な ANSI-C コードを生成するには、マイクロコントローラに依存しない ANSI-C ターゲットと呼ばれる特殊なターゲットを使用します。このターゲットは以下のディレクトリにインストールされます。

```
<install_dir>¥target¥trg_ansi
```

この場合、他の組み込みターゲットとは異なり、生成されるコードには、セグメント化（ページ化）されたハードウェアアーキテクチャに基づくメモリマッピングやデータアクセスに関するコンパイラ固有の「イントリンシック」は含まれません。

ANSI-C コードは、ASCET-SE でサポートされていないターゲットを使用する場合のベースとして使用できます。

ASCET-SE とともにインストールされたターゲットを実際のマイクロコントローラや OS に合わせてカスタマイズする必要がある場合は、本書の当該箇所に記載されているヒントに従ってください。特に、以下の項を参照してください。

- 3.3.5 項「メモリクラスに関する設定」
- 5.2 項「`target.ini` ファイル」
- 5.3 項「`memorySections.xml` ファイル」
- 7.6 項「OSEK 未対応のオペレーティングシステムとのインターフェース」

3.3.2 パスの設定

コンパイラと OS ツールチェーンを使用してアプリケーションをビルドするには、コンパイラと OS がインストールされているパスを ASCET に設定しておく必要があります。これらのツールが前もって PC にインストールされている場合は、ASCET のインストール時にそれらのパスを自動検索することができます。

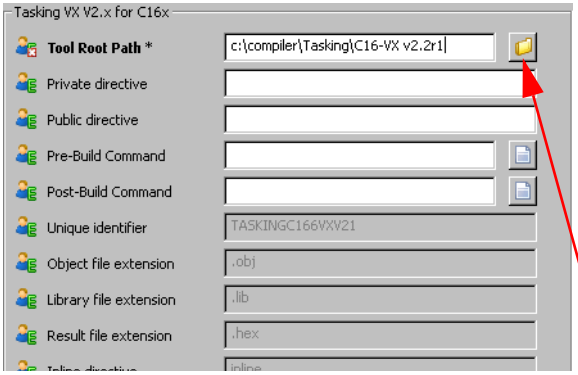
注記

ASCET プロジェクトをビルドする際は、自動検索されたツールチェーンパスの正当性を確認しておくことをお勧めします。またツールのバージョンが ASCET バージョンに対応していることを確認してください。

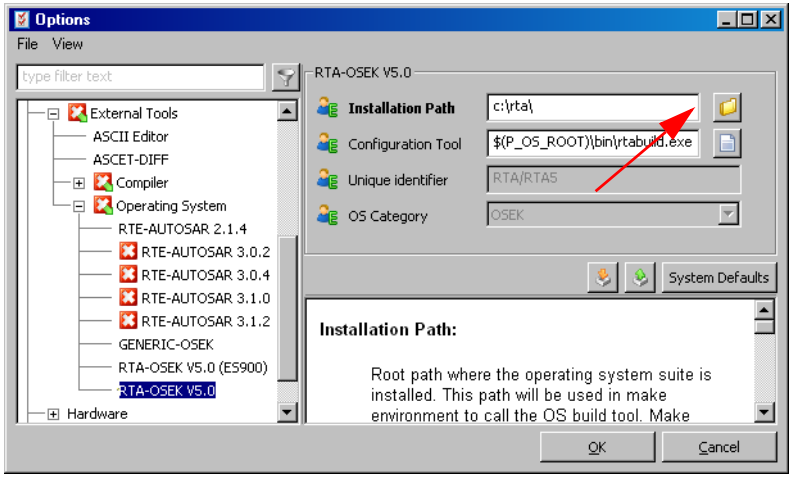
コンパイラと OS ツールチェーンのパスを設定する：

- ASCET コンポーネントマネージャで、**Tools → Options** を選択します。

  “Options”ダイアログボックスが開きます。
- “External Tools/Compiler”ノードを開きます。
- 使用するコンパイラ用のサブノード（“Tasking Vx V2.x for C16x”など）を選択します。
- “Tool Root Path”フィールドの右側のボタンをクリックします。



- “Path Selection”ダイアログボックスでコンパイラとリンカのパスを選択し、ダイアログボックスを閉じます。
- “Options”ダイアログボックスの“Operation System”ノードを開きます。
- 使用する OS のサブノードを選択し、OS がインストールされているパスを選択します。



- **OK** をクリックして変更内容を有効にします。

3.3.3 コード生成に関する設定

コード生成に関する設定は、ASCET のプロジェクトエディタでプロジェクトごとに設定します。ここではビルド処理においてどのコンパイラと OS を使用するかを指定します。

プロジェクトオプションを設定する：

- プロジェクトエディタで **Project Properties** ボタンをクリックします。

  “Project Properties”ウィンドウが開いて、“Build”ノードが表示されます。
- ターゲットとコンパイラを選択します。

  マイクロコントローラターゲットの場合、“Code Generator”コンボボックスの選択肢は `Object Based Controller Implementation` のみです。

- OS を選択します。

  以下のようなオペレーティングシステム（またはその一部）が使用可能です。

`RTA-OSEK Vx.y`	ETAS の OSEK オペレーティングシステム（バージョン x.y）とのインターフェースのためのコードとコンフィギュレーションデータが生成されます。
`GENERIC-OSEK`	一般的な OSEK 用のコードとコンフィギュレーションデータが生成されます。使用する OS 固有のコンフィギュレーション設定が必要になる場合があります。
`RTE-AUTOSAR x.y`	AUTOSAR RTE（バージョン x.y）とのインターフェースのためのコードとコンフィギュレーションデータが生成されます。

- 各サブノードでコード生成オプションを設定します。
- **OK** をクリックして変更内容を有効にします。

コード生成設定についての詳細は、ASCET オンラインヘルプを参照してください。

3.3.4 オペレーティングシステムに関する設定

OS コンフィギュレーションは、OS を ASCET にどのように統合するかを定義するものです。OS の統合は、プロセスをタスクにマッピングして、タスク属性設定を定義し、さらに割り込み属性を定義する、という手順で行います。

コンフィギュレーション設定はプロジェクトエディタの“OS”タブで行います（プロジェクトエディタについての詳細は ASCET オンラインヘルプを参照してください）。

ASCET は、OSEK OS のような優先度ベースのプリエンプティブ OS を想定しています。各タスクにマッピングされたプロセスのスケジューリングは、タスクのスケジューリングにより決まるため、ランタイムにおいて OS がどのようにタスクをスケジューリングするかを理解することが重要です。OS の統合に際しての基本ガイダンスが 7.1 項「スケジューリングと優先度機構」に記載されています。ここに記述されている制約事項が守られていないと、コード生成時にエラーが発行されます。

注記

オペレーティングシステムに RTE-AUTOSAR が選択されている場合、ANSI-C コード生成だけがサポートされているので、OS 設定は必要ありません。RTE-AUTOSAR を使用するプロジェクトを新しく作成する際、“OS”タブに値を設定しても、その内容はプロジェクトエディタを終了すると“OS”タブ自体とともにすべて削除されます。

3.3.5 メモリクラスに関する設定

組み込みマイクロコントローラの場合、PC とは異なり、一般的にデータとコードをメモリの特定のセクション（さらには特定のアドレス）に配置する必要があります。プログラムコードおよび静的データ（定数など）は ROM に配置されるのが普通です。動的データ（変数）は RAM に配置されなければなりません。

また、マイクロコントローラによっては、たとえば、一部のセクションは 8 または 16 ビットアドレスでアドレシングでき、それ以外のセクションは 32 ビットアドレスでしかアドレシングできないなど、アドレシング方法の異なるメモリセクションを一緒に配置できるものもあります。

コントローラメモリ内のエレメントの配置は、インプリメンテーションで割り当てられている「メモリクラス」により決まります。ASCET データモデルにおいて、メモリクラスは、ユーザーが自由に選択できる以下の抽象名で実装されます。

- IRAM - 内部 RAM
- IFLASH1 - 内部フラッシュ ROM メモリの第 1 バンク
- IFLASH2 - 内部フラッシュ ROM メモリの第 2 バンク
- NEAR_RAM - 8 ビットアドレスでアドレシングできる RAM
- FAR_ROM - 32 ビットアドレスでアドレシングできる ROM

名前の定義や、C コードを正確にマークアップするためのコンパイラ固有名への変換の定義は、ターゲットディレクトリにある `memorySections.xml` というファイルに格納されています。このファイルはターゲットごとに用意されていて、そのターゲットの典型的な内容になっています。

`memorySections.xml` 内に定義されているセクション名は、各 ASCET エレメントの“Implementation”ダイアログボックスに反映され、インスタンスのメモリロケーションの選択肢として表示されます。

コード生成の第 2 フェーズにおいて、ASCET-SE は `memorySections.xml` 内の変換情報を使用して、生成された C コードに正しいイントリンシック（通常は `#pragma` 文）を追加します。

メモリクラスの使用法についての詳細は、5.3 項「`memorySections.xml` ファイル」を参照してください。

各ロケーションへの実メモリアドレスの割当ては、リンカ制御ファイル内で行われます。

3.3.6 ターゲット初期化コード

ASCET の各ターゲットには、単純なターゲットコンフィギュレーションが定義されたサンプルアプリケーションが用意されています。プロジェクトを構築する際、ASCET-SE はデフォルトとしてこのサンプルアプリケーションを使用します。使用されるファイルは以下のものです。

```
<install_dir>¥target¥example¥target.[hc]
<install_dir>¥target¥example¥system_counter.c
```

これらのファイルには、メインプログラムのほか、ターゲットハードウェアを初期化してタスクスケジューリング駆動用の 1ms 周期のタイマ割り込みを発生させるためのコードが含まれています。割り込みハンドラ自体は `system_counter.c` に含まれています。実際のプロジェクトにおいては、これらのコードを適切に調整する必要があります。

ASCET で別の割り込みが定義された場合は、割り込みソースを設定し、割り込み優先度レジスタを初期化したりするようなターゲットコードを追加する必要があります。詳細については、OS のマニュアルを参照してください。

ASCET は、C プログラム実行時にメモリセクションが正しく初期化されることを前提としています。デフォルトでは、ASCET はコンパイラとともに提供された C のスタートアップコード（メインプログラムの前に実行されるコード）を使用して C 環境を初期化します。

3.3.7 コンパイルとリンクのカスタマイズ

特定のハードウェアターゲット用にコンパイルやリンクの内容をカスタマイズするには、リンカ／ロケータコントロールファイル内に以下の設定が必要です。

- `memorySections.xml` 内に定義されている ASCET メモリクラスを、適用可能な物理メモリスペースに配置します（79 ページの「リンカ／ロケータの制御」参照）。
- OS 用のメモリセクションを物理メモリスペースに配置します。OS にスタックポインタの位置を知らせる必要もあるので、具体的な手順については、OS のマニュアル（RTA-OSEK の場合は各ターゲット用 RTA-OSEK バインディングマニュアル）を参照してください。

コンパイラとリンカの呼出しに関するカスタマイズは `project_settings.mk` という Make ファイルで行います（5.4.1 項参照）。この Make ファイルで、補助的なヘッダファイルやコンパイル済みオブジェクト、ユーザーライブラリ（ドライバ、外部コード、補間ルーチンなど）などを統合したり、コンパイラ／アセンブラ／リンカオプションやその他の設定を定義することができます。

ターゲットによっては時間測定に関する設定も必要です。

- 入力周波数とタイマプリスケール係数を `project_settings.mk` ファイルに定義してください（5.4.1 項参照）。

`target_settings.mk` という Make ファイル（5.4.2 項参照）が存在する場合、その内容はできる限り変更しないようにしてください。このファイルにはコンパイラ固有の設定が定義されています。

3.3.8 ソースファイル／実行ファイルの生成とターゲット上での実行

アプリケーションをターゲットコントローラ上で実行できるようにするには、実行形式のファイルを作成する必要があり、適合ツールを使用する場合は ASAM-MCD-2MC ファイルも必要です。本項では、ASCET においてソースコードと実行コード、および ASAM-MCD-2MC ファイルを生成する手順について再確認します。

ターゲットによっては以下の作業も必要となる場合があります。

- メモリレイアウトをASAM-MCD-2MCデータファイル`mem_lay.a2l` に入力する（8.2 項参照）
- ETK 用グローバルブロック(TP blob と QP blob)を ASAM-MCD-2MC データファイル `aml_template.a2l` および `if_data_template.a2l` に入力する（8.3 項参照）

それぞれの手順を以下に説明します。

ソースコードを生成する：

注記

物理実験の場合に限り、プロジェクトがなくても各 ASCET モジュールのコードを生成してシミュレートすることができますが、他のコードジェネレータの場合はモジュールをプロジェクトに組み込む必要があります。その際、各クラスまたはモジュールごとに「デフォルトプロジェクト」と呼ばれるプロジェクトを定義します。プロジェクトが存在しないと、インポートされるエンティティの変換式やインプリメンテーション情報を得ることができません。

- 各コンポーネントの用のエディタで **Build → Generate Code** を選択し、ソースコードを作成します。

  コードは、プロジェクト全体、または任意のコンポーネント（モジュールまたはクラス）について生成することができます。この際、必要なすべてのコンポーネントが自動生成されます。
- ソースコードをファイルに保存するには、**File → Export → Generated Code** を選択します。

  この操作を行うまでは、コードは ASCET 内部にのみ存在し、ファイルには保存されません。

プロジェクトの実行コードを生成する：

- 実行ファイルを作成するには、プロジェクトエディタで **Build → Build** を選択します。

  プロジェクト全体のコードが生成され、コンパイルとリンクが行われます。この工程でエラーが 1 つも発生しなかった場合、HEX フォーマットの実行ファイルが `temp.*` という名前で作成されます。コード生成時に作成されたソースコードとオブジェクトコードは、ASCET データベースに格納されます。

実行ファイル生成時、ソースコードを含むすべてのファイルはデフォルト設定で <install_dir>¥cgen¥ ディレクトリに一時的に格納されますが。“Options” ダイアログボックの **Keep files in Code Generation Directory** オプション（オプションについての詳細はオンラインヘルプを参照してください）がオンになっていないと、このディレクトリ下にあるファイルは、ASCET を終了する時にすべて削除されます。

注記

上記のオプションをオンに変更しても、現在のセッション中はその設定変更が有効にならないため、生成されたファイルを保持したい場合は、ASCET を閉じる前にこれらのファイルを別のディレクトリにコピーしてください。
<install_dir>¥cgen¥ に生成されるファイルは、コンパイルされる C ソースファイルではありません。
ソースコードだけを保存するには、**Fle → Export → Generated Code → *** を使用します。コード生成プロセスでそれらのソースコードがデータベースにすでに格納されていた場合は、別のディレクトリに保存するかどうかを選択することができます。

効率上の理由により、ASCET の Make 処理におけるファイル更新状態のチェックでは、すべての依存関係（例、変換式など）が考慮されるわけではありません。そのため、モデル内の変更が全体に及ぼす影響が認識されない場合があります。そこで、モデルに変更を加えた際は、コードの生成を開始する前に、**Build → Touch → Recursive** を実行して、強制的に完全な再生成を実行する必要があります。

実行ファイルが生成された後は、適合システムとのインターフェースに使用される ASAM-MCD-2MC データを作成します。

ASAM-MCD-2MC ファイルを作成する：

- プロジェクトエディタで、**Tools → ASAM-2MC → Write** を選択して ASAM-2MC ファイルを生成します。

  “Write ASAM-2MC To：” ダイアログボックスが開きます。
- ファイルのパスと名前を指定します。

注記

ASAM-MCD-2MC ファイルを作成する際、.¥cgen¥ ディレクトリには保存しないようにしてください。このディレクトリにあるファイルは、ASCET オプションの設定状態によっては ASCET 終了時に削除されてしまう場合があります（ASCET オンラインヘルプを参照してください）。

ここまででターゲット上でプログラムを実行するための準備がすべて揃いました。この後は、デバッガや適合システムを使用して、実行形式のプログラムをコントローラまたは評価ボードにロードします。ASAM-MCD-2MC ファイルは、適合システム（INCA など）による適合と測定作業のために使用されます。

必要に応じて、他のツール（ロジックアナライザ、ソースレベルデバッガなど）も使用できます。

ANSI-C ターゲットの場合の相違点

ANSI C ターゲットの場合、スタートアップコードやメモリレイアウトなどは未定義であるため、リンクは行いません。この「リンクの抑制」は `target.ini` ファイル内の `noLinking` オプションで制御されます。このオプションにはリンクを行わないコンパイラがリストアップされています。

`noLinking` オプションにリストアップされているコンパイラを使用して **Component → Build** または **Component → Rebuild All** を実行すると、ビルド処理は `*.obj` ファイルが生成された時点で終了し、モニタウィンドウに以下のエラーメッセージが表示されます。

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

Microsoft Visual C++ のコンパイラの場合、物理アドレスの計算は意味を持ちません。そのためこれらのコンパイラについてはマップファイル生成の抑制も必要です。これには `target.ini` ファイル内の `noMapFileGeneration` オプションを使用します。これはマップファイルを作成しないコンパイラのリストです。

また、ASAM-MCD-2MC ディスクリプションの生成には実行ファイルが必要ですが、ANSI-C コード生成においてはリンクを行わないため実行ファイルは生成されないので、ASAM-MCD-2MC ファイルも生成できません。

そのため、コード生成オプション **Generate Map File**（ASCET オンラインヘルプの“Project Properties”ダイアログボックスについてのトピックを参照してください）をオフにして、仮想アドレステーブルと `etas.map` ファイルの生成が行われないようにしておくことをお勧めします。

ASAM-MCD-2MC ディスクリプションの生成については 8.4 項を参照してください。

以下の表は、ターゲットとオペレーティングシステムの組み合わせに応じてデフォルト状態において実行される機能を示したものです。

	ターゲット	
オペレーティングシステム	**マイクロコントローラ**	**ANSI-C**
RTA-OSEK	コード生成 コンパイル リンク A2L の生成	コード生成 コンパイル
Generic OSEK	コード生成 コンパイル リンク A2L の生成	コード生成 コンパイル
RTE-AUTOSAR	コード生成 コンパイル	コード生成 コンパイル

3.4 ASCET-SE のインストール内容

本項では ASCET-SE のターゲットディレクトリ `<install_dir>¥target¥trg_<targetname>` にインストールされるファイルについてのクイックリファレンスです。

3.4.1 インストールされるファイル

以下に、ASCET-SE で使用される主なファイルをまとめます。これらのファイルは ASCET のインストールディレクトリ `<install_dir>¥ETAS¥ASCET6.1` のサブディレクトリ `.¥target¥trg_<targetname>` に保存されています。

ディレクトリ `.¥target¥trg_<targetname>`

ファイル	**内容**
`.indent.pro`	“Indent” コードフォーマッティングユーティリティ用のコンフィギュレーションファイル
`aml_template.a2l`	ETK のグローバルコンフィギュレーション BLOB の型定義が含まれるテンプレートファイル（ユーザーによるカスタマイズが必要）→ 116 ページの 8.3 項参照
`build.mk`	リンカ／ロケータの実行時に使用される Make ファイル → 5.4.5 項（78 ページ）参照
`clean.mk`	プロジェクトエディタ内のメニューコマンド **Build → Clean Code Generation Directory** をカスタマイズするための Make ファイル
`codegen.ini`	コード生成用マクロ定義を含むファイル→ 個々のエントリについてはファイル内の注釈を参照

ファイル	内容
codegen_<targetname>.ini	各ターゲット固有のコード生成用オプション設定を含むファイル→ 個々のエントリについてはファイル内の注釈を参照
codegen_ecco.ini	コード生成用の ECCO 設定を含むファイル（コード生成時に ECCO によって読み込まれます。個々のエントリについてはファイル内に説明されています）
compile.mk	コンパイル時に使用される Make ファイル
custom_settings.mk	Make 処理のカスタマイズ用 Make ファイル
depend.mk	生成されるファイルの依存関係を生成するための Make ファイル
do_compile.mk	コンパイラの呼び出しを実行する Make ファイル
generate.mk	ECCO でコード生成を実行する際にのみ使用される Make ファイル（この Make ファイル実行後、プロジェクト内の全モジュールから C ファイルと H ファイルが生成され、ASCET のインストールディレクトリ下の .¥cgen ディレクトリに書き込まれます）→ 5.4.3 項（78 ページ）参照
global_settings.mk	ASCET-SE の内部 Make ファイル
if_data_template.a2l	ETK のグローバルコンフィギュレーション BLOB の型定義が含まれるテンプレートファイル（ユーザーによるカスタマイズが必要）→ 8.3 項（116 ページ）参照
mem_lay.a2l	コントローラのメモリレイアウトが定義された ASAM-MCD-2MC サンプルファイル（ユーザーによるカスタマイズが必要）→ 8.2 項（115 ページ）参照
memorySections.xml	メモリクラスの XML 定義が含まれるファイル → 5.3 項（71 ページ）参照 ANSI-C ターゲット（trg_ansi）の場合はさらに以下の 2 つのメモリクラス定義ファイルが含まれます。 memorySections_Autosar.xml memorySections_Autosar4.xml
OS_<osname>_<version>.template	ASCET-SE が OS コンフィギュレーションファイルを作成する際に使用する <osname>（およびオプションとして <version>）用 OS テンプレートファイル
os_settings.mk	OS 設定用 Make ファイル
postasap.mk	ASAM-MCD-2MC ファイルのポストプロセッシング用 Make ファイル
prj_def.a2l	MOD_PAR セクションが定義された ASAM-MCD-2MC サンプルファイル → 8.1 項（115 ページ）参照

ファイル	内容
project_settings.mk	プロジェクト固有のコンフィギュレーション設定（インクルードされるライブラリなど）やコンパイラ／リンカ設定などを含むファイル → 5.4.1 項（77 ページ）参照
services.ini	算術演算サービスを含むファイル → ASCET オンラインヘルプ「算術演算サービス」参照
settings_<compiler>.mk	全プロジェクトに適用されるコンパイラ／ターゲット固有の設定（ファイル拡張子、プリコンパイラ／コンパイラ／リンカ／その他のプログラムの呼び出し方法、プログラム／インクルードファイル／ライブラリのパスなど）→ 5.4.4 項（78 ページ）参照
smart_compile.mk	スマートコンパイル用 Make ファイル
target.ini	ターゲット固有の ASCET 用デフォルトバリアント設定を含むファイル → 各エントリの詳細は 5.2 項（68 ページ）参照
target_<variant>.ini	ターゲット固有の ASCET 用各種バリアント設定を含むファイル → 各エントリの詳細は 5.2 項（68 ページ）参照
target_settings.mk	ターゲット固有の設定を行うための Make ファイル → 5.4.1 項（77 ページ）参照

ディレクトリ .¥target¥trg_<targetname>¥cp_rules

このサブディレクトリには ECCO が C コード生成時に使用する Perl マクロ（CPR：**C**ode **P**roduction **R**ules）が保存されています。

ディレクトリ .¥target¥trg_<targetname>¥docco

このサブディレクトリには DOCCO 自動コードドキュメンテーションツールが使用するスタイルシートと定義ファイルが保存されています。

ディレクトリ .¥target¥trg_<targetname>¥example

このディレクトリには ASCET-SE 用の簡単なサンプルプロジェクトに使用されるターゲット固有の設定を含むファイルが保存されています。

ファイル	内容
confV50.oil	サンプルプロジェクトのエントリポイントとして使用される OIL ファイル。このファイルには、CPU、OS、COUNTER（時間ラスタ用システムカウンタ）、ISR（システムカウンタ駆動用）、COM、といった OIL オブジェクトが含まれています。
example_rta.exp	サンプルプロジェクトを含む ASCET エクスポートファイル
Howto.html	このディレクトリについての詳細と、サンプルアプリケーションの機能とビルド方法を説明した HTML ファイル
<targetname>_user.<lnk>	リンカ／ロケータ制御ファイルのサンプルで、拡張子 <lnk> はターゲットごとに異なります。→「リンカ／ロケータの制御」（79 ページ）参照

ディレクトリ .¥target¥trg_<targetname>¥include

このディレクトリには ASCET-SE で使用される C のインクルードファイルが含まれます。

ファイル	内容
a_basdef.h	すべての ASCET プロジェクトファイルにインクルードされる、ASCET コントローラ定義用のメインヘッダファイル
a_limits.h	ASCET の標準型の上限値と下限値の定義
a_sect.h	メモリセクション定義を含むヘッダファイル
a_std_type.h	ASCET の標準型（uint16 など）の定義を含むファイル
a_user_def.h	ユーザー定義のデータ型が定義されたヘッダファイル。デフォルト状態ではコンパイルされるコードは含まれません。
message_scheme.h	メッセージバリアント選択用ヘッダファイル → 13.4.3 項（192 ページ）参照
os_inface.h	OS インターフェース定義を含むヘッダファイル（必要に応じて、生成されるすべての C ファイルにインクルードされます。使用する OS に応じてカスタマイズ可能です）
os_rta_inface.h	RTA-OSEK に適応するためのヘッダファイル

ファイル	内容
os_unknown_inface.h	擬似的 OS に適応するためのヘッダファイル
proj_def.h	アプリケーション固有の適応を行うためのヘッダファイル → 5.5.2 項（83 ページ）参照 .
tipdep.h	ターゲット固有の宣言文を含むヘッダファイル

ディレクトリ .¥target¥trg_<targetname>¥Intpol

注記

ASCET 製品に含まれている補間ルーチンはサンプルとして作成されているものです。実際の ECU ソフトウェアで使用されることを前提として作成されたものではありません。詳しくは 2.1 項を参照してください。

ファイル	内容
a_intpol.h	補間ルーチンのインターフェース定義
build_cmd.bat	補間ファイブラリのビルド処理に使用されるバッチファイル **注記：このファイルは直接呼び出さないでください。** これは intpol_<target>_<compiler>.bat というバッチファイルから呼び出すために作成されたファイルです。
customize.pm	指定された型の組み合わせで補間ルーチンを生成する関数を含む Perl マクロ（カスタマイズ可能）
HowTo.html	補間ルーチンの扱いについての説明
intpol_<target>_<compiler>.bat	ターゲット <target> ／コンパイラ <compiler> 用補間ライブラリのビルドを開始するバッチファイル。ソースファイルが以下のサブディレクトリに保存されている必要があります。 .¥target¥trg_<targetname>¥as¥intpol¥src
makeintpol.pl	補間ルーチンの型の組み合わせを生成するための Perl スクリプト
makeintpol_header.pl	補間ルーチンのプロトタイプと共にヘッダファイルを生成するための Perl スクリプト（ASCET-SE が特性カーブ／マップを扱う際に使用します）
path_settings.bat	全ターゲット用のコンパイラパスを設定するためのバッチファイル（バッチファイル intpol_<target>_<compiler>.bat から呼び出されます）
settings_<compiler>.mk	コンパイラ固有の設定を行うための Make ファイル

ディレクトリ .¥target¥trg_<targetname>¥Intpol\lib

ファイル	内容
Disclaimer for interpolation routines.txt	ASCET 製品に含まれる補間ルーチンについての注意事項と免責条項が記述されているファイル
intpol_<target>_<compiler>.<lib>	build.mk（78 ページの 5.4.5 項）に含まれる project_settings.mk 内でプロジェクトにリンクされる補間ルーチンライブラリ。 ライブラリには一部の補間ルーチンしか含まれていません。他のルーチンは、customized.pm によって必要な時に自動的に生成されます。 拡張子 <lib> は、ターゲット固有のライブラリ用拡張子で、ターゲットコンパイラによって定義されます。一般的なものとして *.lib、*.h12、*.a などがあります。

詳しい情報は第 6 章 「補間ルーチン」を参照してください。ご不明の点は ETAS までお問い合わせください。

ディレクトリ .¥target¥trg_<targetname>¥Intpol¥Src

このディレクトリには補間ルーチン用のすべてのソースファイルが保存されています。

ディレクトリ .¥target¥trg_<targetname>¥scripts

このディレクトリには Perl スクリプトがいくつか保存されています。主なファイルは以下のとおりです。

ファイル	内容
convert_hip_db.bat	メモリクラス定義を従来のフォーマット（hip.db/target.ini）から現行フォーマット（memorySections.xml）に変換するためのバッチファイル
convert_hip_db.pl	convert_hip_db.bat が使用する Perl スクリプト

ファイル	内容
`cctolog.pl`	コンパイラが生成したエラー／ワーニングメッセージを ASCET が読み込み可能なフォーマットに変換する Perl スクリプト。変換されたメッセージは自動的に ASCET モニタウィンドウに表示されます。
`lltolog.pl`	リンカが生成したエラー／ワーニングメッセージを ASCET が読み込み可能なフォーマットに変換する Perl スクリプト。変換されたメッセージは自動的に ASCET モニタウィンドウに表示されます。
`ostolog.pl`	OS コンフィギュレーションツール（`rtabuild.exe` など）が生成したエラー／ワーニングメッセージを ASCET が読み込み可能なフォーマットに変換する Perl スクリプト。変換されたメッセージは自動的に ASCET モニタウィンドウに表示されます。

ディレクトリ .¥target¥trg_<targetname>¥source

ファイル	内容
`blkcopy.c`	コントローラ用コード内で配列を初期化する際に使用されるブロックコピールーチンのソースコード
`msgcopy.c`	非アトミックメッセージ（1 マシンワードより大きいメッセージ）のコピーに使用される関数のソースコード
`upmsgcp.c`	保護されないメッセージコピー（これによってメッセージによるプロセス間通信が可能になります）

4 エレメントの実装

ASCET でモデリングを行う際、物理モデルの機能的挙動をテストすることができます。その後は量産ステージに向けて組み込みソフトウェアを徐々に改良していきますが、これはコード生成用のインプリメンテーション（実装情報）を調整することにより行われます。

「実装」とは、連続的、離散的、および論理的に記述された物理モデルの各要素（「エレメント」）を、意味的に正しい形でインプリメント層（ここでは C コード）にマッピングすることを意味し、主な作業は、モデルに含まれる連続的な実数演算を、組み込みターゲットがサポートする離散整数演算（固定小数点演算）に変換することです。この変換時にはすべての要素が量子化されますが、その際に発生する数値誤差は避けることができません。生成されるコードの挙動は、元の物理記述と比べて常に微小な差異があります。

注記

ASCET における「実装コードジェネレータ」とは、「実装実験」や「コントローラへのコード実装」（またはそれぞれの「オブジェクトベースのコントローラへのコード実装」）のために使用されるコードジェネレータを表す汎用的な語です。

実装コードジェネレータは、ユーザーの指定に従って、数値の精度、RAM やスタックの必要量、さらにはコードサイズやコードパフォーマンスを調整しながらコードを生成します。

「インプリメンテーション」は、物理モデルに詳細情報を与えて調整するもので、ASCET で組み込み制御ソフトウェアを作成するために必要なものです。つまりこのインプリメンテーションで、物理的機能を ECU 用コードに変換する際の規則を定義します。このように物理モデルとその実装情報とをそれぞれ別のものとして管理することにより、開発プロセスの構造化が実現できます。

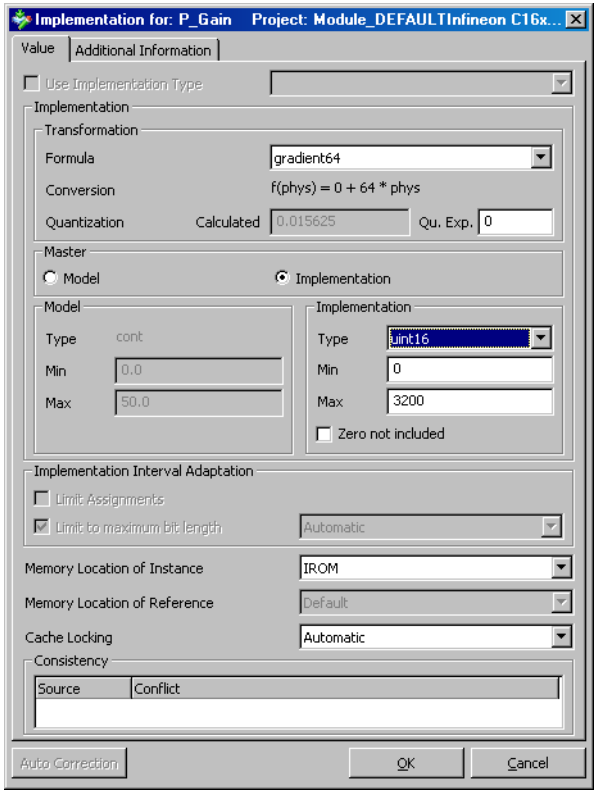
4.1 基本モデル型のインプリメンテーション

エレメントのインプリメンテーションを編集する：

- 実装したいエレメントを右クリックします。たとえば下の例では、パラメータ `P_Gain` を右クリックします。

- **Edit Implementation** を選択します。

以下に示すような、インプリメンテーションエディタが開きます。



上の図は、PID コントローラの比例ゲインである `P_Gain` のインプリメンテーション情報です。物理範囲は 0.0 ～ 50.0 で、量子化の単位は 0.015625 となっていて、以下の変換式が割り当てられています。

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

つまり、この変数は、0 ～ 3200 の範囲の値をとる `uint16` 型として実装されます。以下の表は、物理値から実装値への具体的な変換内容を示したものです。

xphys（物理値）	Ximpl（実装値）	
	整数	二進数
0.000000	0	00000000_00000000
0.015625	1	00000000_00000001
0.031250	2	00000000_00000010
…	…	…
0.984375	63	00000000_00111111
1.000000	64	00000000_01000000
1.015625	65	00000000_01000001
…	…	…
49.968750	3198	00001100_01111110
49.984375	3199	00001100_01111111
50.000000	3200	00001100_10000000

また、この変数は適合パラメータ（通常 ROM 領域に配置されます）なので、メモリクラス `IROM` が選択されています。

以降の項では、エレメントの実装についてのさまざまな側面を説明します。

4.1.1 実装データ型

物理モデルの量を表わすために使用される抽象データ型（continuous、discrete、logical）とは異なり、「実装」においては具体的なデータ型（「実装データ型」）が使用されます。ETAS のツールは以下の実装データ型を使用します。

型	内容	注釈
`sint8`	8 ビット符号あり整数	-128 ～ +127
`uint8`	8 ビット符号なし整数	0 ～ +255
`sint16`	16 ビット符号あり整数	-32768 ～ +32767
`uint16`	16 ビット符号なし整数	0 ～ +65536
`sint32`	32 ビット符号あり整数	-2147483648 ～ +2147483647
`uint32`	32 ビット符号なし整数	0 ～ +4294967296
`real32`	32 ビット IEEE 浮動小数点	一部のターゲットでのみ使用可
`real64`	64 ビット IEEE 浮動小数点	一部のターゲットでのみ使用可
`bit`	直接アドレス指定される単一ビット	一部のターゲットでのみ使用可

注記

一部のプロセッサでは、浮動小数点演算をエミュレートできるソフトウェアライブラリを併用する場合に限り、浮動小数点を実装することができます。しかし、浮動小数点を実装すると実行時間とメモリ消費量が非常に増大するため、一般的な ECU ソフトウェアには適していません。

データ型については以下のような点に注意が必要です。

- モデルデータ型が `udisc` で実装データ型が `sint*` である変数の実装インターバールの下限値は、実装データ型の実際の下限値（負の値）ではなく、0（ゼロ）になります。
- モデルデータ型が `sdisc` で実装データ型が `uint*` である変数のモデルインターバールの上限値は、2147483647 ではなく実装データ型の上限値となります。これは実装データ型が `uint32` の場合にも適用されます。
- モデルデータ型が`cont`または`sdisc`で実装データ型が`uint*`である変数のモデルインターバールの下限値は、モデルデータ型の下限値（負の値）ではなく、0 になります。

生成されるコードにおいては、代入文に限り、浮動小数点数演算と整数演算を以下のように併用することができます。

- 量子化されていない浮動小数点と量子化された整数は、互いに代入できます。
- コードジェネレータは、変換および自動範囲チェックを行うコードを生成します。
- 仮引数と実引数の暗黙のマッピングを行うためのメソッド呼び出しについても同様です。

注記

算術演算や比較の中で浮動小数点と整数のインプリメンテーションを組み合わせて使用することはできません。そのような場合はエラーメッセージが出力されます。

4.1.2 変換式

変換式（Formula）は、モデル量の物理値をソフトウェアで使用される実装値に変換するものです。変換式は、値の有効範囲内において可逆的でなければなりません。ASCET において変換式を定義する際は、必ず物理値から実装値へ、つまり以下の方向で定義されます。

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

変換式は以下のものに適用されます。

- 生成されたコード内の integer にマッピングされる cont 型の物理量

以下のような場合には、identity 変換式（`implValue = physValue`）以外の変換式は使用できません。

- 論理量（ブール値）について、変換式を指定することはできません。
- 離散的物理量（udisc 型または sdisc 型）については、identity 変換式しか使用できません。
- 浮動小数点として実装される cont 型の物理量についても、やはり identity 変換式しか使用できません。

以降の説明では、基本的に物理量を小文字で表記し、対応する実装値を大文字で表記しています。

変換式はプロジェクトエディタの“Formulas”タブで「グローバルに」定義します。新しい式を定義するには、このタブで **Global Formulas → Add** を選択し、定義した変換式をインプリメンテーションエディタで使用します。

ASCET はさまざまな種類の変換式（線形変換式、線形有理式、平方有理式、テーブル、逐語変換式）を認識しますが、コード生成においては以下の形式の単純な線形式だけがサポートされます。

```
X = ax+b
```

この a および b を、それぞれスケール値およびオフセットと呼びます。値の量子化単位は、スケール値の逆数です。

```
q = 1/a
```

以降の説明では、スケール値とオフセットはどちらも有理数であると仮定します。実数は有理数を用いた任意の精度で近似値化できるので、これは実質的な制約とはなりません。整数演算では有理数しか使用できないことに注意してください。

非線形の変換式を記述することもできますが、コード生成においては非線形形式間の自動変換はサポートされていません。

注記

コード生成の内部処理では非線形変換式を identity 式のように扱うので、自動変換は行われません。

非線形量子化を伴う算術演算は不可能です。そのような演算は、たとえば積分器の時定数として、特性値やメソッドを入力するためにしか使用できません。ユーザーの責任において、非線形量子化を伴う量はこのような場合にのみ使用されるようにしてください。コード生成を含むどのツールも、これ以外の使用法には対応していません。

4.1.3 値の範囲（数値量の場合のみ）

量を表す値の範囲は、その数値の有効なインターバル（上下限値）と同じです。指定された値の範囲は、コードジェネレータが中間結果のインターバルを算出する際に使用され、この過程でオーバーフローの発生が検知されます。コードジェネレータは、これを通じて、ソフトウェア内の中間結果の生成方法や計算方法を決定します。必要に応じて、リミッタをイネーブルにする必要があります。

物理値の範囲と実装値の範囲の両方を指定することもできます。可逆の線形変換式では、一方の値の範囲を指定すると、もう一方の指定しなかった方の値の範囲も自動的に更新されます。そのため、ユーザーは物理環境と実装環境のどちら側からでも条件を指定することができます。

ただし、次のような場合には、値の範囲を指定できないか、または指定しても無視されます。

- 論理量（ブール値）と列挙型の量については、値の範囲を指定することはできません。
- 浮動小数点として実装される連続物理量は、指定された実装データ型に、制限を伴うことなくマッピングされます。ASCET のエディタで値の範囲を入力することはできますが、その値は無視されます。代わりに、疑似無限インターバルが採用されます。

4.1.4 インプリメンテーションマスタ

物理モデルとインプリメンテーションのいずれか一方を、インプリメンテーションマスタとして指定することができます。ユーザーがインプリメンテーションマスタ側に入力した値は、マスタ設定や変換式に従って、他方のノンマスタ側の設定に使用されます。

プロジェクトエディタで式をグローバルに変更した後、影響を受けるすべてのインプリメンテーションを、プロジェクトエディタの **Extras → Update Implementations** コマンドを使用して自動更新することができますが、このとき、インプリメンテーションエディタの“Master”オプションでモデル側とインプリメンテーション側のどちらの値の範囲を維持するかを指定しておくことができます。モデル側がマスタとして選択されると、モデル側の設定は変わらず、インプリメンテーション側が更新されます。インプリメンテーション側がマスタなら、反対にモデル側が更新されます。

4.1.5 インプリメンテーション型

インプリメンテーション型 をプロジェクト内に定義しておくことにより、個々の変数のインプリメンテーションをより容易に編集し、また同じインプリメンテーションを物理的に類似するエレメントに容易に割り当てることができます。これはまたクラスまたはモジュールに関しても同様です。インプリメンテーション型には 4.1.1 項から 4.1.4 項に説明されているインプリメンテーションの要素が含まれ、インプリメンテーションエディタ上で各エレメントに割り当てることができます。

インプリメンテーション型の作成と設定の方法は、ASCET オンラインヘルプの「インプリメンテーション型」についてのトピックに説明されています。インプリメンテーション型の使用方法については「インプリメンテーション型の使用方法」についてのトピックを参照してください。

4.1.6 値の範囲

インプリメンテーションの **Limit Assignments** オプションは、エレメントの値を所定の範囲内に制限するかどうかを指定するものです。算出された値が定義された最小値より小さい場合、そのエレメントには最小値が代入され、最大値よりも大きくなった場合は最大値が代入されます。このような「飽和演算」によってオーバーフローやアンダーフローを防止できます。

このオプションがオンになっていると、各代入演算について、指定された値の範囲を確実に維持するためのコードが追加されます。オフになっていると、選択された実装データ型を持ち、かつ限界値を持たない浮動小数点として実装される連続物理量が生成され、値の範囲を維持するのはユーザーの責任となります。

注記

エレメントごとにリミッタ設定を行えるようにするには、“Project Properties” ダイアログボックスの “Integer Arithmetic” ノードで、**Generate Limiters** オプションをオンにしておく必要があります。

Limit to maximum bit length オプションによって、個々のエレメントごとに ASCET が潜在的なオーバーフローをどのようにチェックして回避するかを指定することができます。

- *Reduce Resolution* ：適切な再量子化を行うことによって、潜在的なオーバーフローを回避します。この場合、精度が低下します。
- *Keep Resolution*：値の範囲を適用することによって、潜在的なオーバフローを回避します。値の分解度は変わりません。このオプションの機能を有効にするには、算術演算サービスが必要です。
- *Automatic*：算術演算サービスが有効になっている場合は *Keep Resolution* の方法で潜在的なオーバーフローが回避され、そうでない場合は *Reduce Resolution* の方法が使用されます。

4.1.7 値の範囲からのゼロの除外

ASCET での除算は、「ゼロ除算」の発生を防止するためのオプションがあります。このオプションは、生成されたコード内にランタイムチェックを挿入し、ゼロ除算が発生しないようにするものです。

しかしモデル内の特定のエレメントについて、「その値がゼロになることはない」ということが分かっている場合、**Zero not included** オプションをオンにしてそのエレメントのゼロ除算チェックを省略することができます。つまりこのオプションは、除算の除数の値がゼロにならないようにユーザー自身が注意する、ということを ASCET に対して宣言するものです。

