Moving from Rapid Prototyping to Production

MathWorks Symposium
April 27, 2010
Ren Sang Nah
Who Should Attend

This seminar is intended for:

- Embedded software developers
- Control design engineers
- Systems engineers

Prerequisites:

- Knowledge of MATLAB and Simulink
- Familiarity with concepts of automatic code generation
- Familiarity with concepts of embedded software development
Agenda

1. Introduction

2. **Model architecture**
   - Controlling function prototypes
   - Exporting algorithm

3. **Data dictionary**
   - Controlling data types and storage classes
   - Convert floating-point data to fixed-point

4. **External functions and utilities**
   - Importing custom library functions
   - Importing external code

5. **Model configuration for production code generation**
   - Configuring the target settings
   - Setting code generation objectives and checking compliance

6. Wrap up
Traditional Rapid Prototyping

Simulink

Controller Model

Plant Model

Code Generation

Harness

xPC Target
Gap to Production

- Data design
  - Has less data to monitor
  - May need to control memory sections
  - May need to convert from floating point to fixed-point

- Code interface and reusability
  - Smaller memory footprint
  - May have to match certain calling interface

- Libraries and drivers
  - May want to utilize target-optimized libraries
  - Different drivers

- Target settings
  - Batch automation for building the application
On-Target Rapid Prototyping

Simulink

Controller Model

Plant Model

Code Generation

Harness

Embedded Target
## Rapid Prototyping Comparison

<table>
<thead>
<tr>
<th></th>
<th>Traditional</th>
<th>On-Target</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Useful for testing new ideas and green-field research</td>
<td>Useful for refinement and calibration of designs during development process</td>
</tr>
<tr>
<td><strong>Execution Hardware</strong></td>
<td>Uses PC or non-target HW</td>
<td>Uses ECU or near-production HW</td>
</tr>
<tr>
<td><strong>Code Efficiency, I/O latency</strong></td>
<td>Less emphasis on code efficiency and actual I/O latency</td>
<td>More emphasis on code efficiency and actual I/O latency</td>
</tr>
<tr>
<td><strong>Programs</strong></td>
<td>Works well for new vehicle programs</td>
<td>Works well for delta changes to existing programs</td>
</tr>
<tr>
<td><strong>Engineers</strong></td>
<td>Typically done by systems engineers in R&amp;D or advanced production</td>
<td>Typically done by systems and software engineers in production</td>
</tr>
<tr>
<td><strong>Cost and Convenience</strong></td>
<td>May require custom real-time simulators and hardware, or may be done with inexpensive “off-the-shelf” PC hardware and I/O cards</td>
<td>May use existing hardware, thus less expensive and more convenient</td>
</tr>
</tbody>
</table>
Target Integration Options

- **Full Executable Generation**
  - Target Products From MathWorks
  - Target Products From third parties
  - Build your own links or targets

- **Algorithm Export**
  - Plug the generated code into existing framework
  - Most popular option
Full Executable Generation

Notes:
- Complete executable generated with real-time scheduler
- Automated build via project creation or makefile
- Device driver and timer blocks

Controller Model

Generated Algorithm Code

Input Drivers

Comm Drivers

Optional Target Optimized Code

Output Drivers

Special Device Drivers

Scheduler/Operating System and Support Utilities

Communication Interfaces

Sensors

Actuators

Special Interfaces

Tuning

Notes:
- Complete executable generated with real-time scheduler
- Automated build via project creation or makefile
- Device driver and timer blocks
Link and Target Products

Embedded IDE Link
Makes it easy to verify generated and compiled object code using compiler tool chains (IDEs):
- Altium TASKING
- Green Hills MULTI
- TI’s Code Composer Studio
- Analog Devices VisualDSP++
- Eclipse

Target Support Package
Makes it easy to deploy generated code on particular microprocessors including:
- Freescale MPC5xx
- Infineon C166 family
- STMicroelectronics ST10 family
- TI’s C6000 and C2000
Target Integration Options

- Full Executable Generation
  - Target Products From MathWorks
  - Target Products From third parties
  - Build your own link or target

- Algorithm Export
  - Plug the generated code into existing framework
  - Most popular option
Algorithm Export

