

# 仕向け切り替えのための自動生成コードカスタマイズ

## 本日の内容

1. はじめに：自動生成コードのコード効率について
2. Variantサブシステムについて
3. Variantサブシステムからのコード生成
4. 最適なROM/RAM消費実現のための  
カスタマイゼーション
5. まとめ(効果と制限)

株式会社デンソー

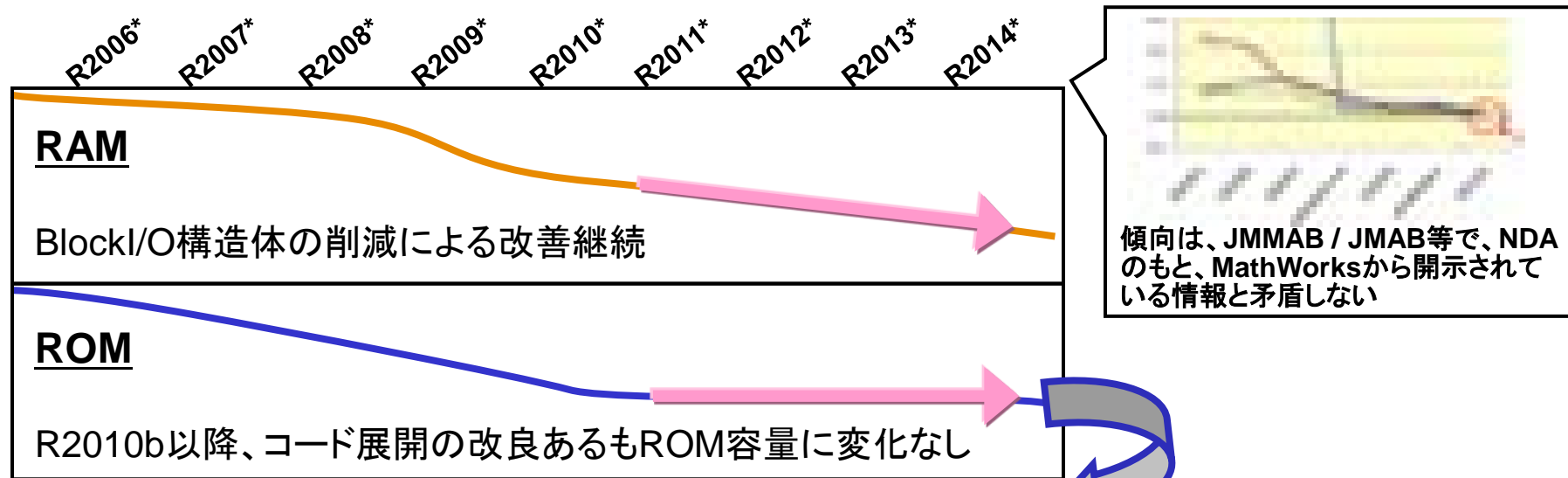
電子基盤システム開発部  
モデルベース技術開発室

福田 勝吉



# 1. はじめに: 自動生成コードのコード効率について

## 弊社の使い方に置けるコード効率(ROM/RAM容量)の改善傾向



## 弊社の使い方でコード効率が悪化する要因

### 1. ポインタ → 我慢

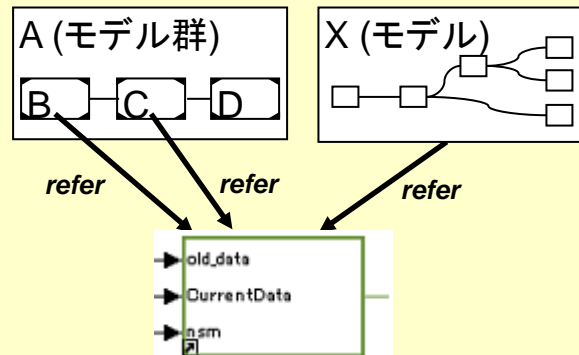
<ハンドコード>

```
if (***) { ptt_map = &Amap[0];  
} else { ptt_map = &Bmap[0]; }  
out = LookUp(in, ptt_map);
```

<オートコード>

```
if (***) {out = LookUp(in, &Amap[0]);}  
} else { out = LookUp(in, &Bmap[0]); }
```

### 2. モデルを跨いだ使い回し → reusable駆使



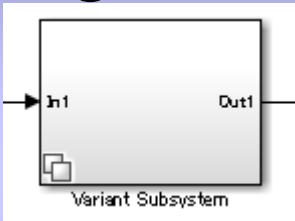
### 3. 複雑な仕向け切替 → Variant Subsystem だけではプロジェクト運用に支障 → 最低限必要な 機能を、カスタマイズ により実現 (本日の説明)

