Stateflowからの量産/組み込みCコード生成 初心者向けチップス集

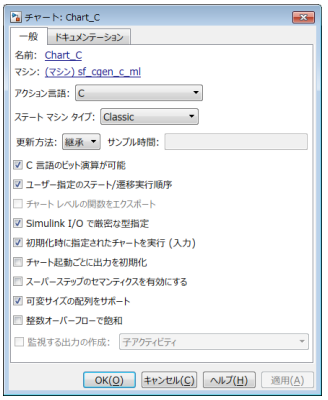
MathWorks Japan

アジェンダ

1. はじめに
2. 基本設定
3. フローチャート
4. 状態遷移図／表
5. 各種関数の利用
6. 外部C関数の呼び出し

はじめに

- 本資料は、Embedded Coderを用いてStateflowモデルから組み込み用Cコードを生成したい方を対象に、基本的な機能や使い方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）
- 例題モデルのチャートプロパティは右図の設定をベースとしています。（例題によっては一部変更アリ）
- R2016b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。
- 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。



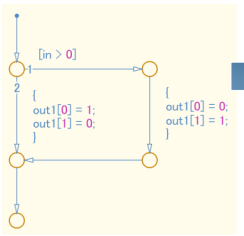
例題モデルについて

- MATLABから sf_cgen_examples.html を開くと例題用Webページが表示されます。
  同ページ内リンクをクリックすると例題モデルが開きます。
- モデルからCコードを生成するにはモデルエディタのビルドアイコンをクリックしてください。
- スライド右下に例題モデル名を記載しています。
  例： >> sf_cgen_c_ml.slx

Stateflowチャートアクション言語の違い 1/2

アクション言語としてC言語/MATLAB言語を選択できますが、両者は似て非なる言語です。
生成されるCコードは処理・記述により同じ場合もあれば異なる場合もあります。下図はベクトルデータの要素設定例です。

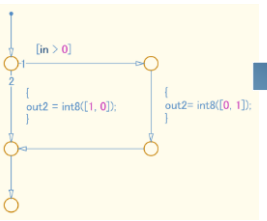
C言語



Code

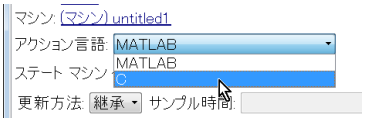
```
if (in > 0) {
  out1[0] = 0;
  out1[1] = 1;
} else {
  out1[0] = 1;
  out1[1] = 0;
}
```

MATLAB言語



Code

```
if (in > 0) {
  out2[0] = 0;
  out2[1] = 1;
} else {
  out2[0] = 1;
  out2[1] = 0;
}
```



チャートプロパティからアクション言語を設定します。
エディタ左下にアクション言語設定を示すアイコンが表示されます。

>> sf_cgen_c_ml.slx

Stateflowチャートアクション言語の違い 2/2

両言語の主な違いはStateflowドキュメントに記載があります。

ヘルプ表示MATLABコマンド：web(fullfile(docroot, 'stateflow/ug/differences-between-matlab-and-stateflow-action-language.html'))
Webヘルプ：https://www.mathworks.com/help/releases/R2016b/stateflow/ug/differences-between-matlab-and-stateflow-action-language.html

MathWorks

Stateflowデータに対するデータオブジェクト関連付け 1/2

- Simulink信号 / 状態 / パラメータと同じように、Stateflowデータに対して信号 / パラメータオブジェクトを関連付けて生成コード内データの名前や記憶クラス等を設定することができます。
- データオブジェクトを関連付けない場合、Coder標準の構造体データが使用されます。

データオブジェクト無し

入力 / 出力 / ローカルデータは構造体、パラメータデータはインラインの数値として生成されています。

```
DW_sf_cgen_data_object_T sf_cgen_data_object_DW;
ExtU_sf_cgen_data_object_T sf_cgen_data_object_U;
ExtY_sf_cgen_data_object_T sf_cgen_data_object_Y;
void sf_cgen_data_object_step(void)
{
  sf_cgen_data_object_Y.out = (int8_T)((sf_cgen_data_object_U.in +
    sf_cgen_data_object_DW.local) + 3);
}

void sf_cgen_data_object_initialize(void)
{
}
```

```
{
out = in + local + param;
}
```

Code

データオブジェクト有り

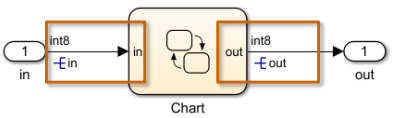
データオブジェクト設定がStateflowデータに反映されています。

```
int8_T in;
int8_T out;
int8_T param = 3;
int8_T local;
void sf_cgen_data_object_step(void)
{
  out = (int8_T)((in + local) + param);
}

void sf_cgen_data_object_initialize(void)
{
  local = 2;
}
```

>> sf_cgen_data_object.slx

MathWorks

Stateflowデータに対するデータオブジェクト関連付け 2/2

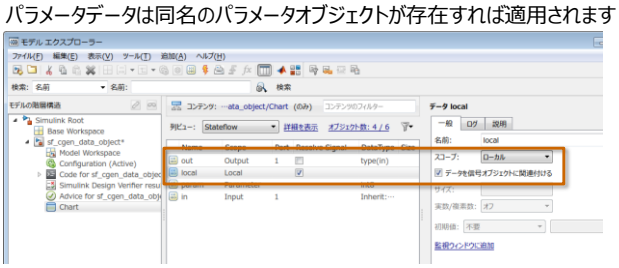


入出力データはSimulink信号に対するデータオブジェクト設定が適用されます

```
int8_T in;
int8_T out;
int8_T param = 3;
int8_T local;
void sf_cgen_data_object_step(void)
{
  out = (int8_T)((in + local) + param);
}

void sf_cgen_data_object_initialize(void)
{
  local = 2;
}
```

```
{
out = in + local + param;
}
```

