Scott Ranville
303-734-8988
scottranville@softwareberet.com

Software Beret Inc.
State-of-the-Art Embedded Software
Tools and Processes

1227 W. Weaver Cir.
Littleton, CO 80120
www.softwareberet.com

Efficient and Flexible ERT Code

By: Scott Ranville
Software Beret Inc.
January 2006

1. Introduction

Many companies from a wide spectrum of industries such as automotive, aerospace, and medical (to name a few) are using Matlab/Simulink/Stateflow as a software development tool. Discrete controller algorithms are being modeled. These algorithms are the software requirements. The models are getting used throughout the software development process from PC based testing (simulations) to rapid prototyping to automatic code generation to automatic test generation. Within this, a number of users need efficient code, especially for embedded and deeply embedded applications. For embedded applications, The Mathworks has developed the ERT target for RTW – also referred to as E-coder. As a general rule, ERT will generate good code. This article will demonstrate the ERT code for different modeling styles. The first modeling style will be a "typical" modeling style that the author has seen used at a number of companies. From this "typical" style, ERT code from additional styles will be generated to show how the good ERT code can be migrated to even more efficient code. With R14 SP3, some modeling styles limit what type of code ERT will generate. The modeling styles will also demonstrate how some of these limitations can be removed based on the modeling style. While it is hard to demonstrate in an article, the "Software Beret" modeling style not only results in very efficient ERT code, but the model creation time is also shorter than the other modeling styles.

2. Process and Code Considerations

When using Matlab as a software development tool, there are a number of considerations such as:

- How to partition the models so that multiple engineers can work simultaneously?
- If legacy code exists, how to migrate this code to a Matlab model?
    - Can functions be migrated one at a time or does the entire legacy base have to be converted before the model(s) can be used?
- How to generate code that can call "other" code or be called by "other" code (for example operating system or driver code)?

There are a number of possibilities for addressing these and other considerations. What the author has seen, is that often times, companies do not consider many of these issues upfront, but rather dive into building models only to find out later, that there are process and or issues with the format of the generated code. When considered at the start, some interesting, and probably unexpected, solutions will emerge that allows the company to achieve significant benefits in productivity and code quality.

This article will not focus on these process and code considerations in great detail. But, the "Software Beret" modeling style does take these issues into consideration and seems to provide a decent solution.

3. Example 1

One process and code consideration is how to model a variable that gets read from and written to in multiple locations. Many companies will partition the model/code so that different functions (also referred to as features) either read the variable or write to the variable. In Matlab, functions can be implemented as a stand alone model, an atomic Subsystem within the model, or within a Stateflow diagram. The next question is, should all these functions be located inside of one model, or can they be in different models so that different engineers can work on the separate functions at the same time. If the functions are in different models, how does one integrate the ERT code from each of these models into the final application? By default, the ERT code from the different models will not easily integrate together as the default assumption by ERT is that model contains the entire application. However, ERT options are available that allow the code from the different models to be more easily integrated together. Modeling styles that use the Model Reference block or library blocks also aid in integrating the code from different models into a single executable. As a stating

point to illustrate the ERT code that results from different modeling styles, consider the following pseudo code:

```
#define Multiplier 7
#define Adder 33

int GlobalVector[7];

double FunctionA( double in_arg1 )
{
    double tmp;
    int i;

    tmp = (in_arg1 * Multiplier)/(in_arg1 + Adder);

    if( tmp > 55.5 )
    {
        /* do nothing */
    }
    else
    {
        for(i = 0; i < 7; i++)
        {
            GlobalVector[i] += 1;
        }
    }

    return( tmp - 55.5 )
}
```

For the situation in which the above code needs to integrate with "other" code, the function interface needs to be maintained. If not, the "other" code will need to be modified to match the ERT interface.

