SAE TECHNICAL PAPER SERIES

2004-01-0269

Multi-Target Modelling for Embedded Software Development for Automotive Applications

Grantley Hodge, Jian Ye and Walt Stuart
Visteon Corporation

Reprinted From: In-Vehicle Networks and Software, Electrical Wiring Harnesses, and Electronics and Systems Reliability (SP-1852)

SAE International™

2004 SAE World Congress
Detroit, Michigan
March 8-11, 2004

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-5760 Web: www.sae.org

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

For permission and licensing requests contact:

SAE Permissions
400 Commonwealth Drive
Warrendale, PA 15096-0001-USA
Email: permissions@sae.org
Fax: 724-772-4891
Tel: 724-772-4028

For multiple print copies contact:

SAE Customer Service
Tel: 877-606-7323 (inside USA and Canada)
Tel: 724-776-4970 (outside USA)
Fax: 724-776-1615
Email: CustomerService@sae.org

ISBN 0-7680-1319-4
Copyright © 2004 SAE International

Positions and opinions advanced in this paper are those of the author(s) and not necessarily those of SAE. The author is solely responsible for the content of the paper. A process is available by which discussions will be printed with the paper if it is published in SAE Transactions.

Persons wishing to submit papers to be considered for presentation or publication by SAE should send the manuscript or a 300 word abstract of a proposed manuscript to: Secretary, Engineering Meetings Board, SAE.

Printed in USA

2004-01-0269

Multi-Target Modelling for Embedded Software Development for Automotive Applications

Grantley Hodge, Jian Ye and Walt Stuart
Visteon Corporation

Copyright © 2004 SAE International

ABSTRACT

Manual 'porting" of source code is often required in order to "reuse" control software in different applications with different target hardware. This process is not cost effective. Maintaining multiple "versions" of the same software also causes problems. This paper describes a way in which multiple target source code can be generated from a single model. A custom data class is developed so that it can be used to define both signal and parameter data types necessary for data dictionary-driven models. This capability allows a single model to be used to generate code for multiple target hardware architectures. A software development process using a generic model to support multiple hardware targets is compared with the hand porting process (e.g. floating-point to/from fixed-point). Auto code generation from a sample multi-target feature model will be presented. The efficiency of the auto code will also be discussed.

INTRODUCTION

Model-based software development is gaining acceptance in the automotive industry. The virtues of modeling and simulating control software are well known. However, it is desirable for a powertrain control system to be reused on different hardware architectures. This results in the need to have unique attributes within the model to accommodate these architecture differences. Unfortunately, this leads to proliferation and lack of reuse at the model level.

Visteon Powertrain has adopted a model-based software process which significantly reduces the difficulty experienced in supporting different hardware architectures by supporting multi-target auto-coding. This process utilizes multiple data dictionaries for the same base model, thus allowing the same control algorithms to be used with microprocessors that support only certain data types.

Visteon Powertrain is actively developing enhancements to the MATLAB® environment so that the code generated from the models can be easily re-targeted to different processor architectures. In addition, these tools are designed to integrate auto-code easily with existing legacy software. Tools have been developed in the following three areas:

1. Auto Code Customization Tools---tools for generating C code in a style that can be integrated in a legacy software environment. Typically, legacy software imposes many constraints on auto-coding, making it difficult to evolve to a model-based process. Visteon Powertrain has solved this limitation.
2. Powertrain Simulink® and Stateflow® library---developed some specific utility libraries for powertrain software functions such as filters, timers, etc..
3. Data Dictionary Tools---developed a way to generate a data dictionary in the modeling environment.

In this paper, the data dictionary tools defining the data class for multi-target modeling will be discussed. The Auto Code Customization tools will be used for automatic code generation.

MANUAL PORTING PROCESS

A traditional software process supporting a variety of hardware architectures is error-prone. This is particularly true when supporting application on both floating point and fixed point microcontrollers. First, fixed-point scaling for all variables and parameters must be determined. The code is then "ported" by re-coding mathematical operations based on the scaling information. In the late stages of development, a change of scaling information may involve significant effort to implement the code changes, particularly if the signal or parameter is referenced in numerous components.