注記

Zero not included オプションは、実装される値がゼロにならないことが明らかである場合にのみオンにしてください。そうでない場合、ECU 稼動時にゼロ除算による深刻な例外エラーが発生する可能性があります。

4.1.8 メモリロケーション

“Memory Location of *”コンボボックスで選択する「メモリロケーション」は、値（および場合に応じてその参照）が配置される ECU メモリを表す抽象メモリセクションの名前です。コードジェネレータは、この情報を使用して、指定された ECU メモリのエレメントレイアウトに従って C コードのデータ構造体を生成します。またこの情報はコンパイライントリンシック（通常は `#pragma` 文）の生成にも使用されます。ロケータはこれらの `#pragma` 文を使用してメモリロケーションを ECU 内の所定のアドレス範囲にマッピングしますが、この際にはユーザー定義された変換テーブルが使用されます。

コードジェネレータは、あるメモリクラスに含まれるすべてのエレメントについて、エレメントエディタの“Memory”フィールドに同じ属性（*Volatile* または *Non-Volatile* ）が指定されているかどうかを調べます。ひとつのメモリクラスが揮発性メモリと不揮発性メモリの両方に属することはできないので、混在が認められた場合はエラーメッセージが出力されます。

コード生成において、各変数の扱いは“Memory”属性に応じて異なり、Volatile メモリに属するものだけが自動的に初期化されます。

データベースを使用している場合、ASCET のコンポーネントマネージャのメニューコマンド、**Tools → Database → Convert → Variables to Volatile** および **Tools → Database → Convert → Parameters to Nonvolatile** を実行すると、属性の混在によって発生するエラーを防ぐことができます。1 番目のコマンドを実行するとデータ内のすべての変数に Volatile 属性が割り当てられ、2 番目のコマンドではすべてのパラメータに Non-Volatile 属性が割り当てられます。

ワークスペースの場合は、このようなグローバルな変換機能は用意されていません。

4.1.9 整合性チェック

もしもインプリメンテーションエディタで整合性のない、つまり矛盾したデータが指定された場合、ASCET はエディタ上に *Consistency* チェックリストを表示してユーザーに警告します。リスト内の項目を 1 つ選択して **Auto Correction** ボタンをクリックすると、それを自動的に修正することができます。

4.1.10 追加情報

必要に応じて、“Additional Information”タブにに詳細なインプリメンテーション情報を入力することができます。これは、ECU のタイプによって必要となる場合があります。またこの情報は、特殊なインフラストラクチャ（DAMOS や MSRDOC 等）に対応するために使用することもできます。用途に応じて、このフィールドには以下のような情報を入力します。

- コード構文、アドレス機構
- ビットパケット用ビットベースアドレスおよびバイナリポジション

このフィールドは、ASCET の基本システムでは使用されず、その構文およびセマンティックス（意味）も本マニュアルでは定義されていません。フィールドの定義はアプリケーションごとに異なるため、オープンなインターフェースを通じて詳細な情報を取得してください。

4.1.11 集合モデル型のサイズ

集合モデル型、つまり配列、マトリックス（行列）、ディストリビューション（軸ポイントの座標データ）、特性カーブおよびマップのサイズ情報は、インプリメンテーションの一部ではなく、ASCET のデータセットとして記述されます。

4.1.12 エレメントの実装についてのまとめ

次の表は、ASCET で使用される各基本モデル型に必要なインプリメンテーション情報をまとめたものです。論理型（`log` 型）と列挙型（`enum` 型）に限り、必ずしもすべてのインプリメンテーション情報を必要とはしません（たとえば、変換式は必要ありません）。その他のスカラ型（`continuous` および `signed/unsigned discrete`）については、インプリメンテーションを構成するすべての情報を入力する必要があります。これは集合型である配列、マトリックス、およびディストリビューションについても同じです。

注記

浮動小数点として実装される連続モデル型については、Identity 変換式（恒等式、つまり係数 1.0 を掛ける変換）を使用してください。離散データ型についても同様で、他の変換式が選択されるとワーニングメッセージが表示されます。

特性カーブおよびマップには、特別な規則が適用されます。これらの集合型については、個々の軸（固有の軸または共通軸）ごとに、実装データ型、変換式、および値の範囲を指定します。さらにデータエディタにおいて、補間モード（線形、丸め、ユーザー定義）を指定できます。

	スカラ			列挙型	配列、マトリックス、ディストリビューション	特性値	
	論理	離散	連続			カーブ	マップ
インプリメンテーション型	○	○	○		○	2*(x,y)	3*(x,y,z)
変換式		△	○		○	2*(x,y)	3*(x,y,z)
実装データ型	○	○	○		○	2*(x,y)	3*(x,y,z)
値の範囲		○	○		○	2*(x,y)	3*(x,y,z)
データ表記 *		○	○		○	○	○

* はパラメータの場合のみ
△は、*identity* 変換式しか使用できません。
○は、プロパティエディタで指定します。

	スカラ			列挙型	配列、マトリックス、ディストリビューション	特性値	
	論理	離散	連続			カーブ	マップ
メモリロケーション	○	○	○	○	○	○	○
“Additional Information”タブ	○	○	○	○	○	○	○
補間モード（線形／丸め／ユーザー定義）						○	○

* はパラメータの場合のみ
△は、*identity* 変換式しか使用できません。
○は、プロパティエディタで指定します。

4.2 複合モデル型（クラス、モジュール、プロジェクト）の実装

複合モデル型（クラス、モジュール、プロジェクト）の実装は、以下のステップで行います。

- そのコンポーネントに含まれているすべての基本モデル型についてのインプリメンテーションを指定します。
- そのコンポーネントに含まれている、他の複合モデル型（他のクラス、モジュール、またはプロジェクト）についてのインプリメンテーションを指定します。
- コンポーネントのデータ構造体用に、個々のメモリクラスや、その他のコンポーネント（ユーザー定義の外部サービスルーチンやハンドコーディングされた関数の呼び出しなど）に固有な設定が必要な場合に限り、そのコンポーネントのインプリメンテーションエディタの“Settings”タブでその設定を行います。

プロジェクト全体のインプリメンテーションは、そのプロジェクト内のすべてのエレメントのインプリメンテーションを規定します。

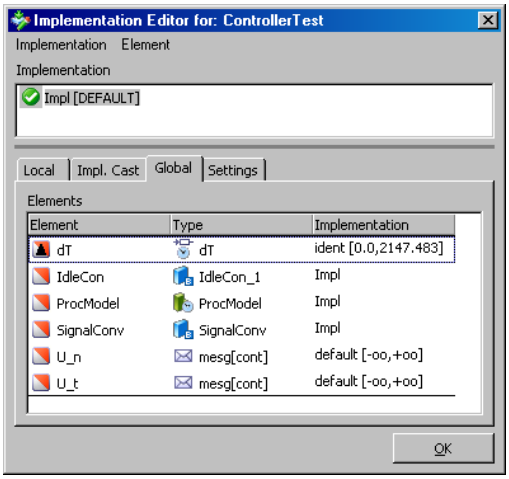
ASCET では、複合モデル型について、さまざまなインプリメンテーションのオプションを使用することができますが、コード生成時には各インスタンスごとにそのうちの 1 つだけが有効になります。

特殊なエレメント（たとえばプロジェクト）については、インプリメンテーションエディタにおいて別のインプリメンテーションに切り替えることができます。モデルの各インプリメンテーションは階層的にリンクされているので、変更されたインプリメンテーションは、その下位にあるすべてのチャイルドエレメントにも適用されます。

プロジェクトまたはコンポーネントのインプリメンテーションを編集する：

- プロジェクトエディタまたはコンポーネントエディタで、**Edit → Component → Implementation** を選択します。

  プロジェクトまたはコンポーネントのインプリメンテーションエディタが開きます。



- “Elements” ペーンで、エレメントの 1 つをダブルクリックします。

  そのエレメント用のインプリメンテーションエディタが開きます。

このようにして、プロジェクトまたはコンポーネント内の任意のエレメントのインプリメンテーションエディタにアクセスすることができます。上の例では標準的なインプリメンテーションしか選択できませんが、他にターゲット用のインプリメンテーションオプションを定義することもできます。

複合モデルエレメントのインプリメンテーションエディタでは、以下のようにして、基本モデルエレメント同士のインプリメンテーションを互いにコピー＆ペーストすることができます。

エレメントのインプリメンテーションをコピーしてペーストする：

- プロジェクト／コンポーネントのインプリメンテーションエディタでコピー元の基本エレメントを右クリックし、**Copy Implementation To Buffer** を選択します。

  選択されたエレメントのインプリメンテーション情報全体が、バッファにコピーされます。

- コピー先の基本エレメントを右クリックし、**Paste Implementation From Buffer** を選択します。

  バッファ内のインプリメンテーション情報全体が、選択されたエレメントにコピーされます。

4.2.1 メソッド呼び出しの最適化

クラス内に定義されたメソッドについて、ASCET では複数のインスタンスを扱うことが可能です。各インスタンスには同じコードが使用されますが、データ構造体（13.3.3「複合（ユーザー定義）オブジェクトのデータ構造体と初期化」を参照してください）はそれぞれ異なるものが使用されます。この場合、C 関数に対して構造体のポインタ（self ポインタ）が渡され、以下のようなメソッド宣言が行われます。

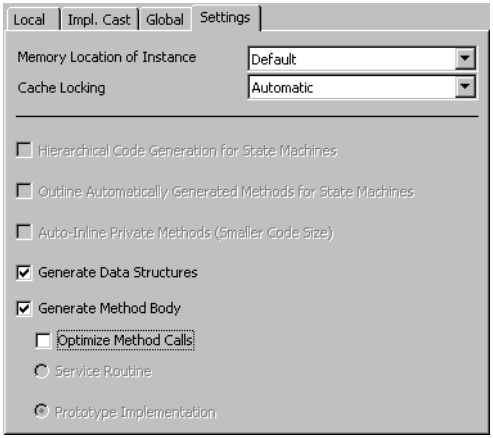
```
sint16 PIDT1_IMPL_out (
     const struct PIDT1_IMPL *self,
     sint16 in);
```

またインスタンスを 1 つしか持たないクラス（「シングルインスタンス」と呼ばれます）に限り、ASCET でメソッド呼び出しが自動的に最適化され、データエレメントが直接アクセスされます。

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

この最適化は、デフォルト状態において有効です。

ただし、ハンドコーディングされた外部コードから ASCET で作成されたメソッドを呼び出す場合、ツールが self ポインタについての最適化を自動的に行う、ということはあまり好ましくはありません。メソッド呼び出しの方法は、モデルの変更に伴って自動的に変更される可能性があるためです。このような場合を想定して、ASCET のクラス用インプリメンテーションエディタの “Settinigs” タブでは、シングルメソッドの最適化を無効にすることができます。



上記のように設定すると、クラスが複数のインスタンスを持つかどうかに関わらず、メソッド呼び出し時に必ず self ポインタが生成されるようになります。

注記

外部でハンドコーディングされた関数から ASCET が生成したメソッドを呼び出したり ASCET が生成した変数やパラメータにアクセスする際は、ASCET によって生成されたデータの型を必ず確認し、各データを同じ型で使用するようにしてください。これは特に self ポインタの場合に重要です。
ASCET によって生成されたコードにおける関数のインターフェースは、システムのバージョンアップによって変わる場合があるためです。

モデル内にインスタンスが 1 つしか存在し得ないクラスに限り、self ポインタを使用しないメソッドインターフェースが生成されるようにすることができます。これには **Optimize method calls** オプションをオンにします。

4.2.2 ユーザー定義サービスルーチン

コードジェネレータでは、クラスのメソッドとプロセスをユーザー定義サービスルーチンとして実装することもできます。この場合、メソッドのボディは ASCET では生成されず、ユーザー自身が、たとえばリンク時にそのコードのリンクを指定することによって供給します。この方法で、高度に最適化されたメソッドをアセンブラコードで実装するようなことも可能です。このサービスルーチンには以下のような条件が適用されます。

- サービスルーチンとして実装されたクラスメソッドについては、メソッドボディは生成されません。ASCET を使用して、ブロックダイアグラム、ESDL、または C コードでモデリングされたファンクションは、マイクロコントローラのコード生成においては無視されるため、ユーザーは適切なコードを外部のソースファイルとして用意しておく必要があります。しかしその場合でも、それらのメソッドの内容は ASCET で行われるシミュレーション実験では必要となるため、ASCET 内にも定義しておくことができます。
- サービスルーチン用に定義されたメソッドとメソッド引数は、それを内包する ASCET モデルから使用することができますが、生成されたコードにはそれらの外部（`#extern`）宣言は存在しません。またクラスにローカルエレメントが含まれている場合、self ポインタが使用され、最適化は行われません（4.2.1 章を参照してください）。

注記

サービスルーチンの引数型の重複割り当てを防ぐため、クラスは、そのローカル変数と同様に、同じメモリクラスに割り当てることをお勧めします。また、サービスルーチンを使用するクラスにはローカルパラメータは持たせず、パラメータが必要な場合は、グローバル定義するか、またはメソッド引数として受け渡すようにしてください。

- プロトタイプクラスからエクスポートされる変数は、それを内包する ASCET モデルから使用することが可能です。生成されたコードには、プロトタイプメソッドの外部宣言が適切な位置に提供されますので、それに対応する適切な変数定義を、ハンドコーディングされた外部ソース内で行ってください。
- ローカルインスタンス変数およびパラメータは、ローカルデータ構造体の一部として生成され、サービスルーチンには self ポインタとして渡されます。インポートされる変数とメソッドローカル変数はメソッドインターフェースには関係しないため、サービスルーチン用に生成されるコードでは無視されます。

サービスルーチンは、以下のようにして定義します。

サービスルーチンを定義する：

- **Edit → Component → Implementation** を選択して、クラスまたはモジュール用のインプリメンテーションエディタを開きます。
- “Settings” タブで、**Generate method body** オプションをオフにします。
- **Service routine** オプションをオンにします。

サービスルーチンの名前は、ASCET の命名規則に従う必要があります。この名前は、クラスまたはモジュールの名前、インプリメンテーションの名前、およびメソッドまたはプロセスの名前をアンダースコアでつなぎ合わせたものになります。インプリメンテーション名自体にアンダースコアが含まれている場合（例、`U8_MASSFLOW_INTEG`）には、その最初のアンダースコアまでの部分がサービスルーチン名に使用されます。

注記

ユーザーは、この命名規則を必ず守ってください。

例：`INTEGRATORK` 型の `MassFlow_Integ` という名前のクラスインスタンスを想定します。このクラスには、`compute` という名前のメソッドの定義が含まれています。このクラスは `U8_MASSFLOW_INTEG` として実装されました。

インプリメンテーション名にデータ型プレフィックスが付いているので、生成される関数は以下のようになります。

```
INTEGRATORK_U8_compute(…),
```

このように、インプリメンテーション名の中のデータ型プレフィックスだけが、生成される関数名に使用されます。そのため、具体的なインプリメンテーションごとにではなく、データ型ごとに 1 つのサービスルーチンを指定すればよいことになります。

複数のインプリメンテーションを扱う場合には、次のような命名規則をお勧めします（強制ではありません）。データ型プレフィックスの後ろの、クラスインスタンス名に相当する名前を選び、必要に応じて連番を付けます（例、`U8_MASSFLOW_INTEG1`）。

これらの命名規則は、プロセッサコマンド（`#define`）についても同様です。

サービスルーチンは生成されたコードから、通常のクラスメソッドと同じ方法で呼び出されます。つまり、ユーザーはルーチン定義内の引数、戻り値、およびローカルエレメントに関するすべての規則に従う必要があります（13.3.6「メソッドの宣言と呼び出し」を参照してください）。

注記

ハンドコーディングされた外部関数を ASCET から呼び出したり、ハンドコーディングで定義された変数やパラメータを ASCET で使用する場合、必ず、ASCET 側で生成されたデータ型を確認し、それと異なる型の使用は避けてください。特に、self ポインタの場合には注意が必要です。

Make メカニズムは、クラスのコード生成、コンパイル、リンクは一切行わないので、代わりにユーザーが適切なコード（関数コードおよび変数／パラメータの定義）を別の方法で提供する必要があります。ASCET 内では、サービスルーチンを、外部 C コードで定義することもできます。

4.2.3 プロトタイプ実装

ハンドコーティングされた外部関数を使用するため、ASCET と ASCET-SE は、ユーザーによるクラスのプロトタイプ宣言をサポートしています。一般的なプログラミング言語における関数プロトタイプと同様、ASCET においてクラスのプロトタイプ宣言を行えば、関数の実体がなくてもインターフェースの宣言が可能になります。これは以下のようにして行われます。

- プロトタイプとして実装されたクラスについては、メソッドボディは生成されません。またプロトタイプクラスのマイクロコントローラ用コード生成においては、ASCET を使用して、ブロックダイアグラム、ESDL、または C コードでモデリングされたファンクションは無視されます。しかしその場合でも、それらのメソッドの内容は ASCET で行われるシミュレーション実験では必要となるため、ASCET 内でも定義しておくことができます。
- ASCET プロトタイプクラス内に定義されたメソッドとメソッド引数は、それを内包する ASCET モデルから使用することができます。外側のモデルのコードには、プロトタイプメソッドの外部（`#extern`）宣言が、呼び出しを行う個所に生成されます。self ポインタは使用されません（4.2.1 章を参照してください）。つまりプロトタイプクラスについての複数のインスタンス生成はサポートされていません。ユーザーは、ハンドコーディングされた外部ソース内に、対応する関数を定義する必要があります。

- プロトタイプクラスからエクスポートされる変数とパラメータは、それを内包する ASCET モデルから使用することが可能です。外側のモデルコードには、プロトタイプメソッドの外部（`#extern`）宣言が適切な位置に生成されます。これらの宣言はプロセッサコマンドによって行われ、必要に応じて無効にすることもできます。ハンドコーディングされたソース内では、適切な定義を行ってください。
- ローカルインスタンス変数、インポートされる変数、およびメソッドローカル変数は、メソッドインターフェースには関係しないため、プロトタイプクラス用に生成されたコード内では無視されます。プロトタイプクラスのローカルエレメントへの直接的なアクセスは、最適化の有無に関わらず、サポートされていません。

メソッドプロトタイプは、以下のようにして定義します。

メソッドプロトタイプを定義する：

- **Edit → Component → Implementation** を選択して、クラス用のインプリメンテーションエディタを開きます。
- “Settings” タブで、**Generate method body** オプションをオフにします。
- **Prototype implementation** オプションをオンにします。

C 関数には厳密な命名規則があり、必ず、クラスまたはモジュール名、インプリメンテーション名、メソッドまたはプロセス名がアンダースコアで接続された名前でなければなりませんが、プロトタイプについては特別な命名規則はありません。ただしプロセッサコマンド（`#define`）を使用して命名規則を定義することも可能です。

注記

ハンドコーディングされた関数を ASCET から呼び出したり、ハンドコーディングで定義された変数やパラメータを ASCET で使用する場合、必ず ASCET 側で生成されたデータ型を確認し、それと同じ型を使用するようにしてください。ASCET によって生成されるコードにおいて、関数インターフェースの部分は、バージョンによって変わる場合があるためです。

Make メカニズムは、クラスのコード生成、コンパイル、リンクは一切行わないので、代わりにユーザーが適切なコード（関数コード、および変数とパラメータ定義）を別の方法（9 章「外部コードの統合」も参照してください）で提供する必要があります。

4.2.4 プロセスとメソッド

プロセスとメソッドについても実装を行うことができます。プロセスとメソッド用のインプリメンテーションエディタでは、3 種類のオプションを選択することができます。

- プロセス／メソッドのコードを割り当てるメモリロケーション
- マイクロコントローラの浮動小数点ユニット（FPU）の使用

  このオプションは OS コンフィギュレーションの生成時に使用され、これにより、タスク切り替え時に FPU のコンテキストを保存するかどうかが判断されます。1 つの OS タスク内のすべてのプロセスとメソッドについてこのオプションがオフになっていれば、そのタスク内には現在の FPU のコンテキストに影響するコードがないと判断できるため、OS はそのタスクについて FPU コンテキストの保存と復帰を行う必要がありません。

  デフォルトでは FPU を使用する設定になっています。

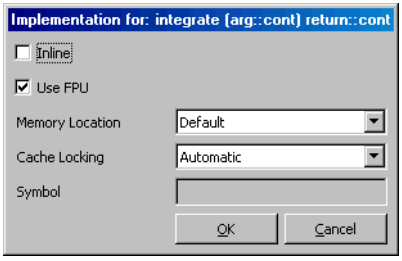
  プロセスまたはメソッドが FPU を使用しないプロセス／メソッドについてこのオプションがオンになっていると、実際の計算に FPU が使用されないのにも関わらず、FPU コンテキストの不必要な保存が行われてしまいます。

  注記

  FPU が搭載されていないマイクロコントローラの場合、このオプションは意味を持ちません。

- メソッドについては、関数のインライン化を行うかどうかを指定することができます。このオプションは、コンパイラコンフィギュレーションで適切なキーワードが“Inline Directive”内で定義されている場合にのみ有効です。現在使用しているコンパイラの現在の設定は、ASCET“Options”ダイアログボックスの“External Tools/Compiler/<*compiler*>”ノード内のエントリで確認できます。

プロセス／メソッド用インプリメンテーションエディタを開く：



- コンポーネントエディタの“Outline”タブで、プロセスまたはメソッドを選択します。

- **Edit → Implementation** を選択して、インプリメンテーションエディタを開きます。

4.3 テンポラリ変数の実装

テンポラリ変数は、演算子および複合モデルエレメントの出力として定義することができます。これを行うためには、希望のエレメントを右クリックし、ショートカットメニューから **Temporary Variable** を選択します。テンポラリ変数を明示的に実装することはできませんが、代わりにメソッドローカル変数を実装することができます（4.5 項を参照してください）。

コードジェネレータは、テンポラリ変数についてインプリメンテーションを自動的に決定します。テンポラリ変数には、実装量が初めて代入されるときに、変換式と値の範囲も割り当てられます。実装データ型には、その変換式と値の範囲にふさわしい型が選択されます。

注記

テンポラリ変数が数式内に挿入されても、その式に対応する演算の生成には影響しません。また 1 つのテンポラリ変数を制御フロー内の異なる複数の分岐先（例、`if` 文の分岐先）で使用することはできません。結果およびインプリメンテーション（例、量子化）は、分岐先ごとに異なる可能性があります。これにより、生成されるコードに算術演算上の重大な間違いが生じてしまう可能性があります。

4.4 インプリメンテーションキャストの実装

インプリメンテーションキャスト（ASCET オンラインヘルプを参照してください）によって、演算やデータ転送などの任意の個所においてインプリメンテーションを任意に変更することができます。このインプリメンテーションキャストは、変数やパラメータとは異なり、メモリを使用しないため、この内容を適合することはできません。

インプリメンテーションキャストにはデータは含まれず、常にモデルデータ型が `const` のスカラ値（単一の値）で、ローカル範囲でのみ有効です（5.2 項を参照してください）。他のエレメントとは異なり、インプリメンテーションキャストの属性は編集できません。インプリメンテーションキャストのインプリメンテーションは、通常の基本モデル型のインプリメンテーションと同じ方法で編集します（4.1.12 項を参照してください）。

4.5 メソッドローカル変数とプロセスローカル変数の実装

メソッドおよびプロセスには、ローカル変数を作成することができます。このためには、クラスエディタまたはモジュールエディタ内のメソッド／プロセスの名前を右クリックし、ショートカットメニューから **Properties** を選択します。そしてシグネチャエディタの“Locals”タブで **Add** ボタンをクリックし、ローカル変数を作成します。

こうして作成した変数には、51 ページ「エレメントの実装についてのまとめ」の項で説明されいる方法で、インプリメンテーションを設定することができます。ユーザーがインプリメンテーションを指定しない場合は、変換式、値の範囲、および実装データ型は、テンポラリ変数の場合と同じ方法で自動的に定義されます。

4.6 演算子インプリメンテーションの自動変換

注記

ASCET V5.0 以降では、インプリメンテーションオプションの **Limit to maximum bit length** と **Zero not included** が「演算子インプリメンテーション」の役割を果たしています。さらに、インプリメンテーションキャストを使用することによって、連鎖的に処理される算術演算の演算子の量子化を、余分なメモリスペースを増やすことなく定義することが可能です。
旧バージョンで作成された演算子インプリメンテーションは、参照したり、インプリメンテーションキャストに置き換えたり、削除することはできますが、編集することはできません。

旧バージョンのモデル内の演算子インプリメンテーションは、削除するか、または新しく導入されたインプリメンテーションキャスト（ASCET オンラインヘルプを参照してください）に自動的に変換することができます。ただしこの自動変換は、コンポーネントごとではなく、データベース全体に適用されます。

注記

インプリメンテーションキャストは、ASCET オンラインヘルプの「インプリメンテーションキャスト」、「ESDL でのインプリメンテーションキャストの使用」、「ブロックダイアグラム内のインプリメンテーションキャスト」についてのトピックに説明されています。

自動変換のルール： 以下の条件が満たされている演算子インプリメンテーションのみが自動変換されます。

- 演算子インプリメンテーションに **Auto** 以外の量子化オプション（addition、subtraction、MIN、MAX、MUX）が含まれていないこと。

  注記

  MIN、MAX、MUX 演算子の自動変換を行うために満たされている必要があるのはこの条件だけです。他の条件は、加減乗除演算にのみ適用されます。

- 演算子の出力が接続されていること
- 演算子の出力が基本エレメントにのみ接続可能であること

  コンポーネント、演算子、階層ブロックには接続できません。

- インプリメンテーションキャストが演算子の出力に接続されている場合、このインプリメンテーションキャストについて、インプリメンテーションエディタの **Use Implemntation Type** オプションの右側のコンボボックスで <No implementation> 以外の項目が選択されていること。
- 演算子インプリメンテーションに特別なプレシフト（乗算と除算の場合）が含まれていないこと
- 除算演算子について、その演算子インプリメンテーションの **Allow zero in phys. interval** オプションがオンになっている場合は、分母について以下の条件が満たされていること
  - 分母入力が接続されていること
  - 分母入力が基本エレメントにのみ接続可能であること

    コンポーネント、演算子、階層ブロックには接続できません。
  - インプリメンテーションキャストが分母入力に接続されている場合、このインプリメンテーションキャストについて、インプリメンテーションエディタの **Use Implemntation Type** オプションの右側のコンボボックスで <No implementation> 以外の項目が選択されていること。

コンポーネント内にこれらの条件が完全に満たされていない演算子インプリメンテーションがある場合（ASCET オンラインヘルプを参照してください）、それらの演算子はマニュアル操作で変換を行う必要があります。

演算子インプリメンテーションをインプリメンテーションキャストに置換する：

- コンポーネントマネージャで **Tools → Database → Convert → Operator Implementations to Impl. Casts** を選択します。

  データベース内のすべての演算子インプリメンテーションが、上記の条件に基づいてインプリメンテーションキャストに変換されます。

演算子（MIN、MAX、MUX 以外）を自動変換すると、以下のような処理が行われます。

- 演算子の出力からの接続線上に、インプリメンテーションキャストが生成されます。
- 除算演算子については、その演算子インプリメンテーションの **Allow zero in phys. interval** オプションがオンになっている場合、その分母の入力への接続線上にインプリメンテーションキャストが生成されます。
- 演算子の出力上に生成されたインプリメンテーションキャストには、接続先のエレメントのインプリメンテーション情報が使用されます。ただしインプリメンテーションキャストのモデル型は、常に cont となります。

- 除算演算子の分母入力への接続線上に生成されたインプリメンテーションキャストには、接続元のエレメントのインプリメンテーション情報（モデル型以外）が使用されます。

注記

演算子にインプリメンテーションが定義されていないコンポーネントのインプリメンテーション（ASCET オンラインヘルプを参照してください）においては、新しく生成されたインプリメンテーションキャストには <No implementation> が選択されています。

- オーバーフローの扱いに関する設定は、以下の規則に従って変換されます。

インプリメンテーションキャスト： / 演算子インプリメンテーション：	Limit to maximum bit length	Reduce Resolution	Keep Resolution
Reduce Resolution	○	○	
Keep Resolution And Limit	○		○
Keep Resolution And Don't Limit			○

  演算子インプリメンテーションの 3 通りの設定は、それぞれ右側の 3 列の組合せに変換されます。

- 演算子インプリメンテーションは、削除されます。

MIN、MAX、MUX 演算子について自動変換の条件が満たされている場合、演算子の削除のみ行われ、インプリメンテーションキャストは生成されません。

自動的に変換されない演算子については、以下のような処理が行われます。

- 演算子出力のすべての接続線上にインプリメンテーションキャストが生成され、<No implementation> が選択されます。

  このインプリメンテーションキャストには、演算子インプリメンテーションをマニュアル変換する際に、適切な実装情報を割り当ててください。

  なお、接続線上にすでにインプリメンテーションキャストが存在していた場合は、インプリメンテーションキャストは追加されません。

- 除算演算子については、演算子インプリメンテーションの **Allow zero in phys. interval** オプションがオンになっている場合、分母入力の接続線上にインプリメンテーションキャストが生成され、<No implementation> が選択されます。

  なお、接続線上にすでに同タイプのインプリメンテーションキャストが存在していた場合は、インプリメンテーションキャストは追加されません。

- 演算子インプリメンテーションはそのまま残ります。

自動変換できない演算子インプリメンテーションがあった場合、以下のようなメッセージが発行されます。

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

このメッセージに対して **OK** を選択すると、“Operator Implementation”ダイアログボックスが開き、変換できなかった演算子インプリメンテーションを含むコンポーネントの一覧が表示されるので、そこで適切なマニュアル変換を行ってください。

5 コード生成に関する設定

生成されるコードのプロパティは、ASCET において以下の 3 通りのレベルで設定／管理されます。

1. 全プロジェクトを対象とするグローバルな設定（コンポーネントマネージャ：**Tools → Options**）
2. プロジェクトごとの設定（プロジェクトエディタ：**File → Properties**）
3. 全プロジェクトを対象に、特定のターゲットについての設定（ターゲットディレクトリ内の `*.ini`、`*.mk`、`*.xml` ファイルを編集）

最初の 2 つの方法については ASCET オンラインヘルプに記述されています。本章では 3 番目の方法について説明します。

全プロジェクトを対象に、特定のターゲットのコード生成は、以下の 3 種類のコンフィギュレーションファイルで制御されます。

1. `codegen[_*].ini`：コードジェネレータのコア部分の制御
2. `target.ini`：ターゲット固有の情報をプロジェクトエディタに提供し、OS コンフィギュレーションの設定に使用
3. `memorySections.xml`：ASCET のインプリメンテーションエディタで使用するメモリクラス名を定義し、対応するターゲット用コンパイラのイントリンシックを提供

ASCET がどのようにコードをコンパイルするかは、GNU Make ファイル（`*.mk`）で制御されます。ASCET はこの Make 処理を実行してプロジェクトをビルドします。

以下の項で、これらの情報について各コンフィギュレーションファイルごとに詳しく説明します。

5.1 `codegen[_*].ini` ファイル

ASCET は以下の 3 つのファイルを用いてコードジェネレータを制御します。

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

  コードジェネレータによって生成されるオブジェクトの命名規則などを定義するマクロが含まれます。このファイルは ASCET 基本システムによってのみ読み込まれます。

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

  コード生成のためのターゲット固有の設定が含まれます。このファイルは、`project_settings.mk` ファイルの `CODEGEN_INI` オプションで参照されます。デフォルトでは、ASCET-SE は `codegen_example.ini` というサンプル用ファイルが使用されます。サンプル用ファイルが使用されないようにするには、`project_settings.mk` ファイルの Make ファイル変数 `EXAMPLE_MODE` を `FALSE` にしてください。

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

  コード生成のためのターゲット固有の設定が含まれます。このファイルは `codegen_<target>.ini` にインクルードされ、ECCO によってのみ読み込まれます。.

これらのファイルは以下のプロパティを制御します。

- コードの外見（変数名など）
- コード生成時の挙動（変数の初期化など）や `#pragma` 文の使用など
- オペレーションシステムのインクルード（メッセージセマンティックの選択、フックルーチンの生成、OIL ディスクリプションファイルの生成など）

`codegen_<target>.ini` ファイルの最初の `[INCLUDE]` セクションには他の `*.ini` ファイルをインクルードすることができます。`codegen_ecco.ini` は自動挿入され、他のファイルは任意に追加します。このセクションは先頭にあるため、先にインクルードされたファイルの設定が処理され、次に `codegen_ecco.ini` の設定が処理されます。このため、特殊な設定を `codegen_ecco.ini` 内に定義しておき、他のファイルの設定をオーバーライドすることができます。

各オプションについての詳細は、`codegen[_*].ini` ファイル内に説明されています。

注記

このコンフィギュレーションファイルは、コード生成開始時に毎回読み込まれるので、変更した内容は次回のコード生成から有効になります。ただしそのためにはプロジェクト内のすべてのコンポーネントについて強制的にコード生成が行われる必要があるので、コード生成開始前に **Build → Touch → Recursive** コマンドを実行するようにしてください。

上記以外に、`codegen.ini` ファイル内の設定を変更することなくユーザー定義の `*.ini` ファイル内でインクルードメカニズムを利用することによって特殊なオプション設定を行うことができます。以下にその方法を示します。

ユーザー定義の `*.ini` ファイルをインクルードする：

- ファイル `<MyInifile>.ini` を作成してターゲットディレクトリに保存します。
- 以下の例のように、`project_settings.mk` ファイル内に `<MyInifile>.ini` をインクルードします。

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

```
CODEGEN_INI =$(P_TARGET)/    ↵
     <MyIniFile>.ini
```

- <MyInifile>.ini ファイル内の先頭位置に [INCLUDE} セクションを追加します。
- テーゲット固有のデフォルトオプションを設定する codegen_<target>.ini ファイルをインクルードします。
- 必要に応じてその他の *.ini ファイルをインクルードします。

```
[INCLUDE]
File1=codegen_<target>.ini
File2=<path>\<filename>.ini
...
```

- [ECCO] セクションを追加してユーザー定義の設定を定義します。

```
[ECCO]
<option1>=<value>
<option2>=<value>
...
```

これらの設定により、インクルードファイル内の設定がオーバーライドされます。

ASCET-SE V6.1 がコード生成時に使用する codegen_*.ini ファイルは、project_settings.mk ファイル内に定義されています。

ASCET-SE V6.1 をデフォルト状態でインストールすると、サンプルディレクトリ内の codegen_example.ini というサンプル用ファイルが使用されます。このファイルが使用されないようにするには、Make ファイル変数 EXAMPLE_MODE を FALSE に設定します。以下に示す project_settings.mk ファイルの先頭部分を変更してください。

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

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

さらに EXAMPLE_MODE を使用するその他の部分も調整が必要です。

5.2 target.ini ファイル

ASCET-SE には、各ターゲットの target.ini というデスクリプションファイルが付属しています。このファイルには、OS エディタの設定（ASCET オンラインヘルプを参照してください）に使用されるエントリが含まれています。また ASCET-SE 内部の設定データも含まれていますが、これらについてはユーザーによる変更はできません。

本項では target.ini で使用されるエントリについて説明します。このファイルは Windows の *.ini フォーマットに従って記述されている必要があります。

デフォルトにおいて target.ini には、汎用の定義、または RTA-OSEK が提供するデフォルトのターゲットマイクロコントローラに関する定義が含まれています。一般的に各ターゲットディレクトリには、複数のバリアントファイル（target_<variant>.ini）が含まれています。この <variant> は RTA-OSEK がサポートするマイクロコントローラターゲットのバリアントタイプに相当します。

1 つのマイクロコントローラのバリアントは、すべて同じ CPU アーキテクチャを使用していますが、ペリフェラルが異なります。つまり、一般的にはバリアントごとに割り込みベクタの数やベクタアドレスと割り込みソースとのマッピングなどが異なるため、ASCET プロジェクトエディタで割り込みを設定するには、正しいバリアントを使用する必要があります。

ターゲットバリアントを選択する：

- target.ini ファイルの名前を target_default.ini に変更します。
- 使用するバリアントファイル target_<variant>.ini を決定します。
- そのファイル名を target.ini に変更します。

target.ini ファイルの内容を以下の表にまとめます。

注記

target.ini ファイルの内容を変更すると、その内容は次回 ASCET を起動した時から有効になります。ターゲットやターゲットバリアントの変更についても同様です。

[Target] *セクション*

type=<*target type*>	各ターゲットの ID。この設定は変更しないでください。
label=<*target name*>	ASCET ユーザーインターフェースに表示するターゲット名
compilerTools=<*compiler list*>	ターゲット用に使用できるコンパイラの一覧。各エントリは空白文字で分割します。
osTools=<*OS list*>	ターゲット用に使用できるオペレーティングシステムの一覧。各エントリは空白文字で分割します。

コンパイラ設定は ASCET オプションダイアログボックスの“External Tools/ <*compiler name*>”ノードで設定します。

OS 設定

maxCoopLevels=<*n*>	協調優先度レベルの最大数。OSEK-OS の場合、6 がデフォルトです。
maxPreempLevels=<*n*>	プリエンプティブ優先度レベルの最大数。numHWLevels + numSWLevel と等しい値です。
numHWLevels=<*n*>	ハードウェア優先度レベルの数。ターゲットのハードウェア割り込み優先度の数と同じです。詳しくは RTA-OSEK ユーザーズガイドまたは各ターゲットの RTA-OSEK バインディングマニュアルのドキュメントを参照してください。
numSWLevels=<*n*>	OS によって定義されたソフトウェア優先度レベルの数。RTA-OSEK の場合、ターゲットに応じて n=16 または 32 です。
event:<*n*>=<*identifier*>, <*x*>,<*y*>,<*address*>[a]	割り込みソースの定義 - n： イベント番号 - identifier： イベント名 - x／ y： 最下位／最上位優先度 - address： 割り込みベクタアドレス

a: これらのエントリは、通常ユーザーが変更するものではありません。

[<osname>] *セクション*

target.ini ファイルには、ターゲットで使用できる各オペレーティングシステムごとに [<osname>] という名前のセクションが 1 つずつ含まれます。

注記

target.ini ファイルにおいては、AUTOSAR RTE はオペレーティングシステムと同等に扱われます。

このセクションでは、ASCET-SE ターゲットがサポートする各 OS のデフォルトパス、ライブラリ名、オプションが定義されます。

`P_OS_INCLUDE`	カンマで区切られた OS 用ヘッダファイルのパス
`P_OS_LIBRARY`	カンマで区切られた OS 用ライブラリファイルのパス
`OS_LIBS`	カンマで区切られた、プロジェクトにリンクする OS ライブラリ
`OS_CONFIG_TOOL_CMD`	OS コンフィギュレーションツール呼び出し用のコマンドラインオプション
`PROJ_OIL_FILE`	サンプルプロジェクトのエントリポイントとして使用される OIL ファイル。OSEK OS の場合にのみ必要です。 **デフォルト：** `<install_dir>¥target¥trg_<targetname>¥example` 内の `conf_<version>.oil` ファイルを参照する `$(EXAMPLE_CONF_OIL)`

これらの値は、ASCET プロジェクトエディタの“Project Properties”ダイアログボックスの“OS Configuration”ノードに自動的に反映されます。これらの設定内容をすべて個々のプロジェクト用に調整する必要はありません。**Enable OS Configuraiton** オプションをオンにしてファイルの内容を読み込み、必要に応じてプロジェクト固有の変更を行ってください。各コンフィギュレーションオプションについての詳細は ASCET オンラインヘルプを参照してください。

デフォルトのパス設定は、OS がインストールされたルートディレクトリを参照する `$(P_OS_ROOT)` からの相対パスで指定されています。このルートディレクトリは、ASCET オプションダイアログボックスの“External Tools/Operation System”の各サブノードで、OS ごとにグローバルに設定されます。

注記

RTA-OSEK 用のデフォルト設定は以下のとおりです。

```
P_OS_INCLUDE = $(P_OS_ROOT)\<targetname>\inc
P_OS_LIBRARY = $(P_OS_ROOT)\<targetname>\lib
OS_LIBS = rtk_s.<lib>
OS_CONFIG_TOOL_CMD = -ds
```

上記の設定は、RTA-OSEK 標準ステータスライブラリを使用し（`rtk_` の次に付加される `s` により指定）、OIL ファイル内の設定に関わらず、RTA-OSEK コンフィギュレーションツールに対して標準ステータスデータ構造体を生成することを強制するものです（コマンド行オプション `-ds` により指定）。
ライブラリやビルドレベルを変更するには、これらのライブラリオプションとツールオプションを変更する必要があります。ライブラリの指示子は、パラメータ `-d` と `s, t, e, ts, tt, te, att, ate` のいずれかです。
例： 拡張（デバッグ）ステータスを使用する場合： `rtk_e.<lib>` および `-de`

5.3 memorySections.xml ファイル

ASCET モデルでは、データとコードをそれぞれ異なるメモリクラスに割り当てることができます。各メモリクラスは抽象的に定義し、一意の名前を割り当てます。たとえば、各メモリセクションを IROM（Internal ROM：内部 ROM）、EXT_RAM（EXTernal RAM：外部 RAM）、FLASH（FLASH memory：FLASH メモリ）といったメモリクラスとして定義します。さらに ASCET コードジェネレータはコンテキスト（参照、仮想パラメータなど）に応じたメモリクラス名を自動生成します。

メモリクラス名は、コード生成プロセスにおいて、実際の名前、コンパイラ固有の `#pragma` 文、および型修飾子に変換される必要があります。メモリクラス名、およびメモリクラス名の変換情報は、XML ベースのメモリセクション定義ファイル `memorySections.xml` 内で定義されます。

各ターゲットのディレクトリにこの名前のサンプルコンフィギュレーションファイルが用意されているので、このファイルの内容を適宜変更して使用してください。`memorySections.xml` ファイルの詳しい記述方法は、各ターゲットディレクトリ内の `ReadMe_memorySections.html` に説明されています。

ANSI C ターゲットには以下の 3 つのサンプルファイルが用意されています。

- `memorySections.xml`：標準のコード生成用メモリセクションが定義されています。非 AUTOSAR コードの生成時に使用します。
- `memorySections_AUTOSAR.xml`：AUTOSAR コード生成用のメモリセクションが定義されています。このファイルは AUTOSAR コードの生成時に自動的に使用されます。各セクションは、AUTOSAR のメモリマッピング（`MemMap.h`）とコンパイラアブストラクション（`Compiler.h`、`Compiler_Cfg.h`）に互換です。
- `memorySections_AUTOSAR4.xml`：AUTOSAR R4.x の手法（関数の引数が参照として渡される場合、const ポインタの代わりにポインタを使用）に基づく AUTOSAR コード生成用のメモリセクションが定義されています。このファイルを使用するには、ファイル名を `memorySections_AUTOSAR.xml` に変更して標準の AUTOSAR 用ファイルの代わりに使用します。