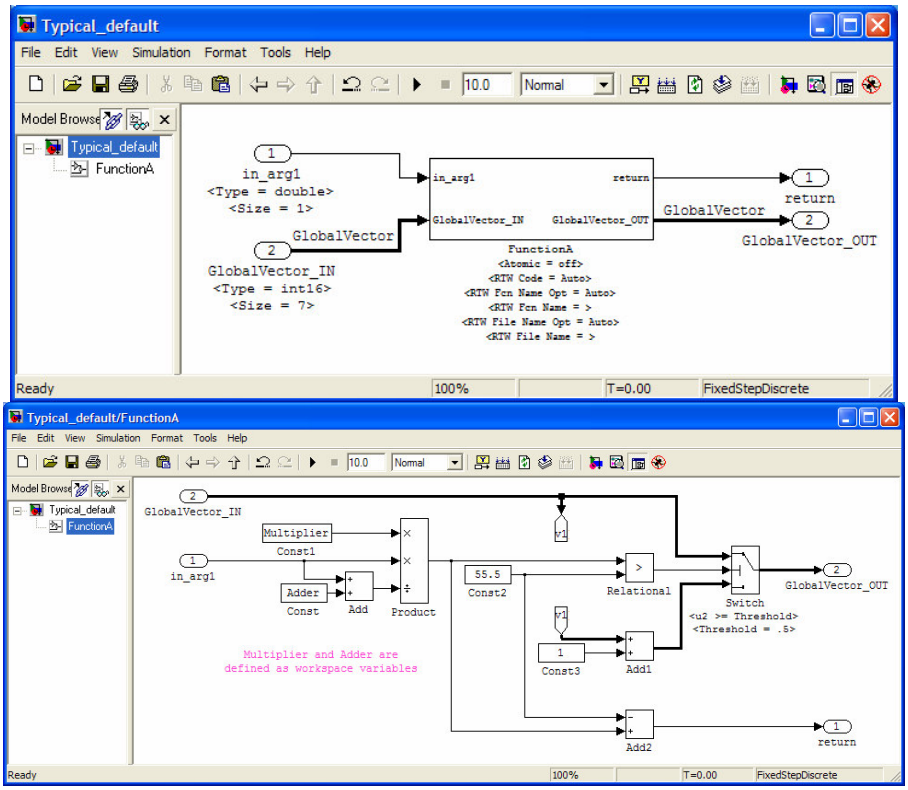
For the situation in which the code is the entire application, the interface is not important, but for embedded applications, the efficiency is still important.

3.1. Example 1 – "Typical" Modeling Style with Default Settings

A typical modeling style for this sample code is to use the Switch block to model the If statement. Other options are available, such as the IF block, but this requires the use of Action Subsystems and could be considered overkill for this simple of a function.

The modeler needs to be aware that Simulink does not allow two blocks in the same Subsystem to have the same name. Thus, the 'GlobalVector' has different names for the Input and Output blocks. Signal lines can have the same label, so the model labels both signals.

For this example, the assumption is that this function is one of many functions that will be modeled in the mdl file. Thus, the function was placed inside of a Subsystem, instead of at the top of the model.



Note: While many companies will use the blocks in the above model, many companies will not have a "nice" looking model, but rather will do things like not line up the blocks, not use "wide signals" to show the vector signals, not use annotations to show important block settings, have crossing signal lines, and include the Constant block for '55.5' twice (makes it harder to maintain the model). Some individuals prefer to not show the block name, which can make the model look cleaner, however, this hurts debugging and traceability to the ERT code. Thus, the author prefers to show block names.

In the ERT code notice:

- Variable names
- All variables are global
- Constant values are part of a global structure
- The code is placed in the '_step' function. If other functions were modeled, the code for them would also reside in this '_step' function, thus, there is no partitioning for the different functions.

File: "main".c

```
void Typical_default_step(void)
{

  /* local block i/o variables*/
  real_T rtb_Product;
  real_T rtb_Switch[7];

  {
    int32_T i1;

    /* Product: '<S1>/Product' incorporates:
     *  Sum: '<S1>/Add'
     *  Constant: '<S1>/Const1'
     *  Constant: '<S1>/Const'
     *  Inport: '<Root>/in_arg1'
     */
    rtb_Product = Typical_default_P.Const1_Value * Typical_default_U.in_arg1 /
```

```
      (Typical_default_U.in_arg1 + Typical_default_P.Const_Value);

    /* Outport: '<Root>/return' incorporates:
     *  Sum: '<S1>/Add2'
     *  Constant: '<S1>/Const2'
     */
    Typical_default_Y.return_d = rtb_Product - Typical_default_P.Const2_Value;
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S1>/Switch' incorporates:
       *  Sum: '<S1>/Add1'
       *  RelationalOperator: '<S1>/Relational'
       *  Constant: '<S1>/Const3'
       *  Constant: '<S1>/Const2'
       *  Inport: '<Root>/GlobalVector_IN'
       */
      if(rtb_Product > Typical_default_P.Const2_Value) {
        rtb_Switch[i1] = (real_T)Typical_default_U.GlobalVector[i1];
      } else {
        rtb_Switch[i1] = (real_T)Typical_default_U.GlobalVector[i1] +
          Typical_default_P.Const3_Value;
      }

      /* Outport: '<Root>/GlobalVector_OUT' */
      Typical_default_Y.GlobalVector_OUT[i1] = rtb_Switch[i1];
    }
  }
}
```

