Embedded Coder
量産／組み込みCコード生成 チップス集

MathWorks Japan
アプリケーションエンジニアリング部
シニアアプリケーションエンジニア
山本　順久

はじめに

はじめに

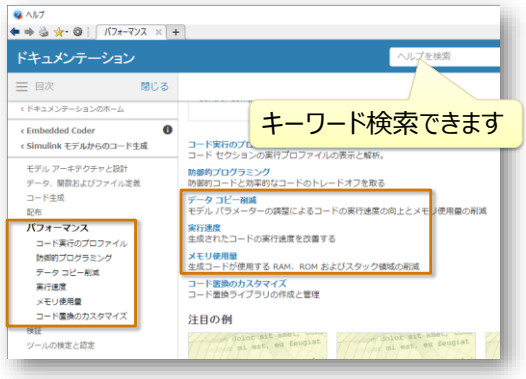
- 本資料は、Embedded Coderを用いて量産 / 組み込み用Cコードを生成したい方を対象に、各種機能や例題を紹介するチップス集となっています。
- 前提として下記の知識・技術が必要です。
  - MATLAB/Simulink/Stateflowの基本的な使い方やモデリングスキル
  - Embedded Coderの基本的な機能・使い方
  - C言語（組み込みソフトやマイコン知識があればベター）
- R2018aを前提に記述されています。
  - 他バージョンでは機能、UI、生成コードが異なる可能性があります。
- 文中のwebコマンドはMATLABヘルプで該当ドキュメントを表示するMATLABコマンドです。
  - 例：Embedded Coderドキュメントトップページ
    `>> web(fullfile(docroot, 'ecoder/index.html'))`
- サンプルモデルのコード生成時にエラーがでる際は、全モデルをいったんクローズ、コード生成フォルダおよびslprjフォルダを削除してから再度試してみてください。
- 本資料に関するご質問・ご要望は当社技術サポートまたは担当営業までご連絡願います。

3

MATLABヘルプとEmbedded Coderチュートリアル

初心者の方はまずはチュートリアルで
基本機能の学習・把握をおススメします

MATLABヘルプに代表的な効率化
チップスが紹介されています



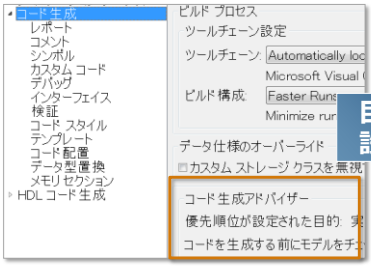
MATLABヘルプにもEmbedded Coder
入門の例題が掲載されています

4

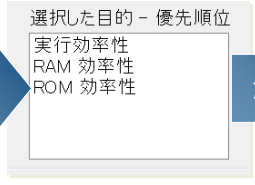
コード生成に向けたモデルコンフィグ・ブロックの基本設定

- 初心者の方はコード生成アドバイザーによるモデルチェックをおススメします。
  - モデルコンフィグやブロックに関するコード生成用推奨値を確認、反映することができます。
- 最終的にはコード生成用の設定値を共通化して運用することをおススメします。

モデルコンフィグ

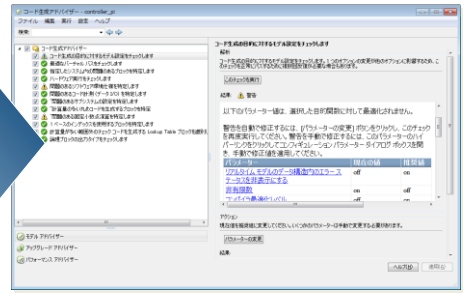


目的設定



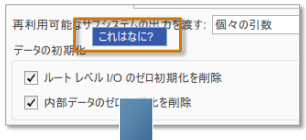
解析

チェック＆修正

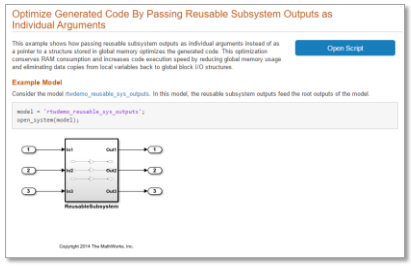


※必ずしも全チェックをパスする必要はありません
（推奨通りに設定できないケースもあるため）

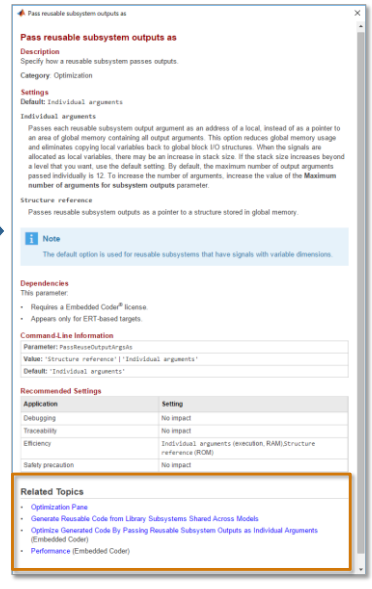
モデルコンフィグの効果・詳細機能を確認したいときは



調べたいコンフィグを
右クリック



コンフィグ
例題



コンフィグ
詳細情報

コンポーネント設計

モデルのコンポーネント設計とコード生成

- 下記の点でコンポーネント設計が重要なのはご存知のことだと思います。
  - モデリング効率性、モデル可読性・理解度向上、開発資産の再利用性・保守性、テスト容易性、差分・派生開発のしやすさ、バージョン管理、等
  - Simulinkマニュアルの「コンポーネント化のガイドライン」では、サブシステム・ライブラリ・モデル参照の特徴・用途を紹介しています。
    >> web(fullfile(docroot, 'simulink/ug/model-architecture-guidelines.html'))
    （R2014bからはSimulink関数というブロックも登場しました。AUTOSARサーバ・クライアントモデル等で使用します）
- コンポーネント設計は、下記の点でモデル生成コードにも影響します。
  - コードの階層構造（関数ツリー）
  - 関数化によるコードの再利用
  - コードのファイル分割
  - グローバル変数の定義ファイル