ローカルデータはモデルエクスプローラから関連付けを行います
パラメータデータは同名のパラメータオブジェクトが存在すれば適用されます

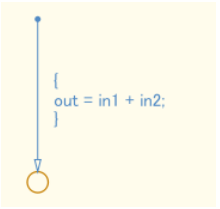


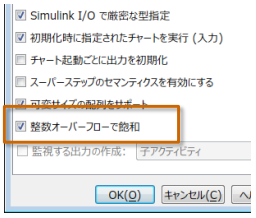
関連付けが有効な場合、データオブジェクト設定を反映したデータが使用されます

MathWorks

チャートプロパティ：整数オーバーフローで飽和

チャートプロパティの **整数オーバーフローで飽和** を有効にすると、整数演算に対して
シミュレーションおよび生成コード双方に飽和処理が入ります。





Code

飽和処理無し

```
out = (int8_T)(in1 + in2);
```

飽和処理有り

```
int32_T tmp;
tmp = in1 + in2;
if (tmp > 127) {
  tmp = 127;
} else {
  if (tmp < -128) {
    tmp = -128;
  }
}

out = (int8_T)tmp;
```

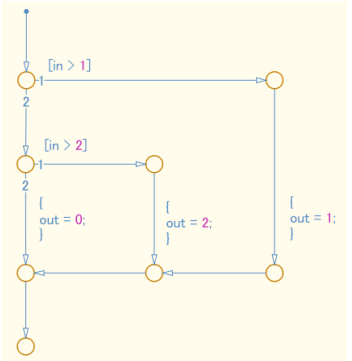
>> sf_cgen_integer_overflow.slx

9

MathWorks

フローチャート：if-else

条件分岐チャートは基本的にif-elseif-else文となります。



Code

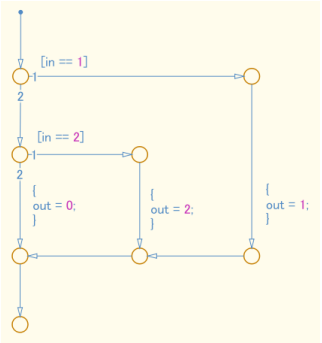
```
if (in > 1) {
   out = 1;
 } else if (in > 2) {
   out = 2;
 } else {
   out = 0;
 }
```

>> sf_cgen_if.slx

10

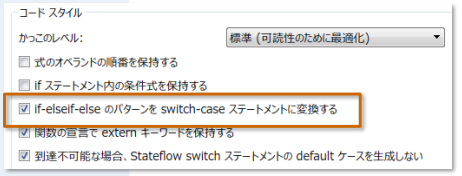
フローチャート：switch-case

下記モデルコンフィギュレーションパラメータにチェックを入れると、特定のif-elseif-else文をswitch-case文として生成します。
コード生成 -> コードスタイル -> if-elseif-elseのパターンをswitch-caseステートメントに変換する



Code

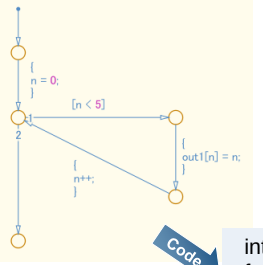
```
switch (in) {
  case 1:
   out = 1;
   break;

  case 2:
   out = 2;
   break;

  default:
   out = 0;
   break;
 }
```



>> sf_cgen_switch.slx

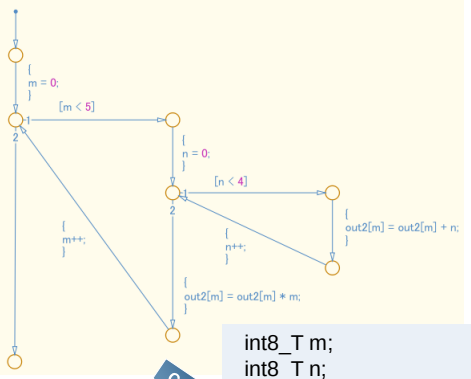
11

フローチャート：for ループ



Code

```
int8_T n;
for (n = 0; n < 5; n++) {
  out1[n] = n;
}
```



Code

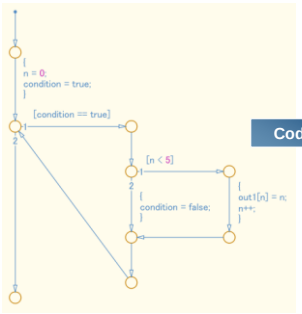
```
int8_T m;
int8_T n;
for (m = 0; m < 5; m++) {
  for (n = 0; n < 4; n++) {
    out2[m] += n;
  }
  out2[m] *= m;
}
```

>> sf_cgen_for.slx

12

フローチャート：while ループ / do-while ループ

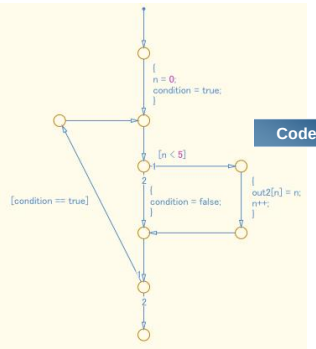
while



Code

```
n = 0;
condition = true;
while (condition) {
  if (n < 5) {
    out1[n] = n;
    n++;
  } else {
    condition = false;
  }
}
```

do-while



Code

```
n = 0;
condition = true;
do {
  if (n < 5) {
    out2[n] = n;
    n++;
  } else {
    condition = false;
  }
} while (condition);
```