## 使い方

- ① VariantサブシステムをSimulinkライブラリからコピー
- ② Variant下に仕向け分のサブシステムを作成
- ③ Variantに仕向け条件式を設定(右クリックでGUI起動)
- ④ WSに条件式に利用された定数を定義

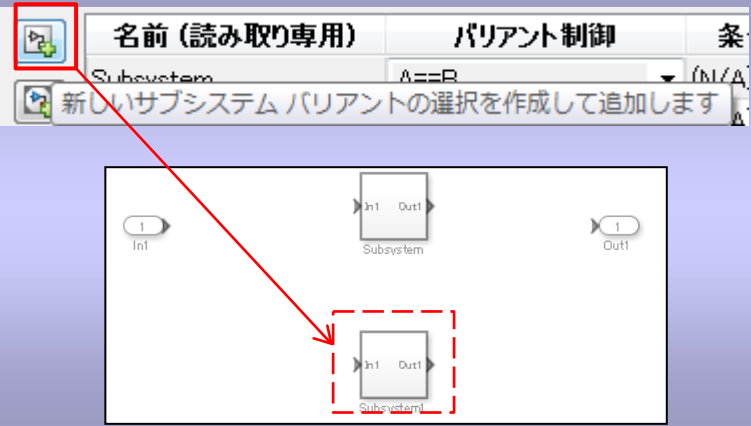
例 : - SW1 = Simulink.Paramter; SW1.Value = 1;

①



Variant Subsystem

②

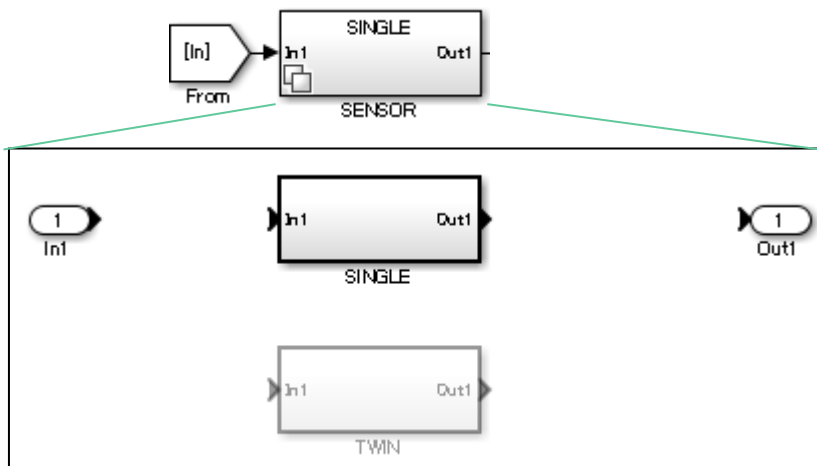


(Variantサブシステム設定GUI)

③

名前 (読み取り専用)	バリエーション制御	条件 (読み取り専用)
Subsystem	SW1==SW2	(N/A)
Subsystem1	(default)	(N/A)

(Variantサブシステム設定GUI)



**Function ブロック パラメーター: SENSOR**

Variant Subsystem

Variant Subsystem ブロックは、シミュレーション用にアクティブなバリエーションを 1 つ含めることができます。バリエーション制御は、どのバリエーションがアクティブであるかを決定します。バリエーション制御は、条件式、条件式を指定する Simulink Variant オブジェクト、既定のバリエーションのいずれかになります。

バリエーションの選択 (子サブシステムまたは Model ブロックのリスト)

前 (読み取り専用)	バリエーション制御	条件 (読み取り専用)
<input type="checkbox"/>	SINGLE SW == SINGLE	(N/A)
<input type="checkbox"/>	TWIN SW == TWIN	(N/A)
<input checked="" type="checkbox"/>	else (default)	の選択も可

バリエーション条件をオーバーライドして以下のバリエーションを使用     コード生成

バリエーション: JESW == u1g\_EJCC\_SINGLE (SINGLE)     プリプロセッサの条件を生成

[バリエーション マネージャーでブロックを開く](#)

OK(Q)    キャンセル(O)    ヘルプ(H)    適用(A)

## インライン展開や#elifの生成が可能

```
#if SW == SINGLE↓
    s2t_Sw1 = variantsample_B.USE_b.iram[0];↓
#elif SW == TWIN↓
    s2t_Sw1 = ((variantsample_B.USE_b.iram[0]) +↓
                (variantsample_B.USE_b.iram[1]));↓
#endif
```

## 関数(サブシステム)の引数渡しも可能

```
#if SW == SINGLE↓
    variantsample_SINGLE(variantsample_B.USE_b.iram,↓
                          &s2t_Sw1);↓
#elif SW == TWIN↓
    variantsample_TWIN(variantsample_B.USE_b.iram,↓
                       &s2t_Sw1);↓
#endif
```