メモリクラスの定義はターゲットとコンパイラに依存します。上記のサンプルファイルを変更する際は、コンパイラのドキュメントを参照してください。

`memorySections.xml` ファイルの先頭部分には、以下の 4 つのメモリクラスカテゴリ用のデフォルトメモリクラスが定義されています。

- `Code` - コード用メモリクラス（メソッド、プロセスなど）
- `Variable` - 変数用メモリクラス
- `Characteristic` - パラメータ用メモリクラス
- `ConstData` - 構造体データ用メモリクラス（コンポーネントの型記述子の情報）

各カテゴリのデフォルトメモリクラスはターゲットに依存します。以下にその例を示します。

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

個々のメモリクラスは <MemClasses> セクションに定義されます。1 つのメモリクラスの定義は、以下のようになります。

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

上記のコード内で*斜体*で書かれている部分には、実際の条件に合わせて適切な値を記述します。各エレメントについての詳細は、各ターゲットディレクトリに保存されている memorySections.xml ファイル（例：...¥ETAS¥ASCET6.1¥target¥tgt_mpc55xx¥memorySections.xml）に記述されています。

文字列エレメントには改行（¥n）を挿入でき、マクロを使用できるものもあります。テンプレート定義に使用できるマクロについても、各ターゲットディレクトリ内の memorySections.xml ファイル内に記述されていますす。

5.3.1 メモリクラスの定義

以下に、メモリクラスを定義して、そのメモリクラスに ASCET 変数を割り当てる手順を説明します。

ステップ 1

変数は、ASCET インプリメンテーションエディタの“Memory Location”コンボボックスで選択されたメモリクラスに割り当てられます。ここに表示されるメモリクラス名は各ターゲットのコンフィギュレーションファイル memorySections.xml（71 ページの 5.3 項を参照してください）内で定義されたものです。

名前を変更したり新しいメモリクラスを追加するには、memorySections.xml ファイルの <MemClassCategories> ブロック内の宣言を編集します。この際、定義する各メモリクラスカテゴリごとにそれぞれ <MemClass> を正しく定義してください。

ステップ2

コンパイル終了時点において、オブジェクトファイル内に存在する各メモリセクションは、マイクロコントローラのメモリ空間内に配置されている必要があります。各メモリセクションとアドレス領域のマッピングはリンカ制御ファイルが行います。リンカ制御ファイルのサンプルが各ターゲットのサンプルディレクトリ（`.¥target¥trg_<targetname>¥example¥`）に格納されているので、これを必要に応じて編集し、使用することができます。

リンカ制御ファイルを独自に作成する場合、`project_settings.mk` ファイル内の変数 `MEM_LAYOUTFILE` を以下の例のように変更し、そのファイルを参照するようにしてください。

```
MEM_LAYOUTFILE  = my_layout_file.inv
```

メモリレイアウトファイルまたはリンカ制御ファイルを変更した場合は、以下の条件が満たされていることを確認してください。

- **`VIRT_PARAM` セクション**

  仮想パラメータは INCA のような適合ツールでのみ使用されるものであるため、このメモリセクションは実メモリ領域の外に配置してください。

- **`VATROM` セクション**

  `VATROM` セクションは実メモリ領域の外に配置し、他のメモリセクションに干渉しないようにしてください。このメモリセクションは、仮想アドレステーブルを集める目的でのみ使用されます。仮想アドレステーブルは、ASAM-MCD-2MC の生成時に HEX ファイルリーダがプロジェクトの全エレメントの正しいアドレスを抽出するために使用するものです。そのため、`VATROM` セクションが使用されているかどうかに関わらず、他のすべてのオブジェクトがメモリ内に配置されている必要があります。

MPC55xx および MPC56x の場合に限り、さらに `a_sect.h` ファイルを追加する必要があります。詳細はコンパイラツールのマニュアルを参照してください。

5.3.2 レガシープロジェクトの移植

ASCET-SE V5.x で開発された ASCET プロジェクトを以降のバージョンの ASCET-SE で使用するには、旧バージョンにおいて 3 つのファイル（`hip.db`、`target.ini`、`codegen.ini`）で定義されているメモリクラスを `memorySections.xml` ファイルに記述しなおす必要があります。

この変換処理を行うための Perl スクリプト（`convert_hip_db.pl`）が各ターゲットディレクトリのサブディレクトリ `.¥target¥trg_<targetname>¥scripts¥` 内に用意されています。

メモリクラス定義を変換する：

- スクリプトファイル `convert_hip_db.pl` を、旧タイプのファイル（`hip.db`、`target.ini`、`codegen.ini`）が保存されているディレクトリにコピーします。

- コマンドラインウィンドウから convert_hip_db.pl を実行します。

注記

必ず、ASCET V6.1 のインストールディレクトリのサブディレクトリに保存されている convert_hip_db.pl を使用してください。旧バージョンの Perl スクリプトを使用すると、変換エラーが発生する可能性があります。

コマンドラインウィンドウに、処理内容が出力され、codegen.ini および target.ini に含まれる所定のエントリと、hip.db からインポートされるメモリクラスが一覧表示されます。
以下の出力例は、codegen.ini に関連するエントリが含まれていない場合の処理結果です。

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

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

T:\M\test>
```

- 必要に応じて、新しく作成された memorySections.xml ファイルの内容を確認してください。

5.4 モデルから実行ファイルへの変換

本項では、ASCET の **Component → Build** メニューコマンドによって実行されるコード生成プロセスについて説明します。ここには、ユーザーライブラリの統合、モジュールの追加などのカスタマイズを行うために必要な情報が含まれています。

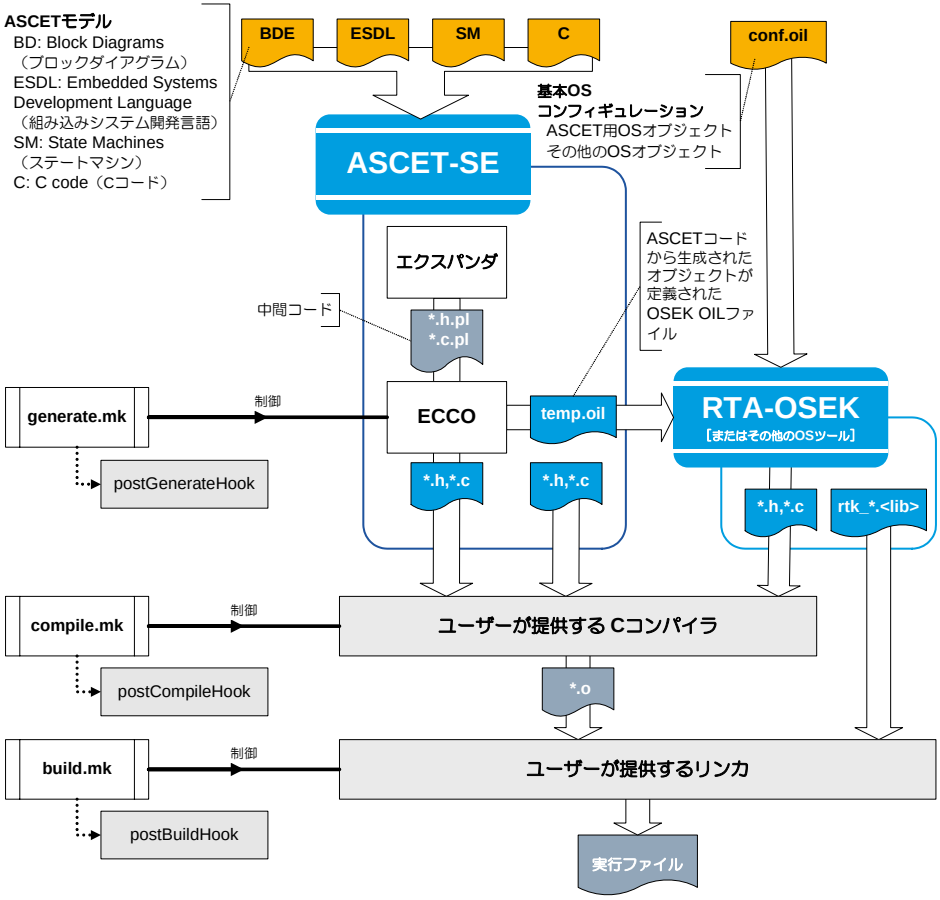


図 5-1　ASCET-SE のプログラム生成プロセス

生成プロセスは以下の 4 つのフェーズで構成されています。

1. **エクスパンダ**

   エクスパンダが ASCET データモデルをハードディスク上の cgen ディレクトリに書き込みます。ASCET で記述された各モジュールは、拡張子 *.db を持つデータベース、拡張子 *.h.pl を持つヘッダファイル、および拡張子 *.c.pl を持つ C ファイルという 3 つのファイルに展開されます。

2. **ECCO**

   cgen ディレクトリ内のデータモデルが ECCO に渡されます。ECCO はデータモデルを分析し、ターゲット固有の調整を行い、コードを最適化します。ECCO は、ターゲットに合った C ファイルおよびヘッダファイルを出力し、この生成プロセスは generate.mk および makefile というファイルにより管理されます。後者はコード生成の各ステップについて、ASCET により自動生成されます。

   また OSEK OIL ファイル（temp.oil）が生成され、基本 OS コンフィギュレーション（confVx.y.oil）から RTA-OSEK が呼び出されて OS データ構造体が生成されます。

3. **コンパイル**

   ECCO と RTA-OSEK により生成された C ソースファイルとヘッダファイルは、ターゲット固有のコンパイラによりコンパイルされます。このプロセスは、いくつかの Make ファイルにより制御されます。ASCET の Make ファイルの拡張子は .mk です（例：project_settings.mk、target_settings.mk）。また ASCET により生成される makefile というファイルには、ユーザーがユーザーインターフェースを通じて入力したすべてのパスや、compile.mk ファイル用のインクルードコマンドが格納されます。以下に、MPC56x をターゲットとした RTA-OSEK の場合の makefile ファイルの一部を紹介します。

   ```
   # path definitions
   P_TGROOT = C:\etas\ascet6.1\target
   P_TARGET = c:\etas\ascet6.1\target\trg_mpc56x
   ...
   P_CCROOT = c:\compiler\diab\5.0.3
   ...
   # phase definition
   include $(P_TARGET)\compile.mk
   ```

   スマートコンパイルによる最適化に続き、コンパイル処理においてさまざまなファイルが生成されて使用され、その結果として、一連のオブジェクトファイルが生成されます。

4. **ビルド**

   コンパイルされたファイルがリンクされて、実行形式のプログラムになります。このプロセスは、build.mk ファイルやターゲット固有の Make ファイルである project_settings.mk および target_settings.mk により制御されます。

以降の項で、このステップがどのように制御されるか、また各ターゲットのサブディレクトリに格納されているコンフィギュレーションファイルを用いてどのようにそれをカスタマイズできるか、という点について説明します。

このプロセスは GNU Make が管理し、Make ファイルが制御します。Make ファイルとビルドスクリプトは、空白文字を含むパスをサポートしています。

- Make ファイルのパスに空白文字が使用されている場合、ASCET はそのパスを Windows のショートネームフォーマットに変換します（例：`c:¥Documents and Settings` というパスは `c:¥DOCUME~1` に変換されます）。
- バッチファイルのパスに空白文字が使用されている場合、ASCET はそのパスを `" "` で囲むか、または Windows のショートネームフォーマットに変換します。

5.4.1 プロジェクト設定 — Make ファイル `project_settings.mk`

この Make ファイルでプロジェクトレベルのコンフィギュレーション設定を定義するもので、各ターゲットディレクトリ（例：`.¥target¥trg_<target>¥project_settings.mk`）に格納されています。

`project_settings.mk` はプロジェクトの要件に合わせてユーザーが編集することができ、このファイルは `compile.mk` および `build.mk` という Make ファイルにインクルードされます。

出荷時における `project_settings.mk` は、「サンプルモードスイッチ」がオン、つまり `EXAMPLE_MODE` 変数が `TRUE` になっています。これはつまり、コード生成時にサンプルファイル（38 ページの「ディレクトリ `.¥target¥trg_<targetname>¥example`」を参照してください）内の設定が使用されることを意味します。これ以外の独自のコンフィギュレーションファイルを使用するには、このスイッチをオフに切り替え（`EXAMPLE_MODE=FALSE`）、`project_settings.mk` 内の設定を適宜変更してください。

また `STOPWATCH_TICK_DURATION` 変数は、dT 時間を参照する際の 1 チック分の時間を ASCET に対して知らせるものです。ASCET 内で正確な dT タイミングを得るためには、実際のターゲットハードウェアのコンフィギュレーションに合わせてこの値を設定してください。

5.4.2 ターゲットとコンパイラ設定 — Make ファイル `target_settings.mk` および `settings_<compiler>.mk`

Make ファイル `target_settings.mk` は、コンパイルとリンクを制御する 2 つの Make ファイル（`compile.mk` および `build.mk`）にインクルードされ、もう 1 つの Make ファイル `settings_<compliter>.mk` をインクルードします。

`settings_<compliter>.mk` は、ファイル拡張子や、プリコンパイラ、コンパイラ、リンカ、その他のプログラムの呼び出し規則、およびプログラムコール、インクルードファイル、ライブラリのパスを定義します。コンパイラおよびリンカ呼び出し時のコマンドラインパラメータもこのファイルに定義されています。

COMPILER SETTINGS セクションの値セットを変更して、プロジェクトプロパティでプリセットされているものと異なるコンパイラを使用することができます。ただしその際は、コンパイラ固有の設定をすべて正しく設定する必要があります。

5.4.3 Make ファイル generate.mk

この Make ファイルの内容は変更しないでください。このファイルは ECCO のプロセスを制御するもので、ここですべてのプロジェクトおよびターゲット固有のファイルが ECCO に渡されます。たとえば、プロジェクト用に生成される全ヘッダを含む Make 変数 FILES_HEADER_PROJ は、ここで定義されます。

5.4.4 Make ファイル compile.mk

この Make ファイルはコンパイルプロセスを制御します。プロジェクトに関連するすべてのファイルが、ここで適切なオプションを使用してコンパイルされアセンブルされます。その結果、すべてのオブジェクトファイルが cgen ディレクトリに書き込まれます。さらにすべてのコンパイラエラーが評価され、必要に応じて ASCET に渡されます。コンパイル時にエラーが発生すると、生成プロセスが終了し、エラーウィンドウが表示されます。

スマートコンパイル

ASCET-SE のコード生成には、前回ビルドが実行された後に内容が変更された C ソースファイルのみを再コンパイルする機能があります。各コードは明示的に比較され、再コンパイルの必要があるかどうかが調べられます。

スマートコンパイルは、2 つの Make 変数によって制御されます。

- COMPILE_MODE：スマートコンパイル機能を使用するかどうかを指定します。値は、smartCompile（デフォルト：スマートコンパイルを行う）または compile（従来のコンパイル）のいずれかです。
- SMART_COMPILE_COMPARE：ファイル比較モードを指定します。値は、smart（デフォルト：コメント内に生成される日時のみを無視）、strict（すべてを厳密に比較）、relaxed（コメントをすべて無視）のいずれかです。

スマートコンパイルの場合、コードの比較時にいくつかの中間ファイルが生成されますが、これらのファイルはユーザーには関係しません。

スマートコンパイルを行うと、前バージョンに比べて Make ファイルの数が増えて複雑になってしまいます。それを避けるため、project_settings.mk ファイル（および必要に応じて target_settings.mk）のみを変更することを強くお奨めします。

5.4.5 ビルド ― Make ファイル build.mk

この Make ファイルはリンクプロセスを制御します。コンパイルされたオブジェクトファイルと必要なライブラリが統合されて実行ファイルとなり、CGen ディレクトリに書き込まれます。

ビルドプロセスをカスタマイズするには build.mk にインクルードされる project_settings.mk を編集します。build.mk ファイル自体の編集はできるだけ避けてください。

リンカ／ロケータの制御

build.mk ファイルによって制御されるビルドプロセスは、コンパイラツールチェーンに含まれるリンカ／ロケータを使用して、実行プログラムの各要素（コード、静的データ、動的データなど）をマイクロコントローラの物理メモリ領域（RAM、ROM など）に割り当てます。このプロセスはリンカ／ロケータ制御ファイルによって制御され、このファイルのフォーマットはコンパイラツールチェーンに応じて異なります。ファイルの内容はマイクロコントローラのバリアント（デバイス、メモリのサイズやレイアウト）に応じて異なるため、適切なファイルを選択する必要があります。

ASCET が使用するリンカ／ロケータファイルは project_settings.mk ファイル内の MEM_LAYOUTFILE 変数によって指定されます（5.4.1 項参照）。この変数は、実際に使用するマイクロコントローラ用の正しいリンカ／ロケータファイルを参照している必要があります。

ASCET-SE には各ターゲット用にリンカ／ロケータファイルのサンプルが用意されていて、このファイルは .¥target¥trg_<targetname>¥example ディレクトリに格納されています。

このファイルを変更する際は、コンパイラとマイクロコントローラのマニュアルを参照してください。

5.4.6 コード生成のカスタマイズ

バナー

生成されるコード内にユーザー定義のバナー（見出し）を挿入することができます。詳しくは ASCET オンラインヘルプを参照してください。

生成されたコードのフォーマット変更 ― コンフィギュレーションファイル .indent.pro

生成されたコードのフォーマットを変更するには、コードフォーマッティングユーティリティ“Indent”を使用します。フォーマットは、各ターゲットディレクトリ内の .indent.pro というファイルに定義します。Indent の機能については、ASCET-SE のドキュメントと共にインストールされるファイル（<install dir>¥..¥ETAS Manuals¥ASCET V6.1¥Tools¥indent.html）に詳しく説明されています。Indent は、“GNU Public License”のもとで再配布されるものです。

コードのポストプロセッシング

ASCET-SE では、生成されたコードを Perl スクリプトを用いて再加工することができます。呼び出されるスクリプトは、以下の例のように、`project_settings.mk` 内の `POST_CGEN_PERL_MODS` という Make 変数で指定します。

```
POST_CGEN_PERL_MODS = postCGenIndent postCGenSample
```

`.¥target¥trg_<targetname>¥scripts` ディレクトリに `postCGenSample.pm` というサンプルファイルが用意されていて、このファイルを変更することによって容易に呼び出しを定義することができます。スクリプトは、以下の規則に従って記述してください。

- `process` という名前の Perl マクロで定義します。
- 呼び出し時の引数を 3 つ用意します。これらの引数は、それぞれソースコードのパス、C ファイルのリスト、H ファイルのリストを表します。

例：

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

出荷時の状態では、ASCET-SE は“Indent”というコードフォーマッティングユーティリティを使用し、このユーティリティも、同じメカニズムで呼び出されます。以下のように記述すると、Indent ユーティリティは機能しません。

以下のように記述すると“Indent”の実行は省略されます。

```
POST_CGEN_PERL_MODS =
```

“Indent”についての詳細は 79 ページ「生成されたコードのフォーマット変更 ― コンフィギュレーションファイル `.indent.pro`」を参照してください。

5.4.7 ビルドプロセスのカスタマイズ

ユーザー定義の Make ファイルをインクルードする

ビルドプロセス内の選択されたポイントにおいてユーザー独自の Make ルールを実行することにより、ASCET の Make プロセスをカスタマイズすることができます。この目的のため、ASCET-SE には以下のような特殊なフックポイントが用意されています。

- `PRE_GENERATE_HOOK` ― コード生成の前
- `POST_GENERATE_HOOK` ― コード生成の後
- `PRE_COMPILE_HOOK` ― コンパイルの前
- `POST_COMPILE_HOOK` ― コンパイルの後
- `PRE_BUILD_HOOK` ― リンクの前
- `POST_BUILD_HOOK` ― リンクの後
- `POST_FILEOUT_HOOK` ― ファイル出力後

フックは custom_settings.mk 内に定義します。

ユーザー定義の Make ファイルは、GNU の Make 構文で記述してください。ASCET-SE 製品には GNU Make マニュアルが含まれていて、製品インストール時に <*install_dir*>¥ETASManuals¥ASCETx.y¥Tools にインストールされます。さらに詳しい情報は、GNU-Make Manual（ISBN：1-882114-80-9、ASCET-SE 製品には含まれません）を参照してください。

ユーザー定義の C ファイルと H ファイルを含める

ハンドコーディングされた C ソースファイルを ASCET-SE の Make プロセスに含めることができます。project_settings.mk ファイル内にファイルのリストを定義し、さらにコンパイラがそれらのファイルを検索する際に使用するパス名リストも定義できます。ここでは以下の Make 変数を利用できます。

- C_INTEGRATION：Make プロセスで C ソースファイルを追加するかどうかを指定します。値は FALSE または TRUE です。

  注記

  RTA-OSEK の場合、ASCET-SE が生成するタスクと ISR のボディは、C コード統合メカニズムによってコンパイルされる個別のファイルに格納されるため、C_INTEGRATION は TRUE に設定しておく必要があります。

- P_C_SRC_FILES： 追加する C ソースファイルのパス（複数可）のリストです。各パスはスペースで区切ってください。
- C_SRC_FILES： 追加する C ソースファイル名（複数可）のリストです。各ファイル名はスペースで区切ってください。ここでファイル名リストを含むファイルを指定する場合、そのファイルのパスが P_C_SRC_FILES で指定され、なおかつ C_INTEGRATION が TRUE である必要があります。
- P_H_SRC_FILES： 追加する H（ヘッダ）ファイルのパス（複数可）のリストを指定します。 各パスはスペースで区切ってください。
- LIBS_USER：ユーザー定義ライブラリ（複数可）のリストです。ファイル名にはパスも含めてください。
- P_ASM_SRC_FILES：追加するアセンブラファイルのパスのリストです。各パスはスペースで区切ってください。
- ASM_SRC_FILES：追加するアセンブラファイルのリストです。各ファイルはスペースで区切ってください。

以下に、Make ファイル変数の使用方法の例として、project_settings.mk の一部を紹介します。

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

C_SOURCE_FILES リストに定義されたファイルが、ASCET の Make プロセスによってコンパイルされ、リンクされます。

ASCET 専用の Make ファイル変数

ASCET では、システム定義されている場所に格納されているファイルにアクセスするための特殊な Make 変数が利用できます。

- $(P_TARGET) ― カレントターゲットのインストールパス（例：.¥target¥trg_mpc56x）
- $(P_TGROOT) ― ASCET のインストールパス内の .¥target パス
- $(P_DATABASE) ― 現在使用されている ASCET データベースのパス
- $(P_CGEN) ― コード生成用ディレクトリ

Make 変数についての詳しい情報は、project_settings.mk 内のコメントを参照してください。

5.5 ASCET ヘッダファイルを用いたコンパイル内容のコントロール

ASCET-SE で生成される C コードには、さまざまな C プリプロセッサ命令が含まれ、これにより、ASCET-SE ヘッダファイルを使用したコンパイル時のコンフィギュレーション設定が可能になります。ヘッダファイルは .¥trg_<targetname>¥include に保存され、例外がある場合はその旨が記述されています。

5.5.1 インクルードファイル a_basdef.h

a_basdef.h ファイルは、ASCET が生成するすべてのファイルにヘッダファイルとしてインクルードされます。このファイルには、以下の内容を持つ別のヘッダファイルがインクルードされます。

- ASCET の標準的な型の定義（a_limits.h、a_std_type.h）
- ターゲット固有の定義（tipdep.h）
- オペレーティングシステムに関する定義（os_inface.h）
- プロジェクト固有のユーザー定義（proj_def.h）

  a_basdef.h ファイルが格納されているフォルダと同じフォルダ内に proj_def.h のテンプレートが用意されているので、その内容を必要に応じて調整してください。

a_basdef.h ファイルの内容は変更できません。

5.5.2 インクルードファイル proj_def.h

出荷時において、このファイルにはいくつかのマクロ定義の他に空のセクションが含まれていて、ここで各アプリケーションに固有な調整を行うことができます。

特に、このファイルには ASCET のコード生成に一貫してオンなプリプロセッサコマンドを含めることができます。以下のスイッチは、コードにおいてそれぞれ特殊な意味を持ちます。

- COMPILE_UNUSED_CODE：このスイッチを定義すると、ASCET モデルから生成されたコードのうちで、モデル内で実際に使用されない部分（一度も呼び出されないメソッドなど）がコンパイルされます。
  例：
  ```
  #define COMPILE_UNUSED_CODE
  ```
- DECLARE_PROTOTYPE_METHODS：ASCET では、クラスをプロトタイプとして実装することができ（4.2.3「プロトタイプ実装」を参照してください）、このスイッチが定義されていると、宣言文（または外部宣言文）が各モデルごとに生成されます。これは、プリプロセッサコマンド（#define）によってメソッド名をマクロに割り当てる場合に使用されます。
  例：
  ```
  #define DECLARE_PROTOTYPE_METHODS
  ```
- DECLARE_INLINE_METHODS：このスイッチを定義すると、インラインモードで実装されたメソッド（4.2.4「プロセスとメソッド」を参照してください）について、関数の宣言文が生成されます。インライン関数用の外部宣言は通常は必要ありませんが、これは関数の内容が展開されるので、実際に使用される前に定義されていなければならないためです。
  例：
  ```
  #define DECLARE_INLINE_METHODS
  ```
- 個々の Extern 宣言文とタイプ宣言文に関するモデル固有のスイッチ
- メッセージコンフィギュレーション用スイッチ：オペレーティングシステムの優先度機構に基づくメッセージコピーに関するデフォルトの最適化は、必ずしもすべてのアプリケーションに適応するとは限りません。そのため、メッセージハンドリングは codegen_ecco.ini ファイル内の modularMessageUse オプションがオンになっている場合に限り、以下の 4 通りに設定されます。

  – デフォルトのメッセージ最適化を行う

  デフォルト設定では、メッセージはオペレーティングシステムの優先度機構に基づいて最適化されます。この場合、コンパイラスイッチを以下のように指定します。

  ```
  #define __MESSAGES __OPT_COPY
  ```

  これはユーザーによって明示的に指定することもできます。

- **メッセージコピーを作成しない**

  メッセージは、グローバル変数として扱われ、コピーは作成されません。この場合、コンパイラスイッチを以下のように指定します。

  ```
  #define __MESSAGES __NO_COPY
  ```

  注記

  モジュール内のメソッドの場合、__OPT_COPY と NO_COPY しか使用できません。他の最適化はサポートされていません。

- **メッセージの最適化を行わない**（コピーは作成する）

  コンパイラスイッチを以下のように指定すると、メッセージは常にコピーされますが、最適化は行われません。

  ```
  #define __MESSAGES __NON_OPT_COPY
  ```

- **OSEK-COM メッセージを使用**（オペレーティングシステムが対応している場合のみ可能）

  ```
  #define __MESSAGES __OSEK_COM
  ```

6 補間ルーチン

注記

ASCET 製品に含まれている補間ルーチンはサンプルとして使用することのみを目的に作成されたものです。実際の ECU ソフトウェアで使用することを目的に作成されたものではありません。詳しくは 2.1 項を参照してください。

プロジェクトが特性カーブ／マップを使用している場合、補間ルーチンを用意する必要があります。ASCET-SE には、補間ルーチンライブラリ `intpol_<target>_<compiler>.<libext>`*[1] とヘッダファイル `a_intpol.h`[2] が含まれています。これらのファイルには、特性カーブ／マップの補間を行うためのルーチンが、さまざまなデータ型の組み合わせで用意されています。

特性カーブおよびマップについては、入力と出力のデータ型の組み合わせは 500 通り以上存在し、そのそれぞれについて独自の補間ルーチンが必要となりますが、実際のプロジェクトで使用される組み合わせは通常 10 通り以下である場合がほとんどです。また、500 以上もの補間ルーチンをすべて組み込むことは現実的でないため、製品に含まれるライブラリには典型的な組み合わせのものだけが含まれています。

このライブラリに含まれていないルーチンは、必要に応じて自動生成することができます。これにはバッチファイル `intpol_<target>_<compiler>.bat`[2] とシステムに付属している Perl インタープリタを使用します。生成されたファイルはコンパイルされ、新しいライブラリファイルが作成されます。

注記

補間ルーチンの作成方法は、補間ルーチンディレクトリ `.¥target¥trg_<targetname>¥Intpol` に格納されている `HowTo.html` というファイルに記述されています。

ライブラリでは、通常の特性カーブ／マップ、およびグループ特性カーブ／マップについて、以下の型の組み合わせをサポートしています（固定特性カーブ／マップの場合は、補間は補間ルーチンを呼び出さずに行われます）。

2D ルーチン： 以下のようなデータ型の組み合わせのルーチンがあります。以下の表記では、最初のデータ型が軸ポイント（補間ノード）を示し、2 番目のデータ型が特性値（出力値）を示します。

`r32r32、s16s16、s16s8、s16u16、s16u8、s32u16、s8s16、s8s8、s8u16、s8u8、u16s16、u16s32、u16s8、u16u16、u16u8、u32u16、u8s16、u8s8、u8u16、u8u8`

1. 格納場所： `.¥target¥trg_<targetname>¥Intpol¥lib`
2. 格納場所： `.¥target¥trg_<targetname>¥Intpol`

3D ルーチン： 以下のようなデータ型の組み合わせのルーチンがあります。以下の表記では、最初のデータ型が X 軸ポイント（補間ノード）、2 番目のデータ型が Y 軸ポイント、3 番目のデータ型が特性値（出力値）をを示します。

```
r32r32r32, s16s16s16, s16s16s8, s16s16u16, s16s16u8,
s16s8s16, s16s8s8, s16s8u16, s16s8u8, s16u16s16,
s16u16s8, s16u16u16, s16u16u8, s16u8s16, s16u8s8,
s16u8u16, s16u8u8, s8s16s16, s8s16s8, s8s16u16,
s8s16u8, s8s8s16, s8s8s8, s8s8u16, s8s8u8, s8u16s16,
s8u16s8, s8u16u16, s8u16u8, s8u8s16, s8u8s8, s8u8u16,
s8u8u8, u16s16s16, u16s16s8, u16s16u16, u16s16u8,
u16s32u16, u16s8s16, u16s8s8, u16s8u16, u16s8u8,
u16u16s16, u16u16s8, u16u16u16, u16u16u32, u16u16u8,
u16u8s16, u16u8s8, u16u8u16, u16u8u8, u8s16s16,
u8s16s8, u8s16u16, u8s16u8, u8s8s16, u8s8s8, u8s8u16,
u8s8u8, u8u16s16, u8u16s8, u8u16u16, u8u16u8, u8u8s16,
u8u8s8, u8u8u16, u8u8u8
```

6.1 補間ルーチンの使用方法

各ターゲット用に ETAS はサンプルの補間ルーチンを提供していますが、これらのサンプルは製品プロジェクト用にそのまま使用されることを前提として作成されたものではありません。これらのルーチンは、補間ルーチンの生成方法やプロジェクトでの使用方法のデモンストレーションを行うためのもので、実際の要件にあった補間ルーチンを作成する際の参考として用いることもできます。

ASCET-SE をインストールすると、各ターゲット用ディレクトリに `¥Intpol` というサブディレクトリが生成されます。

例：`C:¥ETAS¥ASCET6.1¥target¥trg_<targetname>¥Intpol`

補間ルーチンの作成や変更の方法は、`HowTo.html` というファイルに説明されています。

補間ルーチンは、算術演算サービス（「Arithmetic Service」）を用いて作成されています。算出演算サービスについての詳細は ASCET オンラインヘルプを参照してください。

補間ルーチンからは以下の ASCET 算術演算サービスが呼び出されます。

算術演算サービス	機能
`GETAT`	特性カーブ／マップへのアクセス
`GETATFIXED`	固定特性カーブ／マップへのアクセス
`INTERPOLGROUP`	グループ特性カーブ／マップ用補間ルーチン
`SEARCHDISTRIB`	グループ特性カーブ／マップのディストリビューション検索

ASCET における算術演算サービスの使用例として、特性カーブ用の `GetAt()` を用いて新しい補間ルーチンを作成することができます。

uint8 型の値の場合、GetAt() の呼び出しはユーザー固有の CharTable1_getAt_u8_u8() メソッドに置き換わります。このメソッドは、生成される HEX ファイルにリンクされるライブラリなどに統合され、ヘッダファイル a_intpol.h 内に宣言されている必要があります。

サンプルのソースファイルを使用する場合は、HowTo.html というファイルに記述された注意事項をよくお読みいただいた上で、ライブラリの生成や、Make 処理内でのリンクを行ってください。

6.2 補間処理

どのバリエーションについても、補間処理は以下の 2 つのステップで行われます。

1. 補間ポイントの位置（両側の x 軸ポイント）を検索して、オフセット（補間ポイントと、基準となる特性値の x 座標値との距離）を算出します。
2. 目的の座標の、線形補間値を算出します。

グループ特性カーブ／マップの場合、検索結果は中間変数に格納されるので、さまざまな特性カーブ／マップについて値が何度も計算されることはありません。

軸ポイント（補間計算に使用される補間ノード）が等間隔に分布している特性カーブ（固定特性カーブ）の場合、格納されるのは座標値のリストではなくオフセットと距離だけなので、メモリは少なくてすみます。検索は行われず、目的の x 座標値に最も近い固定軸ポイントが使用されます。

6.3 値の許容範囲と精度

提供されている補間ルーチンは、整数インプリメンテーションの計算においては、正確な整数値 ±1.0 以内の値が算出されます。

隣り合う特性値間の距離は、補間時のオーバーフロー発生を防ぐため、無制限に大きくすることはできません。

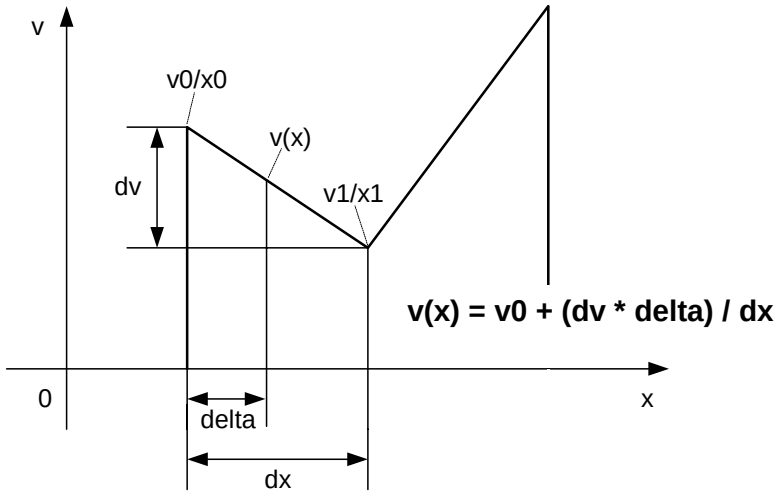


図 6-1 特性カーブの補間

オーバーフローを防ぐための条件は以下のとおりです。

$(dv * dx) < 2^{31}$ （dv > 0、正の傾き）

$(dv * dx) \geqq -2^{31}$ （dv < 0、負の傾き）

したがって、特性カーブの勾配が非常に急な場合（隣り合う特性値間の差が非常に大きい場合）は、軸ポイント（補間ノード）の数を増やす必要があります。

現在のインプリメンテーションでは、データ型 `uint16`、`sint16`、`uint32`、`sint32` を使用するすべてのルーチンが対象となります。オーバーフロー発生により不正な結果が出るのを避けるため、これらのルーチン内で結果がチェックされます。算出された値が隣り合う 2 つの軸ポイントの値の範囲外だった場合は、算出された値の代わりに小さい方の軸ポイントの値が返されます。

浮動小数点数値の補間アルゴリズムは、整数値の補間アルゴリズムと比べてわずかに異なります。理論上は、浮動小数点数値の場合もオーバーフローが発生する可能性があります。

7 オペレーティングシステムの統合

この項では、ASCET-SE とオペレーティングシステムを統合して ASCET プロセスのリアルタイムスケジューリングを実現する方法について説明します。

ここでは OSEK OS、特に ETAS の RTA-OSEK との統合について説明します。他の OSEK 互換オペレーティングシステムの場合、おおまかな内容は同じですが、詳細な点において相違があります。

OS との統合を行うために、ASCET は以下のものを生成します。

- OS を設定して ASCET タスクと割り込みを実行できるようにするための複数の OS コンフィギュレーションファイル
- OS から呼び出される OS タスクと割り込みボディの C コード

また OS との統合を行うために ASCET が必要とするものは、以下のものです。

- ASCET タスクをスケジューリングするために必要な OS オブジェクトを設定するための、システム全体の OS コンフィギュレーションファイル
- ターゲットハードウェアを設定し､所定のアプリケーションモードで OS を起動する main プログラム
- dT モデル変数を提供するためのコールバック関数

7.1 スケジューリングと優先度機構

OSEK OS 内のタスクは、コンフィギュレーション設定時において静的に優先度を割り当てます。優先度ゼロは最低の優先度を示し、数値が大きくなるほど優先度が高くなします。

OSEK 内のタスクは「プリエンプティブ」または「ノンプリエンプティブ」にスケジューリングでき、これを選択するには、ASCET タスクコンフィギュレーションにおいて“Scheduling”オプションを `FULL` または `NON` に設定します（詳しくは ASCET オンラインヘルプを参照してください）。

さらに ASCET は、これら標準の OSEK OS スケジューリングモードに加え、OSEK OS の機能を用いた「協調スケジューリング」もサポートしています。この場合はは、ASCET タスクコンフィギュレーションにおいて“Scheduling”オプションを `COOPERATIVE` に設定します（詳しくは ASCET オンラインヘルプを参照してください）。

「プリエンプティブタスク」は、そのタスクよりも優先度の高いタスクまたは割り込みが随時プリエンプト（中断）できます。

「ノンプリエンプティブタスク」は、プリエンプティブタスクと協調タスクにプリエンプトできますが、他のタスクがこのタスクにプリエンプトすることはできません。ノンプリエンプティブタスクの実行中に、より優先度の高いタスクがレディ状態になっても、そのタスクはノンプリエンプティブタスクの実行が終了するまで待つ必要があります。しかしノンプリエンプティブタスクは割り込みにはプリエンプトされます。

「協調タスク」は、どの時点においても、プリエンプティブタスクとノンプリエンプティブタスク、および割り込みによるプリエンプトが可能です。ただし他の協調タスクによるプリエンプトは、1 つのプロセスが終了した時点においてのみ可能です。

このようなタスクモデルをサポートするため、ASCET は OSEK OS のタスク優先度空間を 2 つのパートに分けています。

1. 協調スケジューリング用優先度
   協調スケジューリングに使用される優先度レベルの値は、コンフィギュレーションオプション **Coop. Levels**（プロジェクトエディタの“OS”タブ）で定義します。この値により、協調タスクの優先度は 0 ～ **Coop. Levels**-1 の範囲に割り当てられます。

   このオプションの最大値は `target.ini` ファイル内の `maxCoopLevels` で定義され、デフォルト値は 6 です。
2. プリエンプティブおよびノンプリエンプティブスケジューリング用優先度
   優先度レベルの値は、使用するターゲットにおいて RTA-OSEK がサポートするタスクの数から協調レベルの最大数を差し引いた値です。これは `target.ini` ファイル内の `numSWLevels - maxCoopLevels` と同じ値になります。

   これにより、プリエンプティブタスクとノンプリエンプティブタスクの優先度は `0` ～ `numSWLevels - 1` の範囲に割り当てられます。

OS コンフィギュレーションが生成される際、OSEK OS の優先度機構の上に上記のような ASCET の「パート分け」が追加されます。

割り込みについては、ASCET は RTA-OSEK の IPL（**I**nterrupt **P**riority **L**evel：割り込み優先度レベル）を使用します。このモデルでは、RTA-OSEK がすべてのターゲットマイクロコントローラ用に IPS を標準化しています。IPL=0 はすべてのタスクが実行される「ユーザーレベル」で、1 以上が「割り込みレベル」[1] になります。割り当てできる最大の IPL は、マクロコントローラがサポートする最高優先度の OSEK OS のカテゴリ 2 の ISR の優先度と同じです。これは各ターゲットディレクトリに保存されている `target.ini` ファイル内の `numSWLevels` の値と同じです。

注記

IPL はタスク優先度とは異なりますので、混同しないように注意してください。IPL=1 はユーザーアプリケーションで使用される最高のタスク優先度よりも高いレベルです。

[1] IPL コンセプトについては RTA-OSEK ユーザーズガイドに詳しく説明されています。各 IPL がどのようにターゲットハードウェアの割り込み優先度に割り当てられるかは、各マイクロコントローラ用 RTA-OSEK バインディングマニュアルを参照してください。

図 7-1 は、OS と ASCET におけるタスクと割り込みの優先度の関係を示しています。

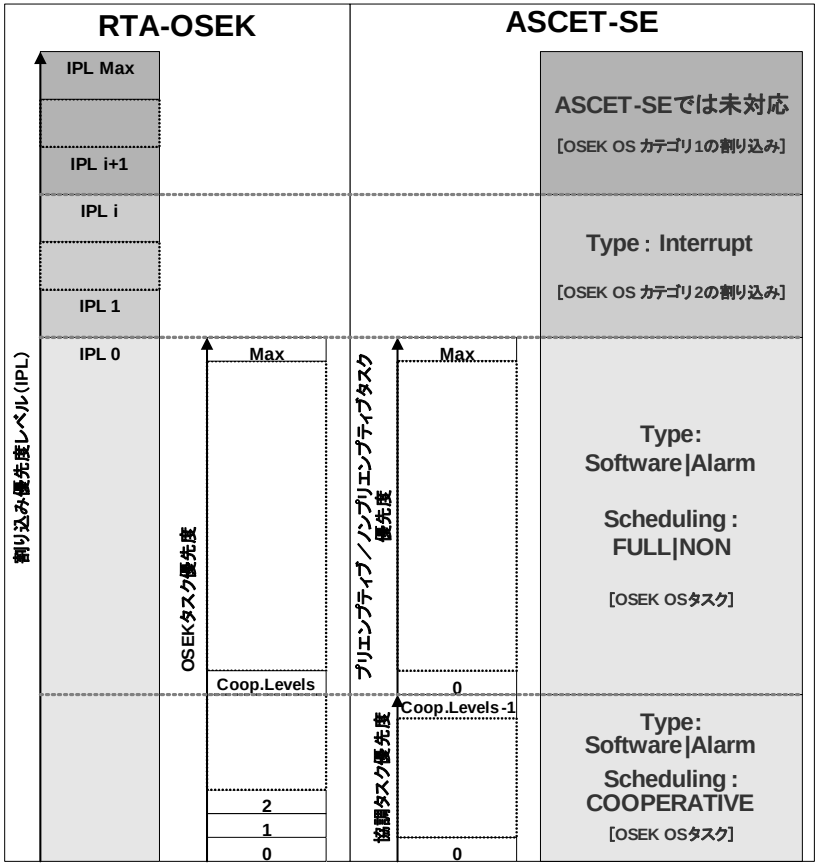


図 7-1 優先度レベル

7.2 プロジェクトの設定

7.2.1 ASCET の OS コンフィギュレーションファイルの生成

RTA-OSEK または Generic OSEK 用のコード生成中に、設定済みの OS コンフィギュレーションファイルを用いて `temp.oil` という OS コンフィギュレーションファイルが自動的に生成されます。このファイルには、ASCET 内に宣言された OS オブジェクト用の OIL（**O**SEK **I**mplementation **L**anguage：OSEK 実装言語）[1] コンフィギュレーション（タスク、ISR、リソース、メッセージ、アラーム、アプリケーションモードなど）が格納されます。

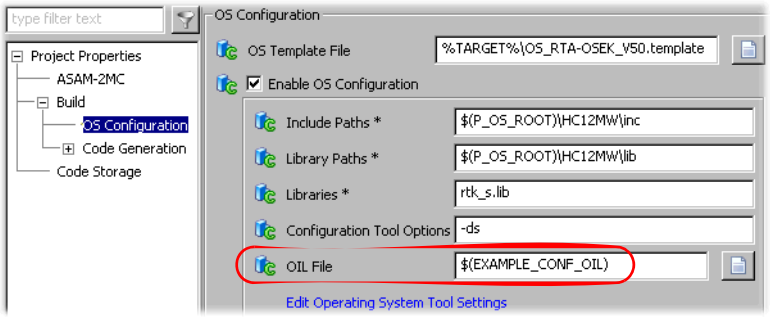


図 7-2　プロジェクト設定における OS とテンプレートの選択

7.2.2 補足的 OS コンフィギュレータの追加

`temp.oil` ファイルに含まれる OS コンフィギュレーションは、完全なものではありません。ASCET と OS の統合にはさらに補足的な OS コンフィギュレーションとして、以下のものを定義する必要があります。

- グローバル OS 設定（ビルドステータス、エラーロギングモード、必要なフックルーチンを含む）を定義する OSEK OS オブジェクト
- ASCET が生成するアラームタスクを駆動するために使用されるカウンタを定義する OSEK カウンタ。カウンタの名前は ASCET の設定（`codegen.ini` ファイル内の `OIL-COUNTER-name` で定義された設定）と一致する必要があります。`OIL-COUNTER-name` のデフォルト設定は `SYSTEM_COUNTER` です。
- カウンタ用リアルタイムチックを供給するための OSEK カテゴリ 2 ISR

これらの補足的コンフィギュレーションは、フレームワーク OIL ファイルとして提供されています。プロジェクトで使用するフレームワークファイルは、図 7-2 に示されるように、プロジェクトプロパティの “OS Configuration” ノードの “OIL File” フィールドで指定します（詳しくは ASCET オンラインヘルプを参照してください）。

[1] OIL についての詳細は www.osek-vdx.org を参照してください。

RTA-OSEK との統合に使用できるサンプルのフレームワーク OIL ファイルが、サンプルアプリケーションと共に下記の場所に用意されています。この場所は `$(EXAMPLE_CONF_OIL)` というマクロで参照できます。

`..¥target¥trg_<`*`targetname`*`>¥example¥conf<`*`version`*`>.oil`

サンプルのフレームワーク OIL ファイルをコピーしてから、そのファイルの設定を必要に応じて変更してお使いいただくことをお勧めします。

`conf<`*`version`*`>.oil` は RTA-OSEK 用のものです。RTA-OSEK は「スマートコメント」（`//RTAOILCFG` または `//RTAOSEK` という形式の OIL コメント）を使用して OIL で定義されていない補助的な OS コンフィギュレーション（割り込み優先度レベルや割り込みベクタアドレスなど）を定義します。

定義されるオブジェクトは以下のものです。

- **CPU** — 他のすべてのオブジェクト用コンテナ
- **OS** — OS 属性の定義
- **COUNTER** — アラームタスクのタイムベースを定義するために必要なシステムカウンタ。ASCET-SE においてこのカウンタのデフォルト名は `SYSTEM_COUNTER` です。デフォルト名を変更するには `codegen_ecco.ini` ファイル内の `OIL-COUNTER-name` オプションを変更してください。

  例：

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

- **ISR** — システムカウンタを駆動するためのカテゴリ 2 割り込み。ISR の名前は重要ではありませんが、ASCET-SE は便宜的に `system_counter` という名前を使用します。

  例：

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