>> sf_cgen_while.slx

フローチャート：パターンウィザード

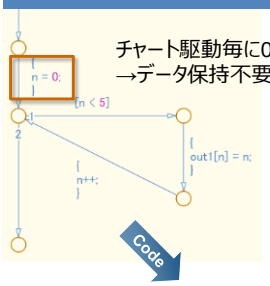
パターンウィザードを使用するとフローチャートを簡単に記述することができます。

ヘルプ表示MATLABコマンド：web(fullfile(docroot, 'stateflow/ug/creating-flow-graphs-with-the-pattern-wizard.html'))
Webヘルプ：https://www.mathworks.com/help/releases/R2016b/stateflow/ug/creating-flow-graphs-with-the-pattern-wizard.html

Stateflowローカルスコープデータがグローバル変数になるケース

Stateflowローカルデータが必ずCコード上のローカル変数になるとは限りません。
チャート計算にデータ保持が必要と判断された場合は、グローバル変数として生成されます。

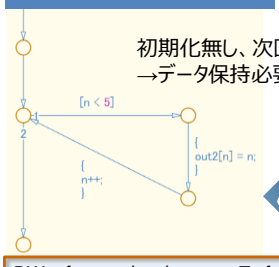
ローカル変数になる例



チャート駆動毎に0初期化される
→データ保持不要、ローカル変数として生成

```
void sf_cgen_local_scope_step(void)
{
  int8_T n;
  for (n = 0; n < 5; n++) {
    out1[n] = n;
  }
}
```

グローバル変数になる例



初期化無し、次回ステップでnの前回値が使用される
→データ保持必要、グローバル変数として生成

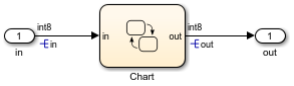
```
DW_sf_cgen_local_scope_T sf_cgen_local_scope_DW;
void sf_cgen_local_scope_step(void)
{
  while (sf_cgen_local_scope_DW.n < 5) {
    out2[sf_cgen_local_scope_DW.n] = sf_cgen_local_scope_DW.n;
    sf_cgen_local_scope_DW.n++;
  }
}
```

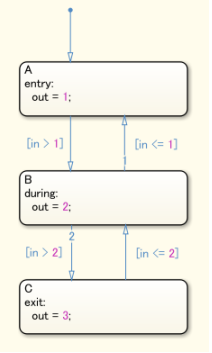
>> sf_cgen_local_scope.slx

15

状態遷移：初期化時チャート実行　1/2

チャートプロパティの**初期化時に指定されたチャートを実行（入力）**
設定によりデフォルト遷移処理位置が変化します
（シミュレーション結果も変化します）





Code

初期化時に指定されたチャートを実行（入力）：OFFでの生成コード

```
#define sf_cgen_state_IN_A             ((uint8_T)1U)
#define sf_cgen_state_IN_B             ((uint8_T)2U)
#define sf_cgen_state_IN_C             ((uint8_T)3U)
…
DW_sf_cgen_state_T sf_cgen_state_DW;
void sf_cgen_state_step(void)
{
 if (sf_cgen_state_DW.is_active_c3_sf_cgen_state == 0U) {
   sf_cgen_state_DW.is_active_c3_sf_cgen_state = 1U;
   sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_A;
   out = 1;
 } else {
   switch (sf_cgen_state_DW.is_c3_sf_cgen_state) {
    case sf_cgen_state_IN_A:
     if (in > 1) {
       sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_B;
     }
     break;

    case sf_cgen_state_IN_B:
     if (in <= 1) {
       sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_A;
       out = 1;
     } else if (in > 2) {
       sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_C;
     } else {
       out = 2;
     }
     break;

    default:
     if (in <= 2) {
       out = 3;
       sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_B;
     }
     break;
   }
 }
}

void sf_cgen_state_initialize(void)
{
}
```

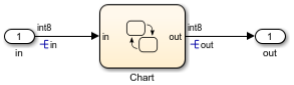
状態用defineマクロ
デフォルト遷移処理
状態A処理
状態B処理
状態C処理

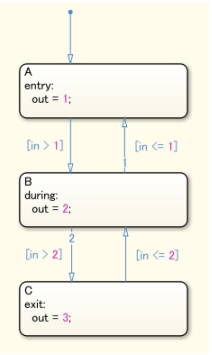
>> sf_cgen_state.slx

17

状態遷移：初期化時チャート実行　2/2

チャートプロパティの**初期化時に指定されたチャートを実行（入力）**
設定によりデフォルト遷移処理位置が変化します
（シミュレーション結果も変化します）





Code

初期化時に指定されたチャートを実行（入力）：ONでの生成コード

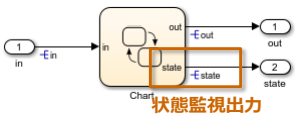
```
#define sf_cgen_state_IN_A             ((uint8_T)1U)
#define sf_cgen_state_IN_B             ((uint8_T)2U)
#define sf_cgen_state_IN_C             ((uint8_T)3U)
…
DW_sf_cgen_state_T sf_cgen_state_DW;
void sf_cgen_state_step(void)
{
  switch (sf_cgen_state_DW.is_c3_sf_cgen_state) {
   case sf_cgen_state_IN_A:
    if (in > 1) {
      sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_B;
    }
    break;

   case sf_cgen_state_IN_B:
    if (in <= 1) {
      sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_A;
      out = 1;
    } else if (in > 2) {
      sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_C;
    } else {
      out = 2;
    }
    break;

   default:
    if (in <= 2) {
      out = 3;
      sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_B;
    }
    break;
  }
}

void sf_cgen_state_initialize(void)
{
  sf_cgen_state_DW.is_c3_sf_cgen_state = sf_cgen_state_IN_A;
  out = 1;
}
```