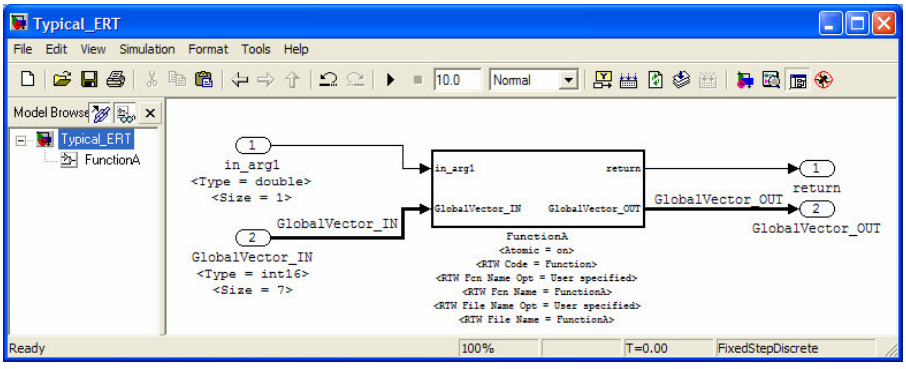
File: "_data.c"

```
Parameters_Typical_default Typical_default_P = {
  55.5 ,                                /* Const2_Value : '<S1>/Const2'
                                         */
  7.0 ,                                 /* Const1_Value : '<S1>/Const1'
                                         */
  33.0 ,                                /* Const_Value : '<S1>/Const'
                                         */
  1.0                                   /* Const3_Value : '<S1>/Const3'
                                         */
};
```

3.2. Example 1 – "Typical" Modeling Style with "ERT" Settings

The same model as above was used, but with one slight change to the signal line labels. This change is needed because the signals are being coded as global variables and ERT does not allow two different signal lines to be coded to the same global variable name.

In this case, mpt.Parameters and mpt.Signals were used to configure the #define constants and the global variables. Some of the default model settings were also changed to improve the code. Finally, the Subsystem was configured differently to allow the contents of the Subsystem to be coded as a function.



In the ERT code, notice:

- For this modeling style, the global input must have a different name than the output version of the variable (this will make integrating code from different models harder)
- With this modeling style, there is no way to get ERT to create a function return nor to pass a function argument. Rather, global variables are always used.
- Constant values are inlined
- The named constants are coded as #define
- The IF statement is inside of the FOR loop for the vector

File: FunctionA.c

```
void FunctionA(void)
{
  /* local block i/o variables*/
  real_T rtb_Product;

  {
    int32_T i1;

    /* Product: '<S1>/Product' incorporates:
     *  Sum: '<S1>/Add'
     *  Constant: '<S1>/Const1'
     *  Constant: '<S1>/Const'
     *  Inport: '<Root>/in_arg1'
     */
    rtb_Product = Multiplier * Typical_ERT_U.in_arg1 / (Typical_ERT_U.in_arg1 +
      Adder);

    /* Sum: '<S1>/Add2' incorporates:
     *  Constant: '<S1>/Const2'
     */
    Typical_ERT_B.Add2 = rtb_Product - 55.5;
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S1>/Switch' incorporates:
       *  Sum: '<S1>/Add1'
       *  RelationalOperator: '<S1>/Relational'
       *  Constant: '<S1>/Const3'
       *  Constant: '<S1>/Const2'
       *  Inport: '<Root>/GlobalVector_IN'
       */
      if(rtb_Product > 55.5) {
        GlobalVector_OUT[i1] = GlobalVector_IN[i1];
      } else {
        GlobalVector_OUT[i1] = (int16_T)(GlobalVector_IN[i1] + 1);
      }
    }
  }
}
```