サブシステム vs. ライブラリ vs. モデル参照

コンポーネント化の手法	その手法が特に適しているモデル化の目的
サブシステム **サブ関数** **コードスニペット**	• 整理する階層を追加し、モデルを視覚的に簡略化します。 • コンテキスト依存の動作に対する継承属性で、最大限に設計を再利用します。
ライブラリ **共有関数**	• 頻繁に使用し、変更はあまり頻繁には行わないモデル化ユーティリティを提供します。 • 1 つのモデルまたは複数のモデル内でコンポーネントを繰り返し再利用します。
モデル参照 **コンポーネント/モジュール**	• 参照モデルはそれを使用するモデルからは独立して開発します。 • 参照モデルの内容は見えないようにできるので、知的財産であるモデルの内容を隠したまま配布できます。 • 冗長なコピーを作成することなく、1 つのモデルを複数回参照します。 • 最上位レベルのコンポーネントに対する定義済みのインターフェイスによって、複数のユーザーによる変更を容易にします。 • 大規模モデル (10,000 ブロックあるモデルなど) に対して、インクリメンタルなモデル読み込み、ブロック線図の更新、シミュレーションおよびコード生成を使用することで全体のパフォーマンスを向上させます。 • 単体テストを実行します。 • 大規模モデルのデバッグを簡略化します。 • モデル構造体を反映するコードを生成します。

Cコードの対応レベルイメージ

```
>> web(fullfile(docroot, 'simulink/ug/model-architecture-guidelines.html'))
```

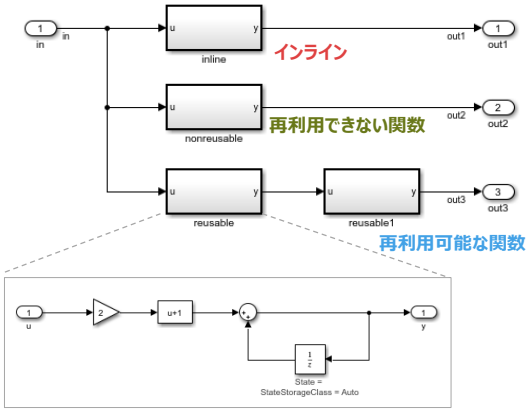

コードの再利用全般に関するドキュメント

```
>> web(fullfile(docroot, 'ecoder/code-reuse.html'))
```

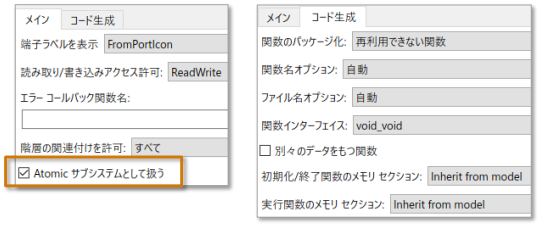
Goal	Referenced Model	Subsystem in Model	Subsystem in Library
Design for explicit code reuse.	√	√	√
Facilitate parallel team development.	√	√	√
Reuse function within a model or across models.	√		
Improve build performance by generating reusable code incrementally.	√		
Verify reusable code with SIL or PIL simulation.	√		
Optimize generated code by configuring code generator to detect opportunities for code reuse.	√	√	√
Maximize reuse with context-dependent behavior.		√	√
Develop a frequently used, and infrequently changed, utility function.			√

サブシステムのコード生成：関数・ファイル名・再利用性　1/2

サブシステムをアトミック化すると、サブシステム単位で関数化することができます。



サブシステムのブロックパラメータ
（ブロック右クリックからメニュー選択）



※ [関数のパッケージ化] を“自動”に設定すると、サブシステムの利用形態（複数利用の有無等）に応じて関数化の有無が自動選択されます

```
>> web(fullfile(docroot, 'ecoder/ug/control-generation-of-subsystem-functions.html'))
```


サブシステムのコード生成：関数・ファイル名・再利用性　2/2

```
#include "subsys.h"

real_T in;
real_T out3;
real_T out1;
real_T out2;
DW_subsys_T subsys_DW;

static real_T reusable_fcn(real_T rtu_u, DW_reusable_fcn_T *localDW);
static void nonreusable_fcn(void);

static void nonreusable_fcn(void)
{
  out2 = (2.0 * in + 1.0) + out2;
}

static real_T reusable_fcn(real_T rtu_u, DW_reusable_fcn_T *localDW)
{
  real_T rty_y_0;
  rty_y_0 = (2.0 * rtu_u + 1.0) + localDW->UnitDelay_DSTATE;
  localDW->UnitDelay_DSTATE = rty_y_0;
  return rty_y_0;
}

void subsys_step(void)
{
  real_T rtb_Sum;
  out1 = (2.0 * in + 1.0) + out1;
  nonreusable_fcn();
  rtb_Sum = reusable_fcn(in, &subsys_DW.reusable);
  rtb_Sum = reusable_fcn(rtb_Sum, &subsys_DW.reusable1);
  out3 = rtb_Sum;
}
```