状態指定defineマクロ
状態A処理
状態B処理
状態C処理
初期化関数でデフォルト遷移処理を実行

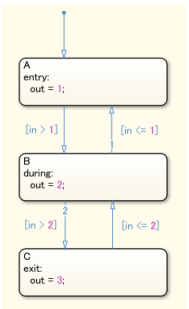
>> sf_cgen_state.slx

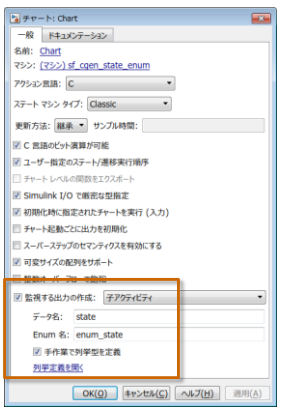
状態遷移：状態変数の列挙型定義　1/2

- 状態監視出力を作成すると、状態値を列挙値にできます。
- 状態監視出力にデータオブジェクトを関連付けることができます。



状態監視出力





Code

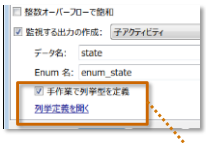
```
enum_state state;
void sf_cgen_state_enum_step(void)
{
  switch (state) {
   case A:
    if (in > 1) {
      state = B;
    }
    break;

   case B:
    if (in <= 1) {
      state = A;
      out = 1;
    } else if (in > 2) {
      state = C;
    } else {
      out = 2;
    }
    break;

   default:
    if (in <= 2) {
      out = 3;
      state = B;
    }
    break;
  }
}

void sf_cgen_state_enum_initialize(void)
{
  state = A;
  out = 1;
}
```

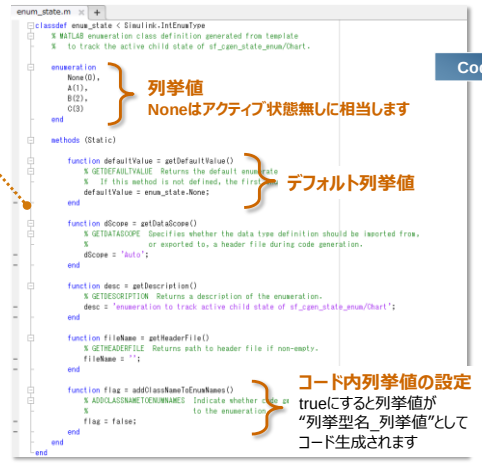
state：状態監視変数
A, B, C：状態列挙値

>> sf_cgen_state_enum.slx

状態遷移：状態変数の列挙型定義　2/2

列挙型の定義（列挙型シンボル・値、等）はMATLABスクリプトで行うことができます。





列挙値
Noneはアクティブ状態無しに相当します

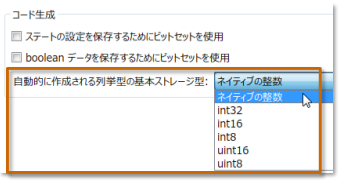
デフォルト列挙値

コード内列挙値の設定
trueにすると列挙値が
“列挙型名_列挙値”として
コード生成されます

Code

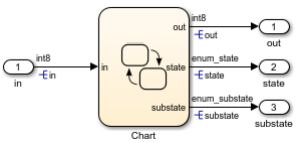
```
typedef enum {
  None = 0,
  A,
  B,
  C
} enum_state;
```

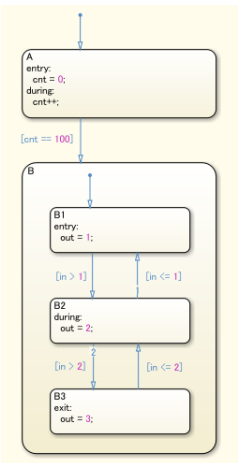
状態列挙型のデータサイズを下記のモデルコンフィギュレーションパラメータから設定できます。
最適化 -> Stateflow -> 自動的に作成される列挙型の基本ストレージ型



階層状態

各階層に状態監視出力を作成することができます。





Code

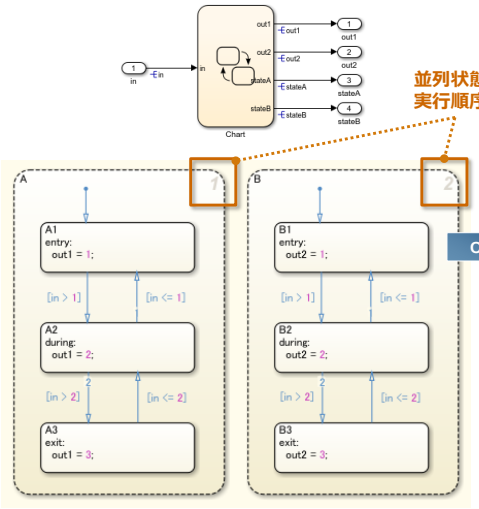
```
enum_state state;
enum_substate substate;
void sf_cgen_sub_state_step(void)
{
  if (state == A) {
    if (cnt == 100) {
      state = B;
      substate = B1;
      out = 1;
    } else {
      cnt = (uint8_T)(cnt + 1);
    }
  } else {
    switch (substate) {
      case B1:
        if (in > 1) {
          substate = B2;
        }
        break;

      case B2:
        if (in <= 1) {
          substate = B1;
          out = 1;
        } else if (in > 2) {
          substate = B3;
        } else {
          out = 2;
        }
        break;

      default:
        if (in <= 2) {
          out = 3;
          substate = B2;
        }
        break;
    }
  }
}

void sf_cgen_sub_state_initialize(void)
{
  substate = B_None;
  state = A;
}
```

