Paper BI-01-2014

An Object Oriented Framework for Simulations in base SAS®

Michael C. Frick, General Motors - Retired, Warren, MI

ABSTRACT

Have you ever wished you could have the power of SAS at your fingertips while coding an object-oriented simulation in a language like C++? Well if you have version 9.3 of SAS and are willing to take on some of the bookkeeping responsibilities normally handled by the complier, you can have the next best thing all while operating completely within base SAS. In this work, I develop an object-oriented framework for simulating the performance of traffic light signals using several features recently added to SAS. Each class of objects in the framework has its own dedicated subroutine (compiled using PROC FCMP) which consists of a hash table to store data associated with each instance of an object and the common code required to operate on the data. Further, each class of objects has an associated SAS macro that can be called to create a new instance of the object. The result is a set of off-the-shelf objects that can quickly be mixed and matched to form an infinite number of different simulations.

INTRODUCTION:

Language defined "data types" (e.g. Integer, Floating Point, String, Pointer, etc.) are primary building blocks when writing code in a traditional language like C or PL/I. Although SAS is less formal in its declaration of variables, it still makes a distinction between the two major data types: numeric and character. The advent of newer languages like C++ and Java brought support for object oriented design and programming which, among other things, give programmers access to user defined data types called classes. Stanley Lippman, [1] in his classic C++ introductory text, states that classes have four associated attributes: (1) data members, (2) data functions, (3) scope limitation (private versus public), and (4) a tag name that can be used to declare a new variable of the user-defined type.

The code segment below shows an Excel Visual Basic [2] example of a class that would facilitate storing, retrieving, and manipulating information on parts stored in a warehouse. The class name is `CItem`. Data members are given first (SKU, QtyOnHand, etc.). The member function shown returns a quote that can be discounted if the quantity on hand exceeds a threshold. A fully functional class would normally have several member functions. Note that defining a class does not create a variable. It is just another data type, like numeric or character. A collection of user-defined classes is called a Class Library. A class library extends the functionality of the native language. An Application Framework is a class library designed to support the building of specific types of applications (e.g . a set of classes aimed at making specific types of simulations easier to perform in SAS.)

```
' Class CItem
Public SKU As String
Public QtyOnHand As Long
Public Price As Double
Public UpperBand As Long
Public Discount As Double

Public Function Quote() As Double
If QtyOnHand > UpperBand Then
   Quote = Price * (1 - Discount)
Else
   Quote = Price
End If
End Function
```

As mentioned earlier, a class is the definition of a new data type. To use a class, we need to declare a new variable that is of the type of our new class. In an object-oriented language, one can simply declare the variable by giving the new variable name and the associated tag name for the new data type using the appropriate syntax for the language. And, of course, just like you can have as many character or numeric variables as you want in a program, you can have as many objects (variables of your new class) as you want. The beauty is the compiler handles everything from there. Behind the scenes, it stores just one copy of the member functions along with a set of the data members for each defined object and is able to keep track of which member functions can be applied to which data members.

In this work, we will explore how one might use hash tables, PROC FCMP, the DATA Step, and the SAS macro facility to emulate Lippman's class attributes. Further, we will explore how one might then pick up the compiler role to bind the data members and data functions together to emulate a working class. Our goal is to lay the foundation for how one might build an application framework in SAS, using the simulation of a common traffic signal as a backdrop. For those not familiar with the object-oriented paradigm, I provide a brief overview in Appendix A intended primarily as a high level introduction and as an advocate for its inherent merits.

1ST ATTRIBUTE OF A CLASS – DATA MEMBERS:

Data members in a class are simply a grouping of variables (say, Name, Grade, and Age of a student). The Names, Grades, and Ages will vary from student to student, but every student will have one of each. To be in the same class, everyone must have identical data members. Often in a language like C++, pointer variables along with data structures like linked lists are used to gain access to specific objects. Some languages, like Visual Basic, allow one to gain access to objects by assigning them to a Collection. All of this should sound familiar to SAS hash table users. The data variables of the hash table behave exactly like class data members. The hash key variables allow us to directly access a specific group of data variables. Hence, emulating the data member portion of a class is straightforward. We will return to exactly how hash tables are used and code examples in a later section where we pull an entire class together.