再利用できない関数
（入出力・状態にグローバル変数）

再利用可能な関数
（入出力・状態が関数引数）

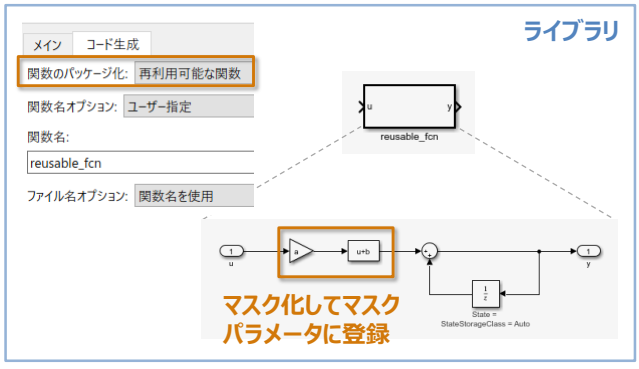
再利用できない関数

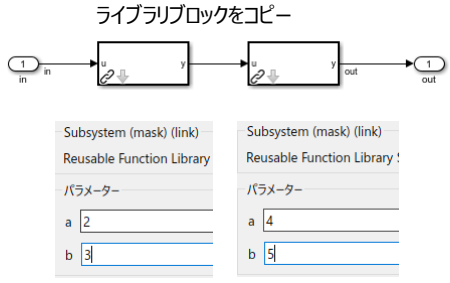
インライン

再利用可能な関数

サブシステムのコード生成：ライブラリブロックの共有関数化 1/4

- ライブラリ内アトミックサブシステムを複数モデル間で共通の関数として生成することができます。
  - サブシステムブロックパラメータの [関数パッケージ化] を“再利用可能関数”に設定、関数名は任意
  - ライブラリを呼び出すモデルのモデルコンフィグ [共有コードの配置] を“共有場所”に設定
  - マスクサブシステムのマスクパラメータは共有関数の引数となります。





>> web(fullfile(docroot, 'ecoder/ug/generate-reusable-code-from-library-subsystems-shared-across-models-4699e100eb8e.html'))

サブシステムのコード生成：ライブラリブロックの共有関数化 2/4

モデルCソース

```
#include "call_lib.h"
#include "call_lib_private.h"

real_T in;
real_T out;
DW_call_lib_T call_lib_DW;

void call_lib_step(void)
{
  real_T rtb_Sum;
  reusable_fcn(in, &rtb_Sum, &call_lib_DW.reusable_fcn_c, 2.0, 3.0);
  reusable_fcn(rtb_Sum, &rtb_Sum, &call_lib_DW.reusable_fcn1, 4.0, 5.0);
  out = rtb_Sum;
}
```

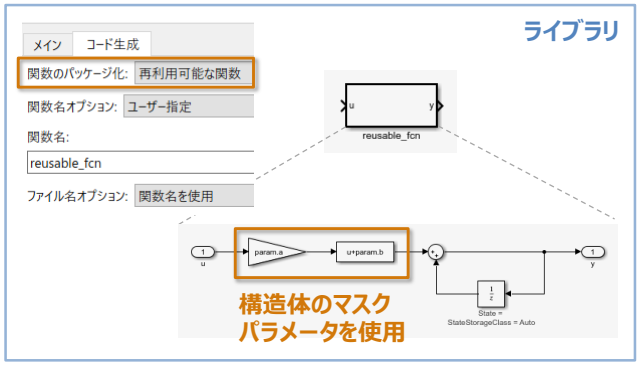
ライブラリCソース

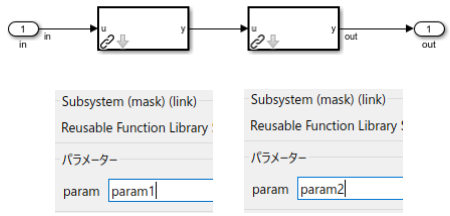
```
#include "reusable_fcn.h"

void reusable_fcn(real_T rtu_u, real_T *rty_y, DW_reusable_fcn_T *localDW,
    real_T rtp_a, real_T rtp_b)
{
  *rty_y = (rtp_a * rtu_u + rtp_b) + localDW->UnitDelay_DSTATE;
  localDW->UnitDelay_DSTATE = *rty_y;
}
```

※ライブラリ内に状態量を持つブロック（Unit Delay等）がある場合、ライブラリのインスタンス毎に状態量変数が生成されます。
※ライブラリ変更後のコード生成でエラーが出る場合は、slprjフォルダをいったん削除してから再度コード生成を行ってください。

サブシステムのコード生成：ライブラリブロックの共有関数化 3/4

- ブロックパラメータが大量にある場合、パラメータを構造体にすることで引数の数を節約できます。
  - Busオブジェクトを用いて構造体のメンバ構成・データ型等を定義できます。






```
>> web(fullfile(docroot, 'simulink/ug/using-structure-parameters.html'))
>> web(fullfile(docroot, 'ecoder/ug/generate-reusable-code-from-library-subsystems-shared-across-models-4699e100eb8e.html'))
```

15

サブシステムのコード生成：ライブラリブロックの共有関数化 4/4

モデルCソース

```
#include "call_lib.h"
#include "call_lib_private.h"

real_T in;
real_T out;
paramBus param1 = {
  2.0,
  3.0
} ;
paramBus param2 = {
  4.0,
  5.0
} ;

DW_call_lib_T call_lib_DW;
void call_lib_step(void)
{
  real_T rtb_Sum;
  rtb_Sum = reusable_fcn(in, &call_lib_DW.reusable_fcn_c, &param1);
  rtb_Sum = reusable_fcn(rtb_Sum, &call_lib_DW.reusable_fcn1, &param2);
  out = rtb_Sum;
}
```