In a manual porting process, there are at least two issues:

1. It takes significant effort to establish the scaling information.

2. Testing of fixed-point overflow/underflow is tedious. Every mathematical operation requires analysis and testing with extreme values to establish the need for overflow/underflow checks.

The above two issues are easily resolved using multi-target modeling. Simulink modeling provides ways to "auto-scale" and to find out where overflow/underflow might occur.

MULTI-TARGET MODELLING

A multi-target model is a model which is independent of data type (i.e., a hardware-independent model). To make the model target-specific, a specific target data dictionary must be loaded into the MATLAB® workspace, and a mechanism to link the data dictionary information and the generic model must be established.

Figure 1 shows a single generic model with two different target data dictionaries.

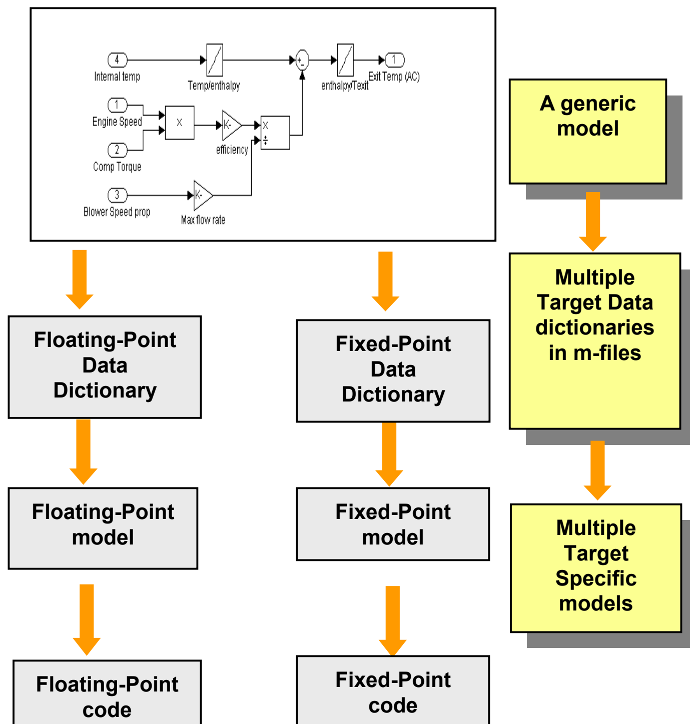

Figure 1: A Multi-target Model with two Target Data Dictionaries.

Automatic code can be generated for both floating-point and fixed-point code from the Target Specific Models. No manual "porting" of floating-point to/from fixed point is required.

MODEL-BASED DATA DICTIONARY

A model-based data dictionary is a way to specify data item information in the MATLAB® workspace. Using the MATLAB default data class definitions, one can define a data item such as:

xyz=Simulink.Signal;

xyz effectively becomes an object in the workspace. However, xyz has limited attributes. Data type, for example, is not in the attribute list. In the next section, we show a customized Data Class design that has all the important attributes for typical powertrain software, such as data type, data scaling, unit, min, max values, etc.

CUSTOM DATA CLASS

A Custom Data Class for powertrain applications was created such that a parameter is now instantiated as follows:

ABC=Visteon.CustomParameter('ABC');

All the attributes will be populated using default values. The user can specify the attributes, for example,

ABC.Value=100;
ABC.Data_Type='U8';

The data type 'U8' is Visteon's data type definition for "unsigned char". Obviously the MATLAB models do not know the meaning of "U8", but a special data class script automatically generates two workspace variables:

<Parameter_or_Signal_name>_Data_Type
<Parameter_or_Signal_name>_Data_Scaling

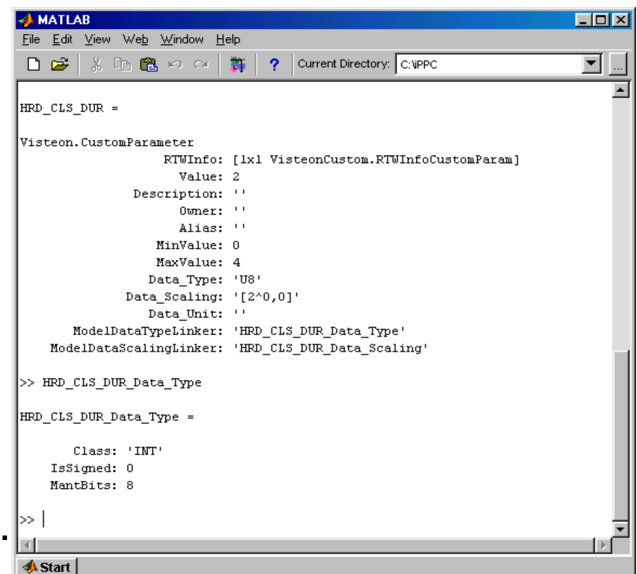

Figure 2: Use of Visteon Data Class

The <Parameter_or_Signal_name>_Data_Type contains the native MATLAB data type as shown in Figure 2.

To change attributes for a data item, one can also use a GUI. Figure 3 shows a GUI for the Visteon Data Class

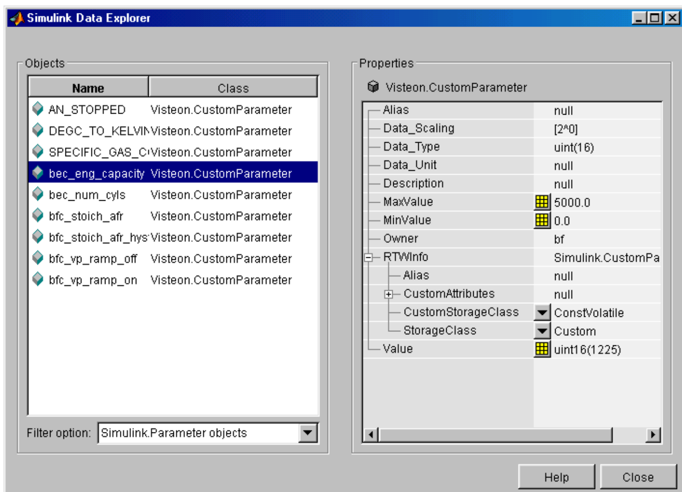

Figure 3: The Visteon Data Class GUI.

LINKING DATA DICTIONARY TO MODEL

One of the advantages of using a data dictionary is to have a generic model with different data dictionary files. By loading an appropriate data dictionary, the generic model can be transferred into a target-specific model. To achieve that goal, we have to have linkages between data type and data scaling in the data dictionary and in the model.

As mentioned before, all data items defined as Visteon data class will automatically have the following information generated:

< Parameter_or_Signal_name >_Data_Type
< Parameter_or_Signal_name >_Data_Scaling

To link the model to the data dictionary, these symbolic names are used in attribute fields for model blocks instead of literal values. Figure 4 shows an example.

OTHER ADVANTAGES OF USING A DATA DICTIONARY

- Central repository for data containing detailed design specifics of both calibration parameters and global/local variables.
- Flexibility of including target-specific information independent of the model to accommodate multi-target models, such as data types, scaling, etc.
- The data dictionary determines how the data is coded in the auto-generated code.
- Automated generation of ASAP2 file template for calibration tools or a specific file type as required.