For readers not already familiar with SAS hash tables, I encourage you to check them out if you do serious DATA step programming. Secosky and Bloom [3] provide a great introduction to the mechanics of the DATA step hash object as it was originally released. Ray and Secosky [4] give an update on functionality added to the hash object in later SAS releases. In addition to giving solid examples of syntax and how one might use DATA step hash objects; Dorfman and Vyverman [5] go behind the curtain to describe the mechanics of how hash tables in general are able to quickly access what appear to be randomly stored objects.

2ND ATTRIBUTE OF A CLASS – DATA FUNCTIONS:

The member functions of a class (interchangeably called behaviors and/or methods of a class) serve as its interface to the outside world. If one needs to set, retrieve, or manipulate the value of a member data field, it is usually done through one of the member functions. Enter PROC FCMP. Introduced for the DATA step in version 9.2, it allows one to write and compile functions and subroutines that can be called directly from a DATA step. By dedicating a specific subroutine to a specific class emulation, we can define a common set of operators to serve as its interface to the outside world. We will return to exactly how PROC FCMP is used and code examples in a later section where we pull an entire class together.

Like hash tables, for those not already familiar with PROF FCMP for DATA steps, I would encourage you to become familiar if you write complicated code in DATA steps. Peter Eberhardt [6] provides a very useful introduction.

3RD ATTRIBUTE OF A CLASS – SCOPE LIMITATION:

Members of a class can usually be defined as Public, Private, or Protected. This functionality allows the class developer the opportunity to hide the inner workings of a class from the class user while still allowing the user access to the class operators. In the Visual Basic example in the Introduction, I declared both the data members and the function as Public. This allows the data members to be manipulated directly by code outside of the class. The upside is simplicity. The code below assigns a value of 50 to the data member QtyOnHand for object clsItem of type CItem. What could be easier? The downside is loss of control. How do I block an end-user of the class library from storing a negative quantity? Normally, one would define data members as private and use public functions to store and retrieve data member values. Data validation is handled inside the member function. Further, public access forces the end-user to have a greater understanding of the inner workings of a class and in the long run, makes it more difficult to modify those inner workings. Think about the SAS hash tables, which are implemented by SAS as objects. As an end-user, your only concern is how to call the member functions (e.g. DEFINEKEY, FIND). Despite the added complexity, it is usually recommended that all communication between a class and the outside world be handled via member functions.

```
clsItem.QtyOnHand = 50
```

We will return to how we emulate Scope Limitation in a later section where we pull an entire class together..

4TH ATTRIBUTE OF A CLASS – TAG NAME:

Lippman's 4th and final attribute is simply a name that can be associated with the class for use in declaring objects of the newly defined class. The task is straightforward in a language where the compiler supports the object-oriented paradigm. Assume we have created a new class and called it "DistributionCenter". To declare a new variable, "myDC" of type "DistributionCenter" in the C++ language, we would simply write:

```
myDC DistributionCenter;
```

For our Visual Basic example from previous sections we would write:

```
Dim clsItem As CItem
Set clsItem = New CItem
```

To emulate this process, we employ a SAS Macro for each of our classes which, when called, will create a new object for the class. We will return to exactly how these macros are used and code examples in a later section.

SIMULATION OVERVIEW

The goal of this work is to lay the foundation for how one might build an application framework in SAS aimed at making specific types of simulations easier to perform in SAS. As the intent was not to actually investigate traffic light behavior, I purposely kept the class functions (behaviors) and data members (attributes) simple. I do not intend to report results from simulation runs; but, even though the included behaviors may be simple, they are functional and the full code has grown to over 1000 lines. Hence, the full code is available from the author by request.