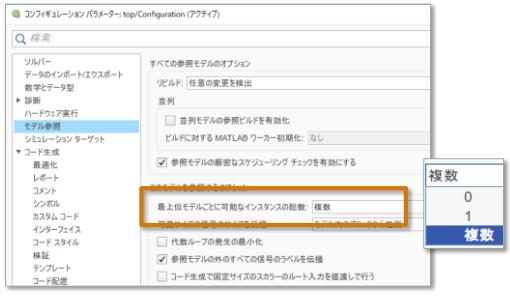
ライブラリCソース

```
#include "reusable_fcn.h"

#include "call_lib.h"
#include "call_lib_private.h"

real_T reusable_fcn(real_T rtu_u, DW_reusable_fcn_T *localDW,
    const paramBus *rtp_param)
{
  real_T rty_y_0;
  rty_y_0 = (rtp_param->a * rtu_u + rtp_param->b) + localDW->UnitDelay_DSTATE;
  localDW->UnitDelay_DSTATE = rty_y_0;
  return rty_y_0;
}
```

16

参照モデルのコード生成：基本情報・コンフィグ設定

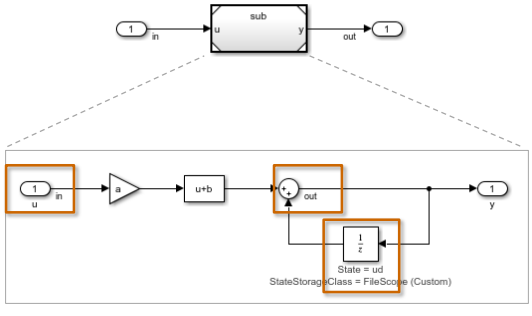
- 親モデルからコード生成を行うと、子モデルが参照モデルとしてコード生成されます。
  - モデル単体で生成したコード（最上位モデルビルド）と参照モデルとして生成されたコード（モデル参照ビルド）は一般に異なります。
  - 参照モデルの内容に変更がないと判断されるとコード生成は省略されます。
  - 参照モデルのコードは"slprj¥ert"フォルダ下に生成されます（モデルコンフィグの [共有コードの配置] が"共有場所"の場合 ）
  - slbuild('参照モデル名', 'ModelReferenceRTWTargetOnly') コマンドで親モデル無しのモデル参照ビルドが可能です。



- 0：参照を許可しない
- 1：1回のみ参照を許可
  - モデルの中にストレージクラスがAuto以外のデータオブジェクトを持つ信号・状態がある場合、1にする必要があります。
  - 関数インターフェースを指定することが可能となります。
- 複数：複数回の参照を許可
  - 複数回呼び出すモデル/関数や共有関数で使用します。

下記ドキュメントにモデル参照に対するコード生成上の制限について説明されています。
>> web(fullfile(docroot, 'rtw/ug/simulink-coder-model-referencing-limitations.html'))

参照モデルのコード生成：1インスタンス



信号をグローバル変数、
状態をファイルスコープ変数（static）に設定

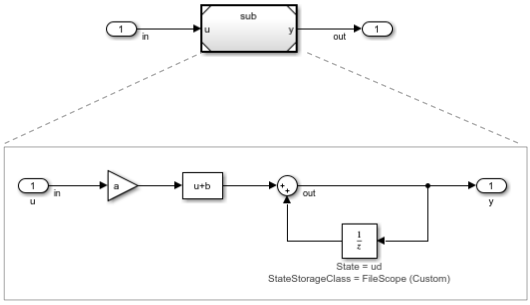
モデルCソース

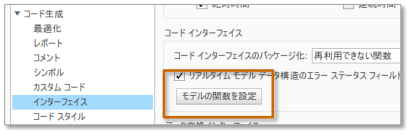
```
#include "top.h"

real_T in;
real_T out;

void top_step(void)
{
  sub();
}
```

参照モデルCソース

```
#include "sub.h"

real_T a = 2.0;
real_T b = 3.0;
static real_T ud;

void sub(void)
{
  out = (a * in + b) + ud;
  ud = out;
}
```

参照モデルのコード生成：1インスタンス（関数インターフェース使用）





subモデルのモデルコンフィグ設定

モデルCソース

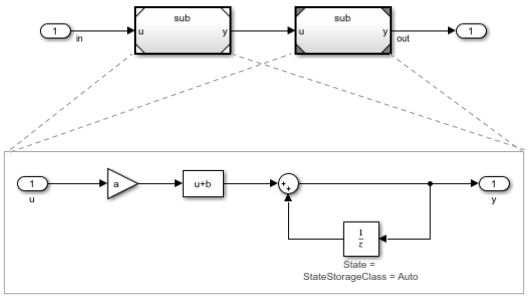
```
#include "top.h"

real_T in;
real_T out;

void top_step(void)
{
  out = sub_fcn(in);
}
```

参照モデルCソース

```
#include "sub.h"

real_T a = 2.0;
real_T b = 3.0;
static real_T ud;

real_T sub_fcn(real_T u)
{
  real_T y;

  y = (a * u + b) + ud;
  ud = y;
  return y;
}
```

19

参照モデルのコード生成：マルチインスタンス



※参照モデル内に状態量を持つブロック（Unit Delay等）がある場合、インスタンス毎に個別に状態量変数が生成されます

モデルCソース

```
#include "top.h"

real_T in;
real_T out;
DW_top_T top_DW;

void top_step(void)
{
  real_T rtb_Model;
  sub(&in, &rtb_Model, &(top_DW.Model_InstanceData.rtdw));
  sub(&rtb_Model, &out, &(top_DW.Model1_InstanceData.rtdw));
}
```