File: header file

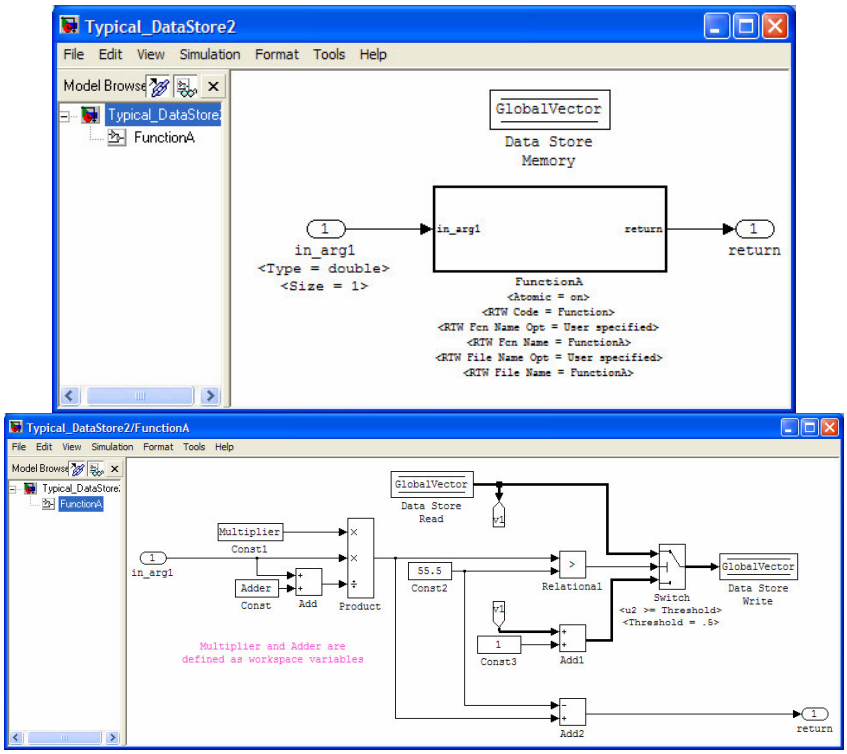
```
#define Adder                          33.0

#define Multiplier                     7.0
```

3.3. Example 1 – Data Store Memory Modeling Style

In the above code, having a different variable name for the global input and output is not desired. These two global variables can be replaced by a single global variable by replacing the Inport and Outport blocks by Data Store Read and Write blocks.

This modeling style does create a problem for automatic test generation tools. This problem is easily solvable, but is a topic for another article.



In the ERT code, notice:

- How the same global variable is both an input and an output
- With this modeling style, there is still no way to get ERT to create a function return nor to pass a function argument. Rather, global variables are always used.
- Constant values are inlined
- The named constants are coded as #define
- The IF statement is inside of the FOR loop for the vector

Depending on how the Data Store blocks are used in the model, an extra temporary variable may be created for each Read and Write block. If the variable is a vector, a FOR loop will be used to assign the global variable to the temporary variable and then vise versa for the output.

File: FunctionA.c

```
void FunctionA(void)
{
  /* local block i/o variables*/
  real_T rtb_Product;
  int16_T rtb_Switch[7];

  {
    int32_T i1;

    /* Product: '<S1>/Product' incorporates:
     *  Sum: '<S1>/Add'
     *  Constant: '<S1>/Const'
     *  Inport: '<Root>/in_arg1'
     *  Constant: '<S1>/Const1'
     */
    rtb_Product = Multiplier * Typical_DataStore2_U.in_arg1 /
      (Typical_DataStore2_U.in_arg1 + Adder);
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S1>/Switch' incorporates:
       *  Sum: '<S1>/Add1'
```

```
         *  DataStoreRead: '<S1>/Data Store Read'
         *  RelationalOperator: '<S1>/Relational'
         *  Constant: '<S1>/Const2'
         *  Constant: '<S1>/Const3'
         */
        if(rtb_Product > 55.5) {
          rtb_Switch[i1] = GlobalVector[i1];
        } else {
          rtb_Switch[i1] = (int16_T)(GlobalVector[i1] + 1);
        }

        /* DataStoreWrite: '<S1>/Data Store Write' */
        GlobalVector[i1] = rtb_Switch[i1];
      }

      /* Sum: '<S1>/Add2' incorporates:
       *  Constant: '<S1>/Const2'
       */
      Typical_DataStore2_B.Add2 = rtb_Product - 55.5;
    }
  }
```

File: header file

```
#define Adder                          33.0

#define Multiplier                     7.0
```

3.4. Example 1 – "Software Beret" Modeling Style

Using the Software Beret modeling style results in the following code.