状態A処理
状態B1処理
状態B2処理
状態B3処理
状態B処理

>> sf_cgen_sub_state.slx

並列状態

各並列状態が実行順序通りにコード生成されます。



Code

```
enum_stateA stateA;
enum_stateB stateB;
void sf_cgen_parallel_state_step(void)
{
  switch (stateA) {
   case A1:
    if (in > 1) {
      stateA = A2;
    }
    break;

   case A2:
    if (in <= 1) {
      stateA = A1;
      out1 = 1;
    } else if (in > 2) {
      stateA = A3;
    } else {
      out1 = 2;
    }
    break;

   default:
    if (in <= 2) {
      out1 = 3;
      stateA = A2;
    }
    break;
  }
  …
```

並列状態
A処理

```
  …
  switch (stateB) {
   case B1:
    if (in > 1) {
      stateB = B2;
    }
    break;

   case B2:
    if (in <= 1) {
      stateB = B1;
      out2 = 1;
    } else if (in > 2) {
      stateB = B3;
    } else {
      out2 = 2;
    }
    break;

   default:
    if (in <= 2) {
      out2 = 3;
      stateB = B2;
    }
    break;
  }
}

void sf_cgen_parallel_state_initialize(void)
{
  stateA = A1;
  out1 = 1;
  stateB = B1;
  out2 = 1;
}
```

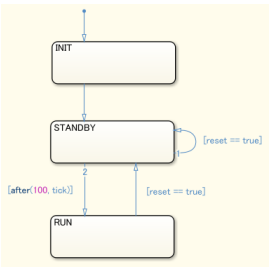
並列状態
B処理

>> sf_cgen_parallel_state.slx

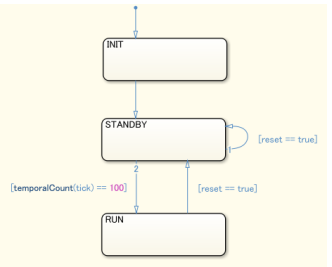
時相論理　1/4

下図のモデルは100ステップ経過後にSTANDBY状態からRUN状態に遷移、リセット信号でSTANDBY状態に戻るロジックを3通りの方法でモデリングしたものです。

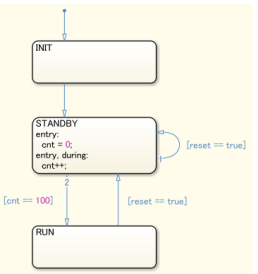
after使用

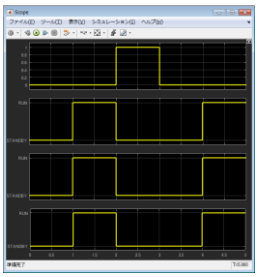


temporalCount使用



直接記述





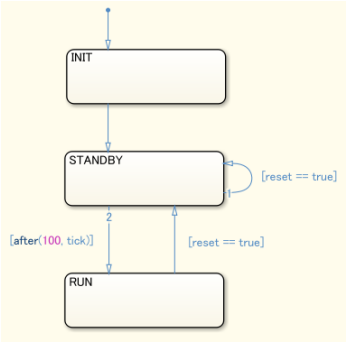
3者のシミュレーション結果は同じです

>> sf_cgen_temporal_logic.slx
>> sim_sf_cgen_temporal_logic.slx

MathWorks

時相論理　2/4

after使用



Code

```
DW_sf_cgen_temporal_logic_T sf_cgen_temporal_logic_DW;
…
if (sf_cgen_temporal_logic_DW.temporalCounter_i1_o < 127U) {
   sf_cgen_temporal_logic_DW.temporalCounter_i1_o++;
 }
```

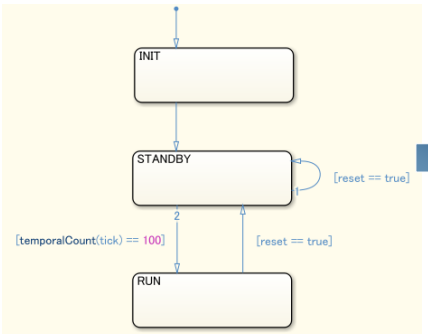
カウント用データを自動生成

```
 switch (state1) {
  case INIT:
   state1 = STANDBY;
   sf_cgen_temporal_logic_DW.temporalCounter_i1_o = 0U;
   break;

  case STANDBY:
   if (reset) {
     state1 = STANDBY;
     sf_cgen_temporal_logic_DW.temporalCounter_i1_o = 0U;
   } else {
     if (sf_cgen_temporal_logic_DW.temporalCounter_i1_o >= 100) {
       state1 = RUN;
     }
   }
   break;

  default:
   if (reset) {
     state1 = STANDBY;
     sf_cgen_temporal_logic_DW.temporalCounter_i1_o = 0U;
   }
   break;
 }
```


MathWorks

時相論理　3/4

temporalCount使用



Code

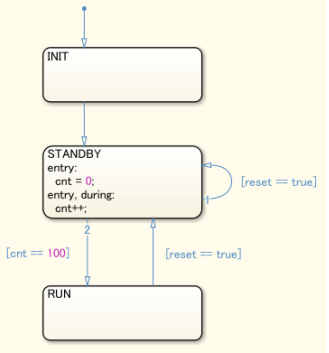
```
DW_sf_cgen_temporal_logic_T sf_cgen_temporal_logic_DW;
…
 if (sf_cgen_temporal_logic_DW.temporalCounter_i1 < MAX_uint32_T) {
    sf_cgen_temporal_logic_DW.temporalCounter_i1++;
 }
```