参照モデルCソース

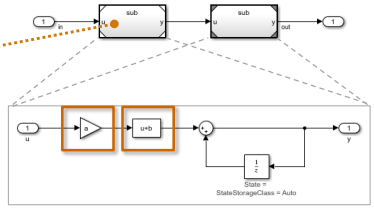
```
#include "sub.h"

real_T a = 2.0;
real_T b = 3.0;

void sub(const real_T *rtu_u, real_T *rty_y, DW_sub_f_T *localDW)
{
  *rty_y = (a * *rtu_u + b) + localDW->UnitDelay_DSTATE;
  localDW->UnitDelay_DSTATE = *rty_y;
}
```

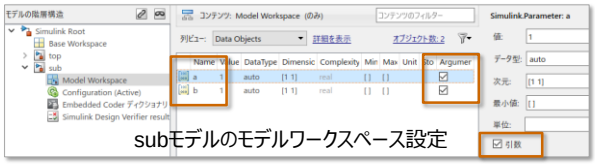
20

参照モデルのコード生成：モデル引数の利用（個々の引数）

（チューナブル）ブロックパラメータをモデル引数として定義することで、ブロックパラメータを引数に持つ関数を生成することができます。一部パラメータのみが異なるロジックを1関数として生成することができます。

Modelブロックの
引数設定





subモデルのモデルワークスペース設定

モデルCソース

```
real_T in;
real_T out;
DW_top_T top_DW;
void top_step(void)
{
  real_T rtb_Model;
  sub(&in, &rtb_Model, &(top_DW.Model_InstanceData.rtdw), 2.0, 3.0);
  sub(&rtb_Model, &out, &(top_DW.Model1_InstanceData.rtdw), 4.0, 5.0);
}
```

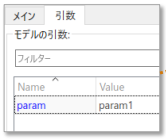
参照モデルCソース

```
void sub(const real_T *rtu_u, real_T *rty_y, DW_sub_f_T *localDW,
    real_T rtp_a, real_T rtp_b)
{
  *rty_y = (rtp_a * *rtu_u + rtp_b) + localDW->UnitDelay_DSTATE;
  localDW->UnitDelay_DSTATE = *rty_y;
}
```

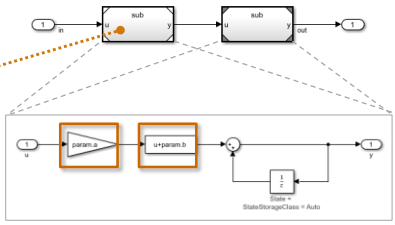
>> web(fullfile(docroot, 'rtw/ug/specify-instance-specific-parameter-values-for-reusable-referenced-models.html'))

参照モデルのコード生成：モデル引数の利用（構造体）

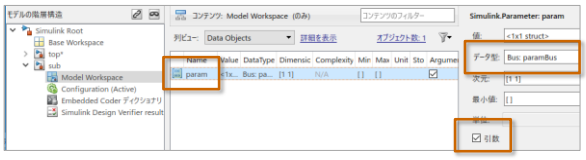
ブロックパラメータが大量にある場合、モデル引数を構造体にすることで引数の数を節約できます。



Modelブロックの
引数設定



引数のデータ型にBusオブジェクトを指定



subモデルのモデルワークスペース設定

モデルCソース

```
real_T in;
real_T out;
paramBus param1 = {
  2.0,
  3.0
} ;
paramBus param2 = {
  4.0,
  5.0
} ;
DW_top_T top_DW;
void top_step(void)
{
  real_T rtb_Model;
  sub(&in, &rtb_Model, &(top_DW.Model_InstanceData.rtdw), &param1);
  sub(&rtb_Model, &out, &(top_DW.Model1_InstanceData.rtdw), &param2);
}
```

```
void sub(const real_T *rtu_u, real_T *rty_y, DW_sub_f_T *localDW,
    const paramBus* rtp_param)
{
  *rty_y = (rtp_param->a * *rtu_u + rtp_param->b) +
  localDW->UnitDelay_DSTATE;
  localDW->UnitDelay_DSTATE = *rty_y;
}
```

参照モデルCソース

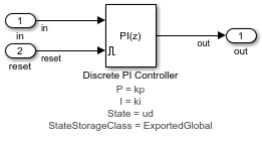
>> web(fullfile(docroot, 'simulink/ug/using-structure-parameters.html'))
>> web(fullfile(docroot, 'rtw/ug/specify-instance-specific-parameter-values-for-reusable-referenced-models.html'))

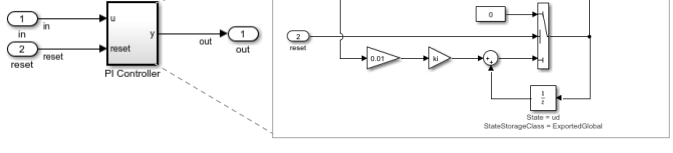
モデリング

効率性・可読性を考慮したモデルのリファクタリング

- 一部のSimulinkブロックからはユーザーから制御できない自動変数が生成される場合があります。
- 基本ブロックを用いて同等な演算処理を自作することで、自動変数を消去できる可能性があります。