Our simulated environment, which serves as the backdrop for our class emulation, consists of three entities ((1) Traffic Lights, (2) Roadways and (3) Vehicles) and an Event Queue that will serve as the traffic cop for messages between entities. Their class functions (behaviors) and data members (attributes) were purposely kept simple. The simulation is a discrete event process driven by an event queue. At simulation start-up, certain key events are pre-loaded to get the simulation up and running. As events are retrieved and executed from the event queue, they have the ability to place future events into the queue. Changing the color of a traffic light is a perfect example. As the light changes to green, it places a future event into the queue at the prescribed time to change the light to yellow.

The event queue is implemented as a regular hash table inside of the main DATA step (code segment below.) All actions that occur in the simulation (e.g. creation of a new traffic light, arrival of a vehicle at a traffic light, etc.) start out life as an event located in the event queue. The primary key for the hash table is TIME_STAMP, which depicts the time an event will occur in the simulation (in seconds from the start of the simulation). Using the "ordered"" option when declaring the hash table makes it possible to dynamically insert new events into the queue at their proper time. Using the "multidata" option allows more than one event to occur in the same time interval.

Besides TIME_STAMP, the hash table contains 5 data members. ENTITY_TYPE tells us which entity the event deals with (event queue, traffic light, roadway or vehicle.) ENTITY_ID tells us which particular entity we are dealing with – i.e. is it Vehicle #1 or Vehicle #2. ENTITY_ID_2, when used, points us to a secondary entity – say perhaps the particular roadway on which a vehicle is traveling. EVENT_CODE is set to the action to be performed on the entity, say, change the traffic light from red to green. Finally EVENT_TEXT is a context-encoded message that carries the information required to carry out the action on the entity. We will return to these parameters in the next section where we describe the Traffic Light class.

```
* declares for Event Queue Hash Table;
 length  Entity_Type $ 16;
 length  Event_Code $ 20;
 length  Event_Text $ 80;
 length  Entity_ID Entity_ID_2 Time_Stamp 8;
 declare hash EventQue(ordered:'a',multidata:'y');
 * EventQue Hash Table Definition;
 rc=EventQue.defineKey("Time_Stamp");
 rc=EventQue.defineData('Time_Stamp','Event_Code','Event_Text','Entity_Type',
                        'Entity_Id','Entity_ID_2');
 rc=EventQue.defineDone();
 call missing(Time_Stamp,Event_Code,Event_Text,Entity_Type,Entity_Id,Entity_ID_2);
```

Figure 1 on the next page gives a pictorial view of the Event Queue. Since I have used the 'multidata' attribute, each key for the hash table can be associated with multiple hash records. Using the 'ordered:a' attribute ensures that records will be inserted into the queue in ascending order. Since our key is Time-Stamp, which records simulation time in seconds from the beginning of the simulation, we can have multiple events occur in the same time period. Remember that for SAS hash tables, only the first record (which I call the primary record) can be found with the FIND method. Secondary events can only be found and accessed through FIND_NEXT and HAS_NEXT methods. Although functional, having to utilize two different access methods is a bit awkward. Since I knew going in that I wanted to run a cleanup process (i.e. collect stats, etc.) at the end of each time frame, I solved this problem by designating the primary record for each entry in the Event Queue as a simulation bookkeeping event. At the start of the simulation, a bookkeeping event is inserted into the Event Queue for each and every time period. Subsequently, actual simulation events are always inserted as secondary records in the Event Queue. The following code segment, which is the main processing loop for the simulation, takes advantage of this structure.