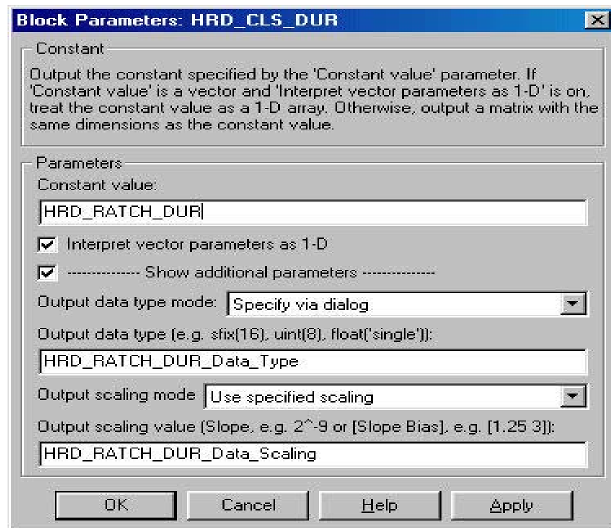

Figure 4: Linking Data Dictionary to Model.

CASE STUDIES

The following model is a component in a powertrain software application. This model can be used for generating floating-point code or fixed-point code depending on which data dictionary is loaded into the workspace.

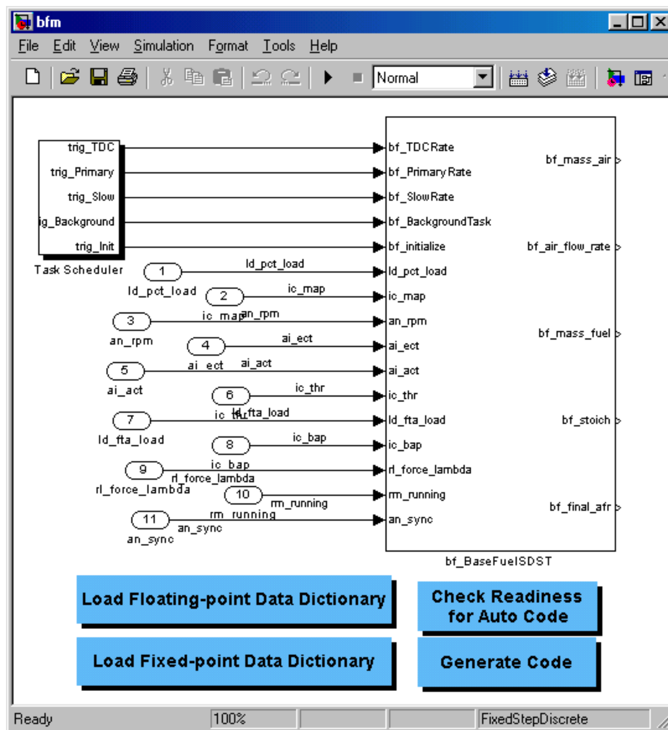

Figure 5: A Component Multi-target Model with Two Data Dictionaries.

DATA DICTIONARY

The following examples of Visteon data dictionary format are defined as an m-file format.

Calibration Constants

Floating-point data dictionary	bfc_vp_ramp_off=Visteon.CustomParameter('bfc_vp_ramp_off');
	bfc_vp_ramp_off.RTWInfo.CustomStorageClass='ConstVolatile';
	bfc_vp_ramp_off.Description='';
	bfc_vp_ramp_off.Owner='bf';
	bfc_vp_ramp_off.Alias='';
	bfc_vp_ramp_off.MinValue=0;
	bfc_vp_ramp_off.MaxValue=0.2;
	bfc_vp_ramp_off.Value=0.1;
	bfc_vp_ramp_off.Data_Type='F32';
	bfc_vp_ramp_off.Data_Scaling='1';

Fixed-point	bfc_vp_ramp_off=Visteon.CustomParameter('bfc_vp_ramp_off');
	bfc_vp_ramp_off.RTWInfo.CustomStorageClass='ConstVolatile';
	bfc_vp_ramp_off.Value=0.1;
	bfc_vp_ramp_off.Data_Type='S16';
	bfc_vp_ramp_off.Data_Scaling='0.00191';

Note that in the floating-point data dictionary, the Data_Scaling attribute will always be 1. The CustomStorageClass determines how the parameter is to be coded.