積分値リセット機能付きPI制御の例






```
DW_pid_block_T pid_block_DW;
void pid_block_step(void)
{
  if (reset || (pid_block_DW.Integrator_PrevResetState != 0)) {
    ud = 0.0;
  }

  out = kp * in + ud;
  ud += ki * in * 0.01;
  pid_block_DW.Integrator_PrevResetState = (int8_T)reset;
}
```

グローバル変数を生成

```
void pid_subsys_step(void)
{
  real_T rtb_Switch;
  if (reset) {
    rtb_Switch = 0.0;
  } else {
    rtb_Switch = 0.01 * in * ki + ud;
  }

  out = kp * in + rtb_Switch;
  ud = rtb_Switch;
}
```

ローカル変数を生成

ベクトル・行列演算：各記述方法の特徴

記述方法	各要素への同一処理	各要素で異なる処理	特徴
信号のベクトル・行列演算	○	△	スカラー拡張（※）等でベクトル・行列信号をそのまま演算できるときにおススメ
For Eachサブシステム	○	△	再利用可能ライブラリ関数を使用すると繰り返し回数を関数の引数にできる（繰り返し回数のみが異なる処理を同一関数化できる）
For Iteratorサブシステム	○	○	ブロック線図での記述が大変な場合がある
MATLAB Functionブロック	○	○	MATLAB構文・関数を利用できるので複雑な処理も記述しやすい
Stateflow	○	○	CおよびMATLAB構文・関数を利用できるので複雑な処理も記述しやすい

※スカラー拡張：スカラーパラメータをベクトル・行列信号に適用するルール
>> web(fullfile(docroot, 'simulink/ug/determining-output-signal-dimensions.html#bsud_ek-10'))

※繰り返し処理に対する生成コードはモデルコンフィグの [ループ展開のしきい値] 設定により、for文または各要素のインライン処理のどちらかが選択されます。
>> web(fullfile(docroot, 'rtw/ug/configuring-a-loop-unrolling-threshold.html'))

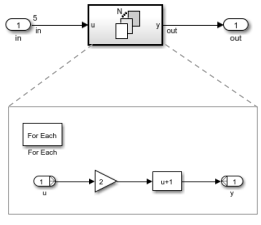
ベクトル・行列演算：各要素に対する同一処理の例

5要素のベクトル信号各要素に対する同一処理

ベクトル信号のスカラー拡張

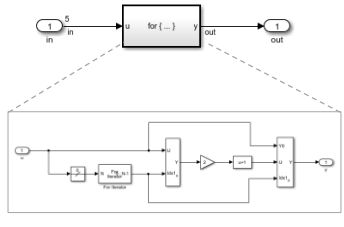

```
int32_T i;

for (i = 0; i < 5; i++) {
  out[i] = 2.0 * in[i] + 1.0;
}
```

For Each サブシステム



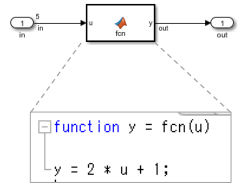

```
int32_T ForEach_itr;

for (ForEach_itr = 0; ForEach_itr < 5; ForEach_itr++) {
  out[ForEach_itr] = 2.0 * in[ForEach_itr] + 1.0;
}
```

For Iterator サブシステム




```
int32_T i;

for (i = 0; i < 5; i++) {
  out[i] = in[i];
}
for (i = 0; i < 5; i++) {
  out[i] = 2.0 * in[i] + 1.0;
}
```

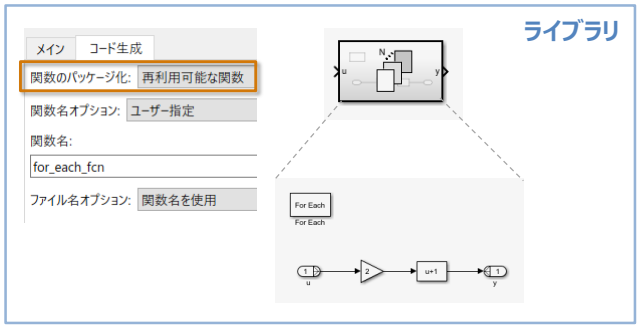
MATLAB Functionブロック




```
int32_T i;

for (i = 0; i < 5; i++) {
  out[i] = 2.0 * in[i] + 1.0;
}
```

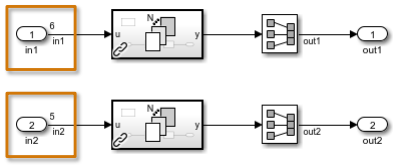
ベクトル・行列演算：For Each サブシステムのライブラリ再利用可能関数

For Eachサブシステムを再利用可能ライブラリ関数にすると、繰り返し回数を関数引数にできます。
（繰り返し回数のみが異なる処理を同一関数化）




```
void for_each_fcn(int32_T NumIters, const real_T rtu_u[6], real_T rty_y[6])
{
  int32_T ForEach_itr;
  for (ForEach_itr = 0; ForEach_itr < NumIters; ForEach_itr++) {
    rty_y[ForEach_itr] = 2.0 * rtu_u[ForEach_itr] + 1.0;
  }
}
```

ライブラリCソース



※R2018aではFor Eachライブラリブロック出力にストレージクラスを持たせるとエラーとなります。同エラーを回避するため、Signal Conversionブロックを挿入しています。

```
real_T rtb_ImpAsg_InsertedFor_y_at_inp[5];
real_T rtb_ImpAsg_InsertedFor_y_at_i_e[6];
int32_T i;

for_each_fcn(6, in1, rtb_ImpAsg_InsertedFor_y_at_i_e);
for (i = 0; i < 6; i++) {
  out1[i] = rtb_ImpAsg_InsertedFor_y_at_i_e[i];
}

for_each_fcn(5, in2, rtb_ImpAsg_InsertedFor_y_at_inp);
for (i = 0; i < 5; i++) {
  out2[i] = rtb_ImpAsg_InsertedFor_y_at_inp[i];
}
```