```
                OS_BASE OS_WCSU {OS_UNDEFINED }; };
    //RTAOSEK OS_TRACE 0;
  };
```

- **COM** — OSEK COM を使用するメッセージ通信のプロパティ定義

  例：

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

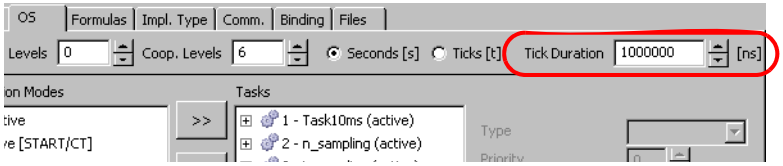
ここには、上記以外の OIL オブジェクトや RTA-OSEK 固有の補足的なコンフィギュレーション情報（詳細は RTA-OSEK のユーザードキュメントを参照してください）も定義することができます。

生成された `temp.oil` ファイルは、RTA-OSEK の補足的 OIL ファイルメカニズムを用いてインクルードされます。このインクルードメカニズムは以下のように OIL の **CPU** ブロックの後ろに配置されます。

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

`system_couner` という ISR は、外部の C コードで実装します。各 ASCET ターゲット用の実装例が、`...¥target¥trg_<target>¥example¥target.c` というファイルに用意されています。詳しい情報は RTA-OSEK ユーザーズガイドを参照してください。

各カウンタチックの間隔 `SYSTEM_COUNTER` はナノ秒単位（一般的には `system_couner` という ISR の起動周期と同じになります）で、コード生成を行う前に ASCET の OS エディタの “Tick Duration” フィールドに入力しておく必要があります。RTA-OSEK ベースのシステムの場合、この値は、生成される `oscomn.h` ファイル内の `OSTICKDURATION_SYSTEM_COUNTER` マクロの値と一致している必要があります。



ASCET は、アラームタスク用のチック／時間変換の目的にのみ、Tick Duration の値を使用します。

7.3 メインプログラムの提供

メインプログラム（通常、`main` と呼ばれます）の役割は、ターゲットハードウェアの初期化と、指定されたアプリケーションモードでの OS の起動です。

デフォルトにおいて、ASCET プロジェクトのビルドでは外部のメインプログラム（`..¥target¥trg_<targetname>¥example¥main.c`）が使用されます。このファイルは組み込みターゲット用のサンプルのメインプログラムで、ハードウェアを設定して `system_counter` 割り込みを 1ms ごとに生成するようにし、RTA-OSEK を `active` アプリケーションモードで起動するようになっています。

その他のメインプログラムを使用するには、`project_settings.mk` ファイル内の Make 変数 `EXAMPLE_MODE` を `FALSE` にセットし、さらに以下のいずれかの設定を行います。

- ASCET-SEが自動的に`conf.c`内にメインプログラムを生成するようにする場合：
  `..¥target¥trg_<targetname>¥codegen_ecco.ini` ファイルの `Os-Config-C_gen_main` を `TRUE` にセット
- ASCET-SE がメインプログラムを生成しないようにする場合：
  `Os-Config-C_gen_main` を `FALSE` にセットし、実際に使用したいソースコードを変数 `P_C_SRC_FILES`（またさらに `P_ASM_SRC_FILES`）に設定

7.4 dT 変数

ASCET は各プロジェクトに dT（**d**elta **t**ime）という名前のモデル変数を生成します。dT は、各タスクおよび割り込みに対して、前回そのタスクまたは割り込みが起動された時から経過した時間をマイクロ秒単位で提供するものです。

dT の時間単位を変更するには、一般的な ASCET 変数と同様の変換式を定義してスケールを変更します。スケーリング処理は ASCET が自動的に実行します。

dT は一般的に「ダイナミック dT（動的 dT）」とも呼ばれ、タスクの連続する 2 回の起動間の実時間を表すダイナミックな値です。dT の値は、タスクや割り込みに対して、プリエンプションによる実行妨害や、リソースの保持や割り込みのディセーブル化によるブロッキングが行われた回数や時間によって変わります。

ASCET が dT の値をセットするには、フリーランニングタイマが必要で、さらにそのタイマのチックの間隔が分かっていなければなりません。この設定は 7.4.1 項に説明されています。

ユースケースによっては、dT にアラームタスク用に設定された周期をセットすることが適切である場合があります。ASCET ではこれを「スタティック dT（静的 dT）」と呼び、この設定は 7.4.2 項に説明されています。

ダイナミック dT とスタティック dT の違い、およびスケーリングされたダイナミック dT とスケーリングされないダイナミック dT の違いを下図に示します。

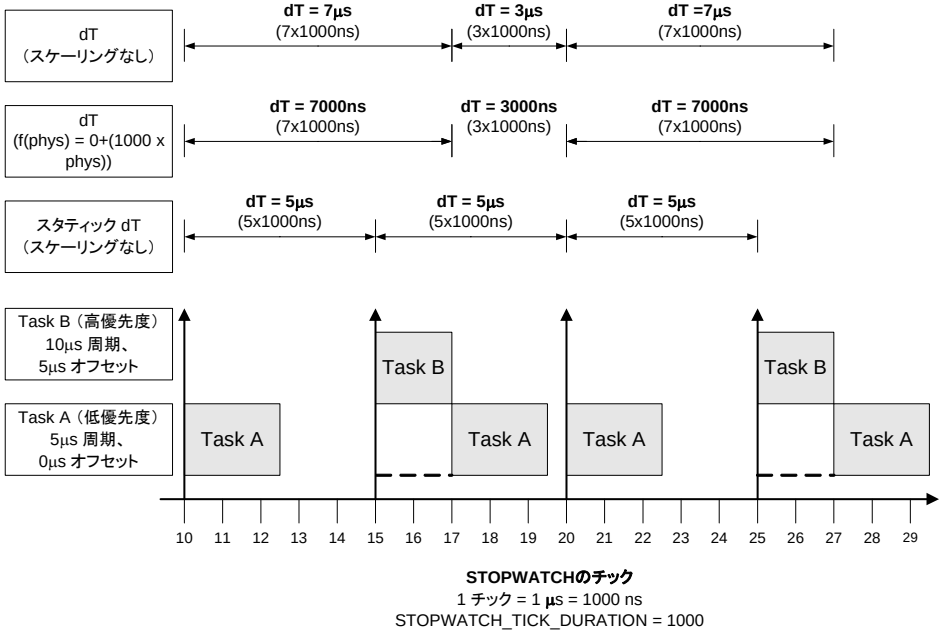


図 7-3 ダイナミック dT とスタティック dT

7.4.1 ダイナミック dT

ダイナミック dT を使用するには、プロジェクトプロパティの **Generate Access Methods for dT (Alternative: use OS dT directly)** というオプションをオンにします。ASCET-SE はランタイムに dT を計算するコードを生成しますが、そのためには ASCET-SE が 32 ビットのフリーランニングタイマソースにアクセスできることが必要です。

ASCET-SE は `setDeltaT()` という関数を生成します。この関数は、生成された各タスクボディ内で使用され、ASCET モデルエレメントである dT（コード内に `dT_PROJECT_IMPL` として生成されます）を更新します。モデルエレメント dT がスケーリングされる場合（つまり複数のインプリメンテーションを使用する場合）、ASCET-SE はそのスケーリングを自動的に正しく処理します。たとえばモデル変数 dT をミリ秒で実装した場合、以下のようなコードが生成されます。

```
void setDeltaT (void)
{
   TimeType dTMicroSeconds =
```

```
        (STOPWATCH_TICK_DURATION*dT)/(TickType)1000;
    (dT_PROJECT_IMPL = ((dTMicroSeconds/1000)));
}
```

ダイナミックdTの計算のための参照時間の提供

ASCETは`GetSystemTime()`というコールバック関数を使用して、SCETモデルが使用するdT値用の参照時間にアクセスします。このコールバック関数は、ターゲットマイクロコントローラのフリーランニングハードウェアタイマの現在値を返す必要があります。

ダイナミックdTが正しく処理されるためには、以下の設定を行ってください。

1. コード生成オプション **Generate Access Methods for dT** をオンにします。

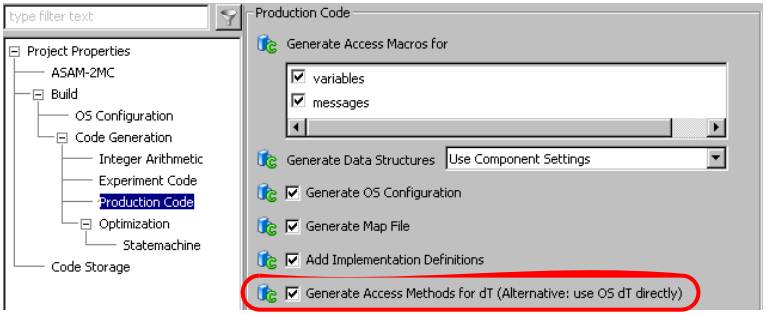


図 7-4 “Production Code" ノードのオプション

2. `codegen_ecco.ini`ファイル内の以下のオプションをオンにします。

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

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

3. `.¥target¥trg_<`*`targetname`*`>¥include¥os_inface.h`というファイル内で以下の行がコメントアウトされている場合は、コメントを解除してこの行を有効にしてください。

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

4. コールバック関数`GetSystemTime()`を作成（実装）し、32ビットのフリーランニングハードウェアタイマの現在の値を戻すようにしてください。

   ASCETをRTA-OSEKと統合する場合は、`GetSystemTime()`をRTA-OSEKの`GetStopwatch()`というコールバック関数に自動的にマッピングされるようにすることができます。そのためには、`project_settings.mk`ファイル内の`ASD_OS_INTEGRATION`を以下のように設定してください。

   ```
   ASD_OS_INTEGRATION = ASD_OS_INTEGRATION_RTA ↵
      MAP_TO_GETSTOPWATCH
   ```

RTA-OSEK のコールバック関数 `GetStopwatch()` は、OS のタイミングビルドと拡張ビルドの両方で必要です。この関数は OS に対して時間測定用 32 ビットフリーランニングハードウェアタイマへのアクセスを提供します（詳しくは RTA-OSEK のドキュメントを参照してください）。つまり RTA-OSEK の `GetStopwatch()` は、ASCET-SE の `GetSystemTime()` が必要とする機能を提供します。なお `GetStopwatch()` は外部 C コード内に実装する必要があります。実装例は各ターゲットのディレクトリ内の `.¥trg_<targetname>¥example¥target.c` というファイルに納められています。以下に `..¥example¥trg_tricore¥target.c` の例を示します。

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

5. `dT` のチックは、`project_settings.mk` ファイル内に定義された `STOPWATCH_TICK_DURATION` マクロによってナノ秒単位で ASCET に通知されます（5.4.1 項を参照してください）。

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

これらの設定により。ASCET がランタイムにおいて `dT` の値を計算し、ASCET モデルから生成されたコードがそれを使用することが可能になります。

7.4.2 スタティック dT

ASCET-SE では、アラームタスクに対してアライバルタイムが「スタティック `dT`」として供給されるようにすることができます。

注記

スタティック `dT` の値はアラームタスク用にのみ定義できます。他のタイプのタスクと割り込みに `dT` を使用するプロセスを含めることはできません。

スタティック `dT` は以下のように設定します。

1. プロジェクトプロパティのコード生成オプション **Generate Access Methods for dT *** をオフにします（図 7-3 参照）。
2. `codegen_ecco.ini` 内のスタティック `dT` オプションを有効にします。`Os-Config-C_gen_dt_static=1`
3. `project_settings.mk` 内の `USE_ASD_CALC_SCALED_DT` を有効にします。

上記の設定が行われると、ASCET は各タスクボディ内に _ASD_TICKS_PER_TASK_PERIOD というマクロを生成し、設定されたタスク周期をシステムカウンタのチック数で定義します。

以下に例を示します。

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

SYSTEM_COUNTER が 10 ms ごとにチックされる場合、上記の例のようにマクロは 10 チックにセットされます（10 チック X 10 ms = 100 ms).

ランタイムの計算時にチック数を時間に変換したり、実装変換式によってモデルの dT をスケーリングしたりするには、proj_def.h 内の ASD_CALC_SCALED_DT マクロを変更する必要があります。デフォルトではこのマクロは 1 対 1 のスケーリングになっていて、dT のチック数は 1 DT チック = 1 VAR です。

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

スタティック dT では、DT の 1 チックの長さは SYSTEM_COUNTER の 1 チック（単位はナノ秒）と同じです。つまり、ASCET の OS エディタ内の Tick Duration と同じ値になります。_ASD_TICKS_PER_TASK_PERIOD をマイクロ秒単位に変換するには、このマクロを変更する必要があります。DT に Tick Duration を掛けた値（DT*10000000）を 1000 で割り、ナノ秒をマイクロ秒に変換するようにします（DT*10000000/1000=DT*10000）。

以下に例を示します。

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

注記

リスケーリングを行うには、ユーザーの責任において、変換時の中間結果がオーバーフローまたはアンダーフローしないようにしてください。

7.4.3 ユーザー独自の dT ルーチンの実装

dT を特殊な用途に用いる必要がある場合、独自の dT を実装することができます。その際は、**Generate Access Methods for dT (Alternative: use OS dT directly)** オプションをオフにします（図 7-4）。

この場合、ASCET-SE は `setDeltaT()` を生成せず、dT 変数を定義しないため、これらのものを外部のユーザーコード内で定義する必要があります。ASCET は、以下の extern 定義に対応する関数と変数を必要とします。

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

`TickType` は `uint32` 以上の型で定義してください。`TickType` 変数の単位はフリーランニングハードウェアタイマの 1 チックで、`GetSysmtemTime()` で取得されます。

```
extern TickType GetSystemTime()
```

`setDeltaT()` は `void/void` 関数として実装し、ユーザーのモデル内で定義されたスケーリングを考慮してグローバル dT 変数を更新させます。

ASCET が生成するコードは C マクロを使用して dT 機能にアクセスします。デフォルトのマクロは `.¥trg_<targetname>¥ include¥os_inface.h` 内に定義されています。これを変更するには、`os_inface.h` 内の以下のマクロを変更してください。

- `DEF_GLB_DT_MEASURE`： このマクロは `conf.c` 内で使用されます。dT の計算に必要なグローバル変数または外部宣言を定義するものです。
- `DEF_TASK_DT_MEASURE`： このマクロは各タスクの先頭部分で使用されます。dT の計算に必要なタスクローカル変数を定義するものです。
- `PRE_TASK_DT_MEASURE`： このマクロも各タスクの先頭部分（`DEF_TASK_DT_MEASURE` の後）で使用されます。この部分に dT を算出するためのコードを挿入します。
- `POST_TASK_DT_MEASURE`： このマクロは各タスクの最終部分で使用されます。この部分に、他のタスク用にグローバル dT 変数をリストアするためのコードを挿入します。

7.5 テンプレートベース の OS コンフィギュレーション生成

ASCET は、テンプレートベースのメカニズムを使用して、OSEK OS の各コンフィギュレーションファイルを生成します。各テンプレート（`*.template` ファイル）はサポートされているすべてのオペレーティングシステム用に提供されていて、`<installation directory>¥target¥trg_<targetname>` ディレクトリに保存されています。

注記

テンプレートは OSEK ベースの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のみ使用されます。AUTOSAR RTE のコンフィギュレーションには使用されません。

“Project Properties”ダイアログボックスの“Build”ノードで OS を選択すると、ASCET-SE は自動的に、その OS 用のデフォルトテンプレートを選択します。使用されるテンプレートは“Project Properties”ダイアログボックスの“OS Configuration”ノードに表示されます。これ以外の設定は必要ありません。

図 7-5 にこれらのノードの設定内容を示します。

表 7-1 は、各 OS で使用されるテンプレートを示しています。表中の **%TARGET%** はテーゲットディレクトリのパスです。

選択された OS に別のテンプレートを使用するには、使用したいテンプレートファイルのフルパスを入力するか、または  ボタンをクリックしてテンプレートファイルを選択します。

“Project Properties”ダイアログボックスの“Build”ノードで OS コンフィギュレーションを変更すると、ASCET-SE は選択された OS 用に現在どのテンプレートが使用されているかを記憶します。

コード生成時、ASCET-SE は、プロジェクトエディタ内で設定された OS のコンフィギュレーション設定と共にテンプレートを使用して、選択されている OS 用のコンフィギュレーションファイルを生成します。コンフィギュレーションファイルの名前は常に `temp.oil` です。

テンプレートメカニズムは非常に柔軟性があり、デフォルトテンプレートまたはユーザーテンプレートを編集するだけで、OS コンフィギュレーションを簡単に変更できます。これは特に、サードパーティの OSEK OS コンフィギュレーションツールを使用するような場合に有効です。

オペレーティングシステム	デフォルトテンプレート
RTA-OSEK 5.0	`%TARGET%\OS_RTA-OSEK_V50.template`
GENERIC-OSEK	`%TARGET%\OS_Generic-OSEK.template`
RTE-AUTOSAR Vx.y	<empty>

表 7-1 サポートされている各オペレーティングシステム用のデフォルトテンプレート

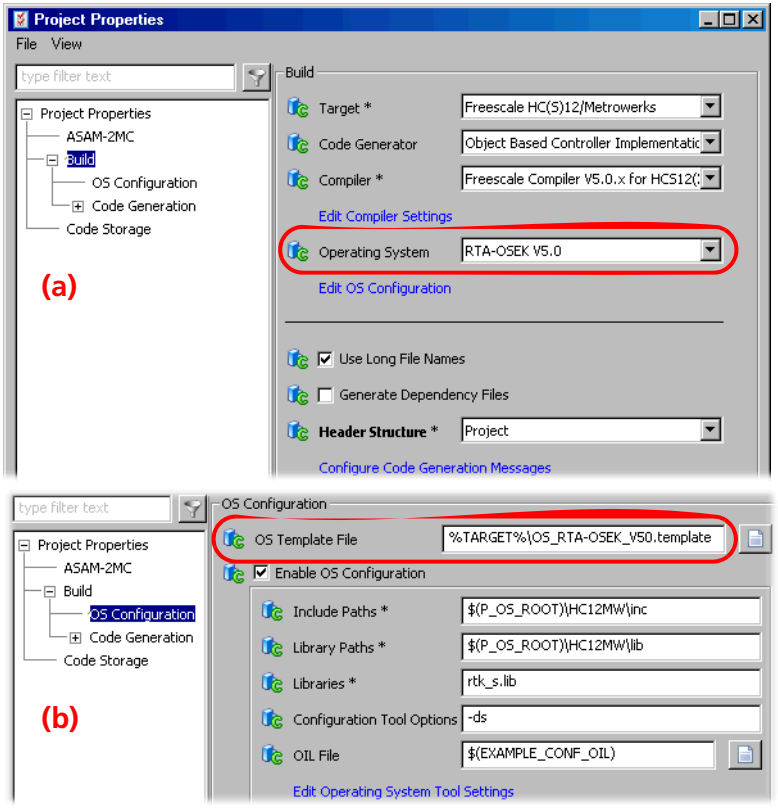


図 7-5 “Project Settings” ダイアログボックスで OS をテンプレートを選択（a: “Build” ノード、b: “OS Configuration” ノード）

注記

テンプレートメカニズムは OS コンフィギュレーションファイルの生成をカスタマイズする目的にのみ使用されるものです。生成される C コードのプロパティを変更するためのものではありません。

7.6 OSEK 未対応のオペレーティングシステムとのインターフェース

ASCET は OSEK に対応しないオペレーティングシステムとインターフェースすることもできます。これは特に ANSI-C ターゲットを使用する場合に有効です。生成されたコードは、ターゲットディレクトリ内の `os_unknown_inface.h` 内の定義を使用して OS インターフェースにアクセスします。

7.6.1 タスクのコンフィギュレーション

ASCET は以下の用にタスクボディを生成します。

- タスク定義は TASK キーワードとタスク名で開始します。

  例：

  ```
  TASK(t10ms){
  ```

- タスクに割り当てるプロセスのリストを関数呼び出しの形式で定義します。

  例：

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

- タスクをターミネートする関数呼び出しを定義します。

  例：

  ```
  TerminateTask();
  }
  ```

提供されている `os_unknown_inface.h` ファイルには TASK マクロと `TerminateTask()` が以下のように定義されています。

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

これを、使用する OS に合わせて変更してください。

デフォルト定義を使用した場合、C プリプロセッサにより以下のコードが得られます。

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

OSEK に対応しない OS を使用する場合、各 ASCET タスクのトリガモードは Software または Init に設定しておくことをお奨めします。トリガモード Interrupt または Alarm を使用するには特殊な OS 機能が必要なので、使用する OS がこの機能をサポートしていない場合は使用しないようにしてください。

7.6.2 OS API とのインターフェース

OS の呼び出しには OSEK OS のネーミング規則を使用しますが、この定義は実装されていません。すべてのオペレーティングシステムの呼び出しは `#define` 文を用いて空の文字列にマッピングされます。

例：

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

これにより、生成されたコード内の `GetResource` 呼び出しはプリコンパイラによって削除され、コンパイル時には無視されます。

注記

ANSI-C ターゲットを使用する場合、デフォルト状態において、OSEK OS の機能に依存する ASCET の機能（リソースなど）はサポートされません。これは C コード内で使用される OSEK 関数呼び出しについても適用されます。

上記の `#define` 文を変更することにより、各関数呼び出しを、OS が提供する関数にマッピングすることができます。

例：

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

7.7 テンプレート言語のリファレンス

本項では、テンプレートの記述方法、および ASCET-SE がアクセスを提供する OS オブジェクトについての情報をまとめます。

7.7.1 テンプレートについて

テンプレートは ASCII テキストファイルです。ASCET-SE V6.1 がテンプレートを処理する際、テンプレートタグ `[%` および `%]` で囲まれていない部分はすべて出力ファイル `temp.oil` に書き込まれます。

注記

テンプレートファイルには必ず拡張子 `.template` を付けてください。この拡張子により ASCET-SE V6.1 がそのファイルをテンプレートであると認識します。

テンプレートメカニズムはテンプレートエンジンとして“Template Toolkit”を使用し、このツールキットがサポートするすべての構造体をカスタムテンプレート内で使用することができます。本項では、ASCET-SE のテンプレート内で使用されるテンプレート言語の制限事項について概説します。テンプレートエンジンの機能についての詳しい情報は http://template-toolkit.org/ を参照してください。

以下のリスト内の 1. はタグを含まないテンプレートです。このテンプレートが ASCET-SE によって処理されると、その結果は、`temp.oil` ファイル内に以下の `2.` のように出力されます。

1. `MyFile.template` の内容

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

2. 生成された temp.oil ファイルの内容

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

命令文

テンプレートタグに囲まれたテキストは、テンプレートエンジンによって命令文として処理され、何らかのアクションが実行されます。各命令文は、テキスト行内の任意の位置に配置でき、複数行にまたがることも可能です。

式： 命令文の 1 つである「式」（expression）は、temp.oil ファイルに出力される評価結果に置き換わります。

一般的に、式は ASCET-SE が提供する OS オブジェクトの各プロパティ値を評価するためのものです。7.7.2 項にオブジェクトとプロパティの一覧が示されています。

以下の例は、OS オブジェクトの属性 numOfHardwareLevels および numOfSoftwareLevels を読み取って割り込みとタスクの優先度レベル数を表示するコメントをテンプレートに追加するものです。

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

条件文： テンプレート言語には条件文も含まれています。以下の例は、OSEK COM メッセージが定義されているかどうかに応じて異なるコメントを temp.oil に出力するものです。

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

反復文： 一般的な OS コンフィギュレーションの生成においては、ASCET-SE V6.1 のプロジェクトコンフィギュレーション内で宣言された各オブジェクトごとにコンフィギュレーションエレメントを追加することが必要となります。そのため ASCET-SE は、ほとんどのコンフィギュレーションオブジェクトへのアクセスをリストとして提供し、反復により各オブジェクト用に適切なコンフィギュレーションを書き出すことができるようになっています。

以下の例は、OSEK OS の 1 つのアプリケーションモードのための適切なコンフィギュレーションを書き出す方法を示したものです。

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

ここで、AppModes というリストに各アプリケーションモードに対応するアイテム（Normal、Diagnostic、LimpHome）が含まれているとすると、上記の例の反復文が処理されることによって以下のような OIL 言語になります。

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

サブルーチン： 共用される処理は、ブロック（“BLOCK”）と呼ばれるサブルーチン内に定義することができます。1 つのブロックには任意のテンプレートテキスト（他の命令文を含みます）を含めることができます。各ブロックには一意の名前を付ける必要があります。

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

各ブロックは、メインテンプレートから PROCESS コマンドを使用して呼び出されます。ブロック内で使用される変数は、引数として渡す必要があります。

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

ブロックは使用する前に宣言する必要はありませんが、呼び出し元と同じファイル内に含まれている必要があります。

他のファイルのインクルード： INCLUDE 文を用いて外部ファイルをインクルードすることができます。この文により、指定されたファイルの内容が出力に含められます。

注記

インクルードされるファイルの内容は、テンプレートエンジンによって処理されません。

パスは相対パスまたは絶対パスを使用できます。相対パスは、テンプレートコード生成パスへの相対パスです。

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

注記

パスはシングルクォーテーションで囲んで記述してください。

コメント： 命令文内のコメントは # シンボルでマークします。コメントは複数行にまたがることができます。以下の例は、一行、および複数行のコメントを示します。

例 1: 一行コメント

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

例 2: 複数行コメント

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

ホワイトスペースの省略： 命令文がその文の行に配置され、その評価結果がNULL の場合、テンプレートエンジンは空白行を出力します。これは、自分自身の行に配置されるあらゆるコントロールフロー文に適用されます。

これを避けるには、先頭の命令タグ [% に続けて等号マーク（=）を挿入します。以下に例を示します。

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

出力結果は以下のようになり、空白行は挿入されません。

```
AAAA
CCCC
```

7.7.2 オブジェクトのリファレンス

テンプレートは、定義済みオブジェクトを使用して OS コンフィギュレーションにアクセスします。一般的に、オブジェクトは OSEK OS 内のコンフィギュレーションアイテムに対応していますが、レガシーオペレーティングシステムのサポートのため、OS オブジェクトでないオブジェクトも含まれています。

アクセスできるオブジェクトは以下のとおりです。

オブジェクト	タイプ	説明
OS	構造体	一般的な OS プロパティ
AppModes	AppMode オブジェクトのリスト	プロジェクト内で定義されている全アプリケーションモード
Tasks	Task オブジェクトのリスト	プロジェクト内で定義されている全タスク（ソフトウェア／アラームタスク）
InitTasks	InitTask オブジェクトのリスト	プロジェクト内で定義されている全初期化タスク
ISRs	ISR オブジェクトのリスト	プロジェクト内で定義されている全割り込みサービスルーチン
Alarms	Alarm オブジェクトのリスト	タスク起動に使用されている全アラーム

オブジェクト	タイプ	説明
Resources	Resource オブジェクトのリスト	プロジェクト内で使用されている全リソース
Messages	Message オブジェクトのリスト	プロジェクト内で使用されている全メッセージ
UsedMessages	UsedMessage オブジェクトのリスト	タスクまたは ISR が使用している全メッセージ
Processes	Process オブジェクトのリスト	プロジェクト内で使用されている全プロセス
Functions	Function オブジェクトのリスト	プロジェクト内で使用されている全関数

各オブジェクトには一連のプロパティが含まれています。オブジェクトのプロパティへのアクセスは、ピリオドを用いた *<object_name>.<property_name>* という形式で記述します（例：`task.prio`）。

注記

オブジェクト名とプロパティ名は大文字と小文字が区別されます。

以下の例は、タスクオブジェクトのリストに反復してアクセスし、プロパティを抽出しています。

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

以降のセクションでは、各オブジェクトのプロパティを説明します。

OS

OS オブジェクトは OS の一般的なプロパティを定義するものです。OS オブジェクトは 1 つだけ定義されています。

プロパティ	タイプ	説明
numOfCoopLevels	整数	協調優先度レベルの数
numOfHardwareLevels	整数	ターゲットがサポートするハードウェア優先度レベルの数
tickDuration	整数	ASCET-SE システムカウンタの 1 チックの長さ（単位：ナノ秒）

プロパティ	タイプ	説明
numOfSoftwareLevels	整数	ターゲットがサポートするソフトウェア優先度レベルの数。組み込みターゲットの場合、ターゲットがサポートするタスクの数（`target.ini` に定義されています）と同じです。 実験ターゲットの場合は、最高のソフトウェアタスク優先度に協調レベルの数を加えた数です。
numOfPreemptLevels	整数	すべてのプリエンプティブレベルの数。以下のように定義されます。 `numOfHardwareLevels` + `numOfSoftwareLevels` - `numOfCoopLevels`
isEnabledOSEKCOM	論理型	ASCET メッセージの代わりに OSEK-COM メッセージをプロセス間通信に使用するかどうかを定義します。OSEK-COM メッセージを使用する場合は `true` で、それ以外は `false` です。この値が `true` の場合、生成された OIL ファイル内にメッセージ定義が含まれます。

AppMode

AppMode オブジェクトは OSEK に似たアプリケーションモードを定義するものです。

プロパティ	タイプ	説明
name	文字列	アプリケーションモードの名前
initTask	文字列	当該アプリケーションモードで OS が起動される際に起動される初期化タスクの名前
timeTable	文字列	当該アプリケーションモードで OS が起動される際に起動されるタイムテーブルの名前。ERCOS^{EK} 固有のものです。

Task

Task オブジェクトは、ASCET プロジェクト内で定義された OS タスクのプロパティを定義するものです。

プロパティ	タイプ	説明
name	文字列	タスクの名前
id	文字列	タスクの ASCET-SE 内部の識別子
prio	整数	当該タスクの優先度。値が大きくなると優先度も高くなります。
prioERCOSEK	整数	ERCOS^{EK} の優先度機構に基づく当該タスクの優先度
schedule	NON / FULL	タスクが他のタスクにプリエンプトできるかどうかを定義します。 OSEK の OIL プロパティ SCHEDULE と同じものです。
activation	整数	タスクに対してキューイングできる起動要求の最大数
autostart	TRUE / FALSE	タスクを自動起動するかどうかを定義します。
autostartAppModes	リスト	タスクを自動起動する際のアプリケーションモードの名前のリスト
usedResources	リスト	タスクが使用するリソースを表すリソース名のリスト
usedMessages	リスト	タスクが使用する OSEK COM メッセージの名前のリスト
usesFPU	TRUE / FALSE	タスクが浮動小数点レジスタ（OS コンテキスト切り替え時に保存／復帰が必要）を使用するかどうかを表すフラグ。浮動小数点コンテキストを保存／復帰意する場合、値は TRUE で、それ以外は FALSE です。
usedProcesses	リスト	タスクが呼び出す ASCET プロセスのリスト
hook	MONITORING / NONE	タスクが使用する OSEK 非対応のフック
deadlineMicroSeconds	整数	タスク起動から処理終了までの最大許容時間（単位：ミリ秒）
usesTerminateTask	TRUE / FALSE	タスクが OSEK の API `TeminateTask()` を使用するかどうかを示すフラグ

InitTask

プロパティ	タイプ	説明
name	文字列	初期化タスクの名前
id	文字列	初期化タスクの ASCET-SE 内部識別子
autostartAppModes	リスト	タスクが自動起動される際のアプリケーションモードの名前のリスト
usedProcesses	リスト	タスクが呼び出す ASCET-SE プロセスのリスト

ISR

プロパティ	タイプ	説明
name	文字列	ISR の名前
prio	整数	ISR の優先度。属性はターゲットに依存し、1 ～ `OS.numHWlevels` の範囲のいずれかの値です。優先度 1 は最低レベルを示します。
prioERCOSEK	整数	ERCOS^{EK} 優先度機構に基づく ISR の優先度
autostartAppModes	リスト	ISR が自動起動される際のアプリケーションモードの名前のリスト。OSEK では使用されません。
usedResources	リスト	ISR が使用するリソースの名前のリスト
usedMessages	リスト	ISR 内で使用される OSEK COM メッセージの名前のリスト
usesFPU	TRUE / FALSE	ISR が浮動小数点レジスタ（OS コンテキスト切り替え時に保存／復帰が必要）を使用するかどうかを表すフラグ。浮動小数点コンテキストを保存／復帰意する場合、値は **TRUE** で、それ以外は **FALSE** です。
usedProcesses	リスト	ISR が呼び出す ASCET プロセスのリスト
category	1 / 2	OSEK 割り込みカテゴリ。ASCET-SE V6.1 はカテゴリ 2 の ISR のみサポートしています。

プロパティ	タイプ	説明
source	文字列	ASCET-SE V6.1 の OS エディタで使用される ISR の表示名。RTA-OSEK と同様の規則が用いられます。
vectorAddress	文字列	割り込みベクタアドレス。アドレスはターゲット依存で、固定されたベクタテーブルの絶対アドレス、またはリロケート可能なベクタテーブル用ベクタロケーションです。RTA-OSEK と同様の規則が用いられます。
hook	MONITORING / NONE	ISR が使用する OSEK 非対応のフック
minPeriodMicroSeconds	整数	ISR の 2 つのインスタンスが連続して起動できる最短時間（単位：マイクロ秒） これは ERCOS^{EK} 固有のものです。

Alarm

プロパティ	タイプ	説明
name	文字列	アラームの名前
taskToActivate	文字列	アラームが満了した時に起動されるタスクの名前
autostart	TRUE / FALSE	アラームを自動起動するかどうかを表すフラグ
autostartAppModes	リスト	アラームを自動起動する際のアプリケーションモードの名前のリスト
delay	整数	アラームが最初に満了するまでのディレイ時間（単位：チック数）
period	整数	アラーム周期（単位：チック数）
delayMicroSeconds	整数	ディレイ時間（単位：マイクロ秒）
periodMicroSeconds	整数	アラーム周期（単位：マイクロ秒）

Resource

プロパティ	タイプ	説明
name	文字列	リソースの名前
property	STANDARD / LINKED / INTERNAL	リソースタイプ。ASCET-SE が生成できるのは STANDARD のみです。
ceilingPrio	TRUE / FALSE	リソースのシーリング優先度

Message

プロパティ	タイプ	説明
name	文字列	メッセージの名前
CDATAType	文字列	メッセージ定義に使用される C のデータ型

UsedMessage

プロパティ	タイプ	説明
name	文字列	メッセージの名前
sentAccessor	文字列	タスクがメッセージを送信する際に使用するアクセサの名前
recvAccessor	文字列	タスクがメッセージを受信する際に使用するアクセサの名前

Process

プロパティ	タイプ	説明
name	文字列	プロセスの名前
usedRessources	リスト	プロセスが使用するリソースの名前のリスト
usedFunctions	リスト	プロセスが呼び出す関数の名前のリスト
usedMessages	リスト	プロセスが使用する OSEK COM メッセージのリスト
usesFPU	TRUE / FALSE	プロセスが浮動小数点レジスタ（OS コンテキスト切り替え時に保存／復帰が必要）を使用するかどうかを表すフラグ。浮動小数点コンテキストを保存／復帰意する場合、値は TRUE で、それ以外は FALSE です。

Function

プロパティ	タイプ	説明
name	文字列	関数の名前
usedRessources	リスト	関数が使用するリソースの名前
usedFunctions	リスト	関数が呼び出す関数（当該関数内でネストされる関数）の名前のリスト
usesFPU	TRUE / FALSE	関数が浮動小数点レジスタ（OS コンテキスト切り替え時に保存／復帰が必要）を使用するかどうかを表すフラグ。浮動小数点コンテキストを保存／復帰する場合、値は TRUE で、それ以外は FALSE です。

8 ASAM-MCD-2MC を用いた測定と適合

ASCET では、測定と適合のための ASAM-MCD-2MC ディスクリプションファイル（A2L ファイル）を生成することができます。このファイルは、ASCET に含まれる静的に定義されたー連のコンフィギュレーションファイルを元に生成されます。本章では各静的ファイルの内容と ASAM-MCD-2MCD データについて説明します。

注記

ASAM-MCD-2MC 内のアライメント定義は、ASCET-SE によって自動的に決定されるようになったため、以前使用されていた `align.a2l` というファイルは必要なくなりました。

8.1 プロジェクト定義（`prj_def.a2l` ファイル）

`prj_def.a2l` というコンフィギュレーションファイルで、ASAM-MCD-2MC ファイル（詳しくは ASAM-MCD-2MC の仕様書を参照してください）の `MOD_PAR` セクションをユーザー定義することができます。このコンフィギュレーションファイルは、ASCET-SE のインストールディレクトリ（`.¥target¥trg_<targetname>`）に格納されていて、出荷時は以下のような内容になっています。

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

このファイルは必要に応じて編集してください。

8.2 メモリレイアウトの設定（`mem_lay.a2l`）

データファイル `mem_lay.a2l` は、コントローラのメモリレイアウトを、ASAM-MCD-2MC 規格で定義された `MEMORY_LAYOUT` として、ASAM-MCD-2MC 構文とセマンティックス（意味）に従って定義するものです。その内容は、生成される ASAM-MCD-2MC データファイルにそのまま挿入されます。このファイルは各ターゲットのディレクトリ（`.¥target¥trg_<targetname>`）

に格納されています。ファイルの内容は、ロケータ呼び出しファイルに定義されているコントローラハードウェアとメモリレイアウトに合わせて調整する必要があります。

注記

このファイルには設定例として提供されているものです。実際に使用するにはファイルをターゲットシステムに合わせて調整する必要あります。

8.3 ETK ドライバの設定（aml_template.a2l および if_data_template.a2l）

aml_template.a2l というファイルには、ETK のグローバル設定用 BLOB（IF_DATA、TP_BLOB など）の型定義が含まれています。

また if_data_template.a2l というファイルには、ETK 用のグローバル設定 BLOB（TP および QP BLOB）が ASAM-MCD-2MC フォーマットで記述されています。

これらのファイルは各ターゲットのディレクトリ（.¥target¥trg_<targetname>）に保存されています。構文は ASAM-MCD-2MC 規格に従い、実際の内容は各適合システムの機能に従うものとなります。

両ファイルの内容は、生成される ASAM-MCD-2MC ファイル内にコピーされます。この際に正しいデータがコピーされるようにするには、if_data_template.a2l 内の IF_DATA の構成が aml_template.a2l に定義された型と一致している必要があります。そのためには、ターゲットディレクトリ内のファイルの内容を更新するか、または内容を適切なファイルへの参照（完全なパスと名前）に置き換えてください。

注記

aml_template.a2l および if_data_template.a2l にはサンプルデータが格納されています。これらの内容は実際に使用するハードウェアに合わせて調整が必要です。

8.4 ディスクリプションファイルの生成

ASCET-SE には、INCA 等の適合ツールでの適合作業に使用できる ASAM-MCD-2MC ディスクリプションファイルを、プロジェクトに合わせて生成する機能があります。このために、必要に応じて、VAT （**V**irtual **A**ddress **T**able：仮想アドレステーブル）と呼ばれるテーブルが、ASCET-SE によってプロジェクト固有の C ファイルの一部として生成されます。

ASAM-MCD-2MC ファイルを生成するために必要な仮想アドレステーブルは、以下のようにして作成します。

仮想アドレステーブルを作成する：

- プロジェクトエディタで、**Project Properties** ボタンをクリックします。

  “Project Properties”ウィンドウが開きます。
- “Production Code”ノードで、**Generate Map File** オプションをオンにします。
- **OK** をクリックして“Project Properties”ウィンドウを閉じます。
- プロジェクトエディタで **Build → Build** または **Build → Rebuild All** を実行し、VAT を含んだコードを生成します。

注記

「ASCET」を「Additional Programmer」として使用する場合は、コード全体の整合性が確立され、VAT が含まれていないことを確認する必要があります。このためには、`codegen_*.ini` ファイル内の `addressTable` オプションを使用して **Generate Map File** オプションをオーバライドしてください。

VAT は複数の C 構造体で構成され、主に、ASAM-MCD-2MC ディスクリプションの一部である、生成されたコードのすべての変数およびパラメータの名前とそれを示すポインタが含まれます。

VAT を含むプロジェクトのコンパイルとリンクが終了すると、生成された HEX ファイル（`temp_vat.*`、拡張子はターゲットコントローラやコンパイラによって異なります ）やその他のファイルに、ASAM-MCD-2MC ファイル生成に必要なすべてのアドレス情報が書き込まれます。

特殊な HEX ファイルリーダによって、HEX ファイルからこのアドレス情報が読み取られ、また VAT からその他の情報（エレメントサイズ、アライメント、バイトオーダー等）が読み取られます。そして `etas.map` という中間ファイルが生成され、そこにすべてのエレメントの名前とメモリアドレスが ASCII テキストで出力されます。

VAT は ECU で実行されるプログラムの一部ではないので、VAT が含まれないもう 1 つの HEX ファイル（`temp.*`）とその他の結果ファイルがリンクされます。

ASAM-MCD-2MC ファイルを作成する：

- プロジェクトエディタで、**Tools → ASAM-2MC → Write** を実行します。

  “Write ASAM-2MC To：” ダイアログボックスが開きます。

- ダイアログボックスで、ファイル名を入力して保存先のディレクトリを選択します。

注記

ASAM-MCD-2MC ファイルは、.¥cgen¥ ディレクトリには保存しないようにしてください。ASCET オプションの設定（ASCET オンラインヘルプを参照してください）によっては、このディレクトリにあるファイルは、ASCET 終了時に削除されてしまいます。

- **Save** ボタンをクリックします。
  ASAM-MCD-2MC ファイルが、指定のディレクトリに指定の名前で保存されます。

以下の図は、コード生成において ASAM-MCD-2MC ファイルを生成する場合としない場合をそれぞれ表したものです。

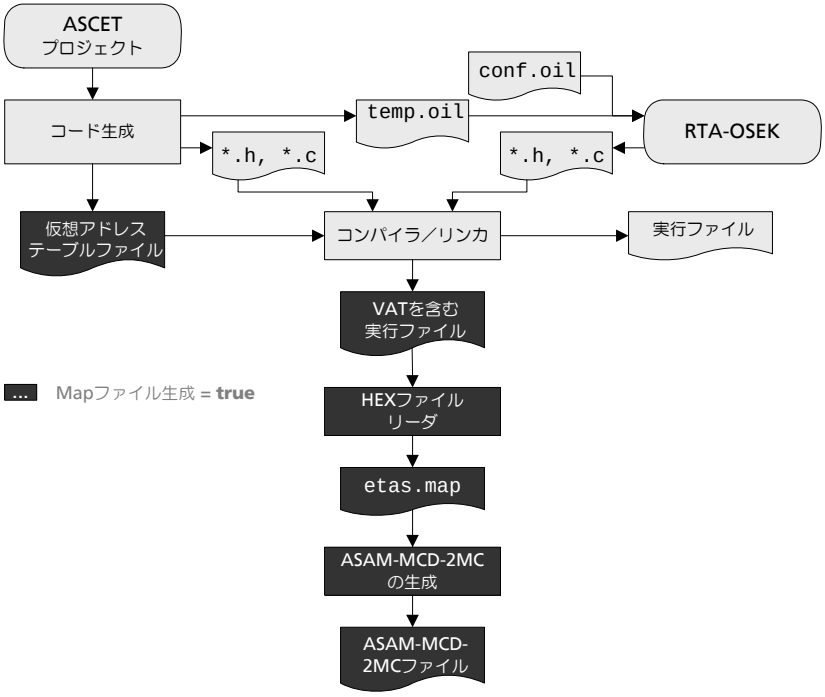


図 8-1 VAT と ASAM-MCD-2MC の生成を含むコード生成と含まないコード生成

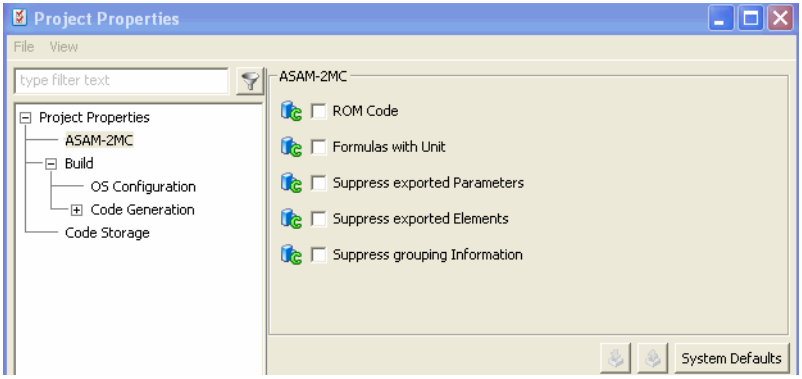
いずれの場合も、VAT は、ECU の物理メモリではないメモリセクションにマッピングされるようにしてください。詳しくは、3.3.5「メモリクラスに関する設定」を参照してください。VAT が通常の ECU メモリに割り当てられると、ASAM-MCD-2MC ファイルに不正なアドレスが出力され、メモリセクションを無駄にマッピングしてしまう可能性があります。

注記

ECU メモリを無駄なく使用するために、VAT は ECU の物理メモリの外に配置するようにしてください。

8.5 エクスポートされるエレメントとパラメータの除外

ASCET では、ASAM-MCD-2MC の生成時に、スコープが Exported のエレメントとパラメータを除外することができます。それにより、これらのエレメントを ASCET 外（サードパーティ製ツールなど）で定義することが可能になります。これはプロジェクトプロパティで設定します。



この機能は、以下の表に示されるように、対象の ASCET オブジェクトのタイプ（モジュール、クラス、プロトタイプクラス）に応じて異なります。プラス記号（+）はエレメントやパラメータが A2L ファイルに生成されることを表し、マイナス記号（-）は生成されないことを表します。

オプション Supress exported -		モジュール		クラス		プロトタイプクラス	
Parameters	Elements	エクスポートされるエレメント	エクスポートされるパラメータ	エクスポートされるエレメント	エクスポートされるパラメータ	エクスポートされるエレメント	エクスポートされるパラメータ
オフ	オフ	+	+	+	+	-	-
オフ	オン	+	+	-	+	-	-
オン	オフ	+	-	+	-	-	-
オン	オン	+	-	-	-	-	-

9 外部コードの統合

ASCET-SE には、ASCET によって生成されたコードと外部 C ソースコード（ハンドコーディングされたものやサードパーティ製ツールで生成されたもの）を統合するための強力な機能があります。これにより実現できる主なユースケースとしては、以下のような 2 通りがあります。

- ASCET を統合プラットフォームとし、モデルから実行ファイルや ASAM-MCD-2MC ディスクリプションファイルを作成するまでの完全な Make プロセスを実行します。
- ASCET によって生成されたコードを、ユーザー独自の Make ツールチェーンに組み込みます。

本章では、ASCET と ASCET-SE によって実現される上記の 2 通りのユースケースについて、特に以下の点に重点を置いて説明します。

- ユーザーが作成した C ファイルとヘッダファイルを、ASCET の Make ツールチェーンに容易に組み込めます。
- ASCET の外部でグローバルに宣言された関数や変数、およびパラメータに ASCET モデルから容易にアクセスできるように、「プロトタイプ」モデルエレメントが導入されました。これは C 関数のプロトタイプに相当するものです。
- メッセージとメソッドインターフェース（シグネチャ）の最適化に関するユーザー設定を行うことにより、外部コードとのインターフェースの信頼性を確保できます。
- ASCET と外部ソースファイルとのインターフェースとして使用される特別なヘッダファイルが、コード生成時に作成されます。

以下の項では、これらの機能についていくつかのケースをご紹介します。

9.1 ASCET モデルからの C 関数の呼び出し

この項では、ASCET モデルから外部関数を呼び出す 2 つの方法について説明します。

9.1.1 プロトタイプの使用

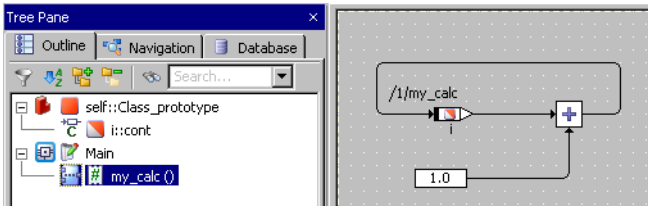
ASCET-SE には、ASCET 環境外で作成された C コードの関数やパラメータおよび変数（外部から提供されるソフトウェア）を使用するためのインターフェース機能があります。これは、ASCET のインプリメンテーションエディタで “Prototype” を生成するユーザーオプションをオンにすることによって実現されます。C 関数プロトタイプと同様に、ASCET のプロトタイプインプリメンテーションによって外部 C コードへのインターフェースディスクリプションが生成されます。このオプションは、サービスルーチンを使用する場合と同じく、クラスのインプリメンテーションエディタで設定します。この機能、および生成されるコードについての情報は、4.2.3「プロトタイプ実装」の項を参照してください。

プロトタイプとして実装されるクラスのコードには外部宣言のみが含まれ、変数やパラメータ、メソッドの定義については一切含まれません。ASCET でモデリングされたプロトタイプエレメントの環境は、プロトタイプのメソッド、グローバル変数／パラメータがどこで使用されるかにかかわらず、外部宣言と同じ意味を持ちます。そのため、ユーザーは外部 C コードにおいてグローバル変数およびパラメータを正しく定義して、ASCET モデルがそれらを使用できるようにする必要があります。

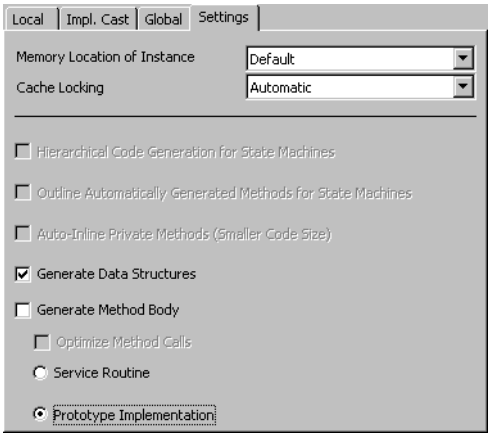
以下の例で、ASCET モデルからグローバル変数を使用して外部 C 関数を呼び出す方法について説明します。