#Define Constants

Floating-point data dictionary	SPECIFIC_GAS_CONSTANT=Visteon.CustomParameter('SPECIFIC_GAS_CONSTANT'); SPECIFIC_GAS_CONSTANT.RTWInfo.CustomStorageClass='Define'; SPECIFIC_GAS_CONSTANT.Value=0.286; SPECIFIC_GAS_CONSTANT.Data_Type='F32'; SPECIFIC_GAS_CONSTANT.Data_Scaling='1';
Fixed-point	SPECIFIC_GAS_CONSTANT=Visteon.CustomParameter('SPECIFIC_GAS_CONSTANT'); SPECIFIC_GAS_CONSTANT.RTWInfo.CustomStorageClass='Define'; SPECIFIC_GAS_CONSTANT.Value=0.286; SPECIFIC_GAS_CONSTANT.Data_Type='U16'; SPECIFIC_GAS_CONSTANT.Data_Scaling='[2^-11]';

Global Variables

Floating-point data dictionary	bf_mass_fuel=Visteon.Signal('bf_mass_fuel');
	bf_mass_fuel.RTWInfo.StorageClass='ExportedGlobal';
	bf_mass_fuel.Data_Type='F32';
	bf_mass_fuel.Data_Scaling='1';
	
	ic_map=Visteon.Signal('ic_map');
	ic_map.RTWInfo.StorageClass='ImportedExtern';
	ic_map.Data_Type='F32';
	ic_map.Data_Scaling='1';

Fixed-point	bf_mass_fuel=Visteon.Signal('bf_mass_fuel');
	bf_mass_fuel.RTWInfo.StorageClass='ExportedGlobal';
	bf_mass_fuel.Data_Type='U16';
	bf_mass_fuel.Data_Scaling='[2^-8,0]';
	
	ic_map=Visteon.Signal('ic_map');
	ic_map.RTWInfo.StorageClass='ImportedExtern';
	ic_map.Data_Type='U16';
	ic_map.Data_Scaling='[2^-8,0]';

Note: 'ImportedGlobal' means external globals, while 'ExportedGlobal' means global defined in the current feature.

Other Data Structures

It should be pointed out that there are many other data structures supported in our Visteon Data Dictionary Tools, such as:

- Structure
- Bit-field structure
- KAM variables (forces certain variables to code in Keep alive memory)
- Static variables
- Local Variables (forces certain variables to code as locals)
- Others

AUTOMATIC CODE GENERATION

Custom Code Options

To generate code that is compatible with the code style in the legacy code environment, some Visteon code options are added as shown in Figure 6.

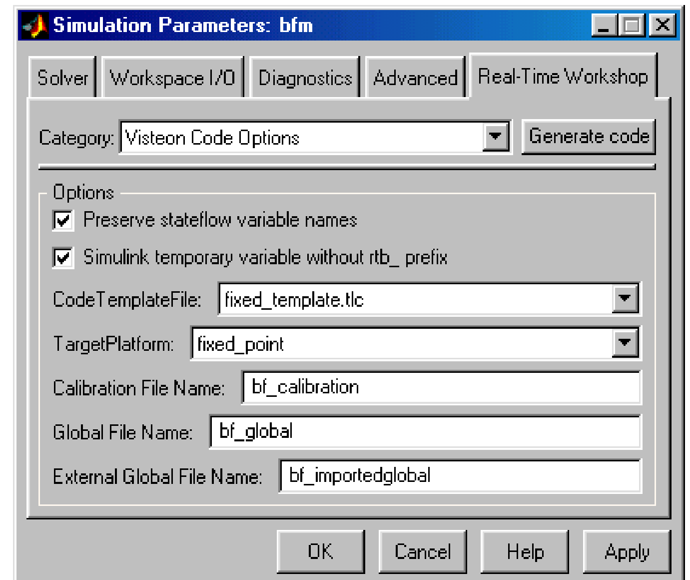

Figure 6: Visteon Custom Code Options.