モデルCソース

27

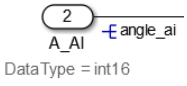
Simulinkデータオブジェクト ストレージクラス

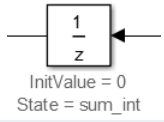
28

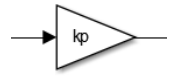
ストレージクラス

- ストレージクラスを適用することでモデル内の信号・状態やパラメータをユーザ指定の名前・記憶クラス・データ形式で生成することができます。
- ストレージクラスを適用するには次の2通りの方法があります。
  - Simulinkデータオブジェクトをワークスペースまたは Simulinkデータディクショナリに作成して関連付ける。
    - SimulinkデータオブジェクトにはSignalオブジェクトとParameterオブジェクトの2種類があります。
  - モデル内の信号、状態、出力端子に直接ストレージクラスを設定する（Signalオブジェクトに相当）。
- ストレージクラス関連ドキュメント・例題
  - >> web(fullfile(docroot, 'rtw/ug/signal-objects.html'))
  - >> web(fullfile(docroot, 'ecoder/ug/define-and-declare-signal-data.html'))
  - >> web(fullfile(docroot, 'ecoder/ug/data-declaration.html'))

ストレージクラス適用例








```
int16_T angle_ai;
```

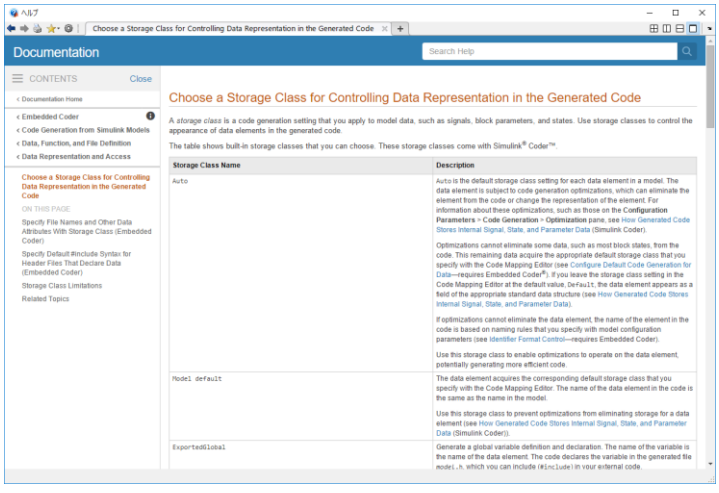
```
static real32_T sum_int;
```

```
#define kp 0.1068F;
```

29

ストレージクラス一覧

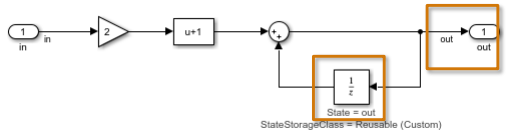
>> web(fullfile(docroot, 'ecoder/ug/choose-a-built-in-storage-class-for-controlling-data-representation-in-the-generated-code.html'))



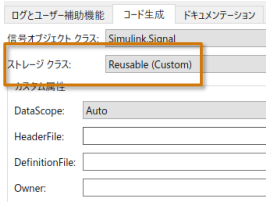
30

Reusableストレージクラスによるグローバル変数の削減・再利用

- Reusableストレージクラスを適用すると、信号・状態を（可能であれば）同一グローバル変数として生成します。グローバル変数が使用するRAM消費量の節約に貢献します。
- 同じ変数になることでコードデバッグの難易度が上がる可能性もありますので注意が必要です。



出力信号名とUnit Delay状態名を
同名にしてストレージクラスをReusableに設定



Reusable無し

```
out = (2.0 * in + 1.0) + ud;
ud = out;
```

Reusable適用

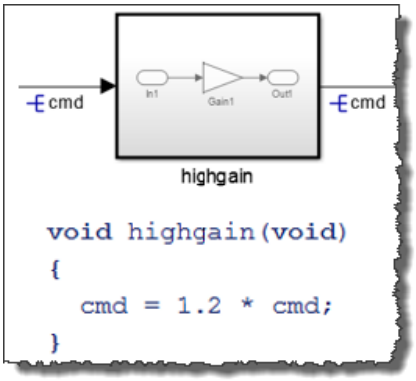
```
out = (2.0 * in + 1.0) + out;
```

```
>> web(fullfile(docroot, 'ecoder/ug/signal-reuse-at-model-and-unit-delay-boundary.html'))
```


グローバル変数再利用によるメモリ消費節約
さらなるメモリ消費節約に貢献

R2016a

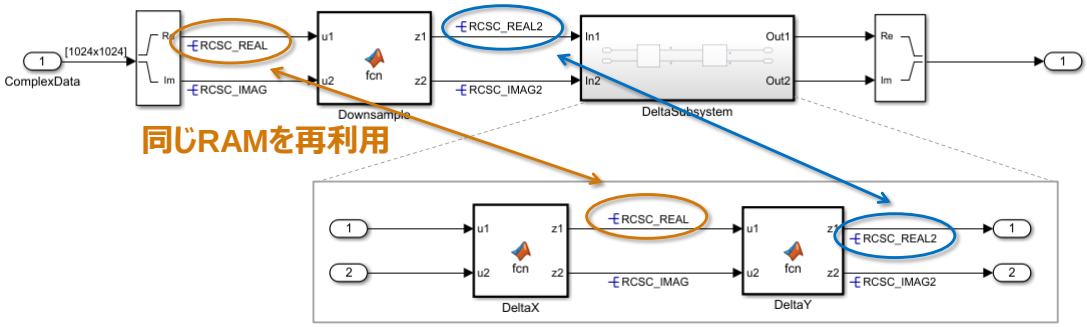
- サブシステム入出力間の信号をReusable ストレージクラスを用いて同一グローバル変数に指定可能に