## その他: Variantマネージャによる設計補助機能

- ・モデル全体のコンパイルSWの可視化
- ・変数・コンフィギュレーションの管理
- ・矛盾のチェック

バリエーション マネージャー: variantsample\_R2014b0K

バリエーションコンフィギュレーション データ

モデルの階層構造 (ベースワークスペースを使用)

名前: variantsample\_R2014b0K

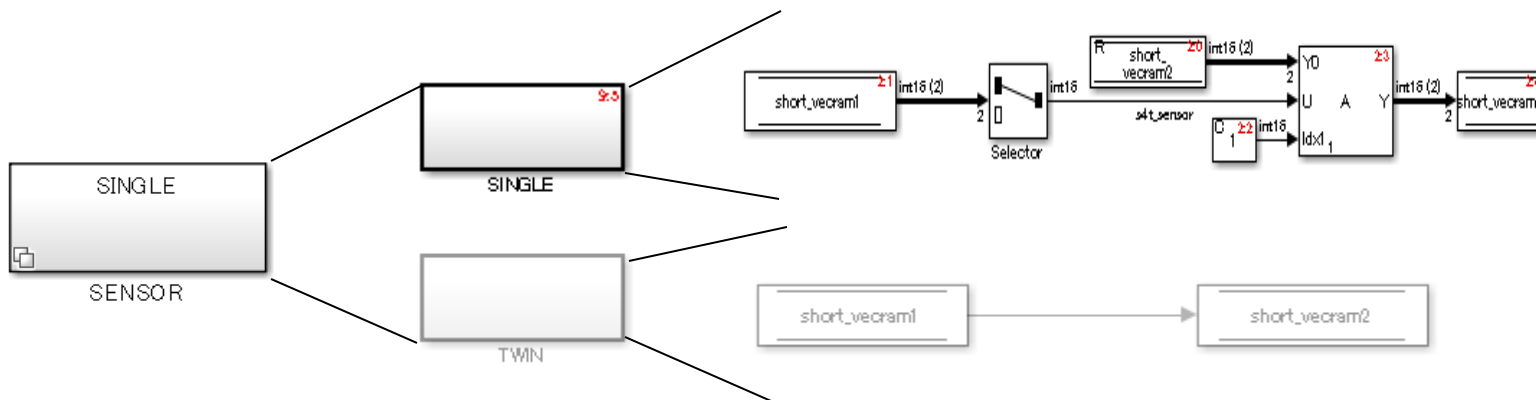
- template
  - template\_8msh1\_Flp
    - VAR11
      - USE
        - SENSOR
          - SINGLE
            - TWIN
              - typeS
                - Subsystem
                  - Subsystem

バリエーション制御

- VAR11 == USE
- SENSOR == SINGLE
- SENSOR == TWIN
- VAR12 == NOT\_USE

# 3-1. Variantサブシステムからのコード生成

モデル



生成コード

```
signed short short_vecram1[2];
signed short short_vecram2[2];
```

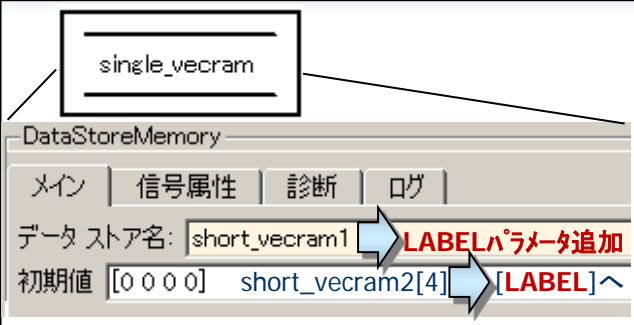
```
#if SENSOR == SINGLE
    short_vecram2[0] = short_vecram1[0];
#elif SENSOR == TWIN
    short_vecram2[0] = short_vecram1[0];
    short_vecram2[1] = short_vecram1[1];
#endif /* SENSOR == SINGLE */
```

# 3-2. Variantサブシステムからのコード生成(ユーザ要望)

|              |  |
|--------------|--|
| <p>モデル</p>   |  |
| <p>生成コード</p> | <pre>signed short short_vecram1[2]; signed short short_vecram2[2];</pre> <p style="text-align: center;">➡</p> <pre>signed short short_vecram1[LABEL]; signed short short_vecram2[LABEL];</pre> <div style="border: 1px solid black; padding: 5px; display: inline-block; margin: 10px auto;">変数サイズのラベル置き換え</div> <pre>for (t_i1 = 0; t_i1 &lt; (LABEL); t_i1++) {     short_vecram2[t_i1] = short_vecram1[t_i1]; }</pre> <p style="text-align: center;">➡</p> <pre>#if SENSOR == SINGLE     short_vecram2[0] = short_vecram1[0]; #elif SENSOR == TWIN     short_vecram2[0] = short_vecram1[0];     short_vecram2[1] = short_vecram1[1]; #endif /* SENSOR == SINGLE */</pre> <div style="border: 1px solid black; padding: 5px; display: inline-block; margin: 10px auto;">定数ラベルへのプリプロセッサ切り替え追加</div> |