In the ERT code, notice (comments removed to protect the modeling style):

- How the same global variable is both an input and an output. With this modeling style, there is never extra temporary variables or FOR loops as is sometimes needed with the Data Store Read and Write blocks
- With this modeling style, ERT DOES create a function return and function argument.
- The IF statement is outside of the FOR loop for the vector thereby increasing CPU throughput

File: FunctionA.c

```
real_T FunctionA(real_T in_arg1)
{
  real_T tmp;
  int32_T i;
  tmp = Multiplier * in_arg1 / (in_arg1 + Adder);
  if(!(tmp > 55.5)) {
    for(i = 0; i < 7; i++) {
      GlobalVector[i] = (int16_T)(GlobalVector[i] + 1);
    }
  }
  return tmp - 55.5;
}
```

File: header file

```
#define Adder                          33.0

#define Multiplier                     7.0
```

3.5. Example 1 – Conclusions

The modeling style and "extra" configurations can have a significant impact on the generated ERT code. Different modeling styles can optimize code size (ROM and RAM) and or code efficiency (CPU). The modeling style also impacts the function interface and how the function needs to be called. This in turn impacts whether different functions need to be in the same model or if the can be placed in different models.

This then impacts how engineers work and if they can edit the model(s) simultaneously or if they have to edit the model one after the other.

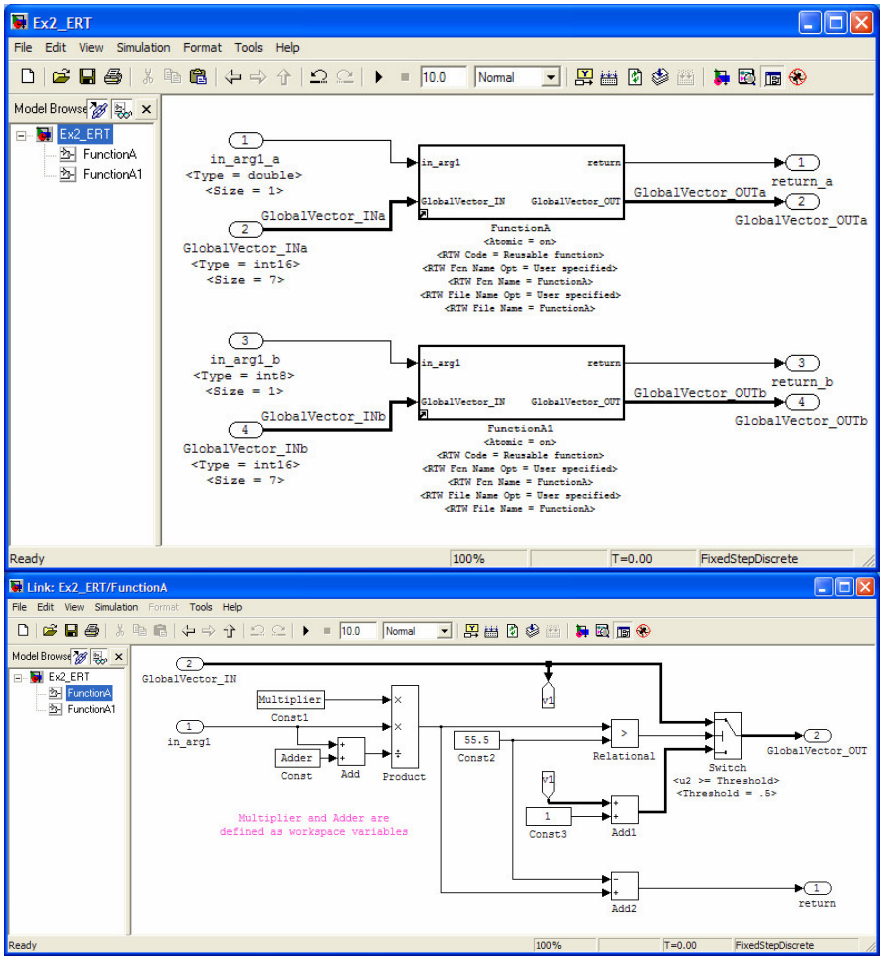
4. Example 2

As a second example, consider the same function as above, but instead of being a standard function, this time treat it as a library function that will get called multiple times.

4.1. Example 2 – "Typical" Modeling Style with "ERT" Settings

Using the "typical" modeling style, the original Subsystem will be placed in a library model and configured to be a "Reusable Function". This modeling style works best when the inputs and outputs are strongly typed and have a fixed size. However, for the case of an "overloaded" library block that will inherit the input type and size, multiple functions can get generated when only one was expected. In the following example, the input size will be held fixed (scalars in this example), but the type of the inputs will change.

The global variables in this example are not implemented well, but to implement them well would complicate this example while not making it more descriptive.