Reusableストレージクラスによるグローバルデータ再利用
更なるRAM消費量削減を実現できます

R2017b

- Reusableストレージクラスを複数回指定可能となりました
  同一RAMの上書き利用によるRAM消費抑制を図ることが出来ます
- 上書き利用ができない場合は、診断メッセージが表示されます



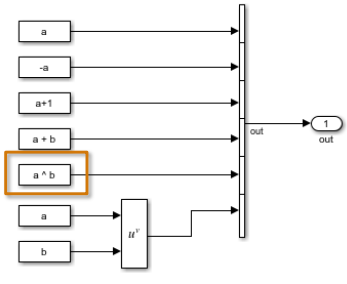
Parameterオブジェクトを含む数式のコード生成

- 調整可能な数式は生成コードにそのまま反映されます。
- 調整可能でない式はその計算結果がインラインで記述されます。
- モデルコンフィグの [診断] → [データ有効性] → [調整可能性の消失を検出] で
  コード生成時に調整可能性の有無をチェックできます。
- 調整可能な演算子・数学関数については
  下記ドキュメントに詳細が記載されています。
  >> web(fullfile(docroot, 'rtw/ug/parameters.html'))

調整可能性の消失診断メッセージ

'tunable expression/Constant4' のパラメーター 'Value' で使用される式 'a^b' には 1 つ以上のサポートされない演算子があるため、生成されたコードはこの式の数値をインライン化します (see ドキュメンテーションを参照)。このブロックのコードは調整可能な変数 (a (base workspace), b (base workspace)) を使用しません。

コンポーネント: Simulink | カテゴリ: Block 警告



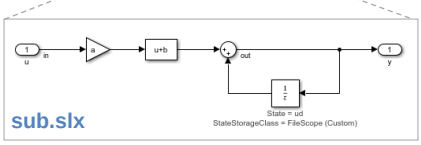

```
real_T a = 2.0;
real_T b = 3.0;

out[0] = a;

out[1] = -a;

out[2] = a + 1.0;

out[3] = a + b;

out[4] = 8.0;

if ((a < 0.0) && (b > floor(b))) {
  out[5] = -pow(-a, b);
} else {
  out[5] = pow(a, b);
}
```

調整可能性が失われた結果、2^3 = 8になっている

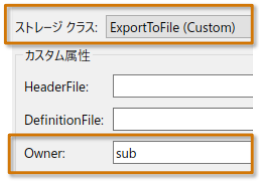
モデル参照利用時のグローバル変数定義ファイル指定

- 定義文が生成されるストレージクラスでは、
  （通常は）最上位モデル生成コードに定義文が記述されます。
- 定義文を参照モデル生成コードにしたい場合は、オーナー属性を持つストレージクラスを利用します。

top.slx



sub.slx



top.c

a,bのオーナーを空にした場合

```
real_T in;
real_T out;
real_T a = 2.0;
real_T b = 3.0;

void top_step(void)
{
  sub();
}
```

a,bのオーナーをsubにした場合

```
real_T in;
real_T out;

void top_step(void)
{
  sub();
}
```

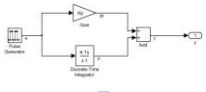
sub.c

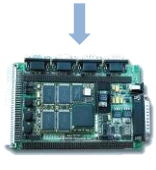
```
static real_T ud;

void sub(void)
{
  out = (a * in + b) + ud;
  ud = out;
}
```

```
real_T a = 2.0;
real_T b = 3.0;
static real_T ud;

void sub(void)
{
  out = (a * in + b) + ud;
  ud = out;
}
```

35

トレーニングコースのご紹介
Embedded Coder





Embedded Coder の多種多様な機能をじっくり学べる！

Embedded Coder による 量産向けコード生成

Embedded Coder による、組み込みターゲットに向けた高効率な C コードの開発方法を学習できます。

Embedded Coder を使用すると、Simulink モデルで作成したアルゴリズムを組み込みターゲットのコード開発環境で使用出来る C ソース ファイルに自動的に変換することができます。ターゲット ハードウェアの制約や、組み込みコードの開発目的に基づいたデータ型・メモリー使用量・アルゴリズムの実行効率性などの要求を満たすコードを、Simulink モデルの設定やブロックのパラメーター、データ オブジェクト、TLC ファイルなどを駆使して自動的に生成させる方法を、この３日間コースで学ぶことができます。

スケジュール：

日程	概要	9	10	11	12	1	2	3	4	5
1日目	- コード生成の手順 - 生成された関数の取り扱い	コード生成の基本手順		生成コードの外部環境へのエキスポート			リアルタイム実行について		関数プロトタイプ	
2日目	- コードの最適化 - データ属性の調整と管理	生成コードの最適化		信号・パラメーターのデータ型・サイズ・etc			データオブジェクト	ストレージクラス	バス信号	
3日目	- コードの構造 - カスタムターゲット - 標準規格への準拠	コードのアーキテクチャ		コード生成・ビルドプロセスの理解			カスタムターゲット		I/Oブロック	標準規格

36