MAXTIME controls how long the simulation runs while "i" keeps track of the time of day. As it is set to 1 when we enter the DO WHILE loop, the initial FIND method retrieves the primary record for time period 1. (Note that error-checking code has been removed from this code snippet for both clarity and brevity.) Immediately, after locating the primary record for TIME STAMP=1, we use the HAS_NEXT method to determine if there are secondary events to be processed for the 1st second of the simulation. Assuming there are, the IF statement resolves to true and we find the 1st secondary event using the hash FIND_NEXT method and immediately delete it prior to processing it. We can do this because the values of the data members in the hash table have already been loaded into their DATA step variable counterparts. That is, just after the FIND_NEXT method is processed, two copies exist of the data variables for the current record in the hash table. The REMOVEDUP method removes the hash table copy without affecting the DATA step copy. We use the DATA step copy to process the event. Hence, the data step variable ENTITY_TYPE is now set to the type of entity we wish to process. Each object class that we define in our class

library must have its own unique value of ENTITY_TYPE. The SELECT statement below uses ENTITY_TYPE to call the subroutine that corresponds to that class. The call to TRAFFICLIGHT subroutine is shown (although the pre and post processing code has been removed for both clarity and brevity). Normally, in an object-enabled language, the compiler would make this connection for us. Since base SAS is not object-enabled, we have to take on this responsibility ourselves. But it is just a case statement. If in the future we wish to enhance our class library and add a new entity, we would need to write a new class subroutine, compile it with PROC FCMP, and add a new CASE to the SELECT statement below. As we will see in the next section, the variable EVENT_CODE, which carries the action to be performed on the entity, is used inside of a SELECT statement in the class SUBROUTINE to execute the appropriate "member function". After processing the current duplicate, the code looks for the next duplicate.

Once all duplicates for a given time unit are removed, the IF statement evaluates to FALSE causing the end of time unit record to be processed and the primary record for the time unit (i.e. the record associated with hash key value) is removed. This process guarantees that all events for a given time unit are processed in the order they were placed into the event queue followed by the simulation housekeeping event. I have removed the detailed code logic from the case statements below for brevity, as my goal here was to convey the structure of how events are processed. The full code for processing traffic light events is available from the author on request.

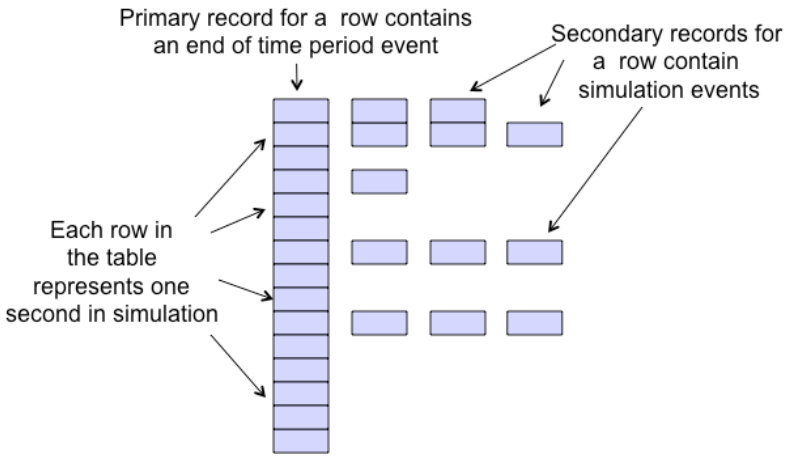

Figure 1. Simulation Event Queue

```
* Processing loop for the Event Queue";
 i=1;
 do while(i<=maxTime);
    Time_Stamp=i;
    rc = EventQue.find();
    EventQue.has_next(result: rcd);
    if rcd>0 then
       do;
         EventQue.find_next();
         EventQue.removedup();
         select (Entity_Type);
           when ("EventQue")     do;  * Event Queue code;    end;
           when ("RoadWay")      do;  * Roadway code;        end;
           when ("Vehicle")      do;  * Vehicle code;        end;
           when ("TrafficLight")
              do;
                . . .
                call TrafficLight(Event_Code,Entity_ID,Entity_ID_2,
                          Time_Stamp,Event_Text,nTrafficLights,rc);
                . . .
              end;
           otherwise             do;  * Error code;          end;
         end;
       end;
    else
       do;
         * Simulation Bookkeeping code;
         rc=EventQue.remove();
         i=i+1;
       end;
  end;
```