Auto Code Samples

The following table shows how the example signals and parameters in the Data Dictionary section are shown in the actual code.

Floating-point code

```
 in bf_calibration.c file
CAL F32 bfc_vp_ramp_off = 0.1F;

 in bf_calibration.h file
#define SPECIFIC_GAS_CONSTANT          (0.286F)

 in bf_global.c file
F32 bf_mass_fuel;

 in bf_importedglobal.h file
extern F32 ic_map;
```

Fixed-point code

```
in bf_calibration.c file
CAL S16 bfc_vp_ramp_off = 52;

in bf_calibration.h file
#define SPECIFIC_GAS_CONSTANT          (586U)

in bf_global.c file
U16 bf_mass_fuel;

in bf_importedglobal.h file
extern U16 ic_map;
```

The following is a specific subsystem within the example implementing an ideal gas equation .

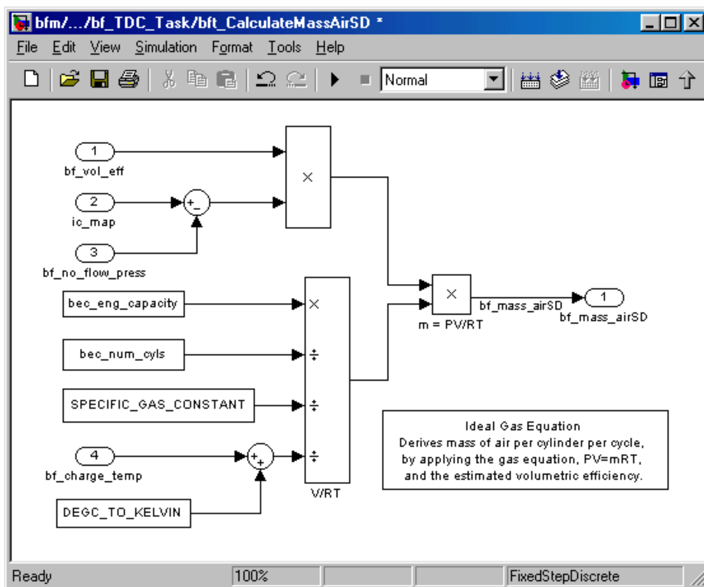

Figure 7: A Portion of a Feature Model.

Floating-point code

```
void ideal_gas_equation(void)
{

  bf_mass_airSD = (bf_vol_eff * (ic_map - bf_no_flow_press)) *
(bec_eng_capacity/ bec_num_cyls / SPECIFIC_GAS_CONSTANT /
(bf_charge_temp + DEGC_TO_KELVIN));
}
```

Fixed-point code

```
void ideal_gas_equation(void)
{

  S32 Product_d;
  U16 V_RT;
  S16 temp26;

  temp26 = ((S16)LSRu(1,ic_map));
  temp26 -= ASR(1,bf_no_flow_press);

  MUL_S32_U16_S16(Product_d,bf_vol_eff,temp26);

  temp26 = bf_charge_temp;
  temp26 += ((S16)DEGC_TO_KELVIN);

  DIV_U16_U16_U8(V_RT,bec_eng_capacity,bec_num_cyls);
  DIV_U16_U16_U16_SL11(V_RT,V_RT,SPECIFIC_GAS_CONSTANT);
  DIV_U16_U16_S16_FLOOR(V_RT,V_RT,temp26);

  MUL_U16_S32_U16_SR16_SAT(bf_mass_airSD,Product_d,V_RT);
}
```