カウント用データを自動生成

```
 switch (state2) {
  case INIT:
   state2 = STANDBY;
   sf_cgen_temporal_logic_DW.temporalCounter_i1 = 0U;
   break;

  case STANDBY:
   if (reset) {
     state2 = STANDBY;
     sf_cgen_temporal_logic_DW.temporalCounter_i1 = 0U;
   } else {
     if (sf_cgen_temporal_logic_DW.temporalCounter_i1 == 100U) {
       state2 = RUN;
     }
   }
   break;

  default:
   if (reset) {
     state2 = STANDBY;
     sf_cgen_temporal_logic_DW.temporalCounter_i1 = 0U;
   }
   break;
 }
```

時相論理　4/4

直接記述



Code

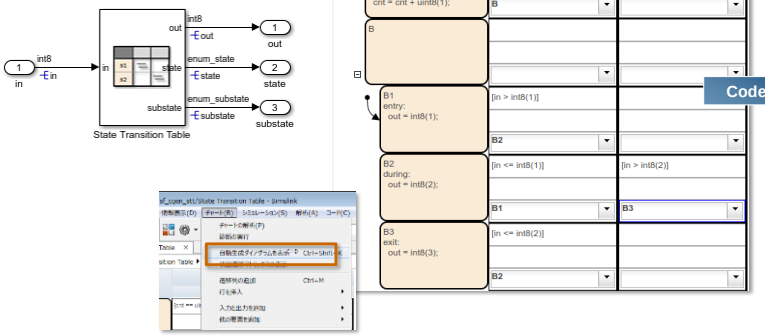
```
static uint8_T cnt;
…
 switch (state3) {
  case INIT:
   state3 = STANDBY;
   cnt = 0U;
   cnt = (uint8_T)(cnt + 1);
   break;

  case STANDBY:
   if (reset) {
    state3 = STANDBY;
    cnt = 0U;
    cnt = (uint8_T)(cnt + 1);
   } else if (cnt == 100) {
    state3 = RUN;
   } else {
    cnt = (uint8_T)(cnt + 1);
   }
   break;

  default:
   if (reset) {
    state3 = STANDBY;
    cnt = 0U;
    cnt = (uint8_T)(cnt + 1);
   }
   break;
 }
```

ローカルスコープデータcntを定義、ストレージクラスFileScopeの信号オブジェクトを関連付けています

26

状態遷移表

状態遷移図と同様にコード生成を行うことができます。
自動生成ダイアグラムを表示メニューで内部の状態遷移図を開いて
チャートプロパティ設定を行うことが可能です。



Code

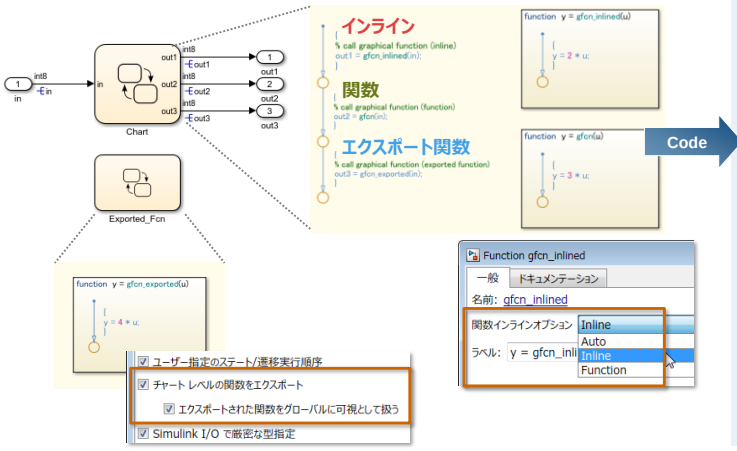
```
enum_state state;
enum_substate substate;
void sf_cgen_sub_state_step(void)
{
 if (state == A) {
  if (cnt == 100) {
   state = B;
   substate = B1;
   out = 1;
  } else {
   cnt = (uint8_T)(cnt + 1);
  }
 } else {
  switch (substate) {
   case B1:
    if (in > 1) {
     substate = B2;
    }
    break;

   case B2:
    if (in <= 1) {
     substate = B1;
     out = 1;
    } else if (in > 2) {
     substate = B3;
    } else {
     out = 2;
    }
    break;

   default:
    if (in <= 2) {
     out = 3;
     substate = B2;
    }
    break;
  }
 }
}

void sf_cgen_sub_state_initialize(void)
{
 substate = B_None;
 state = A;
}
```

>> sf_cgen_stt.slx

27

グラフィカル関数

- グラフィカル関数プロパティの**関数インラインオプション**からインライン or 関数のどちらにするか選択できます（Autoは両者の自動選択）。関数名は「モデル or サブシステム名_グラフィカル関数名」となります。
- 下記チャートプロパティを有効にするとグラフィカル関数を他のチャートから使用することができます。関数名は「グラフィカル関数名」となります。
  チャートレベルの関数をエクスポート -> エクスポートされた関数をグローバルに可視として扱う