```
#include ".¥INCLUDE¥a_std_type.h"

sint16 i;

void my_calc(void)
{
   i++;
}
```

ASCET から `my_calc` 関数を呼び出すために、ASCET モデル内に、グローバル変数 `i` を宣言するクラスと `my_calc` メソッドの宣言を作成します。



上記の BDE で記述された内容をコード生成時には使用されないようにするために、クラスのインプリメンテーションエディタでプロトタイプフラグをセットします。



すると、ASCET 内のそのクラスの環境用に生成されたコードには、クラスへのインターフェースのみが含まれます。

```
#define _Class
#define _i i

#ifndef NO_DECLARE_i
extern sint16 i;
#endif

extern void CLASS_IMPL_my_calc (void);
...
void MODULE_IMPL_process (void)
{
   CLASS_IMPL_my_calc ();
}
```

上の例に示されるように、「プロトタイプ」メソッドの名前はここでは ASCET の命名規則に従って、`<Class>_<Impl>_<Methodname>` の形式（184 ページの「複合（ユーザー定義）オブジェクトのデータ構造体と初期化」を参照してください）となります。この名前と外部コードの実際の名前を合わせるために、ASCET-SE のターゲットディレクトリに `proj_def.h` というヘッダファイルが作成されます。デフォルト状態において、このファイルは ASCET で生成される各ソースファイルにインクルードされるため、ユーザーはこのファイルを利用して、ASCET での名前と外部コードの名前をプリプロセッサ命令（`#define` 文）で対応付けることができます。この例では、`proj_def.h` を以下のように調整します。

```
#define CLASS_IMPL_my_calc() my_calc()
```

プロトタイプのコードでは、グローバル変数／パラメータの extern 宣言はプリプロセッサ命令の #ifdef 文で囲まれます（上のコード例を参照してください）。ここで、必要に応じてユーザー定義の extern 宣言（#define NO_DECLARE_<variablename>）を追加することができます。

たとえば、ASCET 変数 i を外部で宣言された i_user という変数にマッピングするには、以下のように宣言してください。

```
#define NO_DECLARE_i
#define i i_usr
extern uint16 i_usr;
```

このコードは proj_def.h 内に挿入します。

注記

警告： 上記のような変更により ASCET が生成するコードが変化します。それに伴いユーザーは、エレメントやメソッド用の適切なマクロを定義したり、エクスポートされるエレメントを正しく宣言する必要があります。ユーザーは、外部コード自体の挙動だけでなく、ASCET が生成するコードとの相互作用についてすべての責任を負う必要があります。また ASAM-MCD-2MC の生成における問題（以下の記述を参照してください）やその他の問題が発生する可能性もあります。さらに、ASCET で生成されたコードへのインターフェースは、今後リリースされるバージョンによって変更される場合もあります。

ASCET は、エクスポートされるパラメータや、プロトタイプクラスからエクスポートされるエレメントについて、A2L ファイルのエントリを生成しません。これらのエントリが必要な場合は、これらのものを外部に用意し、ASCET 開発プロセス外において、ASCET が生成した A2L ファイルにマージする必要があります。

9.1.2 ASCET 内で記述された C コードからの呼び出し

従来のバージョンと同様に、現行バージョンの ASCET においても C コードの記述は、内部エディタと外部エディタのどちらを使用しても行えます。ASCET 外部で作成された C 関数を、この内部 C コードから extern 宣言を使用して呼び出すことができます。

9.1.3 ASCET の Make 処理に C ソースファイルを含める

ASCET で制御する Make 処理に C ソースファイルを含めるには、ASCET-SE で project_settings.mk 内にファイル名のリストを定義します。さらに、定義されたファイルを ASCET-SE で検索するパスのリストも定義できます。

詳しくは 5.4.7 項 「ビルドプロセスのカスタマイズ」を参照してください。

9.2 外部 C コードから ASCET が生成した関数を呼び出す

ASCET が生成する function_declarations.h ファイルには、ASCET モデルのすべての関数の外部宣言が含まれるので、このファイルをユーザーソフトウェアにインクルードすることにより、外部コードから ASCET のメソッドやプロセスに容易にアクセスすることが可能となります。

プロトタイプ実装されたクラスについての外部宣言コードの生成は、プリプロセッサスイッチによって無効にすることができます。このスイッチは、以下の例に示すように、DECLARE_PROTOTYPE_METHODS と呼ばれます。以下は function_declarations.h の一部です。

```
#ifdef DECLARE_PROTOTYPE_METHODS
extern void CLASS_IMPL_my_calc (void);
#endif
```

9.3 外部のグローバル変数／パラメータを ASCET コードで使用する

9.1.1 項で説明されているように、外部 C コード内で定義されたグローバル変数／パラメータを、ASCET-SE で生成されたコードモデルのプロトタイプインプリメンテーションで使用することができます。これを行うには、各ターゲットがインストールされているディレクトリに格納されている proj_def.h ファイル内で、プリプロセッサ命令（#define 文）を使用して外部コードのネームを ASCET のシンボリックなネームにマッピングします。

また、ASCET によって variable_declarations.h というファイルも生成され、ここには ASCET モデルのすべてのグローバル変数の外部宣言が含まれます。このファイルをユーザーソフトウェアにインクルードすることにより、外部コードから ASCET モデルエレメントに容易にアクセスできます。

プロトタイプとして実装されたクラスでは、特殊なプリプロセッサ命令を使用して、外部宣言のコンフィギュレーションを以下のように設定することができます。

```
#ifdef DECLARE_PROTOTYPE_ELEMENTS
#ifndef NO_DECLARE_i
extern sint16 i;
/* min=-32768.0, max=32767.0, ident, limit=yes */
#endif
#endif
```

上記の例で使用されている DECLARE_PROTOTYPE_ELEMENTS スイッチは、variable_declarations.h でプロトタイプエレメントの外部宣言を一切行わないようにするためのものです。プロトタイプからエクスポートされる個々の変数やパラメータについてのスイッチについては、9.1.1「プロトタイプの使用」の項を参照してください。

9.4 外部データ構造体を使用するコードの生成

デフォルト状態において ASCET-SE は、必要なすべてのデータ構造体を生成するので、プロジェクトは内部で解決されます。しかし、論理モデルは同じでデータの値のみが異なる複数のプロジェクトを扱うような場合、ASCET 内でコード生成を行い、データソースは外部（サードパーティツールなど）から供給する、という方法が効率的です。

このようなワークフローにはさまざまなメリットがあります。たとえばコードの検証が一度終了した後は、軽微なデータ変更を行うたびにコードに「タッチする」危険を避けることができます。

ASCET-SE では、ASCET のデータ構造体の生成を無効にすることにより、このようなワークフローを実現できます。

注記

外部で生成されたデータ構造体を扱うには、ユーザーのシステムを ASCET の外部でビルドする必要があります。つまりこの場合、ASCET を統合プラットフォームとしては使用しません。

データ構造体の生成については、“Project Properties”ダイアログボックスの“Production Code”ノードで設定します。以下の 3 つのモードのいずれかを選択できます。

1. すべてのコンポーネントについて生成する
2. すべてのコンポーネントについて生成しない
3. コンポーネントの設定を使用 ― デフォルト状態において各コンポーネントは、データ構造体が生成されるように設定されています。上記の 2 つのモードは、コンポーネントの設定より優先されます。ASCET で一部のデータ構造体を生成し、その他の構造体を外部コードで生成する場合は、このモードを選択してください。

図 9-1 では、すべてのコンポーネントについてデータ構造体が生成されないように設定されています。

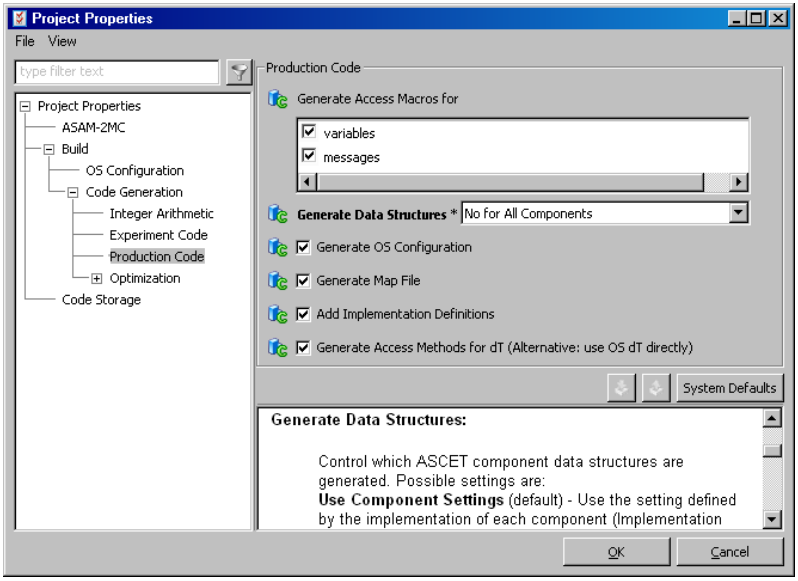


図 9-1　全コンポーネントについてデータ構造体の生成を無効化する

`Use Compoent Settings`（コンポーネントの設定を使用する）モードの場合、図 9-2 のように、各コンポーネントのインプリメンテーションで、データ構造体を生成するかどうかを指定できます。

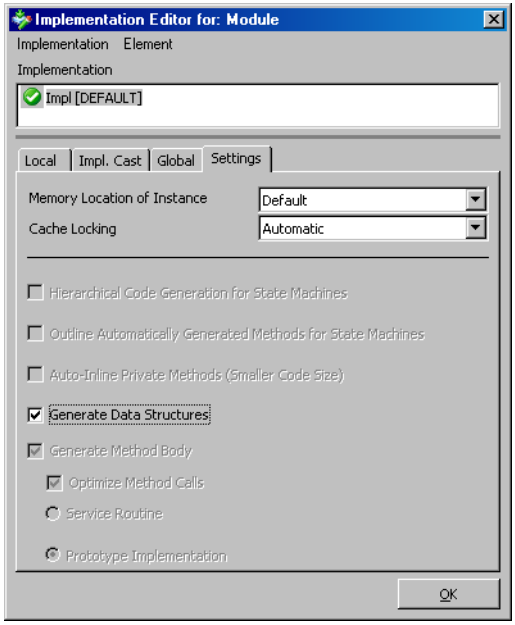


図 9-2　コンポーネントごとにデータ構造体を生成するかどうかを指定する

9.5　ASCET の最適化機能の設定

ASCET で外部コードを使用する場合は、インターフェースの一貫性が保たれることが重要です。しかし ASCET に組み込まれたデフォルトの最適化機能は常に最小かつ最速のコードを生成するため、モデルが変更されると外部インターフェースも変更される可能性がある、という「副作用」を伴います。

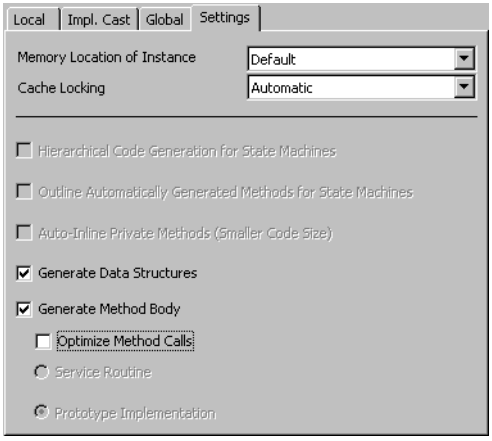
インターフェースの安定性を保つため、このデフォルトの最適化機能を無効にすることもできます。

9.5.1　メソッド呼び出しの最適化に関する設定

複数のインスタンスを持つことができるクラスのメソッドに対しては、ASCET はメソッド引数リストの 1 番目のエレメントとして、ローカルデータ構造体へのポインタを渡します。ASCET においてこのポインタは self ポインタと呼ばれ（184 ページの 13.3.3 項を参照してください）、これは C++ の self ポインタと同じものです。

しかし実際にインスタンスを 1 つしか持たないクラスのメソッドの場合、1 つのデータインスタンスが明確に直接アクセスされるため、このポインタは必要ありません。このような場合は、最適化によってこのような self ポインタを省略することにより、ASCET が生成するコードのランタイムのパフォーマンスが向上し、スタック容量も減少します。この最適化は、コード生成時にデフォルトで行われます。

その一方で、ASCET が生成したコードと外部 C コードを統合するには、ASCET モデルが変わってもインターフェースは常に一定である必要があります。そこで、上記のようなシングルインスタンスのクラスの最適化を無効にすることにより、モデルの違いによってメソッドの呼び出し方法が変わることを防ぐことができます。このシングルメソッドの最適化は、クラスのインプリメンテーションエディタの“Setting”タブで無効にすることができます。



上の図のように設定すると、クラスが複数のインスタンスを持つかどうかにかかわらず、常に self ポインタが生成されます。

注記

外部 C コードから ASCET が生成したメソッドを呼び出したり ASCET が生成した変数やパラメータにアクセスする際は、ASCET が生成しデータ型の定義を必ず確認し、ASCET が生成した型と異なるタイプの型を使用することは避けてください。これは特に self ポインタの場合に重要です。
ASCET が生成したコード内の関数インターフェースは、ASCET のバージョンアップによって変わる場合があります。

クラスのインスタンスがモデル内に 1 つしかないことが明確である場合に限り、**Optimize method calls** オプションをオンにして self ポインタを含まないメソッドインターフェースの生成を有効にすることができます。

9.5.2 メッセージコピーの最適化に関するコンフィギュレーション

設定された OS のタスクタイプと優先度に基づき、ASCET はデータの一貫性を保つために必要な箇所にのみメッセージコピーを生成します（13.4.3 項 「メッセージ」項を参照してください）が、このような最適化を行うには、ASCET がコード生成時においてすべてのデータアクセスを把握している必要があります。

ASCET モデルの外部で定義されたスケジューリングに関するデータアクセスについては、ASCET はまったく認識できません。外部の OS コンフィギュレーションや外部 C コードを用いる際に発生するデータ不整合に関する問題を避けるため、ASCET-SE では、生成して使用するメッセージコピーを定義することができます。詳しくは 13.4.3 項 「メッセージ」を参照してください。

9.6 バリアントパラメータの使用

ASCET でパラメータのコンフィギュレーション設定を行う際、各パラメータのプロパティエディタにおいて **Variants** 属性を設定し、パラメータの複数のバリアントへのアクセスを可能にするかどうかを指定することができます。

ASCET はこのオプションがオンになっているすべてのパラメータを 1 つのメモリセクション内のグループにまとめます。このパラメータセットは 1 つの「バリアント」を定義します。さらに、ASCET は複数のパラメータセット（1 つのセットが 1 つのバリアントに相当します）が存在すると仮定し、外部定義されたオフセットを用いて間接的にパラメータにアクセスするコードを生成します。

この機能は ASCET では実験的に用いられるものです。使用方法についての詳しい情報は、ETAS のサポート窓口までお問い合わせください。

10 モデリングのヒント

本章では、効率的でかつ数値の精度が高い実装コードを生成することに重点を置き、モデルの構築およびインプリメンテーションの設定のための一般的なガイドラインを示します。

ただし一般に、モデルの最適化に求められる各条件は、互いに相反する効果を生むということが前提となります。たとえば、メモリ容量を最小化するためには実行速度と数値精度を犠牲にしなければならず、また反面、実行速度を上げようとすると、メモリ使用量が増加し、コードが読みにくくなる可能性があります。また、精度を高くすると、メモリ使用量は増加します。

10.1 コード実装

コード実装時において、各エレメントのインプリメンテーションは、以下のような要件に基づいて決定されます。

- 物理的な値の範囲
- 求められる精度
- ハードウェアやセンサの特性

10.1.1 変換式の定義

オフセット：変換式のオフセットはゼロにしてください。ゼロ以外のオフセットを指定してもメリットはほとんどなく、数学演算のために余分なコードが必要になります。ただし、以下の例のような場合は例外と考えられます。

- 異なるセンサによる値の差異など、システム仕様の一部としてオフセットが存在しているような場合
- 配列、マトリックス、ディストリビューション、または特性カーブ／マップにおいて、オフセットを使用することによりデータをよりコンパクトに記述して（つまり短いワード長を使用して）メモリスペースを節約できる場合

例：あるセンサ温度の値の範囲が -50 ～ +150°C で、分解度が 1°C であると仮定します。この場合、オフセットを使用しないと 16 ビットのワード長が必要ですが、オフセットを使用すれば 8 ビットで実装することができ、これによって 1 つの量（つまり 1 つの配列要素など）あたり 1 バイトを節約できます。ただしここでは、メモリ使用量と実行速度のどちらを優先させるかを考慮する必要があります。

また、スカラ値の場合、オフセットを使用してもメモリ使用量の最適化のための効果はさほど望めません。

スケール値：スケール値のおよその範囲は、システム全体の物理的特性により決まります。このような数値的要件は論理的に、または実験に基づいて決める必要がありますが、所定の範囲内で実際のスケール値を選択する場合、さまざまな可能性が考えられます。

- スケール値は単純な有理数でなければなりません。たとえば、分数の係数は小さい数で、2 または 10 の累乗であり、大きい素数ではない単純なものでなければなりません（例、8/3、256/100、50）。一般に、スケール値を指定する際には、小数（例、0.1875）よりも分数（例、3/16）の方が好ましく、以下の規則が適用されます。
- 結果が符号なしの値となる式の場合は、$2^K/n$ という形式のスケール値が最適で、符号付きとなるには $2^{K-1}/n$ が最適です。K は対応するワード長（1 ワードのビット数）で、n は実装可能な最大値よりも少し大きい、適切な数です。これにより、値の範囲のほぼ全体を確実に使用できるようになります。
- 使用可能な値の範囲全体を使用するよりも、単純な係数を優先して使用ください。

例：ある値の実際の物理範囲が `[0,9.1]` であり、それの値を 8 ビットで実装する場合は、通常、$2^8/10=25.6$ という単純なスケール値を使用します。この結果、実装されたこの値の量子化単位とインターバルは、`0.039` および `[0,9.96]` となります。

この例で、8 ビットでの最高の精度を得ることを目指す場合には、スケール値に `255/9.1=28.02=2550/91` を用いることもできます。しかしこの場合でも量子化単位は `0.036` で、精度はわずかしか上がらず、制御アルゴリズムにおいて目に見えるほどの数値的向上はありません。一方、生成されたコードにおいて、この複雑なスケール値と他のスケール値との間での変換が必要となる場合、実行時間がかなり長くなったり、精度が損なわれることも考えられます。たとえば、この複雑なスケール値を上述の単純なスケール値に変換しようとすると、`(256/10)/(5824/6375)=2550/91` という、適切とはいえない有理数のリスケーリング係数が発生し、数値精度はかえって悪くなり、また 32 ビットの中間結果も必要となってしまいます。

変換式のを表示する：

- プロジェクトエディタの“Formulas”タブをクリックして変換式を表示します。
- 式のエントリをダブルクリックすると、その式が表示されます。

2 の累乗のスケール値を使用することのメリットについては、いくつかの例ですでに説明しました。リスケーリング処理が単純なビットシフトですむので、スケール値にはできる限り 2 の累乗を使用してください。

10.1.2 インターバルの定義

インターバル、つまり値の範囲を指定するときには、コードジェネレータがそれらの値を数式の変換時に使用する、ということを考慮する必要があります。したがって、インターバルを定義する際は、以下の 2 つの点が重要な目標となります。

- オーバーフロー対策のための処理（右シフトなど）は数値の精度を不必要に損ねるため、避けるようにする。
- 余分なコードを生み出すクリッピングを避ける。

このような条件に基づき、インプリメンテーションには、物理的に妥当な値の範囲を設定するようにしてください。

例：

```
{ A ∈ [0.. 40] } + {B ∈ [ 0,10 ]} = C
```

A と B の値の範囲を同じにして、Aphys[0,40]、B_{phys}[0,40] のようにし、すべての量についてスケールを S = 0.25 にすると、以下のようなインプリメンテーションになります。

A_{uint8}[0, 160], B_{uint8}[0, 160], C_{uint16} [0, 320]

加算の結果には、各加数の 2 倍のビット数が使用されます。

しかしここで、B_{phys}[0,10] という範囲を使用すれば、3 つの量すべてについて同じビット長を使用することができます。

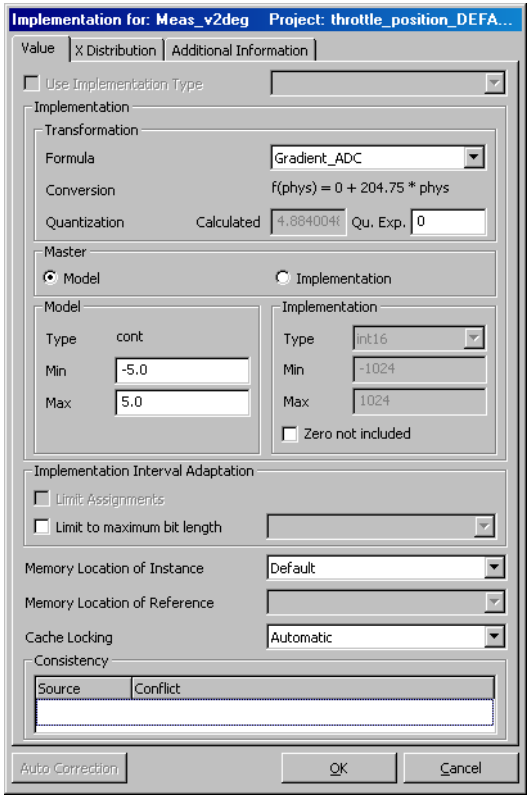
A_{uint8}[0, 160], B_{uint8}[0, 40], C_{uint8} [0, 200]

したがって、実装されるデータ型の値の範囲（例、int8 の場合は [-128, 127]）をそのまま使用するという方法は適切な方法とは言えず、特に、実際に取り得る範囲よりもデータ型の範囲の方が 2 倍以上大きい場合にはお薦めできません。

- 下の例について見てみましょう。

  スロットル位置測定値は特性カーブを使用して電圧から角度に変換されます。

  `Meas_v2deg` のインプリメンテーションエディタを開いて“X Distribution”タブを選択すると、X 軸ポイントのディストリビューションの情報が以下のように表示されます。

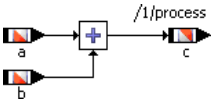


ここで、スロットル位置の測定値は、0 ～ 5 ボルトの 2 つの信号の値の差です。各信号は 10 ビットの A/D コンバータで変換されるため、これらの信号の最高の精度は $5V/2^{10}$ ビットで、スケール値は `1024/5` になります。インターバル `[-5,5]` は、2 つの信号の差から求められます。

10.1.3 互いに関連する複数の変数のインプリメンテーションの定義

互いに代入されるか数学的に接続される変数（またはメソッド引数）の変換式および実装データ型は、可能な限り、互いに一致するように選択してください。この考え方の例を以下に示します。

- 可能であれば、オフセット 0 を選択します。
- **加算と減算**の場合、変数には、同じか少なくとも類似するスケール値を割り当ててください。

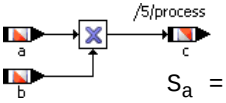


$S_a = S_b = S_c$

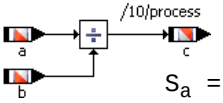
（S：スケール係数）

2 つのスケール値の比率が 2 の累乗または小さな整数、または単純分数である場合、それらのスケール値は「類似している」と言えます。効率化のためには 2 の累乗が最も適していて、単純分数はなるべく避けることをお勧めします。

- **乗算と除算**については、それぞれ両オペランドの積や商をスケールにするのが理想的です。必要に応じて、結果の型を拡張する必要があります。

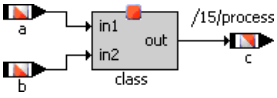


$S_a = S_b * S_c$



$S_a = S_b / S_c$

- **さらに複雑なクラス**の場合、以下のようなスケールをお勧めします。



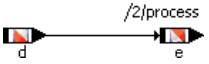
$S_a = S_{in1};\ S_b = S_{in2};\ S_c = S_{out}$

$S_{out} = f(S_{in1}, S_{in2}, S_{p\_int})$

（p_int：内部量）

入力引数とそれに代入される量のスケールや、戻り値とそれに代入される値のスケールは同じです。戻り値のスケールは、引数のスケールおよびクラスの内部エレメントにより決まります。

- **代入**：



— リスケーリングを行うと乗算や除算が追加されるので、モデル内ではリスケーリングを行うのは避けてください。実行時間やメモリが余計に必要となります。

上の例で生成されるコードの場合、たとえば、スケールが変わると、以下のような違いがでます。

$S_d = S_e \quad \Rightarrow$ `(e = d);`

S_d=1/5, S_e=1/3 $\Rightarrow$ `(e = ((d*3)/5);`

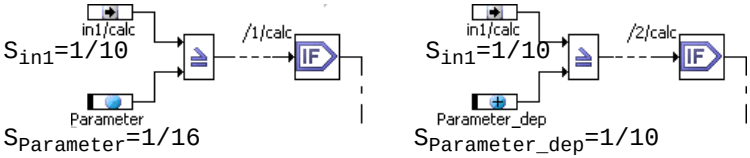
- 固定の底（指標）を用いて量子化を行えば、1 つの乗算または除算だけでリスケーリングすることができます。

  $S_d=10^{-1}$, $S_e=10^{-2}$ ⇒ `(e = (d*10));`

- 底が 2 の量子化を使えば、シフト演算でリスケーリングすることができます。

  $S_d=2^{-2}$, $S_e=2^{-1}$ ⇒ `(e = (d>>1));`

- **依存パラメータ**を使用すると、以下のようなメリットがあります。
  - コンパレータやパラメータを伴う連鎖計算においては、リスケーリングを避けることができます。



  - さまざまな単位を変換する場合、“odd scale factors”（奇数のスケール係数）を取り消すことができます。
  - 実行速度とコードサイズを最適化することができます。*仮想* パラメータを使用することにより、メモリを節約できます。

  ただし、パラメータの数が増える、というデメリットもあります。

- 結果が累積される内部中間メモリ（積分器、フィルタ、ローパスなど）は、少なくとも累積される結果の 2 倍のワードサイズで実装することによって、精度を保証する必要があります。

10.1.4 結果が大きくなる乗算

大きなインターバルを持つ 2 つの量を掛け合わせると、数値の精度が失われる可能性がありますが、これは、コードジェネレータがオーバーフローを回避するために右シフトを使用した場合に発生するものです。

例 1：`X*Y` を計算します。ここで、X と Y はどちらも実装データ型が `uint32` で、32 ビットの範囲をフルに使用します。オーバーフローを避けるために、以下のコードが生成されます。

```
(X>>16)*(Y>>16)
```

これは、数値的には不正確です。たとえば、X または Y が 65536 未満の場合、結果は 0 になってしまいます。

この問題は、複数の乗算演算が続けて行われる場合に特に重大です。

例 2：`X*K*DT` を計算する積分器について考えてみましょう。ここで、X（入力）、K（積分定数）および DT（時間差）は `uint16` 型で、16 ビットの範囲をフルに使用します。中間結果が 32 ビットメモリに格納されるとすると、全部で 16 ビット分の右シフトが必要です。これにより、たとえば、以下のような式になります。

```
((X>>5)*(K>>5))*(DT>>6)
```

しかし、これら 3 つの変数のどれについても小さい値はゼロになり、積分器が常にゼロになってしまいます。これは全面的に、自動的に行われるオーバーフロー対策の結果です。

注記

もちろん、これは特にコードジェネレータが原因で発生する問題ではなく、限られたワードサイズで量子化された演算に伴って発生する一般的な問題です。マニュアル操作によるコーディングの場合にも、同じ問題が発生します。

このような問題を防ぐために、モデリング段階では以下の規則に従う必要があります。

- 乗算のオペランドは、必要以上に精度を高くしないでください。つまり、できるだけ小さいワードサイズで実装してください。
- オペランドの値のインターバルを、物理的に妥当な範囲まで狭めてください。たとえば、積分器の時間差 DT は、10µs の量子化によって 16 ビットで実装することができ、これにより、結果の値のインターバルは、典型的な自動車アプリケーションに十分な 655ms になります。
- 複数の乗算演算を続けて実行する必要がある場合には、量子化およびインターバルの選択は、上記の基準に従って注意深く行い、この部分については充分なテストを行ってください。浮動小数点演算が可能なターゲットの場合は、浮動小数点演算も考慮に入れてください。
- 積分器、ローパス、および類似するフィルタの場合、以下のような式が生成されます。