Controller Model

Generated Algorithm Code

Legacy Algorithm Code

Target Specific Algorithm Code

Input Drivers

Comm Drivers

Output Drivers

Special Device Drivers

Special Interfaces

Scheduler/Operating System and Support Utilities

Tuning

Actuators

Communication Interfaces

Sensors
Model Architecture

- Default code interface
- Controlling function prototypes
- Exporting algorithm
Architecture Component Management

- One aspect of an architecture is how the various system components are integrated and managed
  - Control how the overall system is componentized
  - Control the interface of each component
  - Control the scheduling of each component

- Approaches to support component management:
  - Subsystem/Atomic subsystem
  - Model reference (Model block)

- Ways to control execution sequence:
  - Intrinsic block sort order
  - Conditionally executed subsystem
  - Function call subsystem
Default Code Architecture and Interface

• For single task models (either single or multi-rate), the following entry points are generated:
  ▪ `model_initialize()` – Resets memory and initializes data prior to first time step
  ▪ `model_step()` – Executes one time step
  ▪ `model_terminate()` – Performs cleanup after last step (optional)

• For multitask models, a `model_step()` function is generated for each distinct rate group.
Model Entry Points

Model.h

```c
/* Model entry point functions */
extern void throttle_controller_initialize(void);
extern void throttle_controller_step(void);
extern void throttle_controller_terminate(void);
```

Model.c

```c
/* Model entry point functions */
void throttle_controller_step(void)
{
    /* local block i/o variables */
    real_T rth_BatteryLevelPositionCommand;
    real_T rth_cmd_i;
    real_T rth_err_p;
    real_T rth_integralProduct;
    real_T rth_tlp;
    /* Sum: '<S1>/Error' incorporates:
    * Import: '<Root>/Fbk'
    * Import: '<Root>/Rgt'
    */
    rth_cmd_i = throttle_controller_Upst - throttle_controller_Upst;

    /* Model initialize function */
    void throttle_controller_initialize(void)
    {
        /* Registration code */
        /* initialize non-finites */
        rth_InitInfAndNaN(sizeof(real_T));
        /* initialize error status */
        rtmSetErrorStatus(throttle_controller_M, (NULL));
    }

    /* Model stop function */
    void throttle_controller_stop(void)
    {
        /* Stop code */
        /* Stop non-finites */
        rtmSetInfAndNaN(sizeof(real_T));
        /* Stop error status */
        rtmSetErrorStatus((void)1, (NULL));
    }
}
```
Function Prototype Control

- Allows user to control the prototypes of `model_step()` and `model_initialize()` functions.
  - Function name
  - Ordering of arguments
  - Name of each argument
  - Pass by value/pointer
- Useful for algorithm export
Configuring Function Prototype Control

Generated custom entry points

```c
/* Model entry point functions */
extern void PI_initialize(void);

/* Customized model step function */
extern real_T PI_call(real_T arg_Vbat, real_T arg_Reset, real_T arg_Rqst, real_T arg_Fbk);
```

>> throttle_controller_fp
Componentization Methods

- Model references are useful for subdividing a design into components that can be tested as standalones.

- Library blocks (subsystems) are useful for creating utilities that can be reused.
Scheduling Techniques: Block Ordering and Grouping

Atomic subsystem

Intrinsic block sort order
Scheduling Techniques: Function-Call and Conditionally Executed Subsystems

- Function call is a way to explicitly schedule groups of blocks.
- Function-Called Subsystems are executed by Function-Call Generator blocks or Stateflow events.
- Enabled Subsystems and Triggered Subsystems are other form of conditionally executed blocks driven by a control signal.
Export Functions

- Function-call subsystem is a common approach for explicitly ordering the execution of a group of blocks relative to others.

- Export function applies specifically to function-call subsystems.