In the ERT code, notice:

- The different input types caused two functions to get generated, and how the second function name is "name mangled"

- There is no good way to handle the global variable that the function accesses
- Function arguments are now being passed to the function

File: FunctionA.c

```
void FunctionA(real_T rtu_0, const int16_T rtu_1[7], rtB_FunctionA *localB)
{
  /* local block i/o variables*/
  real_T rtb_Product;

  {
    int32_T i1;

    /* Product: '<S1>/Product' incorporates:
     *  Sum: '<S1>/Add'
     *  Inport: '<Root>/in_arg1_a'
     *  Constant: '<S1>/Const'
     *  Constant: '<S1>/Const1'
     */
    rtb_Product = Multiplier * rtu_0 / (rtu_0 + Adder);

    /* Sum: '<S1>/Add2' incorporates:
     *  Constant: '<S1>/Const2'
     */
    localB->Add2 = rtb_Product - 55.5;
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S1>/Switch' incorporates:
       *  Sum: '<S1>/Add1'
       *  RelationalOperator: '<S1>/Relational'
       *  Inport: '<Root>/GlobalVector_INa'
       *  Constant: '<S1>/Const2'
       *  Constant: '<S1>/Const3'
       */
      if(rtb_Product > 55.5) {
        localB->Switch[i1] = rtu_1[i1];
      } else {
        localB->Switch[i1] = (int16_T)(rtu_1[i1] + 1);
      }
    }
  }
}

/* Output and update for atomic system: '<Root>/FunctionA1' */
void Ex2_ERT_FunctionA(int8_T rtu_0, const int16_T rtu_1[7],
 rtB_Ex2_ERT_FunctionA *localB)
{
  /* local block i/o variables*/
  real_T rtb_Product_b;

  {
    int32_T i1;

    /* Product: '<S2>/Product' incorporates:
     *  Sum: '<S2>/Add'
     *  Inport: '<Root>/in_arg1_b'
     *  Constant: '<S2>/Const'
     *  Constant: '<S2>/Const1'
     */
    rtb_Product_b = (real_T)rtu_0 * Multiplier / ((real_T)rtu_0 + Adder);

    /* Sum: '<S2>/Add2' incorporates:
     *  Constant: '<S2>/Const2'
     */
    localB->Add2_m = rtb_Product_b - 55.5;
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S2>/Switch' incorporates:
       *  Sum: '<S2>/Add1'
       *  RelationalOperator: '<S2>/Relational'
```

```
            *   Inport: '<Root>/GlobalVector_INb'
            *   Constant: '<S2>/Const2'
            *   Constant: '<S2>/Const3'
            */
           if(rtb_Product_b > 55.5) {
             localB->Switch_a[i1] = rtu_1[i1];
           } else {
             localB->Switch_a[i1] = (int16_T)(rtu_1[i1] + 1);
           }
         }
       }
     }
```

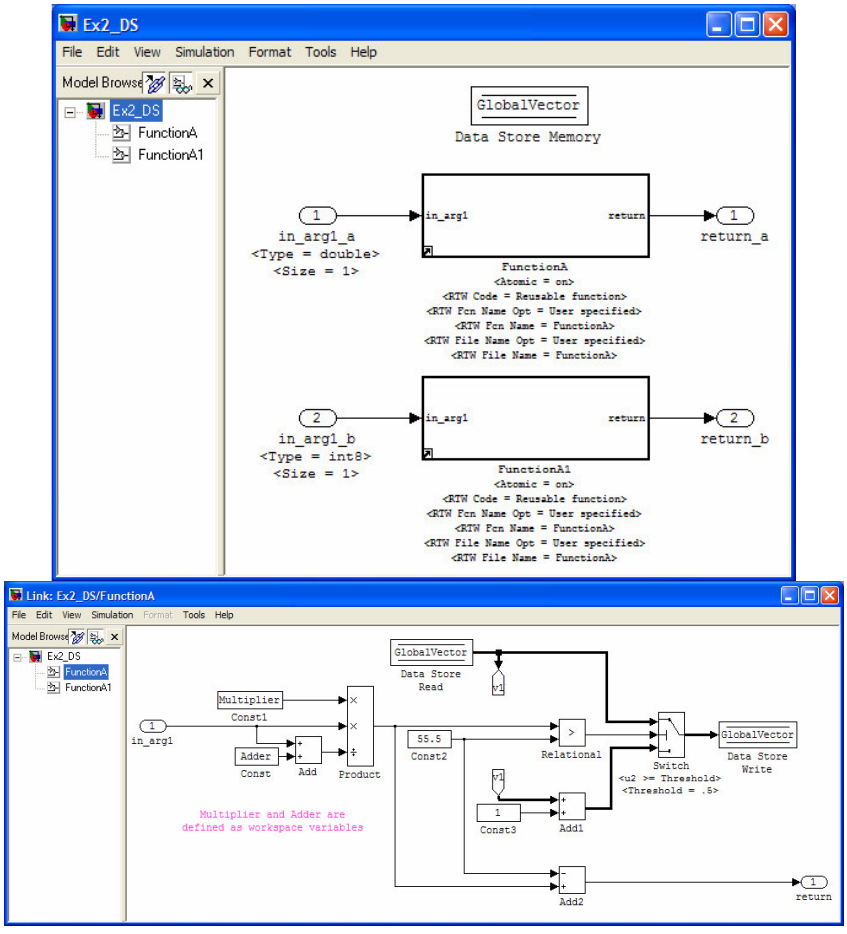
File: header file

```
#define Adder                          33.0

#define Multiplier                     7.0
```