  ```
  in * k * dT
  ```

  この計算が静的タイムフレーム内で実行されると、変数 dT は、（変換式の助けを借りて）定数 k に設定されている固定値に置き換わります。つまり、以下のようになります。

  $$in * (k * dT_{fix}) = in * k_{fix}$$

  このようにすることで、乗算シーケンスおよびそのシーケンスに起因する不正確さを回避することができます。

PID 導関数項の計算における dT の効果を理解する：

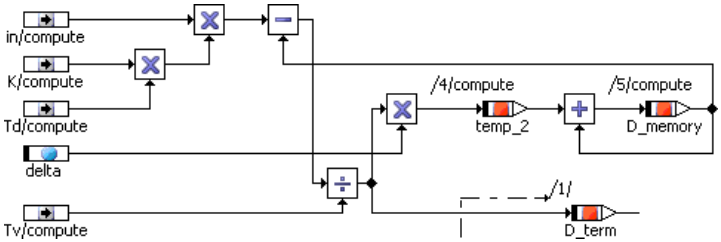
- PIDT1 コントローラ（160 ページの 12.3.9 項を参照してください）の導関数項計算を見てみましょう。

  dT の効果を検討するために、Temp2 の計算に注目します。

$Temp_2$ の計算は dT*t3 で構成されます。ここで t3 は、160 ページの 12.3.9 項ですでに学習した D_term に代入される式です。インプリメンテーションは $dT=2^{14}*dt \in [0,0.1]$ で、中間結果 t3 のスケールは 2^{13} でインターバルは [-42000,42000] です（160 ページの例を参照してください）。

- 乗算 dT*t3 を行うと、9 ビットのオーバーフローになります（つまり、t3 に 7 ビット、dT に 2 ビットの右シフト）。
- この計算は静的なタイムフレームで発生するので、dT をリテラルまたはパラメータで表すことができます。パラメータを使用する場合、オーバーフローを少なくするために、はるかに小さいインターバルを指定することができます。

  - 下図のように、dT の代わりにパラメータ delta を使用します。delta に値 0.001、スケール値 2^{14}、およびインターバル [0,0.001] を割り当てます。



  - この例のために新しいコードを生成し、変化を調べてください。

値のインターバルが小さくなったので、時間ステップを前の例と同じ精度で実装しても、dT*t3 で発生するオーバーフローは 2 ビットだけになりました。この場合、リテラルを使用しても、dT を使用するよりもよい結果を得ることができますが、パラメータを使用する場合ほどの好結果にはなりません。この理由は、リテラルの精度に関する規定にあり（159 ページの 12.3.7 項を参照してください）、ここでは、リテラルは相対誤差が 0.1% 未満になるように実装されます。0.001 の場合、2^{17} というスケール値が必要になるので、5 ビットのオーバーフローが発生します。

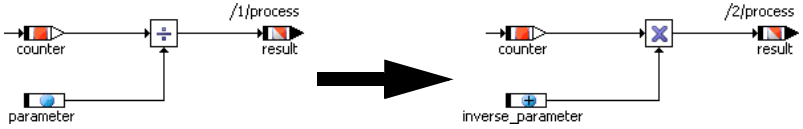
10.2 モデルの構造

本項では、効率的なコードを生成するという観点から、ASCET モデルの最適化について考察します。

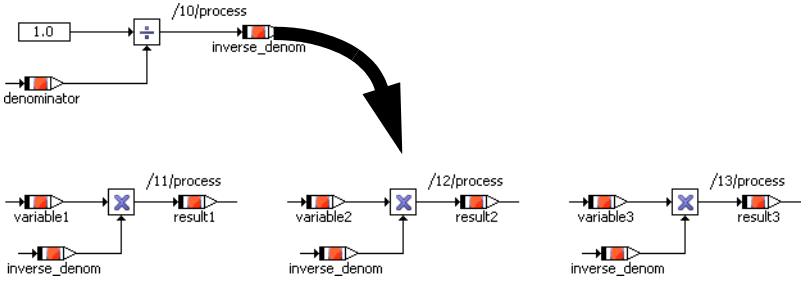
10.2.1 除算

他の部分ですでに説明したように、除算は数値上の問題を数多く引き起こすため、できるだけ避けてください。除算の回避は、たとえば以下のようにして行えます。

- 逆数の値を持つ依存パラメータを使用する。



- 除算の結果を一時的に格納し、それを再利用する。



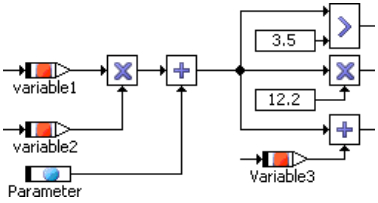
除算については、以下の原則に従ってください。

- 数式内では、除算はできるだけ後で行うようにします。
- 整数に実装する場合は、常に分子を分母よりも非常に大きく（できれば 2 倍のワードサイズに）します。
- 分母には最大有効ワードサイズを使用しないようにします。たとえば有効ワード長が 32 ビットの場合、分母は 16 ビットを超えないようにします。
- 分母のインターバルには、0 が絶対に含まれないようにします。

10.2.2 多重計算、連鎖計算、論理演算子

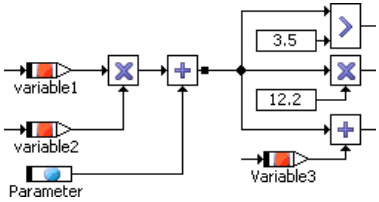
多重計算

たとえば下図のような多重計算は、できるだけ避けてください。このような計算をタイマや積分器内で使用すると、実行時間が長くなる上、間違った結果が算出される可能性があります。



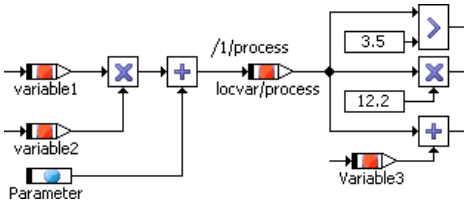
多重計算を避ける方法はいくつかあります。

1. テンポラリ変数を挿入する方法



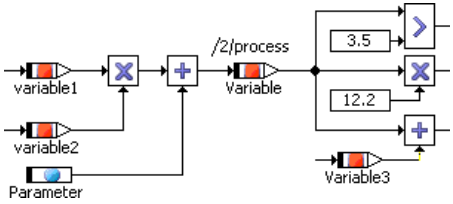
このようにすると、余計なメモリを使わないで中間結果に即座にアクセスできます。しかし、適合システムではテンポラリ変数を実装したり測定したりすることができません。他の個所において利用することもできませんし、シーケンシングを操作することもできません。また、スタック処理の負荷が大きくなります。

2. プロセス／メソッドローカル変数を挿入する方法



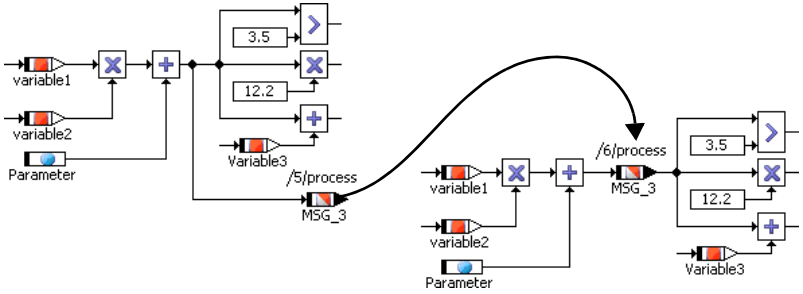
この方法でも、中間結果に即座にアクセスできます。メソッド／プロセスローカル変数はさまざまなコンテクストに実装して何回も使用でき、シーケンシングを指定することもできます。しかしテンポラリ変数と同様に、測定したり、メモリクラスを割り当てたりすることはできません。スタック処理に余分な負荷がかかることも避けられません。

3. 変数を挿入する方法



変数は実装することも測定することもできます。ECU 内に独自のメモリロケーションを持つので、メモリクラスを割り当てることができます。何回も使用でき、様々なメソッドやプロセスで同時に使用することもできます。シーケンシング情報も明示的に指定することができます。しかし一方、変数を使用すると、固定的に確保される RAM スペースが増えます。

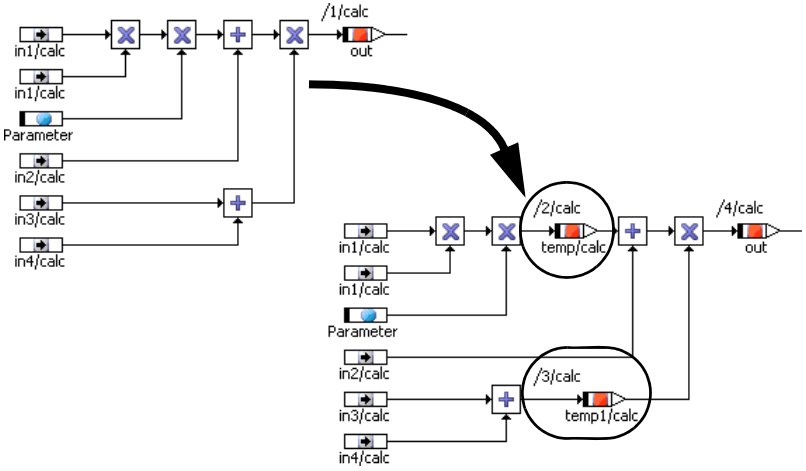
4. 送信メッセージを中間結果として使用すると、それを送受信メッセージに変更することができます。



この方法では、RAM の使用量が増えることはありません。RAM スペースは、既存のメッセージのためだけに必要です。このエレメントは実装や測定が可能で、ECU 内に独自のアドレスを持ち、メモリクラスを割り当てることができます。複数のプロセスで同時に使用することができます。しかし、この方法は、モデル全体のシーケンシングを念頭に置いておく必要があるので、限られた場合にしか使用できません。

連鎖計算

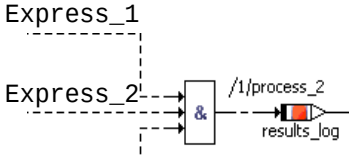
長い連鎖計算には、中間変数（メソッド／プロセスローカル変数）を挿入してください。これを行わないと、コード生成時に生成される一時的な中間結果についてのオーバーフロー対策（右シフト）により、計算結果の精度が失われてしまう可能性があります。



このような場合、中間変数を使用することによって、部分的な結果について希望する精度を指定することができます。

論理演算子

コードジェネレータは論理演算子の入力を、降順に、左から右への連鎖にマッピングします。




```
result_log = ((Express_1)&&(Express_2)&&(Express_3))
```

実行時には、コードはやはり左から右へと処理されます。計算が完了する前に結果を判断できる場合（例、`Express_1 = false`）、その評価処理は中断されます。そこで、論理演算子の入力を計算時間と確率の順序に従って上から下へと並べることをお勧めします。演算子の種類により、特に以下のような式を上側の入力に割り当てることをお勧めします。

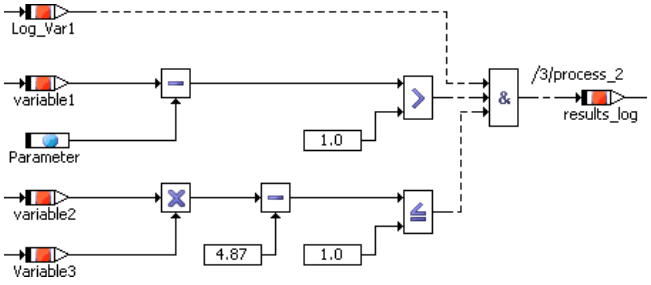
AND 演算子の場合

- 計算時間の短い式
- false になる可能性が高い式

OR 演算子の場合

- 計算時間の短い式
- true になる可能性が高い式

例：



10.2.3 クラスとモジュール

クラスを使用する場合は、以下の点に注意してください。

- デッドビート応答（z^{-1}）は、1 つの変数に置き換えることができます（シーケンシングに注意してください）。

- クラスを不必要にネストさせると、関数呼び出しがネストされ、スタックの使用量が増えて実行時間が長くなります。不必要なネストは避けてください。
- 1 つのクラスについて複数のインスタンスを使用すると、すべてのインスタンスが同じプログラムコードを使用しますが、各インスタンスごとに自分のデータセットを持っています。これによりコード用メモリ（ROM）は節約できますが、各データエレメントへの間接アクセスが必要になります。
- それぞれのクラスは切り離してください。つまり、独立したリターンメソッドまたは直接アクセスを使用し、戻り値を演算処理から独立させてください。一般には、直接アクセスメソッドの使用をお勧めします。

  ターゲットとなるコントローラによっては、**Optimize Direct Access Methods** というコード生成オプションをオンにすることにより（ASCET オンラインヘルプ参照）、リターンのための特別な関数呼び出しが必要なくなります。

  この方法では、戻り値が何度か使用される場合でもクラスでの計算は一度しか行われないので、実行時間が短縮されます。内部アルゴリズムと戻り値の計算は、同じ速度で実行する必要はありません。また古い戻り値と新しい戻り値の両方にアクセスすることもできます。この方法の欠点は、計算結果を格納するための中間メモリとして、追加の変数を使用しなければならなくなることです。
- メソッドのインライン化を行う場合、コンパイラはメソッドのプログラムコードをモジュールのプログラムコード内に直接組み込むので、関数呼び出しは必要なくなります。それにより実行速度が最適化されますが、このメソッドが 2 回以上使用される場合にはメモリがよけいに必要になります。
- ASCET はクラスの各インプリメンテーションごとに別のプログラムコードを作成するため、1 つのインプリメンテーションを繰り返し使用すると、メモリ所要量は少なくなります。しかし、この方法の有用性は限られています。

モジュール内でメソッドを使用する際は、以下の点に注意してください。

- モジュール内のメソッドに含まれるメッセージやリソースへのアクセスは可能ですが、モジュール内のメッセージについての最適化は、`_OPT_COPY` と `_NO_COPY` しかサポートされていません。他のバリアント（`_NON_OPT_COPY`、`_OSEK_COM`、`_OSEK_COM_STACK_BUFFER`）が選択されていると、コード生成時にエラーが発生します。
- モジュール内のメソッドがメッセージを使用している場合、このメソッドは 1 つのタスクからしか呼び出せません。これは、メッセージがアクセスされる部分のコードについて、タスク優先度が静的に割り当てられる必要があるためです。

10.2.4 ステートマシン

ステートマシンの最適化は、要件に応じて以下の 3 つの観点から行うことができます。

- 応答時間
- 実行速度
- コードサイズ

最適化用の各種オプションについては、ASCET オンラインヘルプに詳しく記述されています。

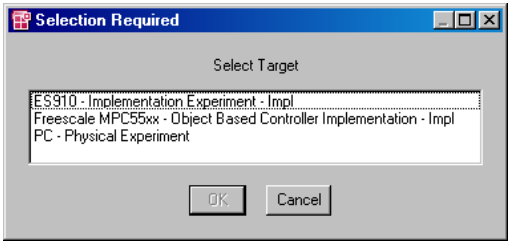
11 プロジェクトのターゲットを変更する

ASCET-SE では、あるプロジェクトを異なるターゲット用のものに変換することができます。その際は、既存のターゲットや実験タイプ、インプリメンテーションから新しいターゲット用に C コードと OS 設定をコピーします。

プロジェクト全体の C コードをコピーする：

- プロジェクトエディタにおいて、新しく使用するマイクロコントローラに合わせてターゲットとコード生成オプションを選択します。
- **Extras → Copy C-Code From**... を選択します。

  “Selection Required”ダイアログボックスが開きます。



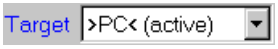
- コピー元のターゲットを選択して、**OK** をクリックします。

  ターゲットコードが、使用するマイクロコントローラ用にコピーされます。

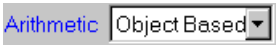
クラス／モジュール単位で C コードをコピーする：

あるクラスやモジュールの C コードを他のターゲットや実験タイプ、インプリメンテーションにコピーするには、以下のいずれかの方法で行います。

1. **Tools → Code Variants → Copy To** メニューを使用



- C コードエディタで、コピーするモジュールまたはクラスを開きます。
- “Target”コンボボックスで、C コードが書かれた時のターゲットを選択します。

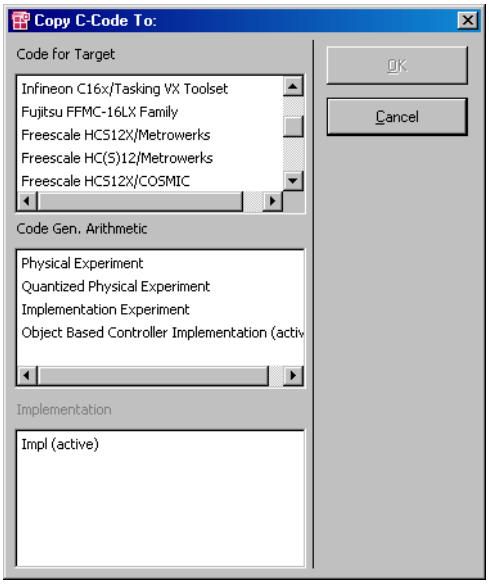


- “Arithmetic”コンボボックスで、C コードが書かれた時の実験タイプを選択します。

- “Implementation”コンボボックスで、C コードが書かれた時のインプリメンテーションを選択します。

- **Tools → Code Variants → Copy To** を選択します。

  “Copy C-Code To:”ダイアログボックスが開きます。



- “Code for Target”フィールドで、使用するターゲットを選択します。
- “Code Gen. Arithmetic”フィールドで、実験タイプを選択します。
- “Implementation”フィールドで、インプリメンテーションを選択します。

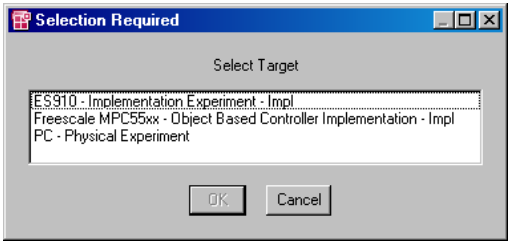
  これらすべての選択を行うと、**OK** ボタンがアクティブになります。
- **OK** をクリックしてダイアログボックスを閉じます。

  C コードが、選択されたコントローラ用にコピーされます。

2. **Tools → Code Variants → Copy From** メニューを使用
   - C コードエディタにおいて、“Target”、“Arithmetic”、“Implementation”の各コンボボックスで、使用したいターゲットと実験タイプ、インプリメンテーションを選択します。

- **Tools → Code Variants → Copy From** を選択します。

  “Selection Required”ダイアログボックスが開きます。



- コピー元とするターゲットのバリエーションを選択して **OK** をクリックします。

  C コードが、現在のターゲット用にコピーされます。

オペレーティングシステムの設定をコピーする：

- プロジェクトエディタで、“OS”タブを選択します。
- **Operating System → Copy From Target** を選択します。
- “Selection Required”ダイアログボックスで、OS 設定のコピー元とするターゲットを選択します。
- **OK** をクリックしてダイアログボックスを閉じます。

  オペレーティングシステムの設定が、現在のターゲット用にコピーされます。

ターゲットに合わせてコード生成条件を設定する方法については、第 5 章「コード生成に関する設定」を参照してください。

12 量子化演算について

本章では、ASCET で記述されたアルゴリズムをもとにコードジェネレータがコード生成を行う方法について詳しく説明します。変換の規則については、後の項でさらに詳しく説明します。まず基本演算を変換してから、インプリメンテーション情報を使用して数式を最適化する様子を、例を用いて説明します。また整数演算の数値誤差についても概説します。

実装コード生成における最も基本的な処理は、物理モデルの算術演算をターゲット実装用の量子化された算術演算に自動変換することです。必要な変換処理および補正係数が生成され、オーバーフローが自動的に回避または補正されます。従来のマニュアル操作によるコーディングのプロセスでは、このステップでの信頼性は低くなってしまうことは明らかです。そのため、信頼性のある自動コード生成を利用して、ソフトウェアの品質を向上させることが重要です。

生成された整数算術演算機構は、さらに最適化することが可能です。

論理（ブール）演算、制御用構造体、およびメソッド呼び出しは、実装コード生成と物理コード生成の両方において同じように変換されます。両者の主な相違点は、実装コード生成では整数の算術演算が生成されるのに対して、物理コード生成では生成されないことです。

実装コード生成の主目的は、ユーザーにより指定されたインプリメンテーションを考慮しながら物理記述を意味的に正しく変換することです。量子化や整数の除算による数値の誤差は付き物ですが、その誤差を最小限に抑えることはできます。それによって生成されるコードは信頼性の高いものとなり、たとえば、実行時のオーバーフローも回避することが可能です。

12.1 自由度と最適化の度合い

ユーザーが定義する変数／パラメータのインプリメンテーションは、コードジェネレータにとって必須項目ですが、これらの「固定」インプリメンテーションを含む数式にさえ、ある程度の自由度があるのが普通です。自由度とは、つまり中間結果用のインプリメンテーションの選択肢のことです。自由度はコードジェネレータにより定義されていますが、ターゲットに関する制約、特に整数量を表すのに使用できる最大ビット長については考慮が必要です。

自由度は、コードジェネレータが以下の基準に基づいて最適化を行うためのものです。

- 数値誤差を最小限にする
- オーバーフローを回避、または補正する
- 実行時間とメモリ必要量（コードサイズと、RAM およびスタックスペース）を最小化する

これらの最適化の目標は、互いに矛盾する部分があります。許容範囲内のオーバーヘッド内で完全な最適化プログラムを作成することはできません。そこで、コードジェネレータでは以下の 2 つの基本要素からなる発見的手法が用いられます。

- 個々の基本演算を適切に変換するためのローカル規則

- 一層最適な数式が生成されるようにローカル変換を調整するための全体的な制御方針

この手順では、場合によっては期待どおりの結果が得られない可能性があります。そのような場合、ユーザーはマニュアル操作で介入し、ジェネレータに許されている自由度を減少させる必要があります。これは、数式内のあるポイントに、定義済みのインプリメンテーションを使用してテンポラリ変数を導入することにより行います。

また固定小数点コードについての特別なオプションを選択して、さらに進んだ最適化を行うことも可能です（ASCET オンラインヘルプの“Interger Arithmetic”ノードについての説明を参照してください）。

12.2 整数算術演算機構の数値的誤差について

物理算術演算を整数算術演算に変換すると、数値誤差が発生します。この誤差には、量子化と整数の除算という 2 つの原因があります。

12.2.1 量子化による誤差

実際の量が量子化されるときには、量子化単位の 2 分の 1 以内の誤差が発生します。

この誤差を避けることはできません。理論上は、量子化の精度を上げることにより、この誤差を小さくすることができます。しかし、量子化を小さくすればするほど、結果となる整数は大きくなります。もちろん、実際には量およびそれらに対して行われる計算（つまり中間結果）の両方の量子化表現には、制限範囲内（つまり、32 ビットで表せる範囲）の値しか使用できません。

このようなわけで、実現できる精度は、量子化表現の選択（つまり値の範囲と量子化単位）により決まります。量子化を選択するときには、数値の精度とメモリ所要量との妥協点を見つける必要があり、さらにターゲットで使用できるワードサイズについて考慮する必要もあります。

12.2.2 整数の除算による誤差

原則として、整数の算術演算のうち加算、減算、乗算は、オーバーフローさえ発生しなければ正確に計算されます。しかし整数の除算では商の小数部分が切り捨てられてしまうため、誤差が発生します。たとえば、2/3 は 0 になり、9/5 は 1 になります。原則として、結果を切り上げることにより、誤差を最大で半分まで小さくすることができますが、除算は、特に誤差の伝播という点で好ましくない動作の原因になります。

数値の精度をできる限り損なわないようにするために、整数の除算を使用するときには以下の規則に従ってください。

- 可能であれば、除算をまったく使用しないようにします。
- たとえば 32 ビット /16 ビットのように、分子が分母よりも顕著に大きくなるようにしてください。一般に、分子のワードサイズを分母のワードサイズの 2 倍にし、この追加されたビットをフルに使用するようにします。

- 一連の数学演算が続いている場合は、除算をできるだけ後回しにします。たとえば、x*y がオーバーフローしないという条件下では、一般的に (x*y)/z の方が (x/z)*y よりも高精度です。

12.2.3 誤差の伝播

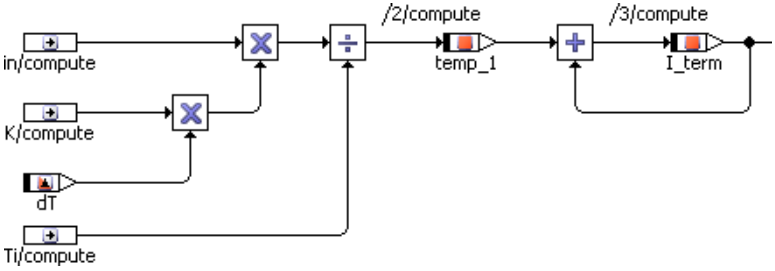
量子化や除算による誤差は、数々の数学演算を通じて伝播され、すぐに大きくなってしまう可能性があります。通常、入力量に誤差が含まれていなければ正しく行われる加算などの演算でも、誤差は伝播されます。

組み込み制御ソフトウェアを実現していく過程で、量子化を選択してから、結果の数値精度が満足できるものであるかどうかを調べてください。満足できない場合には、以下の方法を試してみてください。

- 使用できるワードサイズとの関連で、可能な限りより精密な量子化を選択する
- 「戦略的な」量子化を選択し、式のリスケーリング処理で除算が自動生成されないようにする
- 数式の変換／簡素化／近似により、除算または乗算による誤差の伝播を減らす
- アルゴリズム全体を修正して、数値的に一層安定性のあるものにする

誤差の伝播を減らす例：

- 下図に示す I_term の計算について考えてみましょう。



PID コントローラの積分項を表す式は、一般に以下のように記述されます。

$$I\_term = f_{integral}(in*(K/Ti)*dT)$$

関数 In および係数 K/Ti は、積分を求める前に計算されます。このような処理における数値誤差は、先に除算が行われている（しかもその後乗算で拡大されている）ことによってだけではなく、オーバーフロー対策（つまり、除算の前に行われる K の左シフト - これについては後述します）によっても生じます。そのため、アルゴリズムをブロックダイアグラムで示す方が、通常の数学的な表現よりも精度が高まります。

さらによい方法は、PIDT1_MOD モジュール内の特性マップに Ti の逆数を設定することにより、除算を完全になくすことです。しかし、こうすると、通常のパラメータとの直接的関係が失われてしまいます。

12.3 整数コード生成の規則

本項では、モデル内で物理的に定義された基本演算が、コードジェネレータによってターゲット用の量子化された整数算術演算機構にマッピングされる際に適用される「*ローカル規則*」について説明します。また、複合数式の最適化についても説明します。

各基本演算の変換規則は、以下の原則に基づいて決定されます。

- **数値精度を維持すること**

  数値の精度を犠牲にするのは、オーバーフローを避けるためにやむを得ない場合だけです。

- **中間結果のオーバーフローが発生しないこと**

  コードジェネレータが優先しなければならないのは、中間結果でのオーバーフローを防ぐことです。必要に応じて数値の精度を犠牲にしてでも粗めの量子化を自動選択します。

- **追加する演算の数を最小限にすること**

  中間結果の量子化をカスタマイズする際、追加する処理の数を最小限にする必要があります。

- **指定された値の範囲を守ること**

  コードジェネレータはユーザーにより指定された値の範囲を守ることを保証します。必要に応じて、明示的なリミッタが生成されます。

12.3.1 代入

物理量の代入、たとえば y := x は、どのようにして適切な量子化が行われた C コードに変換されるのでしょうか。これを説明するために、量子化されたソース値 X を、異なる量子化が行われた可能性のあるターゲット値 Y に代入する場合を考えてみましょう。

```
assignment (phys.): y := x
source:             X  = ax+b
target:             Y  = cy+d
```

ソースとターゲットの変換式が同じであれば、実装値を直接代入することができます。

```
Y := X
```

そうでない場合には、代入の前に、ソースをターゲットの変換式に変換する必要があります。

```
Y := fx,y(X)
```

コードジェネレータの優れている点の 1 つは、この変換式を自動生成することです。まず、ソースに適切な変換係数、つまり、ターゲットのスケールとソースのスケールの比率をかけることにより、ターゲットに適合するようにソースをリスケールします。

```
X1 := X*(c/a)
```

それから、ターゲットにあわせてオフセットを適用します。

```
Y = X2 := X1 + d - b*(c/a)
```

リスケーリング、つまり、有理数であっても大抵は整数でない変換係数をかける処理には問題が多く、基本的に、この乗算は様々な方法で整数演算に変換される可能性があります。以下の変換方法では、係数 `c/a` は単純分数であると想定しています。

- 先に乗算を行う：`(X*c)/a`

  これが最も適切な方法です。中間結果にオーバーフローが発生しない場合は、必ずこれを選択してください。

- 先に除算を行う：`(X/a)*c`

  この方法では除算による誤差が次の乗算でふくらみ、非常に大きな数値誤差を生じてしまいます。

- 近似式：`(X*c')/a'`

  ここで、`c'/a'` は `c/a` の「単純な」、つまり小さい係数を用いる有理近似式でなければなりません。一般に、アルゴリズムを用いてこのような近似式を設定するのは非常に困難です。コード生成で使用されるこの試みは、いわゆる連続的分数拡張と呼ばれるものです。

このアプローチを、以下のような例で分かりやすく説明します。

`X*(20/13)` を計算すると想定します。`X` の範囲は `[0,80]` で、計算には 8 ビットの数（`0` ～ `255`）だけを使用でき、`x` の現在値は `73` であるとします。

浮動小数点演算での計算結果は、`112.31` です。

また整数演算では、以下の問題が浮上します。

- 乗算を先に行って `(73*20)/13=112` という答えを得ようとすると、中間結果が `73*20=1460` となり、大きくなりすぎてしまうので、現実的ではありません。
- 除算を先に行うと、以下のように結果が非常に不正確になってしまいます。
  `(73/13)*20=5*20=100`

一方、必要な除算 `20/13=1.538` についてユーザーが近似式 `3/2=1.5` を選択すると、この結果は `(73*3)/2=19/2=109` になります。この結果はほぼ正確な上、中間結果もオーバーフローしません。

コードジェネレータは、使用可能なワードサイズで数値精度をできるだけ高くしようとします。そこで、リスケーリングには以下のアルゴリズムを使用します。

1. 一般に、個々の量のスケールを単純な比率で近似します。これを行うことで、`c/a` というリスケーリング係数には大きい係数がなくなります。

2. 使用可能なワードサイズ内で中間結果を実装できる場合には、乗算を先に行います。

```
(X*c)/a
```

3. そうでない場合には、中間結果のオーバーフローの量（ビット数）について調べます。それから、状況に応じて、以下の 2 通りの方法のうち数値的に正しい方のアプローチが選択されます。
   - 先に除算を行い、次に乗算を行う

     ```
     (X/a)*c
     ```

   - s ビット分右シフトしてから、上のステップ 2 のように進める

     ```
     (((X>>s)*c)/a)<<s
     ```

     スケールを 2 の累乗として指定できる場合には、主としてこの方法が使用されます。その場合、最後のシフト演算は削除されます。

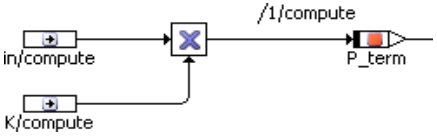
プロセス全体をまとめると、代入は以下のステップを経てコード化されます。

1. ソースをターゲットのスケールにリスケーリングする
2. オフセットを調整する
3. 必要に応じて、結果の値のインターバルを制限する
4. 変換後の実装値を変数に代入する

メソッド呼び出しの実引数と仮引数の間の代入も、同じように行われます。

代入の例：

- 下図の `P_term` の計算について考えてみましょう。



ここでは、中間結果 `In*K` が `P_term` に代入されます。このインプリメンテーションは以下のようになります。

```
In    = 2048*in ∈[-2,2],
K     = 64*k ∈[0,50],
P_term= 256*pterm ∈[-100,100]
```

したがって、中間結果のスケールは `2048*64` で、範囲は `[-100,100]` です。これを `P_term` に代入するには、以下のリスケーリングが必要です。

$$1/512 = 256/2048/64 \quad (\text{つまり } 2^{-9} = 2^{8-11-6})$$

すべてのスケール値が 2 の累乗なので、これは右シフトだけで簡単に行うことができます。限界値は必要ないので、以下のようなコードが生成されます。

```
P_term = ((In * K) >> 9);
```

12.3.2 加算と減算

加算と減算は同じように扱われるので、ここでは加算だけについて説明します。

2 つの量を加算する場合にはまず、両者の量子化を同じスケール値にする必要があります。その後でオフセットが加算されます。たとえば、メートル単位の長さとキロメートル単位の長さは、どちらかを先にリスケーリングしてからでないと足し合わせることはできません。

加算のためのコード生成は、以下のステップで実行されます。

リスケーリング： 両方のオペランドを同じスケールにします。精度の損失の程度を少なくするために、細かく量子化された方のスケールが選択されます。しかし、使用できるビット長において、粗めに量子化された値を細かく量子化された値に合わせて変換することができないような場合、粗い方の量子化表現が使用されます。

加算： リスケーリングされたオペランドを、オフセットが指定されている場合にはそれも含めて加算します。

オーバーフローの扱い： 指定された値の範囲によりオーバーフローが発生する可能性が認められる場合には、片方または両方のオペランドを加算前に右シフトします。これにより精度は下がりますが、オーバーフローをなくすことができます。

例：x+y を計算します。ただし、

X = 3*x かつ
Y = 5*y で、

どちらも範囲は [0,100] です。有効な結果は 8 ビットのみであると想定します。

- まず、X を、それよりスケールが精密な Y(5) に合わせてリスケーリングします。中間結果の X*5 は 1 バイトでは収まらないので、先に除算を行います（精度は低下します）。

  ```
  X':=(X/3)*5
  ```

- 中間結果の X' の値の範囲は [0,165] です。X'+Y という加算の結果はオーバーフローになるため、両オペランドを右シフトしてダウンスケールしてから加算します。
- この加算演算全体について生成されるコードは、以下のようになります。

  ```
  (((X/3)*5)>>1)+(Y>>1)
  ```

注記

一般に、加算は入力同士を交換できる可換演算であると考えられていますが、ターゲットコード生成ではさまざまなシフト演算が適用されるため、可換演算ではありません。ユーザーは特殊な状況、特に複合算術式の場合には充分な注意が必要です。

12.3.3 乗算

乗算では、加算とは異なり、かけ合わせる 2 つの量の量子化が互いに異なっていても実行できます。たとえば、X=ax+b と Y=cy+d をかけると、以下のようになります。

```
X*Y = acxy + adx + bcy + bd
```

ただし、両オペランドがオフセット b=d=0 で実装される場合には、整数の結果は簡素化されます。すると、整数の結果は物理結果の単なる線形スケールになります。

```
X*Y = (a*c)*(x*y)
```

その結果、乗算のためのコードが以下のようなステップで生成されます。

オフセットを 0 にする： 両オペランドのオフセットを 0 にします。

乗算： その結果同士をかけ合わせます。

オーバーフローの扱い： 指定された値の範囲によりオーバーフローが発生する可能性が認められる場合には、乗算でオーバーフローが起きなくなるまで両オペランドを右シフトします。この、最低限必要な精度の低下は、各オペランドの有効ビット数に応じた割合で両オペランドに割り振られます。

例：x*y を計算します。ただし、

X = 50x+3 ∈ [3,203] かつ
Y = 4y ∈ [0,10] で、

8 ビットの算術演算だけが有効であると想定します。

- まず、X をシフトして、オフセットを 0 にします。つまり、
  X’= X-3 ∈ [0,200]
- X'*Y という乗算の結果はオーバーフローになります。つまり、新しいインターバル ∈ [0,2000] になります。8 ビット以内に収めるために、3 ビット分の右シフトが必要です。大きい方の値 X' を 2 ビット分シフトし、Y を 1 ビット分シフトします。
- 乗算について生成されるコードは、以下のようになります。

  ```
  ((X-3)>>2)*(Y>>1)
  ```

- この結果のスケール値は 200/8=25、オフセットは 0 になります。

注記

右シフトを行ってオーバーフローを回避すると精度が低下するので、大きな値を用いる乗算がいくつか連続する場合には、不満足な数値精度になりやすいといえます。しかし、これはコードジェネレータに起因する誤差ではなく、使用可能なワード長が限られていることによる潜在的問題です。そこで、連続する乗算は、非常に慎重に行う必要があります。必要に応じて、変数を挿入して中間結果を所定のスケール値に強制的にスケーリングする必要があります。

12.3.4 除算

除算も、乗算の場合と同様に、互いにスケールの異なるオペランドを使用して行うことができます。この場合もまず、オペランドのオフセットを 0 にする必要があります。それから、除算の結果を 2 つのスケール値の比率でスケーリングします。

```
X=ax, Y=cy and X/Y=(a/c)*(x/y)
```

乗算とは異なり、ここではオーバーフローが発生することはありません。また実行時にゼロ除算が発生することもありません。このことは、コードジェネレータによる値の範囲のチェックにより保証され、分母の範囲に 0 が含まれている場合には、エラーメッセージが出力されます。

すでに説明したように、整数の除算ではかなりの数値誤差が発生する可能性があります。この誤差を少なくするために、コードジェネレータは以下の規則を使用します。

- できる限り、分子のワードサイズを分母のワードサイズの 2 倍にします（たとえば、分母が 8 ビットの場合には 16 ビットで分子を実装します）。これは、マイクロコントローラターゲットでの除算に使用される通常のアセンブラ命令に相当します。
- 分子は、ワードサイズ分をフルに使用するようにします。

まず、以上のようになっていない場合には、分子を左シフトして大きくする必要があります。

相応に物理指定された除算用のコードは、以下のステップで生成されます。

オフセットを 0 にする： 両オペランドのオフセットを 0 にします。

分母がゼロにならないかどうかをテストする： 分母の値の範囲に 0 が含まれている場合には、コードジェネレータはエラーメッセージを出力して停止します。

分子を大きくする： 可能であれば、分子のワードサイズが分母のワードサイズの 2 倍になるように分子を適切なビット数だけ左シフトして大きくし、ワードサイズをフルに使用するようにします。

除算： 最後に、除算を行います。

例：x/y を計算します。ただし、

X = 3x ∈[0,255] かつ
Y = y ∈[2,10] であるとします。

- X を 8 ビット分左シフトして、16 ビットワードサイズをフルに使用します。
- 次に、16 ビットを 8 ビットで割る除算を行います。生成されるコードは、以下のようになります。

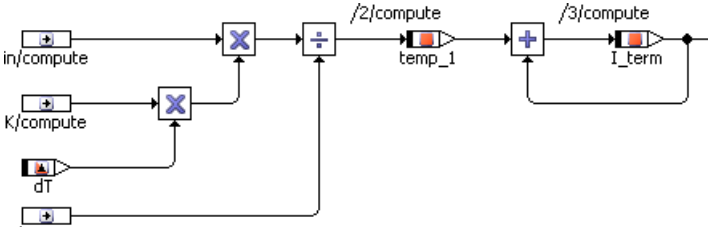
  ```
  (X<<8)/Y
  ```

- この結果のスケール値は 3*256、オフセットは 0、値の範囲は [0,32640] です。

PID コントローラ内の積分項を算出する：

- 下図の I_term の計算について考えてみましょう。

  これは代入、加算、乗算および除算の演算を組み合わせたものです。



インプリメンテーションは、以下のように定義されています。

```
in      = 2048*in ∈[-2,2],
K       = 64*k ∈[0,50],
dT      = 214*dt ∈[0,0.1],
Ti      = 1024*ti ∈[0.005,2],
temp_1  = 1024*temp1 ∈[-2,2],
I_term  = 256*iterm ∈[-100,100]
```

中間結果が 32 ビット長になってしまう可能性があるので、temp_1 というテンポラリ変数を使用して、式を 2 つの部分に分けて計算します。

- 最初の乗算である K*dT のスケール値は $64*2^{14}$ で、範囲は [0,5] です。この乗算には 23 ビットしか必要ないので、結果はオーバーフローしません。
- 次の乗算である K*dT*in のスケール値は $2^6*2^{14}*2^{11}=2^{31}$ で、範囲は [-10,10] です。これにより、4 ビットのオーバーフローが生じます。そこで、in（12 ビット）と K*dT（24 ビット、符号あり）の有効ビット数に応じた割合で、右シフトを割り振ります。
- 次に、結果を T_i で割ります。分子はすでにワードサイズをフルに使用しているので、左シフトは必要ありません。結果を temp_1に代入するには、$1/128=2^{-7}=2^{10+10-6-14-11+4}$ のリスケーリングが必要です。こうして、以下のようなコードが生成されます（クリッピングが必要ですが、ここでは示されていないので注意してください）。

  $temp\_1 = (((in>>1)*((K*dT)>>3))/T_i)>>7;$

- 第 2 の部分は temp_1 + I_term という加算です。通常、リスケーリングには細かい方の量子化（つまりここでは temp_1 の量子化）を使用しますが、結果を再び I_term に代入する必要があるので、$1/4=2^{-2}=2^{8-10}$ というリスケーリングが使用されます。これにより、リスケーリング処理が 1 つ省かれます。これについての詳細は、160 ページの 12.3.9 項を参照してください。こうして、以下のようなコードが生成されます（ここでも、クリッピングは示されていません）。

```
I_term = ( (temp_1>>2) + I_term);
```

- この例で生成されるコードをもう一度調べ、上の式を確認してください。特に、リミッタの実装方法に注意してください。

12.3.5 比較

比較演算子への入力は、一般的な変換式に変換する必要があります。この変換式のカスタマイズは 2 つのステップで行われます。まず、スケーリングが適合され、次にオフセットが調整されます。

通常、比較は、不必要な精度の損失を防ぐため、2 つ値のうちの細かく量子化されている方に合わせて実行されますが、細かい方の量子化を使用することによってリスケーリング後の値が使用可能ワードサイズを超えてしまう場合には、粗い方に合わせて量子化されます。

12.3.6 スイッチとマルチプレクサ

比較の場合と同様に、スイッチおよびマルチプレクサのすべての入力も、一般的な変換式に変換する必要があります。これは、比較演算子の場合と同様に行われます。それから、C の通常の制御文、つまり、`if/else`、`case`、`(a?b:c)` により、選択が行われます。

12.3.7 リテラル

リテラルには、ASCET でインプリメンテーション情報を指定することはできません。コードジェネレータはそれぞれのコンテクストに合った変換式を使用して、リテラルを自動変換します。

例：X が 10 倍にスケーリングされる場合、モデル内の `x+1.0` という計算は `X+10` に変換されます。

リテラルが非常に粗く量子化されている場合、その周囲の変数の自動的な量子化適合により、不満足な結果が得られる可能性があります。このような事態は、中間結果を伴う数式の中でリテラルが乗算や除算に用いられる場合に、特に発生する可能性があります。たとえば、次の式について考えてみましょう。

```
y := x*1.049,
```

ここで、x および y は 0.1 で量子化されます。x の値の範囲によっては、リテラル 1.049 は整数 `10`（つまり、物理値 `1.0`）で近似することができます。このような場合には、リテラルは式から完全に姿を消してしまいます。

```
y := (x*10)/10 = x
```

この影響を抑えるために、リテラルは改良されたスケール値を取得します。目標は相対誤差を `0.1%` 未満に保つことです。上の例では、リテラル `1.049*10=10.49` が `671/64=10.484` として実装されます。したがって、前ページの式は以下のようになります。

```
y := (x*671)/640,
```

これで、係数はほぼ適切に近似化されます。

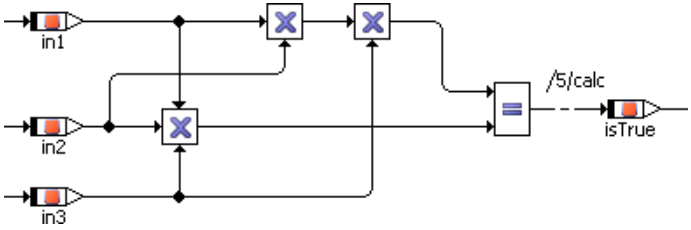
注記

0.001 という精度しきい値はハードコーディングされているので、ユーザーがこれを微調整することはできません。そのため、自動的に調整されたリテラルの量子化があまりにも不正確になってしまうことが、まれにあります。このような場合には、モデル内でリテラルの代わりに定数を使用することができます。そうすれば、ユーザーが変換式を供給することができます。

12.3.8 3 個以上の入力を持つ演算子

3 個以上の入力を伴う数学演算子は、2 項演算のシーケンスに分解され、それらの各演算についてコードが順に生成されます。

結果的に、下図の結果は常に `True` になります。



注記

一般的には、加算は入力同士を交換できる可換演算であると考えられますが、ターゲットコード の生成においてはさまざまなシフト演算が適用されるため、可換演算としては扱われません。特殊な状況、特に複合算術式の場合には、ユーザーが充分な注意を払うことが必要です。

12.3.9 数式の最適化

コードジェネレータは数式を最適化するためにヒューリスティックな制御方針を使用します。この制御方針は 2 つのフェーズで機能します。まず、ボトムアップの意味解析で、数式内の各中間結果について最適化データが収集されます。それから、このデータに基づくトップダウンの生成フェーズで、それぞれの結果についてのターゲットスケーリングが定義されます。自由度（149 ページの「自由度と最適化の度合い」という項を参照してください）により、数式全体について最適なスケールを選択することができます。目的は、リスケーリング時に使用される追加計算の数を最小限にすることです。

最適なスケール値は、正規化スケール、つまり、2 の累乗ではないトータルスケール値内の係数を使用して決定されます。

たとえば、正規化スケールである 3 は、中間結果を $3*2^{N-1}$ で、つまり 3/2、3、6、12 などでスケーリングできることを示します。これは、以下の理由で重要です。

- 数値の精度をカスタマイズしてオーバーフローを防ぐことができるように、スケールの範囲は可変でなければなりません。
- このようなカスタマイズは、シフトにより実行されます。
- これは、シフト演算は乗算や除算よりも効率的であるという仮定に基づいています。このことは、ほとんどのターゲットに当てはまります。

このアプローチを説明するために、簡単な例を紹介します。

例：4 つの変数 v、w、x、y の加算を行い、その結果を z に代入します。