- This export mechanism is useful for extracting only the code for individual function-call subsystems without their scheduling code.
Export Functions

```
void t_ltic_A_Init(void)
{
    rtDWork.Delay_DSTATE_o = 1;
}

void t_ltic_A(void)
{
    rtB.Delay_b = rtDWork.Delay_DSTATE_o;
    rtB.Add = rtU.U1 + (real_T)rtB.Delay;
    rtDWork.Delay_DSTATE_o = (int8_T)(-rtB.Delay_b);
    rtY.TicToc1[0] = rtB.Delay_b;
    rtY.TicToc1[1] = rtB.Delay_c;
}
```

>> rtwdemo_export_functions
Data Dictionary

- Controlling data types
- Controlling data storage (storage classes)
- Convert floating-point data to fixed-point
Data Management

- Govern how data of the overall system and its various components is defined and packaged:
  - Control how much memory each data element occupies
  - Control how memory is allocated and its ownership
  - Control how data is shared or passed
  - Control how to export data for external use
  - Control how to import externally defined data

- Approaches to support data management:
  - In-model configurations (block attributes and signal properties)
  - Data dictionary approach (via data objects)

- Two key aspects of data characteristics:
  - Data types
  - Storage Classes
Data Type Specification

Data Types can be explicitly specified for signals, parameters, and states.

Data types can be propagated via inheritance or arithmetic rules.

Intrinsic Data Types
- Floating Point (double, single)
- Signed Integer (int8, int16, int32)
- Unsigned Integer (uint8, uint16, uint32)
- Boolean
- Fixed Point

User-Defined Data Types
- Enumerations
- Numeric (basic typedef)
- Alias (typedef)
- Bus (struct)

Allows users fine-grain control over data types.
Data Storage Specification

Storage class governs how the data of a model is:

- Declared
- Defined
- Accessed

```c
/* Exported Global Signals */
/* Note: Exported global signals are block signals with an * storage class designation. RTW declares the memory for * and exports their symbols. */

/* Exported block signals */
extern real32_T reset; /* '<Root>/Reset' */
extern real32_T rqst; /* '<Root>/Rqst' */
extern real32_T fbk; /* '<Root>/Fbk' */

if (reset != 0.0F) {
  rtb_err = 0.0;
} else {
  rtb_err = throttle_controller_obj_DWork.UnitDelay_DSTATE;
}
```
Data Storage Specification

- Built-in storage classes:
  - Auto/SimulinkGlobal
  - ExportedGlobal
  - ImportedExtern
  - ImportedExternPointer

- Custom storage class (CSC):
  - Extended capabilities to specify definition and header files
  - Enable optimization of storage via qualifiers and memory section
  - Allow custom packing via structures or bit fields
  - Allow interface to external code via get and set methods
  - Predefined CSCs exist
Default Storage Classes

Model inputs

States

Model outputs

Parameters

typedef struct {
    real_T UnitDelay_DSTATE;
} D_Work_throttle_controller;

typedef struct {
    real_T Vbat;
    real_T reset;
    real_T reqst;
    real_T fnk;
} ExternalInputs_throttle_controller;

typedef struct {
    real_T Cmd;
} ExternalOutputs_throttle_controller;

typedef struct {
    real_T ResetValue_Value;
    real_T ProportionalGain_Value;
    real_T IntegralGain_Value;
} Parameters_throttle_controller_1;

>> throttle_controller

model.h
Specification in a Data Dictionary

- Using the data dictionary driven approach, you can keep algorithm separated from data.
- MATLAB workspace becomes a data dictionary.
  - Composed of Simulink data objects that define characteristics of signals, states, and parameters
  - Any user-defined data types used by these data objects

![Data Dictionary Table]

![Simulink Diagram]

>> throttle_controller_obj
Model Explorer as a Data Dictionary Interface

Display both editable and read-only attributes for each signal and parameter
Data Objects

- A means through which characteristics of data can be defined in MATLAB workspace:
  - Signal objects for governing signals, states, and data stores
  - Parameter objects for governing tunable parameters and constants

- There are Simulink and MPT variants of each, resulting in:
  - Simulink.Signal
  - Simulink.Parameter
  - mpt.Signal
  - mpt.Parameter

- MPT data objects extend Simulink data objects capabilities by:
  - Provide additional data placement options
  - Provide additional initialization options
Data Object Dialog

- Data type and an assistant to help set data type string
- Storage class and its attributes
- Other attributes of interest
  - Dimension
  - Complexity
  - Sample time
  - Minimum and maximum
  - Optional initialization
  - Units
  - Description
To apply a signal object in Simulink:
1. Create a signal object in the base workspace.
2. Use the object name to label a signal in the model.
3. Indicate that the signal must resolve to a signal object.
Associating Parameters with Data Objects

To apply a parameter object in Simulink:
1. Create a parameter object in the based workspace.
2. Use the parameter object as a block parameter.
3. Enable **Inline parameters** optimization in the model configuration.
Simulink Fixed-Point Data Types