The above floating-point auto code shows that all the expressions are folded into one big statement. This is the result of the so-called “Expression Folding” option in the Embedded Coder®. The fixed-point auto code of course results in more lines of code because it requires fixed-point manipulation. The fixed-point arithmetic macros are automatically generated in bf_macros.h file. The following table shows some of the macros.

```
#define LSRu(nBits,C) (((unsigned int)(C))>>(nBits))

#define ASR(nBits,C) ( (C)>>(nBits) )

#define MUL_S32_U16_S16(C,A,B) \
 { \
   C = (((int)(A)) * ((int)(B))); \
 } \

#define DIV_U16_U16_U8(C,A,B) \
 { \
      \
   if ( (B) == 0) \
   { \
        \
     (C) = (U16)((0xFFFFU)); \
   } \
   else \
   { \
        \
        \
     (C) = (U16)( ((unsigned)(A)) / ((unsigned)(B)) ); \
   } \
 } \
```

The Auto Code Efficiency

This component's auto-generated code has built with all other components in a powertrain application without errors. The code size comparison between hand code and auto code is show in Table 1.

Table 1: Code Size comparison between a fixed-point hand code and auto code.

		Code Size
Hand Code		928
Auto Code	No overflow/underflow check	904
	Check OF/UF everywhere	1562
	Check only where necessary	934

*Based on Tasking Compiler for ST10

The above table shows that the overflow/underflow checking increases code size significantly, so it is not practical to do this check everywhere. Fixed-point modeling is the best way to determine where the check is necessary. It is clear that the customized Embedded Coder® is able to generate efficient fixed-point code.

Table 2 shows ROM and RAM comparisons between hand code and auto code for a floating-point component in some typical powertrain software.

Table 2 ROM and RAM comparison between a floating-point hand code and auto code.

	Hand Code	Auto Code
ROM	6408	6192
RAM	132	112

The auto code has less size of ROM and RAM compared to that of hand code. The auto code is readable and peer reviewed, and checked with the QAC static analysis tool. Most importantly, the auto code is implemented in a real-world powertrain application.

CONCLUSION

A custom data class allowing data type and data scaling information to be incorporated into the model is discussed. Multi-target modeling provides a basis for software development for multiple targets. The advantage is faster and lower cost implementation by "re-using" the control algorithms or models. A target-specific code can be auto-generated with an appropriate target data dictionary. Visteon Auto Code Customization tools make the Embedded Coder® flexible and effective for generating production powertrain control software. Visteon Powertrain has demonstrated that model-based software development can generate quality software in less time, and the automatic code ROM & RAM sizes are equal to or better than hand written code.

ACKNOWLEDGMENTS

We'd like to acknowledge the helpful discussions in Visteon MBDG meetings with MBDG team members. The Table 2 results are from a work done by Sam Abihana at Visteon Powertrain.

REFERENCES

1. MATHWORKS press release about Visteon's model-based and auto coding capability. http://www.mathworks.com/company/pressroom/index.shtml/article/415
2. Jian Ye, Ph.D., Walt Stuart and Grantley Hodge, 'Model-Based Development of Visteon Powertrain Software', The Mathworks International Automotive Conference (IAC), 2003.
3. Mark Hsu, Maher El-Jaroudi, Eric Bender "Accelerated Life Cycle Development for Electronic Throttle Control Software using model-based/auto-code technology", SAE Paper, 2004

CONTACT

Jian Ye, jye5@visteon.com

Walt Stuart, wstuart@visteon.com

Grantley Hodge, ghodge@visteon.com

DEFINITIONS, ACRONYMS, ABBREVIATIONS

MBDG: Model-Based Development Group in Visteon Powertrain.

Multi-target Model (Generic Model): A model that contains no data type information (other than double precision).

Target Specific Model: A generic model with a target specific data dictionary loaded into the workspace and linked to the model.

ROM: Read Only Memory.

RAM: Random Access Memory.

KAM: Keep Alive Memory.

MATLAB®: A modeling environment consisting of a suite of software.

M-file: A Matlab file containing parameter specification as well as logic script.

Simulink®/Stateflow®: components of the MATLAB suite of tools.

•Embedded Coder® is a trademark of The Mathworks, Inc.

•MATLAB®, Simulink®, Stateflow®, and Real-Time Workshop are registered trademarks of The Mathworks, Inc.