```
z := ((v+w)+x)+y
```

これらの変数が以下のようにスケーリングされると想定します。

変数	スケール値	標準化
v	4	1
w	3	3
x	8	1
y	5	5
z	10	5

まず、ボトムアップの意味解析においては、すべての中間結果について、正規化スケール値（つまり、2 の累乗の係数以外）を使用して一連の最適スケール値を収集します。

次に、コード生成フェーズでは、このローカルデータを使用して、式全体を通じて下方向にバックトラッキング（トップダウン）することにより、各結果について最適なスケール値を選択します。

中間結果	最適スケール値	コメント
v+w	1 または 3	どちらのスケール値もリスケーリングを 1 回しか必要としないので、同程度に適切に機能します。これ以外のスケール値を使用すると、両方の入力をリスケーリングしなければならなくなります。

`(v+w)+x`	1	x のスケール値は 1 なので、v+w も 1 でスケーリングするのが最適です。これにより、余計なリスケーリングを 1 つ省くことができます。そのようなわけで、ここでは 1 の方が 3 よりも適しています。
`((v+w)+x)+y`	1 または 5	ここでもやはり、どちらを選択しても同じくらいうまく機能します。少なくとも片側だけはリスケーリングが必要です。
`z:=((v+w)+x)+y`	5	代入の前にリスケーリングが必要なくなるように、この式全体をスケール値 5 でコード化します。

これらのスケール値は、必要なリスケーリングおよびシフト演算に従って挿入されます。中間結果ではオーバーフローが発生しないと仮定して、下記のコードが式用にコンパイルされます。

中間結果	**スケール値 ***
`t1 := v+((w>>2)/3)`	4
`t2 := (t1<<1)+x`	8
`z := ((t2*5)>>2)+(y<<1)`	10

* 正規化されていません。

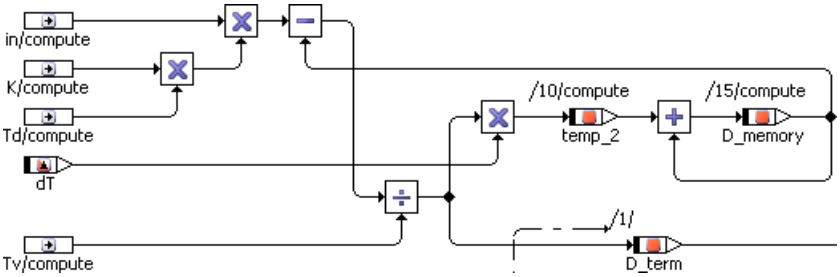
ここでは、分かりやすいように中間結果を個別に示しています。実際のコード生成では、C コード用の長い数式が生成されます。個々の演算の生成は制御方針に従ってローカルに行われます。これらの演算については、グローバルな最適化は行われません。

これまでは、PID コントローラのすべての例では 2 の累乗のスケール値を使用していました。そのため、すべてのリスケーリングはシフト演算によって行われましたが、これでは、いつ最適化が発生したかがあまり明白ではなくなってしまいます。たとえば、積分項の計算（前の記述を参照してください）の最後の加算である `temp1+I_term` は、`I_term` への最後の代入で行われるシフト演算を 1 つ省くために、粗めのスケール値で行われました。

次の例では、必ずしも完全な 2 の累乗ではないスケール値を使用します。

導関数の項の計算を最適化する：

- 下図に示す PID コントローラの例の、導関数項の計算について考えてみましょう。



ここでは、D_term の計算だけに的を絞ります。

- 式内の量のインプリメンテーションを確認します。

インプリメンテーションは以下のように定義されています。

```
in          = 2048*in ∈[-2,2],
K           = 64*k ∈[0,50],
Td          = 640*td ∈[0,2],
Tv          = 640*tv ∈[0.005,2],
D_memory    = 1024*dmem ∈[-10,10],
D_term      = 256*dterm ∈[-100,100]
```

ここでもやはり、中間結果は 32 ビットになります。D_term の計算は、以下のように行われます。

- 積分項の計算の場合と同様に、最初の 2 つの積算である K*Td*in では 3 ビットのオーバーフロー（つまり、3 ビットの右シフト）が発生します。結果のスケールは $2^{6}*640*2^{11}*2^{-3} = 5*2^{21}$ になり、範囲は [-200,200] になります。
- 次に、結果から D_memory を引きますが、まず両オペランドのスケールを同じにする必要があり、ここで最適化が発生します。以下のスケール値を考慮する必要があります。

オペランド	スケール値	正規化
K*Td*in	$5*2^{21}$	5
D_memory	2^{10}	1
Tv	$5*2^{7}$	5
D_term	2^{8}	1

- この減算の結果である K*Td*in-D_memory には、5 または 1 のいずれかの正規化スケールを使用することもできます。1 を使用すると、Tv で割る次のステップではスケール値 5 が再び採用されます。この結果を D_term への最後の代入に使用するためには再びスケーリングする必要があるので、全部で 3 回のリスケーリングが必要になります。

  このようなわけで、正規化スケールの 5 を選ぶ方が得策です。Tv による除算では 5 によるスケーリングをしなくなるので、最後の代入でもリスケーリングが必要なくなります。結局、リスケーリングは 1 回だけ（つまり、D_memory のスケール値を 5 にするだけ）で済みます。
- 結果的に、この減算については D_memory がリスケーリングされて $5*2^{21}$ になります（正規化されていません）。しかし、両オペランドを右シフトしてオーバーフローを防ぐ必要があるので、実際のスケール値は $5*2^{20}$ になります。
- 最後に、この結果を Tv で割ります。分子はすでに 32 ビットをフルに使っているので、左シフトは必要ありません。結果を D_term に代入するためには、$5*2^{7}*2^{8}/(5*2^{20})=2^{-5}$ というリスケーリング（つまり、右シフト）が必要です。

  中間結果は以下のようになります。

中間結果	スケール値
`t1 := (in>>1)*((K*Td)>>2)`	$5*2^{21}$
`t2 := (t1>>1) - D_memory*5120`	$5*2^{20}$
`D_term := (t2/Tv)>>5`	2^{8}

上記の中間結果は、分かりやすいように個別に示されていますが、実際の C コードでは、1 つの長い式が生成されます。

- この例で生成されるコードをもう一度調べ、上の式を確認してください。その際、リミッタがどのように実装されているかに注意してください。

13 生成されるコードについて

本章では、ASCET-SE が生成するソフトウェアの特性について説明します。生成されるコードの内容を理解し、開発工程においてコードレビューを行う際の参考となるように、ASCET モデルの内容と構造体を C コードに変換する際の基本ルールを説明します。

13.1 モジュラー性

ASCET-SE のコード生成はモジュラー単位で行われます。C コードとヘッダファイルは ASCET の複合エレメント（プロジェクト、モジュール、クラス）ごとに生成され、各モジュールとそのエレメントについて、階層構造のデータ構造体が 1 つずつ生成されます。これを行う際、システム全体についての知識は必要ありませんが、1 つのモジュール用のコードとその階層構造が正しくコード化されるためには、すべての依存モジュールについてパブリックインターフェース（エクスポートされる変数、パブリックメソッド）が明確になっている必要があります。そこで、コードジェネレータは、プロジェクトと各モジュール／クラスについてクラスインターフェースと呼ばれる内部構造体を生成します。エレメントのコード生成を行う場合に必要なのは、それが参照するすべてのエレメントの*クラスインターフェース*だけです。C コードのマニュアルプログラミングにおいては、プロトタイプ宣言をヘッダファイルに格納してそれを使用する、という手法がよく使われますが、これはその手法によく似ています。

13.2 生成されたコードの複数ファイルへの分割

各エレメント（クラス、モジュール、プロジェクト）ごとに生成された C コードは、いくつかのファイルに分かれます。ファイル名は最大 255 文字まで使用できる Windows ファイルフォーマットで自動生成され、必要に応じて MS-DOS 互換の 8.3 フォーマットも使用できます。

ここでは以下の規則が適用されます。

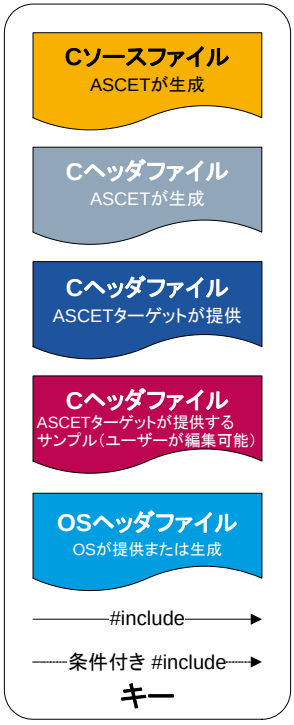
- プロジェクト、各クラス／モジュール、各インプリメンテーションごとに C ソースコードファイル（`*.c`）が 1 つづつ生成され、さらに“Project Properties”ダイアログボックスの“Build”ノードの“Header Structure”オプションの設定に応じて以下のように C ヘッダファイル（`*.h`）が生成されます。
  - `Component`（デフォルト）：ヘッダファイルはプロジェクト用、および各モジュール／クラス用に生成されます。
  - `Module`： ヘッダファイルはプロジェクト用、および各モジュール用に生成されます。
  - `Project`： ヘッダファイルは各プロジェクト用にのみ生成されます。
- 外部 C コードを含むモジュールの場合、外部コードが含まれる 2 つのファイル（`*E.c` および `*E.h`）がさらに生成されます。

- プロジェクト用に生成されたすべてのヘッダファイルは、`FILES_HEADER_PROJ` 変数に割り当てられます（5.4.3 項を参照してください）。
- ASCET モデル内のすべての関数の外部宣言を含んだヘッダファイル、`function_declarations.h` が生成されます。
- ASCET モデル内のすべての変数とパラメータの外部宣言を含んだヘッダファイル、`variable_declarations.h` が生成されます。

13.2.1 インクルード階層

生成されるコードのインクルード階層は、Project Properties”ダイアログボックスの“Build”ノードの“Header Structure”オプションの設定に応じて変わります。

以下の図 13-1、図 13-2、図 13-3 は、3 つのオプション設定に応じた違いを同じキー（ファイルのカテゴリ）を用いて表したものです。



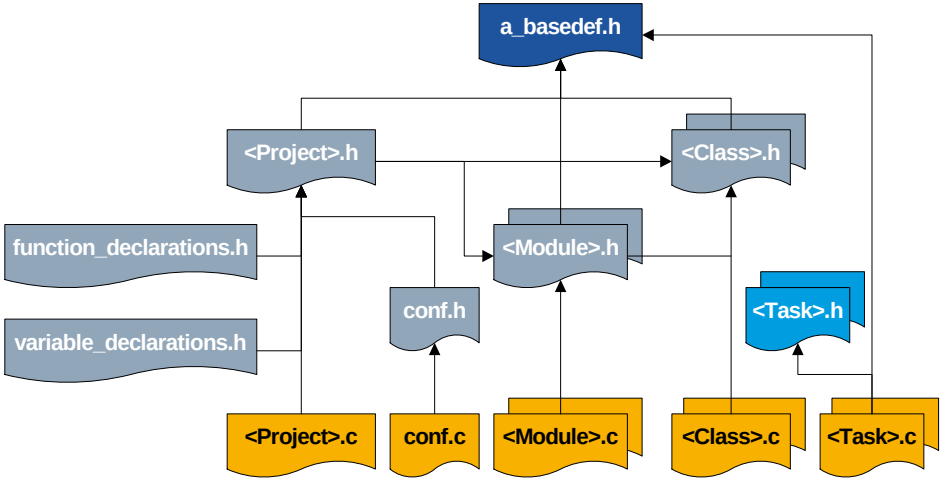


図 13-1 インクルード階層：コンポーネントヘッダ

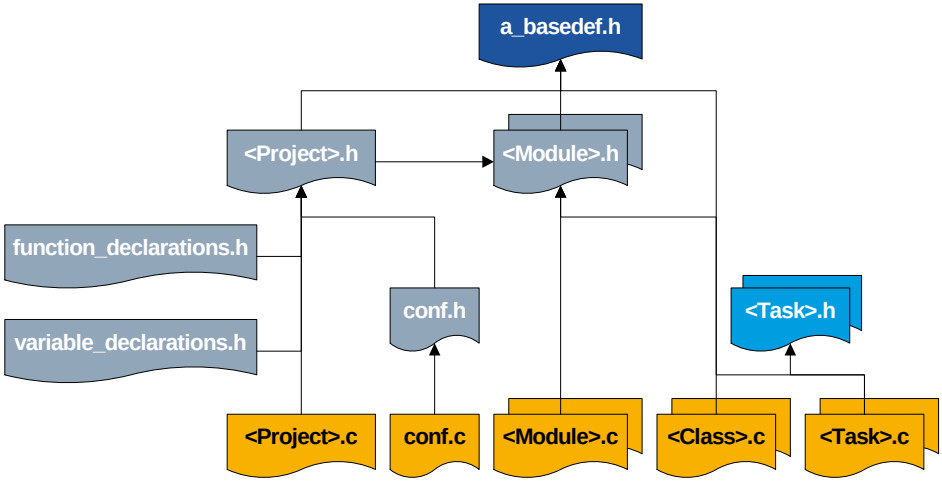


図 13-2 インクルード階層：モジュールヘッダ

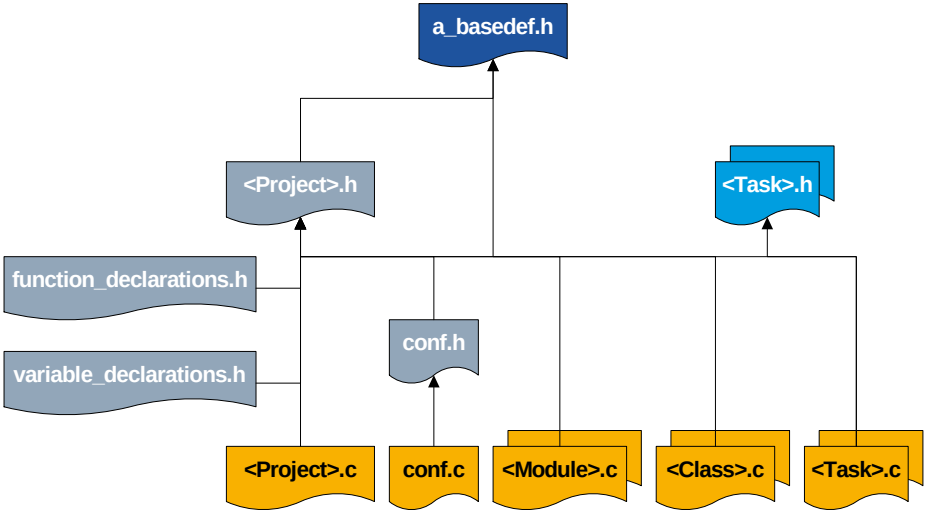


図 13-3 インクルード階層：プロジェクトヘッダ

a_basedef.h 自体のインクルード階層は、オプション設定に関わらず 図 13-4 のようになります。

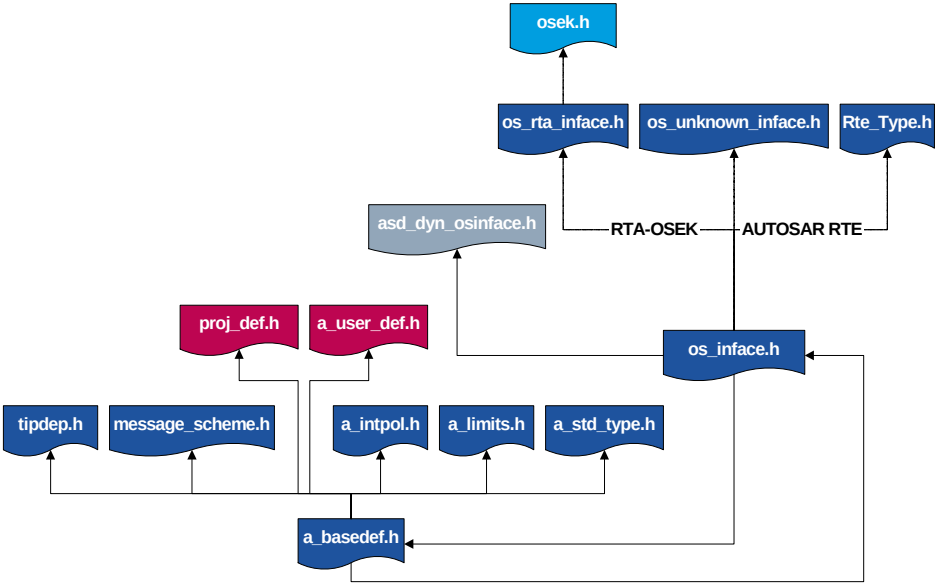


図 13-4 a_basedef.h のインクルード階層

13.3 ソフトウェアアーキテクチャ

「ソフトウェアアーキテクチャ」とは、ASCET モデルの論理記述とデータが C コードに変換される際の、あらゆる基本ルールです。これにはいろいろな事柄が含まれますが、命名規則、ストレージシステム（データのメモリ格納に関する方法）、およびデータ構造体の変換などがあります。各ターゲット用 ASCET-SE に対して共通の「基本ソフトウェアアーキテクチャ」が適用され、本項ではその基本部分について説明します。

このソフトウェアアーキテクチャの設計基準として以下のような条件があげられます。

- コントローラにおけるデータの*インスタンス生成*、およびそれによるメモリの予約は、完全に「静的」に行われます。動的アロケーションを使用することはできません。たとえば、変数に関してポインタ管理や malloc の呼び出しによって生じるメモリ所要量や実行時間のオーバーヘッドは許容できません。
- 選択されたデータ構造体では、クラスの静的な「多重インスタンス生成」が可能でなければなりません。これにより、同じコードを同じインプリメンテーションのすべてのインスタンスに使用できるようになります。同じコードを複製することは、メモリの浪費です。
- 最適化はシステム全体を通じて行われます。
- ユーザー定義メモリクラス内のデータストレージをサポートします。
- パラメータなど、すべての静的データは「静的に初期化」する必要があります。

上記の条件に基づいて、以下のような仕様が定められました。

- エクスポートされるパラメータは、エクスポートする C ファイル内でグローバル C 変数として静的に作成されて初期化され、それらをインポートするファイル内で external 変数として宣言されます。パラメータは ROM 領域に割り当てられます。
- エクスポート／インポートされる変数も同様に扱われますが、これらは RAM 領域内に静的に作成され初期化されます。ただし、Non-volatile 指定の変数については静的な初期化は行われません。初期化が必要な場合はユーザーが初期化コードを記述する必要があります。
- クラスおよびモジュールのローカルエレメントは、それぞれ個別の C 構造体に格納されます。それらが複数の異なるメモリクラスに属する場合、各メモリクラスに C 構造体が追加され、これらは C ポインタを使用してアクセスされます。モデルの構造に基づいて、各モジュール用にネスト構造（あるモジュールの中に、他の別のクラスのインスタンスがエレメントとして含まれます）の「インスタンスツリー」が作成されます。インスタンスは、このように構造体に組み込む以外に、モデル内で **As reference** オプションがオンになっている場合はポインタでアクセスすることもできます。後者の方法は、2 つのオブジェクトが相互に参照しあう場合（車軸の回転など）に必要です。

- 受信インスタンスのメモリ領域のポインタが各メソッド呼び出しで渡されるので、ひとつのメソッドのコードを、同じインプリメンテーションのすべてのクラスのインスタンスで使用できます（このポインタは *self* ポインタと呼ばれ、インスタンスが複数生成される場合、またはエレメントのインプリメンテーションで明示的に指定された場合にのみ使用されます）。
- 各メモリ領域ごとに、各コンポーネントのエレメントは 1 つの構造体にまとめられます。各コンポーネント（ただしデータを含むもののみ）ごとに 1 つの構造体が存在し、そこからメモリクラス構造体を参照することができます。
- すべての暗黙的初期化は、静的に行われます。
- 特性カーブおよびマップ用には、固定的なストレージシステム（レコードレイアウト）のみサポートされます。

13.3.1 命名規則

ASCET コンポーネント（クラス、モジュール、またはプロジェクト）用の C ネームは、以下の規則に従って割り当てられます。

< コンポーネント名 >_< インプリメンテーション名 >

クラス名には、インプリメンテーション名を付加する必要があります。なぜなら、1 つのクラスの複数のインスタンスが、互いに異なるインプリメンテーションでモデル内に発生する可能性があるからです。以降では、この名前を `ClassIdentifier`（「クラス識別子」）と呼びます。

モジュールやプロジェクトは 1 つのインスタンスしか持つことができないので、インプリメンテーション名を付加する必要はありません。しかし、一貫性をもたせるために、これらのコンポーネントの命名についても上記の規則が適用されます。

実験ターゲット用のコードを生成する場合とは対照的に、この命名規則では、「1 つのプロジェクト内のクラス、モジュール、およびプロジェクトの名前は一意でなければならない」という制約 があります。この制約を守らないと、誤動作やコンパイラ／リンカエラーが発生する可能性があります。そのため、Make メカニズムにおけるコード生成処理の開始時に、名前に曖昧さがないかどうかが確認されます。

ユーザーはエクスパンダコンフィギュレーションファイル `codegen.ini` 内で、クラスおよび変数の名前を作成するための規則を部分的に修正することができます。詳細は、5.1「`codegen[_*].ini` ファイル」を参照してください。

13.3.2 基本オブジェクトのストレージシステム（データ構造体と初期化）

実験用ターゲットの場合、シミュレーション実行中の任意のポイントでデータを記録したり変更したりできる汎用オブジェクト構造体が使用されますが、ここでの動的メモリアロケーションに必要なメモリおよびランタイム要件は、ECU のターゲットコントローラで使用するには厳しすぎます。シミュレーションで使用される補足データは ECU には必要ないので、この基本ソフトウェアアーキテクチャによりプリセット された圧縮構造に置き換えられ、ユーザーがこれを修正することはできません。

実装データ型の汎用定義（`uint16` など）はターゲットコントローラでも使用されるので、グローバルシステムヘッダファイルに定義されます。

注記

以下の例でわかるように、グローバルエレメントのデータ構造体は孤立的に作成される（つまりインスタンスツリーには組み込まれない）ため、宣言および初期化のために生成されるデータ構造体を記述することができます。ローカルエレメントの場合、宣言および初期化コードは指定されたとおりに生成され、インスタンスツリーに組み込まれます。

スカラ値と論理値

グローバルなスカラ値と論理値は、指定された実装データ型の C 変数にコード化されます。

```
uint16 scalarVariable;
```

グローバルなスカラパラメータと変数の初期化は、定義文によって静的に行われます。

```
const uint16 scalarParameter = 123;
```

ローカル変数の定義と初期化は、データ構造体の一部として行われます。
Non-volatile 指定の変数は、ローカル変数、エクスポート変数を問わず、初期化は行われません。

注記

Non-volatile 指定の変数は、どのような場合であっても初期化は一切行われません。

配列とマトリックス

配列とマトリックス（行列）は、指定された実装データ型の C の配列にコード化されます。

```
sint32 array[size];
uint16 matrix[size];
```

配列のサイズは固定なので、実行時に変更することはできません。このサイズはモデル内のサイズに相当します。マトリックスは、C コードの 1 次元配列として生成されます。

注記

複数インスタンスの場合、すべてのデータレコードについて同じ 1 つのオブジェクト定義が使用されるので、配列とマトリックスのサイズはすべてのインスタンスで同じでなければなりません。サイズはオブジェクト定義と一緒には格納されません。これは、モデル内で配列のサイズにアクセスすることはできないので、必要ないためです。

メモリ上では、配列は順に増加するインデックスに従って格納されます。マトリックスの場合は列ごとに格納されます。つまり最初に第 1 列めが順に格納され、続いて次の列が格納されます。

以下はマトリックスの例です。

```
1 2 3
4 5 6
7 8 9
```

上記のようなマトリックスは以下のように格納されます。

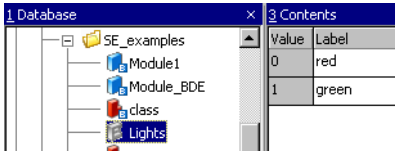
```
1 4 7 2 5 8 3 6 9
```

グローバルパラメータ配列の初期化は、C コードの定義文の中で静的に行われます。

```
const uint16 arrayParam[4] = { 10,1,4,9 };
```

配列およびマトリックスは `#define` 文としてコード化されるので、これらを定数またはシステム定数として*作成することはできません*。このため旧バージョンの ASCET からプロジェクトを移行する場合には、既存のシステム定数をマニュアル操作でパラメータに切り替える必要が生じます。

列挙型データ



列挙型データは、C コードの最小の整数データ型に割り当てられるため、C の `enum` 型とは異なり、マシンワードより小さいワードサイズで済みます。

```
uint8 Lights;
```

シンボル名（この例では `red` および `green`）は整数値に割り当てられます。ASCET で生成される ASAM-MCD-2MC デスクリプションファイル内では、それぞれのシンボル名が各整数値に割り当て直されるので、適合システムからこれらの名前にアクセスすることができます。

```
/begin COMPU_VTAB enum_Lights_tab_ref
""
TAB_VERB
2
0 "red"   1 "green"
DEFAULT_VALUE "Error"
/end COMPU_VTAB
```

特性カーブ

特性カーブには、以下のような単純なストレージシステムが使用されます。

格納される値	説明	バイト数
n（先頭アドレス）	軸ポイントの数	1 または 2 バイト（下記参照）
X1		
X2		
…	軸ポイント	n*x バイト、インデックスは順に増加
Xn		
W1		
…	特性値	n*w バイト、インデックスは順に増加
Wn		

表 13-1 ストレージシステム - 特性カーブ

軸ポイントと特性値（X と W）がどちらも 1 バイトで実装される場合は、軸ポイント数は 1 バイトに格納され、それ以外の場合には 2 バイトに格納されます。

このようなストレージシステムでは、軸ポイント数や、軸ポイントと値の実装データ型が変わる可能性があるので、C の汎用構造体定義を使用することはできません。そこで、コード生成においては個々の特性カーブごとに別の構造体定義が生成されます。またモジュールと初期化コードは別々に生成されるので、この定義の名前が C のヘッダコードに登録されます。

特性カーブ の例：

KL には 3 つの軸ポイントがあります。入力および出力のデータ型は、ともに sint16 です。

C コードでは、この構造体はコンポーネントのヘッダファイル <component>.h に以下のように定義されます。

```
struct PIDT1_MOD_IMPL_KL_TYPE {
   uint16 xSize;
   sint16 xDist [3];
   sint16 values [3];
};
```

グローバルエレメント KL の静的初期化は、宣言文の中で行われます。

```
const struct PIDT1_MOD_IMPL_KL_TYPE KL =
{
   3,
```

```
    {
       -2, 1, 4
    },
    {
       5, 6, 7
    }
};/*** KL ***/
```

ローカルな特性カーブは、ネストされたデータ構造体の一部として定義され、初期化されます。

ストレージシステムでは、軸ポイントの現在数と最大数とを区別しません。また今後、適合時に軸ポイント数を調整できるようになる予定もありません。`struct` 内のベクタのサイズが、生成時に設定された軸ポイントの数に相当します。

アクセスは、アクセスルーチンによって行われます。特性値にアクセスするには 2 つの方法があり、1 つは補間ルーチンを使用する「リニア（線形）」方式で、もう 1 つは特性値をルックアップテーブルとして使用する「丸め」方式です。ASCET-SE には両方のアクセスルーチンが含まれています。上記の例の場合、リニアアクセスは以下のように行われます。

```
pwm_out = CharTable1_getAt_s16s16
        ((void *)&KL, xin);
```

また丸めアクセスの場合は以下のようなコードが生成されます。

```
pwm_out = CharTable1_getAtR_s16s16
        ((void *)&KL, xin);
```

注記

アクセスルーチンを作成する場合、構造体エレメントのストレージ（「アライメント」）はコンパイラによって定義されるので、注意してください。

特性値のデータエディタにおいて、ユーザーはリニアアクセスと丸めアクセスのどちらを使用するかを指定できます。

特性マップ

特性マップのストレージシステムは、下の表のとおりです。これは、特性カーブのものとよく似ています。

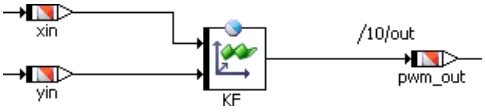
格納される値	説明	バイト数
n（先頭アドレス）	X 軸ポイントの数	1 または 2 バイト（下記参照）
m	Y 軸ポイントの数	1 または 2 バイト（下記参照）
X1		
…	X 軸ポイント	n*x バイト、インデックスは順に増加
Xn		
Y1		
…	Y 軸ポイント	m*y バイト、インデックスは順に増加
Ym		
W1,1		
W1,2		
…	特性値	(n*m)*w バイト、カラムごと（Y のインデックス m の方が X のインデックス n より先に増加）
Wn,m-1		
Wn,m		

表 13-2 ストレージシステム - 特性マップ

ここでは、すべての軸ポイントと特性値（X、Y、W）が 1 バイトで実装される場合には、X と Y の軸ポイント数（n と m）はどちらも 1 バイトに格納され、それ以外の場合には、2 バイトに格納されます。

特性カーブの場合と同様に、コード生成においては個々の*特性マップごと*に別の struct 定義が生成されます。

特性マップ の 例：



KF には x 軸に 3 つ、Y 軸に 4 つの軸ポイントがあります。入力のデータ型はともに sint16 で、出力は uint16 です。

C コードでは、コンポーネントのヘッダファイル <component>.h に以下のように定義されます。

```
struct PIDT1_MOD_IMPL_KF_TYPE {
   uint16 xSize;
   uint16 ySize;
   sint16 xDist [3];
   sint16 yDist [4];
   sint16 values [3 * 4];
};
```

グローバルエレメント KF の静的初期化は、やはり宣言文の中で行われます。

```
const struct PIDT1_MOD_IMPL_KL_TYPE KF =
{
   3,
   4,
   { 1, 3, 5 },
   { 0, 1, 8, 15 },
   { -5, -3, 0, 1,
      0, 1, 4, 6,
      8, 5, 4, 4 }
};/*** KF ***/
```

ローカルな特性マップは、ネストされたデータ構造体の一部として定義され、初期化されます。

軸ポイントと値は、順に増加するインデックスを使用して格納されます。それぞれの格納スペースが、生成時に設定された軸ポイント数に合わせて確保されます。値の格納はカラムごとに行われます。他はすべて、特性カーブの場合と同じです。以下に示すリニアアクセス（補間ルーチン呼び出し）の例で分かるように、アクセスの方法も同様です。

```
pwm_out
   =  CharTable2_getAt_s16s16s16(
      (void *)&KF, xin, yin);
```

丸めアクセスの場合も、やはり以下のようなコードが生成されます。

```
pwm_out
   =  CharTable2_getAtR_s16s16s16(
      (void *)&KF, xin, yin);
```

特性値のデータエディタにおいて、ユーザーはリニアアクセスと丸めアクセスのどちらを使用するかを指定できます。

グループ特性カーブおよびマップについては、順に増加するインデックスを使用する配列に値のみが 格納されます。

格納される値	バイト数
W1（先頭アドレス）	
…	n*w バイト、インデックスは順に増加
Wn	

表 13-3 ストレージシステム - グループ特性カーブ

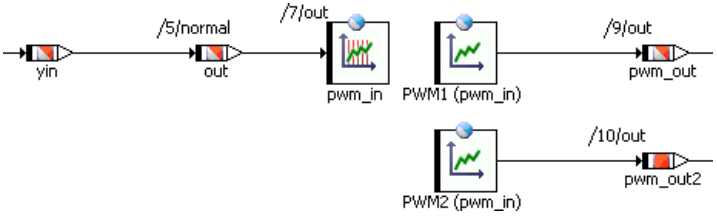
各軸ポイントは、別の「軸ポイントディストリビューション」オブジェクトに保存されます。

格納される値	説明	バイト数
n（先頭アドレス）	軸ポイント数	2 バイト
X1		
X2		
…	軸ポイント	n*x バイト、インデックスは順に増加
Xn		

表 13-4 ストレージシステム - 軸ポイントディストリビューション

軸ポイントディストリビューションは、複数のグループ特性カーブまたはマップに共通して使用することができます。

軸ポイントディストリビューションおよびグループ特性カーブの例：



pwm_in に定義されているように、PWM1 および PWM2 にはそれぞれ 6 個の軸ポイントがあります。pwm_in の入力データ型は uint16 です。どちらのカーブの出力データ型も uint16 です。

C コード内の静的な定義は、コンポーネントのヘッダファイル <component>.h 内で以下のように行われます。

```
struct PIDT1_MOD_IMPL_pwm_in_TYPE {
  uint16 size;
  uint16 dist [6];
};
struct PIDT1_MOD_IMPL_PWM1_TYPE {
  sint16 values [6];
};
struct PIDT1_MOD_IMPL_PWM2_TYPE {
  sint16 values [6];
};
```

さらに、それぞれの軸ポイントディストリビューションごとに、補間結果を格納するための中間メモリとして、3 つの変数が生成されます。これらの変数は、グループ特性カーブにアクセスするために使用されます。

```
uint16  pwm_in_index;
uint16  pwm_in_offset;
uint16  pwm_in_distance;
```

これらのエレメントはエクスポートされるので、データ構造体の初期化は、やはり別の構造体の中で行われます。中間変数は個別に初期化されません。

```
const struct PIDT1_MOD_IMPL_pwm_in_TYPE pwm_in =
{
  6,
  {
    0, 4, 8, 10, 12, 13
  }
};/*** pwm_in ***/
const struct PIDT1_MOD_IMPL_PWM1_TYPE PWM1 =
{
  {
    1584, 16, 16, 0, 0, 0
  }
};/*** PWM1 ***/
const struct PIDT1_MOD_IMPL_PWM2_TYPE PWM2 =
{
  {
    16, 16, 1584, 0, 0, 0
  }
};/*** PWM2 ***/
```

ローカルなディストリビューションテーブルとグループ特性カーブは、ネストされたデータ構造体の一部として定義され、初期化されます。

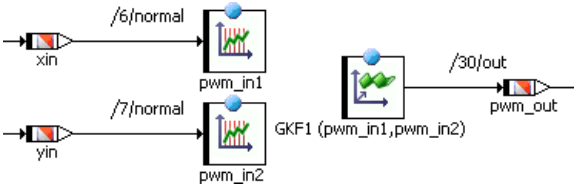
アクセスは、モデルの場合と同様に、2 つのステップで行われます。まず、軸ポイントの検索が行われます。

```
Distribution_search_u16(
   (void*)&pwm_in.dist,
   (uint16)pwm_in.size,
   (uint16)out,
   (void *)&pwm_in_index,
   (void *)&pwm_in_offset,
   (void *)&pwm_in_distance);
```

軸ポイントの検索結果は中間変数 pwm_in_index、pwm_in_offset および pwm_in_distance に格納されます。その後、これらの結果には専用の補間ルーチンでアクセスすることができます。こうして、軸ポイントの 1 回の検索に基づいて、複数の異なる特性カーブおよびマップを評価することができます。

```
pwm_out
   =  GroupTable1_getAt_u16s16(
      (void*)&PWM1,
      pwm_in_index,
      pwm_in_offset,
      pwm_in_distance);

pwm_out
   =  GroupTable1_getAt_u16s16(
      (void*)&PWM2,
      pwm_in_index,
      pwm_in_offset,
      pwm_in_distance);
```

軸ポイントディストリビューションおよびグループ特性マップの例：



GKF1 には 4 つの X 軸ポイント（pwm_in1 に定義されています）と 3 つの Y 軸ポイント（pwm_in2 に定義されています）があります。pwm_in1 および pwm_in2 の入力データ型は uint16 で、特性マップの出力データ型は sint16 です。

C コード内の静的な定義は、コンポーネントのヘッダファイル <component>.h で以下のように定義されます。

```
struct PIDT1_MOD_IMPL_pwm_in1_TYPE {
   uint16 size;
   uint16 dist [4];
};
```

```
struct PIDT1_MOD_IMPL_pwm_in2_TYPE {
  uint16 size;
  uint16 dist [3];
};
struct PIDT1_MOD_IMPL_GKF1_TYPE {
  sint16 values [4 * 3];
};
```

この場合も、それぞれの軸ポイントディストリビューションごとに、3 つの中間変数が生成されます。

```
uint16  pwm_in1_index;
uint16  pwm_in1_offset;
uint16  pwm_in1_distance;
uint16  pwm_in2_index;
uint16  pwm_in2_offset;
uint16  pwm_in2_distance;
```

データ構造体の初期化：

```
struct PIDT1_MOD_IMPL_pwm_in1_TYPE pwm_in1 =
{
  4,
  {
    0, 4, 8, 12
  }
};/*** pwm_in1 ***/
struct PIDT1_MOD_IMPL_pwm_in2_TYPE pwm_in2 =
{
  3,
  {
    1, 2, 3
  }
};/*** pwm_in2 ***/
struct PIDT1_MOD_IMPL_GKF1_TYPE GKF1 =
{
  {
    -5, -3, 0,
    0, 1, 4,
    8, 5, 4,
    19, 7, 0
  }
};/*** GKF1 ***/
```

ローカルなグループ特性マップは、ネストされたデータ構造体の一部として定義され、初期化されます。

軸ポイントの検索は、各軸ポイントディストリビューションごとに個別に行われます。

```
Distribution_search_u16(
   (void *)&pwm_in1.dist,
   (uint16)pwm_in1.size,
   (uint16)xin,
   (void *)&pwm_in1_index,
   (void *)&pwm_in1_offset,
   (void *)&pwm_in1_distance);
Distribution_search_u16(
   (void *)&pwm_in2.dist,
   (uint16)pwm_in2.size,
   (uint16)yin,
   (void *)&pwm_in2_index,
   (void *)&pwm_in2_offset,
   (void *)&pwm_in2_distance);
```

軸ポイントの検索結果は中間変数に格納されます。その後、これらの結果には専用の補間ルーチンでアクセスすることができます。

```
pwm_out
   =  GroupTable2_getAt_u16u16s16(
      (void *)&GKF1,
      pwm_in1_index,
      pwm_in1_offset,
      pwm_in1_distance,
      (uint16)pwm_in1.size,
      pwm_in2_index,
      pwm_in2_offset,
      pwm_in2_distance,
      (uint16)pwm_in2.size);
```

固定特性カーブ／マップ

固定特性カーブおよびマップには等間隔の軸ポイントが含まれるため、軸ポイントを別のディストリビューション配列に格納する必要がなく、代わりに、軸ポイント数、第 1 ポイントのオフセット、およびポイント間の距離に応じて、データ構造体に内部ディスクリプションを格納します。

格納される値	説明	バイト数
n（先頭アドレス）	軸ポイント数	2 バイト
Xoff	最初の軸ポイントのオフセット	2 バイト
Xdist	軸ポイント間の間隔	2 バイト
W1		
…	特性値	n*w バイト、インデックスは順に増加
Wn		

表 13-5 ストレージシステム - 固定特性カーブ

格納される値	説明	バイト数
n（先頭アドレス）	X 軸ポイント数	2 バイト
m	Y 軸ポイント数	2 バイト
Xoff	最初の X 軸ポイントのオフセット	2 バイト
Xdist	X 軸ポイント間の間隔	2 バイト
Yoff	最初の Y 軸ポイントのオフセット	2 バイト
Ydist	Y 軸ポイント間の間隔	2 バイト
W1,1		
…	特性値	(n*m)*w バイト、カラムごと（Y のインデックス m の方が X のインデックス n より先に増加）
Wn,m		

表 13-6 ストレージシステム - 固定特性マップ

固定特性カーブ の例：

固定特性カーブ FKL1 には、軸ポイントが 5 つあります。軸ポイント間の間隔は 2、最初の軸ポイントのオフセットは 0 です。

C コードでは、エクスポートされるこの特性カーブは、コンポーネントのヘッダファイル <component>.h に以下のように宣言されます。

```
struct PIDT1_MOD_IMPL_FKL1_TYPE {
   uint16 xSize;
   sint16 xOffset;
   uint16 xDistance;
   sint16 values [5];
};
```

固定特性カーブの場合、定義と静的初期化は、以下のようになります。

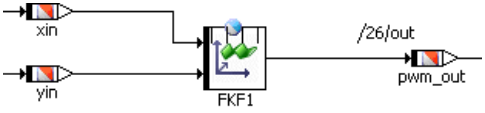
```
const struct PIDT1_MOD_IMPL_FKL1_TYPE FKL1 =
{
   5,
   0,
   2,
   {
      0, 1, 2, 3, 4
   }
};/*** FKL1 ***/
```

ローカルな固定特性カーブは、ネストされたデータ構造体の一部として定義され、初期化されます。

固定特性カーブおよびマップの場合、軸ポイントが固定的に等間隔で配置されているので、特別なサブルーチン（検索ルーチン）を使用しなくても、インデックスを直接計算することにより評価できます。この例では、C コードは以下の形式になります。

```
pwm_out = CharTableFixed1_getAt_s16s16(&FKL1,xin);
```

固定特性マップ の 例：



固定特性カーブ FKF1 には、x 軸上に 4 つ、y 軸上に 5 つの軸ポイントがあります。X 軸ポイントのオフセットは 2 で間隔も 2 です。Y 軸ポイントのオフセットは -3 で間隔は 3 です。

C コードでは、エクスポートされるこの特性マップは、コンポーネントのヘッダファイル - <component>.h 内で以下のように宣言されます。

```
struct PIDT1_MOD_IMPL_FKF1_TYPE {
   uint16 xSize;
   uint16 ySize;
   sint16 xOffset;
   sint16 yOffset;
   uint16 xDistance;
   uint16 yDistance;
   sint16 values [4 * 5];
};
```

グローバルの固定特性マップの定義と静的初期化は、以下のようになります。

```
const struct PIDT1_MOD_IMPL_FKF1_TYPE FKF1 =
{
   4,
   5,
   2,
   -3,
   2,
   3,
   {
      23, 23, 24, 25, 26,
      23, 15, 16, 17, 18,
      23, 7, 8, 9, 10,
      23, -1, 0, 1, 2
   }
};/*** FKF1 ***/
```

ローカルな固定特性マップは、ネストされたデータ構造体の一部として定義され、初期化されます。

C コード内での呼び出し部分は以下のようになります。

```
pwm_out =
CharTableFixed2_getAt_s16s16s16(&FKL1,xin,yin);
```

13.3.3 複合（ユーザー定義）オブジェクトのデータ構造体と初期化

クラス

ユーザーが定義する各クラスごとに 1 つの C 構造体が定義されます。そこにはクラスのインスタンス変数が、メモリクラスの順に含まれます。構造体の名前はクラスの C ネーム（クラス+インプリメンテーション名、13.3.1 項 「命名規則」を参照してください）です。各メモリクラスごとに、個別の構造体が生成されて参照されます。この構造体を通じて、例外なくすべてのインスタンス変数に直接アクセスできます。PID コントローラの例で使用したクラス PIDT1 の構造体定義は、以下のとおりです。

```
struct PIDT1_IMPL_RAM_SUBSTRUCT {
   sint16 temp_1;
   sint16 temp_2;
};
struct PIDT1_IMPL {
   struct PIDT1_IMPL_RAM_SUBSTRUCT *PIDT1_IMPL_RAM;
   sint16 memory_D_term;
   sint16 D_term;
   sint16 P_term;
   sint16 I_term;
};
```

ユーザー定義クラスのインスタンスは、C コード内に、PIDT1_IMPL クラス型の構造体を作成することにより作成されます。

通常、メソッド内のクラスインスタンス変数にアクセスする場合、構造体内に格納されている値に直接アクセスできます。しかし、同じクラスについて複数のインスタンスの存在が許されている場合は、それは不可能です。この場合には、レシーバ引数（*self* ポインタ）を追加して使用します。そうすれば、1 つのメソッド用の同じコードをそのクラスのすべてのインスタンスに使用することができます。ここでも、PIDT1 クラスを例にすると、compute メソッドの呼び出しは以下のようになります。

```
void PIDT1_IMPL_compute (const struct PIDT1_IMPL
   *self, sint16 in, uint16 K,
   uint16 Tv, uint16 Ti, uint16 Td) {
   sint32 _t1sint32;
   sint16 _t1sint16;