- Represent real-world values using integers
- Enable generation of integer-only code from designs
- Support integers with a word size of up to 128 bits
- Support slope-bias and binary-point-only scaling
Specifying Fixed-Point Data Types

Specify fixed-point attributes:
- Word size
- Sign
- Scaling

Automatically compute scaling with the highest achievable resolution based on output min & max

Shows representable range and resolution of the type
Fixed-Point Conversion

- Determine the optimum integer data type for each applicable signal and parameter to avoid over/under-flow, while maximize resolution.
- Cast each operand to its optimum data type.
- Perform the fixed-point operation.
- Cast the result to the output data type if it is explicitly specified to be different from the optimum result data type.
Fixed-Point Advisor

- Fixed-Point Advisor facilitates the conversion of a floating point model or subsystem to a fixed-point equivalence.

- Fixed-Point Advisor helps the user
  - Set model configuration to support automatic scaling
  - Set block attributes to enable automatic scaling
  - Perform the initial scaling for the model
  - Validate the results of the initial scaling against the floating point benchmark
  - Prepare the model for code generation
Fixed-Point Advisor (Continued)

The Fixed-Point Advisor facilitates converting a floating-point model or subsystem into a fixed-point representation. It performs four main tasks:

1. Prepare Model for Conversion
   - Evaluate model wide configuration options.
   - Create floating-point base line data set.
2. Prepare for Data Typing and Scaling
   - Evaluate block specific configurations.
   - Add design minimum and maximum information to the model.
3. Perform Data Typing and Scaling
   - Propose fixed-point data typing and initial scaling to the blocks.
   - Analyze the resulting fixed-point model behavior.
4. Prepare for Code Generation
   - Examine issues resulting in inefficient code.

The Fixed-Point Advisor provides you with feedback on the results of the task. If the task fails, the Fixed-Point Advisor provides you with information on how to modify the model to complete the task. For more information, click Help.

>> throttle_controller_f2f
External Functions and Utilities

- Importing custom library functions
- Importing external code
Tools for Importing User Code and Utilities

- Enable existing code and custom utilities to be integrated with the code generated from the model

- Most common methods for importing external code:
  - Target Function Library: For custom library functions
  - Legacy Code Tool: For existing user-written algorithms
  - Hand-crafted inlined S-function: For creating device drivers
Target Function Library

- ANSI-C code can be further optimized for processors with built-in arithmetic instructions
- Target Function Library Replacement API enables generation of optimized, processor-specific arithmetic code
- Benefits:
  - Improved execution performance
  - Reduced code size
  - Applies generically across all products
  - Model and release independent
int32_T add_s32_s32_s32_sat(int32_T a, int32_T b)
{
    int32_T tmp;
    tmp = a + b;
    if ((a < 0) && (b < 0) && (tmp >= 0))
    {
        tmp = MIN_int32_T;
    }
    else if ((a > 0) && (b > 0) && (tmp <= 0))
    {
        tmp = MAX_int32_T;
    }
    return tmp;
}

int32_T tricore_add_s32_s32_s32_sat(int32_T a, int32_T b)
{
    return (__sat int)a + b;
}

Optimized for TriCore

Generic ANSI-C Code

>> rtwdemo_tfladdsub
Legacy Code Tool

- A MATLAB-based tool for integrating external code into Simulink models for simulation and code generation by:
  - Creating an S-function block that calls the external function
  - Creating a complimentary S-function TLC file that provides instruction on how to emit code for the S-function block

```matlab
%% Initialize LCT Structure
s = legacy_code('initialize');
s.SFunctionName = 'LookUpTable_sfun';

% Cput Fcn:
% float LookUpTable(
%    float xIn,
%    float xAxis[],
%    float yAxis[],
%    int axisLength)

s.OutputFcnSpec = [{'single y1 = LookUpTable(''...
    'single u1,...' % xIn
    'single u2[]','...' % xAxis
    'single u3[]','...' % yAxis
    'int32 size(u2,1)'); % axisLength
```
Legacy Code Tool Registration File

/* LookupTable.h */
/* Provided the function prototype for a Simple Lookup Table */
extern float LookupTable(float xIn, float xAxis[], float yAxis[], int axisLength);