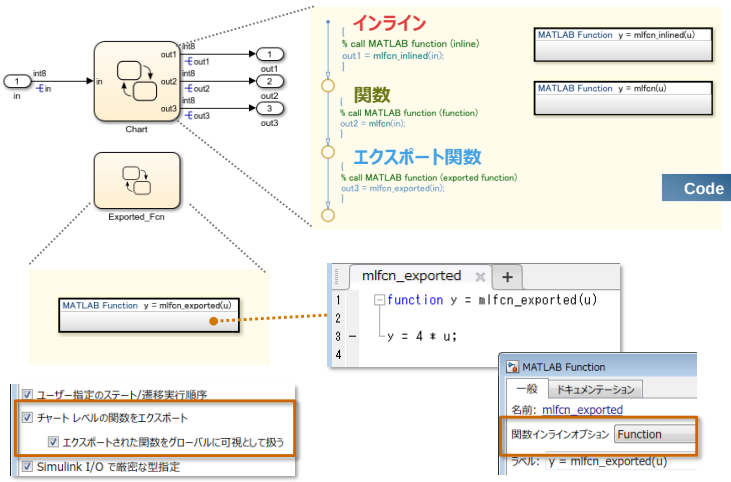

```
extern void gfcn_exported(int8_T u, int8_T *y);
static int8_T sf_cgen_graphical_fcn_gfcn(int8_T u);
static int8_T sf_cgen_graphical_fcn_gfcn(int8_T u)
{
  return (int8_T)(u * 3);
}

Void gfcn_exported(int8_T u, int8_T *y)
{
  *y = (int8_T)(u * 4);
}

void sf_cgen_graphical_fcn_step(void)
{
  int8_T tmp;
  out1 = (int8_T)(in * 2);        インライン
  out2 = sf_cgen_graphical_fcn_gfcn(in);  関数
  gfcn_exported(in, &tmp);        エクスポート関数
  out3 = tmp;
}
```

>> sf_cgen_graphical_fcn.slx

29

MATLAB関数

グラフィカル関数と同様の設定を行うことができます。




```
extern void mlfcn_exported(int8_T u, int8_T *y);
static int8_T sf_cgen_ml_fcn_mlfcn(int8_T u);
static int8_T sf_cgen_ml_fcn_mlfcn(int8_T u)
{
  return (int8_T)(3 * u);
}

void mlfcn_exported(int8_T u, int8_T *y)
{
  *y = (int8_T)(u * 4);
}

void sf_cgen_ml_fcn_step(void)
{
  int8_T tmp;
  out1 = (int8_T)(in * 2);          インライン
  out2 = sf_cgen_ml_fcn_mlfcn(in);  関数
  mlfcn_exported(in, &tmp);         エクスポート関数
  out3 = tmp;
}
```

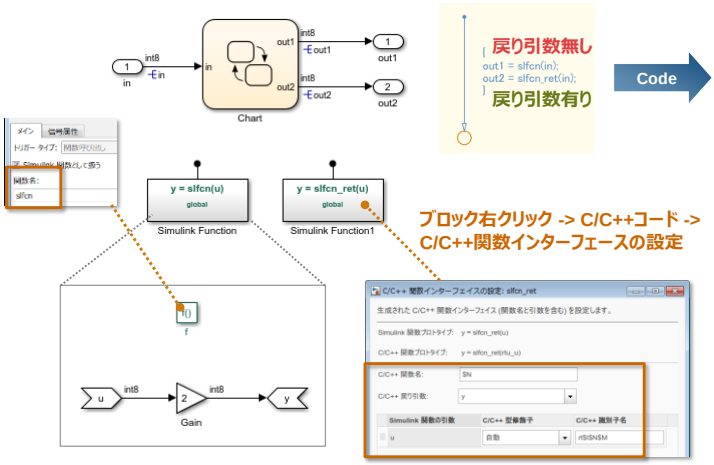
>> sf_cgen_ml_fcn.slx

30

MathWorks

Simulink関数

Simulink関数ブロックプロパティから関数インターフェースを設定することができます。関数名は「Simulink関数名」となります。
コンテキストメニュー -> C/C++コード -> C/C++関数インターフェースの設定




```
void slfcn(int8_T rtu_u, int8_T *rty_y)
{
  *rty_y = (int8_T)(rtu_u * 2);
}

int8_T slfcn_ret(int8_T rtu_u)
{
  return (int8_T)(3 * rtu_u);
}

void sf_cgen_sl_fcn_step(void)
{
  int8_T tmp;
  slfcn(in, &tmp);    戻り引数無し
  out1 = tmp;
  out2 = slfcn_ret(in);    戻り引数有り
}
```

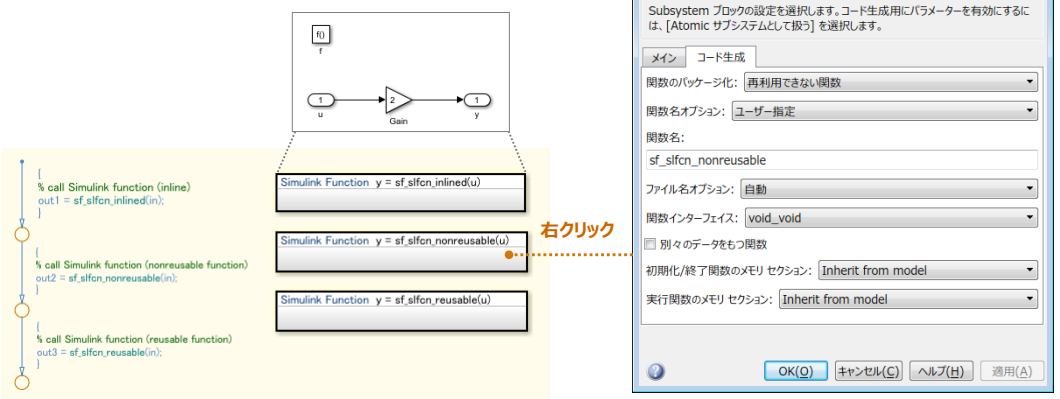
>> sf_cgen_sl_fcn.slx

MathWorks

Stateflow内Simulink関数　1/2

Simulink関数パラメータからコード生成設定を行うことができます。下記は代表的なパラメータです。

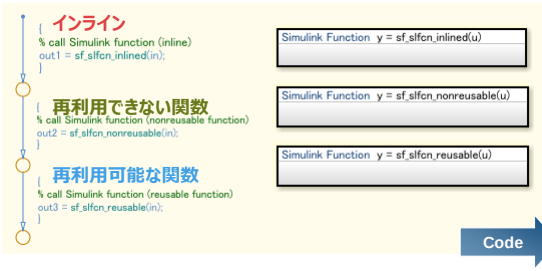
- 関数のパッケージ化：インライン / 再利用できない関数 / 再利用可能な関数
- 関数名：自動 / サブシステム名 / ユーザー指定
- ファイル名：自動 / サブシステム名 / 関数名 / ユーザ指定
- 関数インターフェース：void_void / 引数を許可