4.2. Example 2 – Data Store Modeling Style

Using the "Data Store" modeling style, and placing the Subsystem in a library model results in code that is similar to the typical modeling style in that the overloaded library function gets coded as two functions.



In the ERT code, notice:

- The different input types still cause two functions to get generated, and how the second function name is "name mangled"
- The global variable is handled much better than in the "typical" modeling styling

- Function arguments are being passed to the function

File: FunctionA.c

```
void FunctionA(real_T rtu_0, rtB_FunctionA *localB)
{
  /* local block i/o variables*/
  real_T rtb_Product;
  int16_T rtb_Switch[7];

  {
    int32_T i1;

    /* Product: '<S1>/Product' incorporates:
     *  Sum: '<S1>/Add'
     *  Inport: '<Root>/in_arg1_a'
     *  Constant: '<S1>/Const1'
     *  Constant: '<S1>/Const'
     */
    rtb_Product = Multiplier * rtu_0 / (rtu_0 + Adder);
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S1>/Switch' incorporates:
       *  Sum: '<S1>/Add1'
       *  DataStoreRead: '<S1>/Data Store Read'
       *  RelationalOperator: '<S1>/Relational'
       *  Constant: '<S1>/Const3'
       *  Constant: '<S1>/Const2'
       */
      if(rtb_Product > 55.5) {
        rtb_Switch[i1] = GlobalVector[i1];
      } else {
        rtb_Switch[i1] = (int16_T)(GlobalVector[i1] + 1);
      }

      /* DataStoreWrite: '<S1>/Data Store Write' */
      GlobalVector[i1] = rtb_Switch[i1];
    }

    /* Sum: '<S1>/Add2' incorporates:
     *  Constant: '<S1>/Const2'
     */
    localB->Add2 = rtb_Product - 55.5;
  }
}

/* Output and update for atomic system: '<Root>/FunctionA1' */
void Ex2_DS_FunctionA(int8_T rtu_0, rtB_Ex2_DS_FunctionA *localB)
{
  /* local block i/o variables*/
  real_T rtb_Product_o;
  int16_T rtb_Switch_j[7];

  {
    int32_T i1;

    /* Product: '<S2>/Product' incorporates:
     *  Sum: '<S2>/Add'
     *  Constant: '<S2>/Const'
     *  Constant: '<S2>/Const1'
     *  Inport: '<Root>/in_arg1_b'
     */
    rtb_Product_o = (real_T)rtu_0 * Multiplier / ((real_T)rtu_0 + Adder);
    for(i1 = 0; i1 < 7; i1++) {

      /* Switch: '<S2>/Switch' incorporates:
       *  Sum: '<S2>/Add1'
       *  DataStoreRead: '<S2>/Data Store Read'
       *  RelationalOperator: '<S2>/Relational'
       *  Constant: '<S2>/Const2'
```

```
         *  Constant: '<S2>/Const3'
         */
        if(rtb_Product_o > 55.5) {
          rtb_Switch_j[i1] = GlobalVector[i1];
        } else {
          rtb_Switch_j[i1] = (int16_T)(GlobalVector[i1] + 1);
        }

        /* DataStoreWrite: '<S2>/Data Store Write' */
        GlobalVector[i1] = rtb_Switch_j[i1];
      }

      /* Sum: '<S2>/Add2' incorporates:
       *  Constant: '<S2>/Const2'
       */
      localB->Add2_k = rtb_Product_o - 55.5;
    }
  }
```

File: header file

```
#define Adder                          33.0

#define Multiplier                     7.0
```

4.3. Example 2 – Software Beret Modeling Style

Using the Software Beret modeling style results in the following code.