# 4. 最適なROM/RAM消費実現のためのカスタマイゼーション

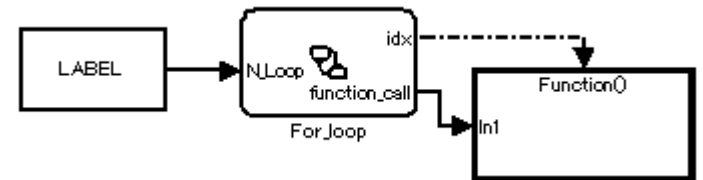
変数サイズの  
ラベル置き換え



LABEL指定するパラメータを設け、  
独自ツールにて書き換え

```
signed short  short_vecram1[LABEL];
signed short  short_vecram2[LABEL];

for (t_i1 = 0; t_i1 < (LABEL); t_i1++) {
    short_vecram2[t_i1] = short_vecram1[t_i1];
}
```



定数ラベルへの  
プリプロセッサ  
切り替え追加

```
自動生成コード
#define LABEL 1

const calibration_DATA = 1;

プリプロセッサ追加用ファイル
#if SENSOR == SINGLE
    LABEL = 1;
#elif SENSOR == TWIN
    LABEL = 2;
#endif

#if DEST == USA
    calibration_DATA = 1;
#elif DEST == JPN
    calibration_DATA = 2;
#endif
```

+

独自  
ツール

```
マクロ定数
#if SENSOR == SINGLE
    #define LABEL 1
#elif SENSOR == TWIN
    #define LABEL 2
#endif

Const定数
#if DEST == USA
    const calibration_DATA = 1;
#elif DEST == JPN
    const calibration_DATA = 2;
#endif
```

# 5. まとめ(効果と制限)

変数サイズの  
ラベル置き換え

**効果**

- 変数サイズの動的確保  
(常に最大値を確保する運用  
に比べRAMの効率利用)  
short\_vecram1[LABEL];
- 配列違いのコードを同一ロ  
ジックにまとめられる事による  
検査効率の向上  
for (t\_i1 = 0; t\_i1 < LABEL; t\_i1++)  
{short\_vecram2[t\_i1] }

定数ラベルへの  
プリプロセッサ  
切り替え追加

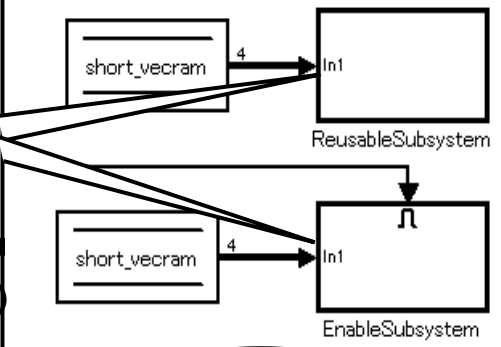
- 同一のConstant定数名の  
複数仕向け適用  
(仕向毎にラベル定義するの  
に比べROMの効率利用)  
#if DEST == USA  
tempram =calibration\_USA;  
#elif DEST == JPN  
tempram =calibration\_JPN;
- 従来ハンドコードからの同  
一ラベル名での移植

## 制限(個別アプローチの限界)

**関数引数のベクトルの扱い**

関数コール側はOKだが、  
Reusablefunc(vecram);  
Enablefunc(en,vecram);

関数実体ではBlock/Oに、  
サイズ情報が含まれNG  
void func(const f4 rtu\_In1[4])  
{  
vecram[0] = rtu\_In1[0];  
...  
vecram[0] = rtu\_In1[4];



このような機能が  
将来、標準サポート  
される事を希望

<プリプロセッサ追加用ファイル>

```
#if DEST == USA
DATA1 = 1;
#elif DEST == JPN
DATA1 = 2;
#endif

#if DEST == USA
DATA2 = DATA1; ←ファイル内で定義済みのラベル参照
#elif (DEST != USA) && (DEST == JPN) && (DEST == EUR) ←整合性
DATA1 = DATA2; ←矛盾や間違い
#endif
```

**※生成したコードのModifyでは、ユーザーの複雑な設定への対応に限界。(モデル情報を用いたチェックが必要)**