>> sf_cgen_sf_sl_fcn.slx

MathWorks

Stateflow内Simulink関数 2/2

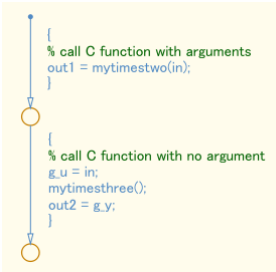


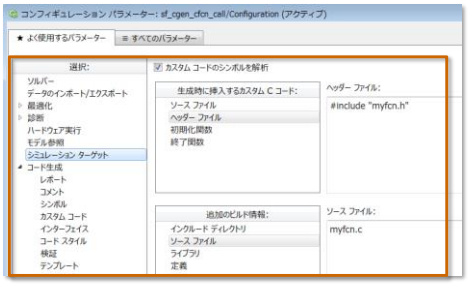

```
DW_sf_cgen_sf_sl_fcn_T sf_cgen_sf_sl_fcn_DW;
extern void sf_slfcn_reusable(int8_T rtu_u, int8_T *rty_y);
extern void sf_slfcn_nonreusable(void);
void sf_slfcn_nonreusable(void)
{
  y = (int8_T)(3 * u);
}

void sf_slfcn_reusable(int8_T rtu_u, int8_T *rty_y)
{
  *rty_y = (int8_T)(rtu_u * 4);
}

void sf_cgen_sf_sl_fcn_step(void)
{
  out1 = (int8_T)(in * 2);    インライン
  u = in;
  sf_slfcn_nonreusable();  再利用できない関数（void_void）
  out2 = y;
  sf_cgen_sf_sl_fcn_DW.u_l = in;
  sf_slfcn_reusable(sf_cgen_sf_sl_fcn_DW.u_l, &sf_cgen_sf_sl_fcn_DW.Gain);
  out3 = sf_cgen_sf_sl_fcn_DW.Gain;                    再利用可能な関数
}
```


MathWorks

外部C関数の呼び出し 1/2

- **モデルコンフィギュレーションパラメータ -> シミュレーションターゲット** に ソース／ヘッダファイルやインクルードディレクトリ等を設定することで、 Stateflowチャートからc関数を呼び出すことができます。
- コード生成を行うと関数呼び出し処理がそのまま生成されます。
- 外部ソースで定義されたグローバル変数をStateflowデータとして定義せずに 外部参照することができます。





myfcn.c

```
#include "myfcn.h"

signed char g_u;
signed char g_y;

signed char mytimestwo(signed char u)
{
  return 2 * u;
}

mytimesthree()
{
  g_y = 3 * g_u;
}
```

myfcn.h

```
extern signed char g_u;
extern signed char g_y;

extern signed char mytimestwo(signed char);
extern mytimesthree();
```

>> sf_cgen_cfcn_call.slx

外部C関数の呼び出し 2/2

```
{
% call C function with arguments
out1 = mytimestwo(in);
}
```

関数引数経由のデータ交換

```
{
% call C function with no argument
g_u = in;
mytimesthree();
out2 = g_y;
}
```

グローバル変数経由のデータ交換
外部コード上のグローバル変数（g_u, g_y）を
Stateflowデータに代入して利用します

Code

```
out1 = (int8_T)mytimestwo(in);

g_u = in;
mytimesthree();
out2 = (int8_T)g_y;
```

35

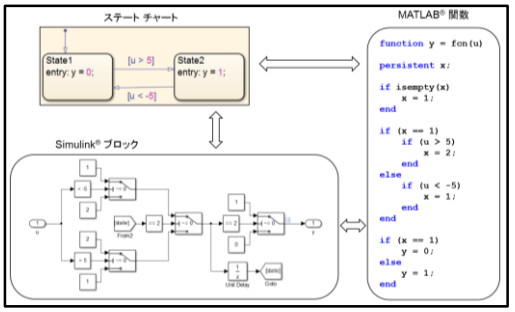
トレーニングコースのご紹介
Stateflow

フローチャートやステートマシンをシミュレーションに活用！

自動車分野向け
Stateflow 基礎

Simulink では表現が苦手な条件分岐、繰り返し処理、モードロジックやスケジューリングのアルゴリズムは、Stateflowを使って表現することにより、見た目やメンテナンス性がよくなるばかりか、モデリング効率も格段に上がります。
本トレーニングは、流れ図（フローチャート）や状態遷移図（ステートマシン）を用いて効率よくモデルを作成し、シミュレーションする方法を学びます。
また、講師の指導の元、自動車分野で使用される実践的な例題や、多くの演習を解くことにより短期間で Stateflow の使い方・効率的なモデリング、デバッグの方法を理解することができ、受講者は即座に業務で活用できるようになります。

Stateflow により、シンプルな表現が可能



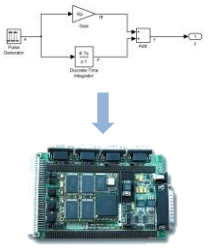
※ 本コースの受講には、Simulink の基礎知識が必須です

スケジュール（目安）：

日程	9	10	11	12	1	2	3	4	5
1日目	流れ図のモデル化			ステートマシンのモデル化			階層的ステート	パラレルステート	
2日目	チャートでのイベントの使用			チャートからの関数呼び出し		真理値表 状態遷移表		Stateflowの設計上の注意事項	

36

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	

37