In the ERT code, notice (comments removed to protect the modeling style):

- The different input types now result in only 1 function getting generated with a type cast to accommodate the different input types (note there is no Data Type Conversion block in the model)
- The function now has the desired one input argument and a return value

File: FunctionA.c

```
real_T FunctionA(real_T in_arg1)
{
  real_T tmp;
  int32_T i;
  tmp = Multiplier * in_arg1 / (in_arg1 + Adder);
  if(!(tmp > 55.5)) {
    for(i = 0; i < 7; i++) {
      GlobalVector[i] = (int16_T)(GlobalVector[i] + 1);
    }
  }
  return tmp - 55.5;
}
```

File: header file

```
#define Adder                          33.0

#define Multiplier                     7.0
```

4.4. Example 2 – Conclusions

Overloaded library functions can result in more functions getting generated than some users may have expected. However, this does allow for a very flexible modeling style in that only one library block is needed instead of many strongly typed library blocks. The Software Beret modeling style retains the modeling flexibility with only one library block, but has ERT add in the needed type casts (without using the Data Type Conversion block) to allow only one function to be generated.

4.5. Conclusions

This paper has briefly described a few of the available modeling styles. Simulink is a very powerful modeling tool and supports a wide range of modeling styles. As this paper has started to illustrate, the modeling styles impact the generated code, which in turn impacts the entire software development process. For example, the "typical" modeling style with no ERT customizations works best when the entire application is being modeled in one mdl file. Conversely, the Software Beret modeling style makes it easier to integrate ERT code that is generated from different models and also with "other" generated code such as legacy code. This then allows the model creators to create the models in parallel using standard models as opposed to other processes and modeling styles that require the modelers to work with library blocks for example. While library blocks are powerful, editing a library can be interesting when unintended library links are created within the library. Independent of the partitioning of the functionality into one or multiple models, the modeling style also impacts how quickly the models can be created. Different modeling styles require different numbers of mouse clicks which impacts the model creation time.

While not addressed in this paper, the modeling style also has significant impact on the rest of the software development process from requirements through testing. Because Simulink offers so many degrees of freedom, it is possible to create modeling styles that are hard to understand. Many companies will have reviews of the model, and these hard to understand modeling styles make the reviews difficult and longer in duration. Some companies will review the models by running Matlab while other companies will print out the models. The Matlab-based reviews allow the reviewers to double click on blocks and look at important block settings. However, this is not possible when reviewing the models that have been printed out. Thus, a slightly different modeling style is needed when the process involves reviews of printed models.

Verification and Validation of the Simulink algorithm and the resulting ERT code is important. There are commercially available tools that help with this testing in the form of formal methods tools and tools that automatically generate "safety critical" tests such as for the MCDC testing requirement. Unfortunately, these tools do not work equally well with all modeling styles. The Software Beret modeling style has been constructed to try to optimize the performance of these automatic test generation tools.

Model-based software development appears to gaining momentum as it offers significant benefits in improved code quality and reduction in overall development time and cost. However, the author has seen instances where companies may not realize this potential, and even end up with development times that are longer than with "traditional" software development methods. One reason for this is that the companies did not investigate how the modeling style impacts the entire software development process. The companies appear to get excited about the possibilities, and then start modeling with the first modeling style that seems obvious to them. However, as they move through the software development process, they encounter problems with tools that do not behave the way they assumed that they would. They then have to find workarounds, wait for tool vendor changes, or accept less improvement than what they were expecting. By making the upfront effort to find the modeling style that works throughout the entire software development process, these "last minute" headaches can be avoided and model-based software development can achieve the benefits that companies are expecting and need in today's fast paced, competitive world.

Bio

Scott Ranville started working on model-based embedded software development shortly after graduating in 1993 with a Masters from the University of Michigan. Scott has worked with a number of tool vendors to evolve the tools to their current state. Scott has taken this detailed knowledge of the tools and has been helping aerospace and automotive companies to implement model-based processes. Scott consults on both high level process issues as well as low level tool details. Some example tools include: tracing requirements from high level through tests, automatic style guide checkers, automatic code generation, formal methods, automatic test generation, and graphical model coverage tools.

Scott Ranville, 303-734-8988, scottranville@softwareberet.com, www.softwareberet.com