/* LookupTable.c */
/* Provided the function definition for a Simple Lookup Table */
#include "LookupTable.h"

float LookupTable(float xIn, float xAxis[], float yAxis[], int axisLength)
{
    /* This routine assumes monotonically increasing values of X */
    float outValue;
    int axisLoc;
    axisLoc = (float) (axisLength * 0.5); /* Start in center of table */

    if (xIn >= xAxis[axisLength - 1])
    {
        outValue = yAxis[axisLength - 1];
    }
    else if (xIn <= xAxis[0])
    {
        outValue = yAxis[0];
    }
    else
    {
        float xLoc;
        xLoc = xIn - xAxis[axisLoc];
    }
    return outValue;

    % Initialize the LCT Structure
    s = legacy_code('initialize');
    s.sFunctionName = 'LookupTable_sfun';
    % Output Fcn:
    % float LookupTable(
    % float xIn,
    % float xAxis[],
    % float yAxis[],
    % int axisLength)
    s.OutputFcnSpec = 'single y1 = LookupTable('';
    % single u1, ... % xIn
    % single u2[], ... % xAxis
    % single u3[], ... % yAxis
    % int32 size(u2,1))'; % axisLength
    s.HeaderFiles = ('LookupTable.h');
    s.SourceFiles = ('LookupTable.c');
Legacy Code Tool Generated S-Function

% Generate Simulation Files
% Generate the C-MEX S-function source file
legacy_code('sfcn_cmex_generate',s);
% Compile the C-MEX S-function MEX file
legacy_code('compile',s,'-g');
% Create a masked subsystem for the S-function
legacy_code('slblock_generate',s)
Legacy Code Tool Generated Code

```matlab
% Generate Code Generation Files
% Generate the S-function TLC file
legacy_code('sfcn_tlc_generate',s);
% Generate the S-function make configuration file for building SIM and RTW targets
legacy_code('rtwmakecfg_generate',s);

/* Product: '<Root>/Product' incorporates:
 * Constant: '<Root>/Constant'
 * Constant: '<Root>/Constant1'
 * Import: '<Root>/Vbat'
 * S-Function (LookUpTable_sfun): '<Root>/LookUpTable_sfun'
 */

cmd = rtb_Integral * LookUpTable( throttle_controller_lut_U.Vbat,
throttle_ctrl_lut_lut_vbat_inmap, throttle_ctrl_lut_lut_vbat_outmap, (int32_t)7);```
Model Configuration for Production Code Generation

- Configuring the target settings
- Setting code generation objectives
- Checking compliance against objectives
Basic Code-Generation Workflow

Once the model has been tested and verified via simulation, here's an overview of the workflow for code generation:

1. Run Model Advisor and revise
2. Configure code generation options
3. Generate code
4. Inspect the generated code and report
5. Test the generated code
Basic ERT Code Generation Settings

- Select ert.tlc as the target
- Select language
- Enable generate code only
- Select the discrete fixed step solver
- Enable HTML Report for easy review
Select a device or specify details for custom device
Software Environment

Specify and constrain the software environment
ASAP2 File Generation

- Generates ASAM/ASAP2 calibration file as an artifact of code generation
- Supports signals and parameters
- Customizable script lets you include memory address information

```matlab
>> throttle_controller_asap2
```
Code Generation Advisor

Allow the user to select and prioritize the following code generation objectives

- Execution efficiency: To achieve fast execution time
- ROM efficiency: To reduce ROM usage
- RAM efficiency: To reduce RAM usage
- Traceability: To preserve mapping between model elements and code
- Safety precaution: To increase clarity, determinism, robustness, and verifiability of the code
- Debugging: To enable debugging of the code generation build process

Can create custom objectives:

- Rapid prototyping
- Production
Code Generation Advisor Workflow

Select and sort objectives

Check model against set objectives and report violations
Verifying Generated Code with Software-in-the-Loop Testing

>> rtwdemo_sil_block
Wrap Up

- Use Simulink for both Rapid Prototyping and Production to fully leverage your Model-Based Design investment
- Migration is streamlined if you spend time early on to establish the right architecture and ensure model portability
- Real-Time Workshop Embedded Coder offers many approaches for production deployment, some were discussed today
- Contact MathWorks to arrange a more detailed presentation and discussion on these topics