   ...(the rest of code for method "compute")

};
```

注記

各クラスごとにインスタンスが 1 つしか使われない場合に限り、レシーバは省略され、適切なコンポーネントがグローバルな分析により決定されます。
self ポインタの最適化は、クラスのインプリメンテンションエディタ内で無効に設定することができます。

プロトタイプクラス

- #define 文による外部宣言のカプセル化
- 関数のボディを含まない
- ローカルデータ構造体を含まない

サービスルーチン

- 関数のボディを含まない
- ローカルデータ構造体を含む

- 特殊な命名規則

モジュール

コードジェネレータは、モジュールをクラスのように扱います。また、モジュールには、モジュールの階層的エレメント構造に含まれるすべてのローカルエレメント用の階層データ構造体（「インスタンスツリー」と呼ばれます）のルートが含まれます。

各モジュールについて定義できるインスタンスは 1 つだけなので、モジュールのすべてのインスタンス変数およびパラメータに直接アクセスすることができます。クラスの場合とは異なり、self ポインタは必要ありません。プロセスは void-void 型関数として実装されます。`PIDT1_MOD` 内の `normal` プロセスは、以下のようになります（コードの大部分は省略されています）。

```
void PIDT1_MOD_IMPL_normal (void) {
   ...
   PIDT1_MOD_IMPL_TP_cmd_d =
      CharTable1_getAt_s16u16((CharTable1*)&
      (PIDT1_MOD_IMPL_Cmd_pct2deg),
      (sint16)_t1sint16);

   ...(the rest of code for process "normal")

};
```

論理テーブル

論理テーブルは、コード生成時にはクラスのように扱われます。論理テーブルにはパラメータが含まれないという点だけが異なります。

次の例（生成されるコードを簡略化したもの）で分かるように、論理テーブルに定義される論理依存関係が、論理演算子のシーケンスに変換されます。

```
sint8 CLASS_BOOLTAB_Y1
          (struct CLASS_BOOLTAB_Obj *self)
{
   return  ( (sint8) ( (
           ((!_X1) && _X2)
           || (_X1 && (!_X2)) )
           || ((_X1 && _X2)
              && _X3) ) );
}
```

条件テーブル

条件テーブルは内部的に ESDL クラスに変換されてからコード化されます。このテーブルの機能等については ASCET オンラインヘルプを参照してください。

13.3.4 ローカル変数とローカルパラメータ

ローカルエレメントは、データ構造体（170 ページの 13.3.2 項を参照してください）の一部として生成されます。生成されたコードにおいて、これらのエレメントへのアクセスは、それが属する構造体によって提供されるパス名を使用して行われます。コードを読みやすくするため、複雑な階層名には、プリプロセッサ定義文によって単純な名前が割り当てられます。

例：

```
#define _a ModuleA_IRAM.Class.a
#define _b ModuleA_IRAM.Class.b
...
void CLASS_IMPL_calc (void)
{
  _a = _b;
}
```

13.3.5 エクスポートされる変数とインポートされる変数

エクスポートされる変数やメッセージは、それらをエクスポートするモジュールのコード内に定義されるグローバル C 変数として実装されます。

ASCET では、エクスポートされる変数を、1 つのクラスのすべてのインスタンスに 1 回だけ存在するという意味で、「*クラス変数*」として扱っています。

インポートされる変数の実装は、それをインポートするモジュール用に生成されたコード内で、直接その変数のグローバル C 変数名を使用することによって行われます。このためには、この変数はインポートするモジュールのヘッダで `external` として宣言されます。したがって、その変数をインポートするモジュールのコードには、その変数をエクスポートするモジュールのコードへの直接参照が含まれることになり、その意味では完全なモジュラー化が実現されていないことになります。シミュレーションコードにあるような、リンケージのためのポインタ割り当ては存在しません。

注記

エクスポートされる変数について名前や型などの変更を行った場合には、ユーザーはエクスポート／インポート構造内のモデル全体を明示的にコード生成し直す必要があります。これは、コード生成の前に **Build → Touch → Recursive** を選択することによって行えます。

この場合も、エレメント名はプリプロセッサ定義文によって割り当てられます。

13.3.6 メソッドの宣言と呼び出し

メソッドの C ネームは、クラス識別子とメソッド名をアンダースコアでつないだものになります。

```
classIdentifier_methodName()
```

メソッドの仮引数の C ネームは、モデル名と一致しています。

```
returnType classIdentifier_methodName(argType1
  argName1, argType2 argName2)
```

引数や戻り値などのようなパラメータの渡し方は、そのパラメータの型によって異なります。

- **スカラパラメータ**と**ブールパラメータ**：所定の実装データ型の値 として直接渡されます。
- **特性カーブ**と**特性マップ**：その特性カーブ／マップの構造体へのポインタを渡します。
- **配列**と**マトリックス**：先頭エレメントへのポインタを渡します。
- **複合オブジェクト**：所定のクラス構造体へのポインタを渡します。

このルールは ASCET 全体に共通するもので、物理実験においても同様です。つまりスカラおよびブールパラメータは値渡しで、他の型はすべて参照渡しです。

複数のインスタンスが正しく処理されるように、`self` という C ネームのパラメータがパラメータリストの先頭位置に追加されます。そしてメソッド呼び出しのレシーバへのポインタ、またはそのインスタンス変数構造体へのポインタが、このパラメータで渡されます。このパラメータは、以下のような場合には必要ないので省略されます。

- モジュールのプロセス（モジュールのプロセスのインスタンスは 1 つしか生成されないため）
- インスタンス変数を持たないクラスのメソッド（レシーバは無意味であるため）

ただし、クラスのインプリメンテーションエディタ上で、このパラメータが強制的に生成されるように設定することができます。

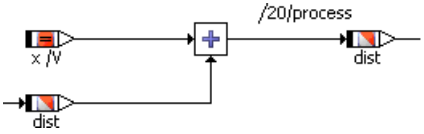
一例として、`PIDT1` クラスの `out` メソッドの形式は、以下のとおりです。

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

13.3.7 定数とリテラル

リテラルは、C コードにおいていわゆるリテラル（文字）として実装されます。リテラルは、インプリメンテーションの内容に従い、必要に応じて変換されます。定数の場合もこれと同じで、どちらもインプリメンテーションによる実装は行えません。リテラルと定数は、C コード内で `#define` を使用してコード化されます。

例：



この例で使用されている定数は、生成された C コードには以下のように実装されます。

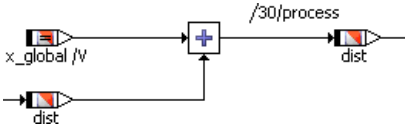
```
/**** constants defined by module MOD_IMPL ****/
#ifndef MOD_IMPL_X
#define MOD_IMPL_X 2.0
#endif
```

この例では、以下のようなコードが生成されます。

```
void MOD_IMPL_process (void) {
   dist = ((dist + (sint16)2));
   /* min=-10, max=10, hex=1phys+0 */
   /* end of process MOD_IMPL_process */
```

それに対して、グローバルエレメントとして生成される定数には、他のグローバルエレメントの命名規則に従って、プロジェクト名やインプリメンテーション名を含まない名前が付けられます。

例：



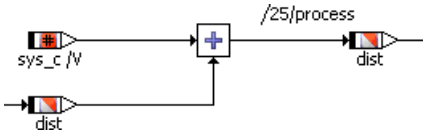

```
/**** エクスポートされる定数 ****/
#ifndef X_GLOBAL
#define X_GLOBAL 5.0
#endif
```

生成されるコードは、ローカル定数の場合と同様です。

```
void MOD_IMPL_process (void) {
   output = ((dist  +  (sint16)5));
   /* min=-20, max=20, hex=1phys+0 */
   /* end of process MOD_IMPL_process */
```

13.3.8 システム定数

システム定数は、C コード内で #define 文を使用して生成され、シンボリックに使用されます。これらはインプリメンテーションによる実装を行うことができます。下の例では、システム定数は量子化 1/2 で作成されました。



このシステム定数の定義として、以下のコードが生成されます。

```
#ifndef SYS_C_MOD_IMPL
#define SYS_C_MOD_IMPL 2
#endif
```

このシステム定数は、関数内で以下のようにシンボリックに使用されます。

```
void MOD_IMPL_process (void) {
   dist = ((((sint16)SYS_C_MOD_IMPL << 1) + dist));
   /* min=-10, max=10, hex=1phys+0 */
   /* プロセス MOD_IMPL_process の終わり */
```

グローバルエレメントとして生成されるシステム定数は、グローバル定数（189 ページ参照）と同様に、プロジェクト名やインプリメンテーション名を含まない名前が付けられます。名前には必ず大文字が使用され、ASCET モデル名は必要に応じて調整されます。

13.3.9 仮想パラメータ

仮想パラメータは、実際には ECU メモリに存在しないパラメータで、これを使用して、仮想でない実際の依存パラメータを定義することができます。このメカニズムをサポートする適合システム（INCA など）と併用すれば、仮想パラメータを適合することによって一度に複数のパラメータを操作することが可能です。

例：車輪の半径が、仮想パラメータとして定義されていると仮定します。そのため、これは ASCET モデルで直接使用することはできませんが、これに依存するパラメータとして定義された車輪の直径と円周は、モデル内の様々なロケーションで使用できます。

注記

仮想パラメータは ECU に実際には存在しないので、これを ASCET モデル内で直接使用することはできません。

仮想パラメータが使用される際は、memorySections.xml ファイル内に独立したメモリクラス VIRT_PARAM が定義されます。仮想パラメータとして定義されたパラメータは、すべてこのメモリクラスに割り当てられます。

クラスまたはモジュール用の C 構造体が生成される際、他のすべてのメモリクラスの場合と同様に、このメモリクラスについても仮想パラメータ用の独立したサブ構造体が生成されます（184 ページの「クラス」の項を参照してください）。通常のメモリクラス用のサブ構造体とは異なり、このサブ構造体はメイン構造体（MOD_IMPL）からは参照されません。

```
struct MOD_IMPL_IRAM_SUBSTRUCT {
   uint16 cont;
};
struct MOD_IMPL_VIRTPAR_SUBSTRUCT {
   uint16 radius;
};

struct MOD_IMPL {
   struct MOD_IMPL_IRAM_SUBSTRUCT *MOD_IMPL_IRAM;
   uint16 diameter;
};
```

このように特別な扱いを受けるのは、仮想パラメータ用のメモリ領域が物理的に ECU メモリの外側にアロケートされるためです。そのため、このメモリ領域をコードが参照することはできません。ECU 内に存在しないメモリ領域をメモリコンフィギュレーションで指定するだけで、これを実現することができます（3.3.5「メモリクラスに関する設定」の項も参照してください）。

13.3.10 依存パラメータ

生成されるコードを見る限り、依存パラメータは通常のパラメータと変わりありません。しかし、その初期値はユーザーが直接指定するのではなく、定義されている依存関係に従ってコードジェネレータが間接的に決定します。そしてこの依存関係は、それ以外にはコード内にまったく反映されません。この依存関係は適合システムが使用する ASAM-MCD-2MC ファイルに含まれます。

13.4 リアルタイム構成体

13.4.1 タスク

「タスク」は一連のプロセスで構成され、アプリケーションやオペレーティングシステムによって起動されます。タスクが起動されても、そのタスクがすぐに実行されるとは限りません。タスクの実行開始のタイミングは、オペレーティングシステムによってスケジューリングされます。タスクの属性には、オペレーティングモード、起動トリガ、優先度、スケジューリングモード等があります。タスクの実行が開始されると、そのタスクに割り当てられたプロセスが、指定の順番で実行されます。

OSEK オペレーティングシステムでは、タスクは C ソースコードにおいて TASK() マクロを使用してマークされます。このマクロの展開は各メーカーの OS によって異なり、これによって OS がタスクボディを正しく呼び出すことができます。

ASCET と ASCET-SE は、以下のタスクスケジューリングモードをサポートしています。

- Alarm：アラームタスク
- Interrupt：割り込みタスク
- Software：ソフトウェアタスク
- Init：初期化タスク

初期化タスクは、各アプリケーションモードごとに 1 つだけ割り当てることができます。

13.4.2 プロセス

「プロセス」は 1 つの機能単位で、複数のプロセスが並行して実行されます。各プロセスはタスクに割り当てられ、タスクは一連のプロセスから成り立っています。

プロセスには引数と戻り値はありません。すべてのターゲット（ANSI C を含む）に対して、プロセスは以下の例のような `void/void` 関数としてコード化されます。

```
void MOD_IMPL_process (void) {
   CL_IMPL_calc();
}
```

この例のプロセスは、クラス `CL` のメソッド `calc` の呼び出しのみを実行します。

13.4.3 メッセージ

「メッセージ」は、リアルタイムに実行されるプログラムにおいて常にデータの整合性を保つために使用されるものです。通常のグローバル変数を使用すると、たとえばあるプロセスが変数を使用している時に、それより優先度の高いプロセスが同じ変数にアクセスしてその変数の内容を変えてしまう可能性があり、データの整合性が損なわれる危険性があります。

メッセージを使用する場合、グローバルな分析によって必要と認められたすべてのケースについてメッセージのコピーが生成されます。これについてユーザーが介入する必要はありません。

ただし、ユーザーは各メッセージが 1 つのプロセスからしか送信されてないことを確認する必要があります。リアルタイム環境で 1 つのメッセージに対して複数のプロセスが書き込みを行った場合、レシーバが受け取るメッセージがどのセンダからのものなのかを判断する方法がないためです。

注記

メッセージコピーの最適化機能は OSEK オペレーティングシステムのタスク優先度機構に基づいて行われます。そのため、ECU 上で実行されるすべてのタスクとそのプロパティが ASCET において正しく把握されている必要があります。それが不可能な場合（つまり OSEK 準拠でないオペレーティングシステムを使用する場合や、ハンドコーディングされた外部ソースからメッセージがアクセスされる場合など）、メッセージコピーの最適化が適切に行われない可能性があり、生成されたコードが危険な挙動を示す恐れがあります。このような場合は、メッセージ最適化機能は使用しないでください。

デフォルト状態において行われるメッセージコピーの最適化がすべてのアプリケーションに有効であるとは限らないため、メッセージハンドリングについては詳細なユーザー設定が必要です。

コンパイル時にメッセージコピーバリアントを選択する

`codegen[_*].ini` ファイルの内容を変更し、サポートされているコピーバリアントがすべて C コードに出力されるようにすることができます（`modularMessageUse=true`）。この際、各バリアントのコードはプリコンパイラ文 `#if ... #endif` で区切られて出力されます。

これにより、コード生成時ではなくコンパイル時にメッセージコピーのバリアントを選択することが可能となります。

メッセージコピーバリアントの選択は C マクロ `__MESSAGES` の定義で実現されます。このマクロは、ユーザー定義のヘッダファイル `message_scheme.h` 内に含めるか、またはコンパイラオプション（`project_settings.mk` 内の `PROJECT_DEFINES` という Make 変数）で定義します。

以下の 4 つのオプションがあります。

- メッセージコピーの最適化（デフォルト）

  メッセージコピーがオペレーティングシステムのタスク優先度機構に基づいて最適化されます。このバリアントは以下の C マクロ定義で有効になります。

  ```
  #define __MESSAGES __OPT_COPY
  ```

  前提条件：このメッセージコピーバリアントを使用する場合、ASCET が、メッセージを使用する各タスクと ISR の優先度を把握している必要があります。この情報が不完全な場合、メッセージコピーとして生成されるコードが不正なものになる可能性があり、それによってランタイムにデータが破壊されてしまう危険性があります。

- メッセージコピーを生成しない：

  メッセージはグローバル変数として扱われ、コピーは作成されません。このバリアントは以下の C マクロ定義で有効になります。

  ```
  #define __MESSAGES __NO_COPY
  ```

注記

モジュール内のメソッドからアクセスされるメッセージの場合、`__OPT_COPY` と `NO_COPY` しか使用できません。他の最適化はサポートされていません。

- メッセージ最適化を行わない（常にコピーを作成）：

  メッセージは常にコピーされます。このバリアントは以下の C マクロ定義で有効になります。

  ```
  #define __MESSAGES __NON_OPT_COPY
  ```

  ここでは最適化は行われません。このバリアントは最も柔軟性があり、ASCET 内で OS コンフィギュレーションが完全に設定されていなくても使用可能です。

- OSEK COM を使用：

  OSEK COM を使用してメッセージ通信を行います。これはオペレーティングシステムが OSEK-COM メッセージをサポートしている場合にのみ可能です。このバリアントは以下の C マクロ定義で有効になります。

  ```
  #define __MESSAGES __OSEK_COM
  ```

  この場合、OSEK オペレーティングシステムによってすべてのメッセージとそのコピーが定義されます。メッセージの現在の値へのアクセスは、各プロセスの前後で OSEK-COM API[1] の `ReceiveMessage()` および `SendMessage()` を使用して行われます。

コード生成時にメッセージコピーバリアントを選択する

1 つのメッセージコピーバリアントのみが C コードに出力されるようにするには、`codegen[_*].ini` 内の `modularMessageUse` の値を `false` に設定し、さらに `messageUsageVariant` でメッセージコピーバリアントを指定します（詳細は `codegen_ecco.ini` ファイル内のコメントを参照してください）。この場合、指定されたメッセージコピーバリアントのコードしか生成されないので、コンパイラマクロ `__MESSAGES` の定義は不要です。

[1] OSEK Communication Specification：http://www.osek-vdx.org/ を参照してください。

13.4.4 リソース

「リソース」は、OSEK オペレーティングシステムの GetResource ／ ReleaseResource メカニズムによって保護されています。生成されたコードは、他のオペレーティングシステムで使用したり、ハンドコーディングされたソースコードと共に使用することが可能で、制約条件はありません。

RTA-OSEK は OSEK のリソース管理機能である RES_SCHEDULER（OSEK の仕様書を参照してください）をサポートしています。リソースのシーリング優先度は OS のスケジューリング優先度と同じです。ASCET においては、このリソースは C コードからのみ使用できます。このためには、まず **Resource** ボタン（  ）をクリックして C コードモジュール内にリソースを定義し、そのリソースに名前を付けます（例：RES_SHEDULER）。

これにより、C コードエディタから ASCET マクロ経由でリソースにアクセスすることが可能となります。以下にマクロの使用例を示します。

```
ASD_RESERVE(RES_SCHEDULER);
   /* user code */
   ...
ASD_RELEASE(RES_SCHEDULER)
```

上記のマクロから、以下のようなコードが生成されます。

```
...
DeclareResource(RES_SCHEDULER);
...
void process(void)
   {
   ...
   GetResource(RES_SCHEDULER);
   /* user code */
   ...
   ReleaseResource(RES_SCHEDULER);
   ...
}
```

13.4.5 アプリケーションモード

「アプリケーションモード」は、システム全体が状況に応じて異なるモードで稼動できるようにするためのものです。これによって、まったく異なる機能を実行する複数のシステムモードを、柔軟かつ容易に設計し管理することが可能となります。モードの例としては、スタートアップ、通常の運転モード、シャットダウン、診断、EEPROM プログラミング、といったものがあげられます。各アプリケーションモードごとに、タスクの構成や優先順位、タイマコンフィギュレーション等を独自に設定することができます。

ASCET は、OSEK OS のアプリケーションモードをサポートします。使用するアプリケーションモードは OS の StartOS() に引数として渡されます。モードの制御や切り替えは、ASCET の範囲外でおこなわれます。

ASCET を RTA-OSEK V5.x と統合する場合は、OS を異なるアプリケーションモードで再起動することができます。ただしこのような機能は OSEK OS の規格外のものであるため、他の OSEK OS ではサポートされていない可能性があります。

14 ASCET-SE の内部情報

本章では ASCET-SE コードジェネレータの内部構造や動作原理などについて概説します。この内容は参考情報として提供するもので、ASCET-SE を使用する際に必須となるものではありません。

14.1 コードジェネレータの構造

コード生成サブシステムはレイヤ構造になっています。各レイヤが行う処理について、以降の項で簡単に説明します。

14.1.1 フロントエンドコンバータ

各機能記述形式（ブロックダイアグラム、ステートマシン、ESDL）ごとに、専用の「フロントエンドコンバータ」が用意されています。ここでは、機能記述の構文解析が行われます。たとえば、グラフィック入力においては、ブロック上のポートがすべて接続されているかどうかが調べられ、ESDL の場合には、パーサーが使用されます。記述が構文的に正しければ、フロントエンドがこれらのファイルを MDL フォーマット に変換します。

C コードモジュールは、ユーザーがインプリメンテーションレイヤに対して直接作業を行う、例外的な記述方法です。この方法では、ターゲットごとに手作業で C コードを入力します。このような特殊な位置付けのため、C コードモジュールはコード生成にはあまり影響しません。C コードモジュールについては本書で後述します。

14.1.2 MDL と MDL ビルダ

MDL（**M**ethod **D**efinition **L**anguage: メソッド定義言語 ）は、各機能記述形式で作成された物理モデルを一律の形式で実装するために内部的に使用される中間フォーマットで、ユーザーが見ることはできません。MDL はオブジェクトベースのビューを提供し、クラスおよびメソッドを宣言して定義します。さらに、MDL にはリアルタイム動作を実装するエレメント、つまり、プロセス、メッセージなどが含まれます。アルゴリズムは、ここではまだターゲットに依存しない物理表現のままで、ユーザー指定による量子化（例、補正係数によるリスケーリング）は、後のコード生成プロセスで行われます。しかし、すべてのエレメント（変数、メソッド引数など）は、このフォーマットの中で、インプリメンテーション情報とともに詳しく記述されます。

セマンティック解析

MDL ビルダではまず全般的なセマンティック解析が行われ、その後に実装コード生成のための分析が行われます。演算のスタックベースの動作モードに従って数式が意味的に分析され、さらに以下のチェックが行われます。

- 非線形の変換式が使用されていないか。使用されている場合にはエラーメッセージが出力されます。

- 浮動小数点と実数が不当に混用されていないか。不当に混用されている場合にはエラーメッセージが出力されます。
- ユーザーが指定したインプリメンテーションが、最大ビット幅を超えていないか。超えている場合にはエラーメッセージが出力されます。
- 除算の場合、分母の物理範囲に 0 が含まれていないか。含まれている場合にはエラーメッセージが出力されます。
- 代入の場合、代入する式の物理範囲が代入される変数の物理範囲に適合しているか。適合していない場合には、警告メッセージが出力されます。この場合、リミッタの生成を有効にすることをお勧めします。

最適化データの収集

セマンティック解析の後、MDL ツリーのセットアップ時に追加情報（スケーリング係数、値の範囲など）についての計算が行われます。このデータは、算術演算機構の変換を最適化するために使用されるもので、MDL ツリー内の各ノードと共に格納されます。

中間結果の物理範囲の計算：すべての変数、パラメータ、メソッド引数、および戻り値の範囲はユーザーによって指定されますが、数式内で使用される中間結果については、インターバル演算を使用して値の範囲を計算する必要があります。

最適化データの計算：生成されるコードの精度と効率のバランスをとるためには、中間結果の量子化をうまく選択することが重要です。数式内の各演算ごとに最適なスケールのリストが作成されますが、これらは、再量子化を行う演算の回数を最小限にすることを前提としています。最適化データは、コード生成フェーズにおいて決定要素として機能します。

14.1.3 コードジェネレータ

コードジェネレータは、MDL のオブジェクト型構造を関数型構造にマッピングするもので、このレイヤには、MDL よりも C によく似た単純な言語機能が含まれています。

コードジェネレータも、やはりターゲットには依存しません。しかし、ECU ターゲットの場合に限り、このレイヤにおいて特別な最適化が行われるので、実験ターゲットと ECU ターゲットは区別されます。

ASCET では 4 種類のコードジェネレータがあり、それらを選択することができます。各ジェネレータの主な違いは、算術演算機構の変換方法にあります。これら 4 種類のコードジェネレータは統合型開発工程の 4 つの実験フェーズをサポートするものです。初めの 3 つのフェーズは実験ターゲットを使用して行われ、最後のフェーズは実際のマイクロコントローラターゲット（ECU）を使用して行われます。

- Physical Experiment（物理実験）は、物理エレメントおよび浮動小数点演算（量子化なし）を生成します。インプリメンテーション情報は必要ありません。

- Quantized Physical Experiment（量子化物理実験）は、量子化を伴う物理シミュレーションを行います。浮動小数点演算が使用されますが、各エレメントについても値の範囲と量子化を指定することができます。インプリメンテーションは部分的に指定され、実行時に変更することもできます。
- Implementation Experiment（実装実験）は、インプリメンテーションレイヤ上でシミュレーションを行います。すべてのインプリメンテーション情報（データ型、変換式など）を指定する必要があります（後のコントローラ実装 で必要なため）。アルゴリズムは、ターゲットシステムの固定小数点演算機構に自動的に変換されます。
- Object Based Controller Implementation（オブジェクトベースのコントローラ実装）は、ECU 用にさらに最適化を行います（例：インポートされるエレメントは直接参照されます）。命名規則は変更され、その名前がデータベース ID の代わりに使用されます。固定小数点演算のコード化方法は実装実験の場合と同じなので、それと同じ挙動が得られます。

  各ターゲット用 ASCET-SE には、オブジェクトベースのコントローラ実装の機能のみが盛り込まれています。つまり、モデル内で記述されたオブジェクト構造がそのままコントローラソフトウェアにマッピングされます。

物理実験においては、プロジェクト環境なしに ASCET モジュールのコードを生成し、シミュレーションを行うことができますが、それ以外のコードジェネレータを使用する場合には、モジュールは必ずプロジェクトに組み込まれている必要があります。この場合、プロジェクトなしにはインプリメンテーション情報にアクセスする方法がなく、インポートされるエレメントの変換式やインプリメンテーション情報が得られません。

エクスパンダ

エクスパンダはターゲットに依存しない中間コード（`*.pl` ファイル）を作成し、このコードが、最終的なターゲット固有の C コードの生成に使用されます。エクスパンダはソフトウェアの基本となる要素を作成します。これはつまり、MDL の物理的な数式を、後に C コード内で実際に使用される計算に変換する、ということです。この変換は、コードジェネレータにより、標準化された内部インターフェースを使用して指示されるので、ユーザーは、コードジェネレータとは無関係にエクスパンダを選択することができます。

MDL ビルダとは異なり、エクスパンダは関数指向です。MDL ツリーのノードに対応する個々の演算用の中間コードを生成するために、MDL ツリーが上から下へと繰り返し走査されます。まず、ローカルコンテクスト、つまりその演算だけについての基本原則に従って、個々の演算についてのコード生成が行われます。その後、値のインターバルとセマンティック解析時に計算された最適化データを使用して、それぞれの数式全体について、最適なコードが生成されます。

エクスパンダはインプリメンテーションレイヤで稼働します。つまり、物理表現ではなく C データ型を使用します。

ECCO

最後に、エクスパンダが生成した中間コードを ECCO が最終的な C コードに変換します。

14.2 コード管理

ここで説明する*コード管理システム* は、コードジェネレータの構成要素ではありません。コードジェネレータの処理を支援し、自動生成されたコードやマニュアルコーディングされたコードを安全かつ永久的に保存するためのものです。

14.2.1 Make メカニズム

Make メカニズムは、整合性のある最新のコードバージョンを作成します。コードはモジュール構造になっているので、モデルチェンジを行った後はモジュール単位についてコードを生成し直してコンパイルすればよいため、処理時間も最小限におさえることができます。Make メカニズムは ASCET データモデルに基づいて依存関係ネットワークを生成します。このネットワーク内の各モジュールのタイムスタンプが分析され、どのモジュールのコードを再生成する必要があるかが判断されます。

しかし残念ながら、タイムスタンプだけでは再生成が必要かどうかを判断できない場合があります。タイムスタンプが変わっていたとしても、実際に再生成の必要があるかどうかを自動的に認識することはできません。

Make メカニズムは*物理実験* コード生成用に最適化されているので、処理にかかる時間を短くすることが重要視されています。そのため、場合によっては必要以上に多くのモジュールについて再生成が行われたり、あるいはまれに、必要なモジュールが再生成されなかったりすることがあります。このような状況を防ぐため、モデル構造体に大幅な修正（変数／メソッドの作成／削除、エクスポート／インポートされる変数の変更など）を加えた後は、コード生成を行う前に、**Build → Touch → Recursive** を実行してください。

14.2.2 コードマネージャ

コードマネージャは、コード生成およびストレージ用の一元的なインターフェースとして、内部的に機能します。このインターフェースを通じて他のすべてのサブシステムが、コード生成、Make メカニズム、およびコードストレージについての要求をやりとりします。このインターフェースによって制御される機能の例を以下に紹介します。

- コンポーネント用のソースコードを生成する（**Build → Generate Code** を実行します）。
- 実行コードを生成する（**Build → Build** を選択するとコードが生成され、コンパイルされ、リンクされて ASCET データベースに格納されます）。
- コードをターゲットにロードする（たとえば、**Build → Experiment** を実行します）。

- ソースコードをファイルに保存する（**File → Export → Generated Code → *** を実行します。）このオプションは、コードがすでにデータベースに格納されている場合に限り使用できます。
- “Touch”を実行する（**Build → Touch → *** を実行すると、Make メカニズムのためのタイムスタンプが更新されます）。

コード管理により、ソフトウェアが生成したコードやマニュアルコーディングされたコードが、ASCET データベースに安全にしかも永久的に格納されます。ASCET の各コンポーネント（モジュール、ステートマシン、クラスなど）について、複数のコードバリエーションをそれぞれ独立したエンティティとして、一度にデータベースに格納することができます。

基本的に、コードバリエーションは、*Code Generation Settings* で選択されたターゲット、*コードジェネレータ*、およびエクスパンダの種類によって決まります。

注記

ターゲットとエクスパンダは相互の関連性によって選択されるので、実際にはターゲットとコードジェネレータだけでコードバリエーションを識別することができます。

これらのオプション設定のいずれかが変更されると、新しいバリエーションが作成され、別のコードバリエーションとして格納されます。また他のコード生成オプションのいずれか（**Protected Division**、**Generate Limiters** など）が変更された場合には、既存のバリエーションのコードに変更後のコードが上書きされます。

複数のインプリメンテーションを使用できないコードジェネレータ（“Physical Experiment”など）が選択された場合は、システムが生成したコードやマニュアルコーディングされたコードがコンポーネントとともに格納されます。

注記

現在の ASCET では、上記の条件に当てはまるコードジェネレータは“Physical Experiment”だけです。

複数のインプリメンテーションを使用できるコードジェネレータが選択されると、システムが生成したコードとマニュアルコーディングされたコードは、別のロケーションに格納されます。インプリメンテーション情報を使用してコードを生成するために必要なデータ（変換式、グローバル変数など）はプロジェクト内に存在するため、生成されたコードはプロジェクトとともに格納され、またマニュアルコーディングされたコードは、個々のインプリメンテーションのコンポーネントとともに格納されます。

14.3 CPR（コード生成ルール）のディレクトリ構造

CPR（**C**ode **P**roduction **R**ules：コード生成ルール）は、以下の構造を持つディレクトリに格納された Perl プログラムです。

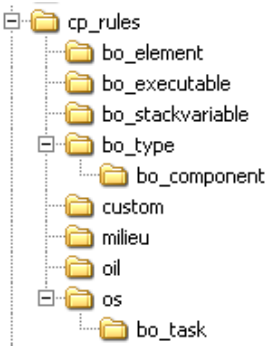


図 14-1 CPR のディレクトリ構造

`cp_rules`（生成用ベース）	コード生成についての全般的なルール
`bo_*`	各オブジェクト（エレメント、型、コンポーネント、実行ファイル）のために生成されたコードを調整するためのルール
`milieu`（ターゲット固有）	ECCO が生成したコードを、ターゲット、オペレーティングシステムのコンフィギュレーション、CPU 周波数、プリスケーラに適応させるためのルール
`custom`（ユーザー定義）	コード生成についてユーザーが定義したルール
`oil`	OIL のコード生成ルール
`os`	OS のコード生成ルール

CPR レベルで再利用するための技術上の前提条件は、以下のような Perl の特徴に基づいています。

関数（マクロ、CPR）を検索するために、Perl は実行開始時に渡されるディレクトリリストを参照します。この検索は、マッチする符号を持つ**最初の**関数が見つかると終了し、またマッチする関数が見つからない場合にはエラーメッセージを出力して終了します。

ASCET-SE に含まれている各コンポーネントの CPR が個別のディレクトリに格納されていて、さらに Perl インタープリタに対して Make ファイル内で「特別な CPR から一般的な CPR へ」という順序に従うディレクトリリストが提供されている場合には、ユーザー固有の CPR で標準 CPR を上書きすることができます。

15 ASCET-SE の制約条件

本章では、コード生成の構造上存在する制約と、それらを回避する方法について説明します。既知のエラーも紹介します。

15.1 一般的な制約

15.1.1 インターバル演算

数式内の中間結果の範囲は、インターバル演算で計算されます。インターバル演算では、機能的な関連を認識することはできません。中間結果は常に、値の範囲が互いに一貫性のない入力変数から計算されるため、ワード長が不必要に長くなったり、オーバーフローが誤って検出されたり、数値精度が必要以上に損なわれたりしてしまう可能性があります。またあるいは、代入時において、不必要なリミッタ用のコードが生成されてしまう可能性もあります。

例：x ∈ [9.0,99.0] の場合、式 x/(x+1) については x が式の分子と分母の両方に含まれているので、式の結果のインターバルは、[0.9,0.99] です。しかし、インターバルアルゴリズムが先に分母のインターバル（x+1、つまり [10.0,100.0]）を計算してしまうと、x/(x+1) のインターバルは実際のインターバルをこの分母のインターバルで割ることにより求められる（[9.0,99.0]/[10.0,100.0]=[0.09,9.9]）ため、実際のインターバルよりも 2 階数も大きな範囲となってしまいます。

さらに、この式がこれより大きな式、たとえば (x/(x+1))*y のような式の一部分になっている場合、インターバルが大きくなりすぎたためにオーバーフローが発生する可能性があり、それを防ぐための右シフトが中間結果に（たぶん y にも）発生してしまう可能性があります。この結果、システムの計算性能が著しく悪化する可能性があります。

このような影響を防ぐため、ファンクションの既知の依存性に基いて変数を追加し、中間結果の範囲を明示的に指定するようにしてください。

15.1.2 リテラルの量子化が行われない

まれに、リテラルの量子化がそのコンテクストに基づいて自動的に設定され、そのためにリテラルが非常に粗く実装されてしまう場合があります。しかしこのようなケースは非常にまれで、一般に非有理のリテラル値だけに発生します。

パラメータ、または実装されたテンポラリ変数を使用すると、この問題の改善に役立ちます。

15.1.3 ASCET の直接アクセスと特性マップ

ネストされたクラス内で特性マップへの直接アクセスを行うと、動作としては正しくても、効率的でないコードが生成される可能性があります。

以下のように呼び出される補間ルーチン内で特性カーブまたはマップを取得するには、単純な式を使用する必要があります。

```
CharTable2_getAt_s8s8s8(ASD_CHTBL_PTR(Two_D),
(sint8)1,(sint8)1);
```

これが守られていない場合、最適化オプション **Optimize Direct Access Methods *** がオフになっていると、生成されるコードは「不正」ではなくても「非効率的」なものとなってしまいます。

例：

```
ASD_INPL_CharTable2_getAt_s8s8s8(                    ↵
   INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(        ↵
   (struct MIDDLE_IMPL *)&self->Middle))).xSize,     ↵
   (const sint8 *)                                   ↵
   (INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(       ↵
   (struct MIDDLE_IMPL *)&self->Middle))).xDist),    ↵
   INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(        ↵
   (struct MIDDLE_IMPL *)&self->Middle))).ySize,     ↵
   (const sint8 *)                                   ↵
   (INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(       ↵
   (struct MIDDLE_IMPL *)&self->Middle))).yDist),    ↵
   (const sint8 *)                                   ↵
   (INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(       ↵
   (struct MIDDLE_IMPL *)&self->Middle))).values),   ↵
   (sint8)1, (sint8)1) ;
```

回避策（パフォーマンスの最適化）：特性カーブ／マップを参照する getAt メソッドが呼び出される際、以下のメソッド呼び出しにより取得される一時変数をモデル内で使用する、という方法があります。

```
res = Middle.Inner().Two_D().getAt(1,1);
```

ここでは一時変数への参照が代入されから、その一時変数の getAt メソッドが呼び出されます。

```
_Two_D_REF = Middle.Inner().Two_D();
res = _Two_D_REF.getAt(1,1);
```

その結果、より効率的なコードが生成されます。

```
_Two_D_REF = INNER_IMPL_getTwo_D(                    ↵
   (MIDDLE_IMPL_getInner((struct MIDDLE_IMPL *)      ↵
   &self->Middle)) );

ASD_INPL_CharTable2_getAt_s8s8s8(                    ↵
   _Two_D_REF->xSize,(const sint8 *)                 ↵
   _Two_D_REF->xDist,                                ↵
   _Two_D_REF->ySize,(const sint8 *)                 ↵
```

```
_Two_D_REF->yDist,(const sint8 *)                ↵
_Two_D_REF->values,                              ↵
(sint8)1, (sint8)1);
```

15.2 ASCET-SE の使用に関する制約条件

15.2.1 特性カーブおよびマップの入力

制約：特性カーブおよびマップの入力は静的変数（RAM に保存される変数）でなければなりません。ASCET-SE ソフトウェアアーキテクチャでは、これはエクスポートまたはインポートされるクラス変数やモジュールのローカルインスタンス変数である必要があります。メソッド引数やメソッドローカル変数、およびクラスのインスタンス変数は使用できません。

原因：近年の適合システムや ASAM-MCD-2MC フォーマットにおいては、たとえば現在のブレイクポイントを表示するような場合、個々のカーブやマップごとに、スタックメモリではなく静的な RAM 変数に格納する入力変数の名前とメモリアドレスを指定しなければなりません。RAM 変数でなく、またグローバルでも可視的でもない式または変数を使用すると、特性カーブの適合がまったく行えないか、あるいは制限付きでしか行えなくなります。

確認方法：上記の制約条件に該当する場合、コード生成において、パラメータが適合不可能であることを示す警告メッセージが出力されます。

回避策：必要に応じて、適切な静的中間変数（RAM 変数）をモデル内に挿入してから、特性カーブに入力してください。

15.2.2 軸ポイントの検索と補間を個別に行えない

制約：通常の個別の特性カーブおよびマップについて、軸ポイントを検索するプロセスと補間を行うプロセスを別々に行うことはできません。つまり、`search` メソッドと `interpolate` メソッド（特性カーブおよびマップの拡張インターフェース）の両方を使用することはできません。

原因：特性カーブオブジェクトは、コントローラの静的メモリ領域（ROM/FLASH）に格納されているので、さらにストレージスペースを持たせることはできません。軸ポイントの検索を別に行えるようにするには、RAM 上に別の変数を常に追加作成し、そこに検索結果を格納するようにしなければなりません。これは、効率上の理由で行われていません。

確認方法：上記に該当する場合には、コード生成時に障害レポートが表示されます。

15.2.3 補間メソッドを選択できない

制約：シミュレーション時のように特性カーブおよびマップごとに異なる補間メソッドや外挿メソッド（丸め、線形）を選択することはできません。補間および外挿の動作は、使用される補間ルーチンにより、グローバルに決定されます。

原因：補間のタイプを各特性カーブごとに個別に選択するようにすると、補間のタイプごとに別のルーチンを提供する（つまりコードの量が増えます）か、汎用ルーチンを使用してそれを呼び出す時に補間タイプをスイッチとして渡す（つまりコードの量が増え、実行時間が長くなります）ことが必要になります。そのため、効率上の理由で、この機能はコントローラには備わっていません。

確認方法：なし。特性カーブの型とデータ型のそれぞれの組み合わせ用に提供されている（またはユーザーが供給した）補間ルーチンが必ず呼び出されます。

15.2.4 コンポーネント名の一意性

制約：コンポーネントの名前は、プロジェクト内でユニークでなければなりません。さらに、プロジェクトの名前も、その中にあるどのコンポーネントの名前とも違うものでなければなりません。

原因：コントローラコード内のファンクションおよび変数の C ネームは、それを見れば内容を想像できるようなものでなければならないので、コンポーネントの名前を含める必要があります。プロジェクト内の 2 つのコンポーネントの名前が同じだと、コード内で名前の衝突（コンパイラ／リンカエラー）が起こる可能性があります。

確認方法：上記に該当する場合、Make メカニズムで、障害レポートが表示されます。

15.2.5 コントローラおよび固定小数点演算の Make メカニズム

制約：Make メカニズムでは、*実装実験やコントローラ実装* において、式の変更などによって生じる、プロジェクト全体または個々の部分の再生成を必要とするような依存関係を、*すべて*認識できるわけではありません。それらをすべて認識しようとすると、プロジェクト全体を分析しなければならなくなり、完全な再生成と同じくらい長い時間がかかってしまいます。

原因：*コントローラ実装用の* Make メカニズムは、物理シミュレーションの場合と同じように機能し、モデル内の変更によるグローバルな影響については、認識されない部分もあります。

回避策：グローバルな影響を伴う変更については、ユーザーは **Component → Touch → Recursive** を選択してプロジェクトの完全な再生成を強制的に実行する必要があります。このように、コードの一貫性はユーザーが管理することになります。

15.3 ASCET-SE コード生成における既知の問題点

ASCET-SE について、現時点において以下のようなエラーが認識されています。ここでは、ASCET-SE を使用するコントローラコード生成に特に関連するエラーだけを紹介します。ASCET に関連する一般的な制約は、ここでは紹介しません。

15.3.1 ASCET 終了後の実行コード構築

Build → Build を実行すると、実行形式のプログラムがテンポラリディレクトリ（`..¥ascet5.2¥cgen`）に生成され、ASCET データベースに格納されます。そして ASCET の **Keep files in Code Generation Directory** オプション（ASCET オンラインヘルプを参照してください）がオフになっていると、`.¥cgen¥` 以下のディレクリの内容は、ASCET が終了するたびに削除されます。またこのオプションを変更しても、現在のセッションについては反映されませんので注意してください。

なお `.¥cgen¥` のコードが削除された後も実行コードはデータベース内に残っていますが、それを読みとる方法はありません。現在のコードが必要な場合は、ASCET を再度起動して実行形式のプログラムを作成し直す前に **Build → Touch → Flat** を実行して、強制的にコンポーネントを再コンパイルしてリンクし直します。

16 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	Phone:	+49 711 8 96 61-0
70469 Stuttgart	Fax:	+49 711 8 96 61-105
Germany	WWW:	http://www.etas.com/

日本支社

イータス株式会社

〒 220-6217	Phone:	(045) 222-0900
神奈川県横浜市西区	Fax:	(045) 222-0956
みなとみらい 2-3-5	E-mail:	sales.jp@etas.com
クイーンズタワー C 17F	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:	http://www.etas.com/ja/contact.php
技術サポート	WWW:	http://www.etas.com/ja/hotlines.php

索引

へ

ほ

ま

め

も

ゆ

り

れ

ろ

わ