TRAFFIC LIGHT CLASS - CLASS EMULATION:

The ability to use hash tables inside of PROC FCMP, introduced in version 9.3, was the final piece to the puzzle. Henrick, et al [7] provides an introduction to this new capability, which we use to create our Traffic Light class in the following code segments. The subroutine that will serve as a container for our class emulation is shown in the top half of page 7 and has 4 major pieces: (1) the subroutine declaration, (2) local variable declarations, (3) a hash table declaration that implements the class members, and (4) a SELECT statement that implements the class operators. This structure is slightly different than one would find in an object-enabled language. Here, we must communicate directly with one subroutine, which in turn uses a case statement to perform required operations. Limitations as to how FMCP functions and hash tables seem to be able to communicate directly with each other led me to this alternate structure. Fortunately, the goal was to emulate object-oriented functionality, not object-oriented structure. In the true object-oriented case, we would access a class' method function directly. Here, we pass the name of the method as a parameter to a common function and utilize a select statement inside the subroutine to execute the required code - different structure, same result. See Appendices B and C for an overview of the classes for our other 2 entities in the simulation environment – VEHICLES and ROADWAYS.

Refer back to the Event Queue processing loop described earlier. When an event is pulled from the queue with an Event_Type of "TrafficLight", a call is made to the TRAFFICLIGHT subroutine. The first 5 parameters passed to the subroutine TRAFFICLIGHT directly align with data members of the event queue hash table. As the class operators may need to alter these parameters, each are included in the OUTARGS statement. The other two parameters allow the class to communicate error codes (RC) and the current number of traffic lights (nTrafficLights) stored in the hash table to the outside world.

By using the "ordered" attribute for the TRAFFICLIGHT hash table and assigning each new traffic light an integer hash ID, we are able to directly access a particular traffic light or sequentially walk through the set of traffic lights. Although the code to create the hash table is relatively straightforward with the normal DEFINEKEY, DEFINEDATA, DEFINEDONE, and CALL MISSING statements, there is one minor twist. These statements are separated from the variable and hash table declarations. We defer actually creating the hash table until we receive an event from the Event Processing Loop saying it is time to create the traffic light entity (i.e until we receive an Event that is pulled from the Event Queue which has an ENTITY_TYPE of "TRAFFICLIGHT and an EVENT_CODE of "TL_Create" (detail for TL_Create is shown in the 2nd code segment page 7.). Processing the "TL_Create" event leaves us with a functional, but empty, traffic light hash table. Traffic lights are added to the hash table via the "TL_ADD" event, which will be reviewed in depth below.

As previously discussed, my object-oriented structure relies on calling a single subroutine which then determines what action is required and executes the required method via a case statement. This implies that the arguments to the subroutine must be able to carry all required information for all methods – which means interpretation of those arguments is required. Also, those same arguments must be used to pass information back to the main-processing loop. As the TL_ADD method provides examples of both, I will use it to demonstrate how the objects are able to interact with the event queue and other objects. The primary purpose of the TL_ADD method is to respond to a request from the main event-processing loop to add a new traffic light entity to the traffic light hash table. In essence, create and initialize a new object of type TRAFFICLIGHT. A secondary purpose of TL_ADD is to create a new event that can be placed into the event queue that will change the state of the light (say from green to yellow) at some future time. Perhaps the best way to explain the TL_Add method and how objects interact with the Event Queue is to follow how the subroutine parameters are processed. Detail for the TL_ADD case of the SELECT statement is shown on Page 8. Complete code for the other cases in the select statement is available from the author by request

Let's assume that the current event pulled from the Event Queue has the following values.

- Entity_Type: TrafficLight
- Event_Code: TL_ADD
- Entity_ID: Null
- Entity_ID2: Null
- Time_Stamp: 10
- Event_Text: `'CommonDequindre,60,0.75,0.20,Normal GYR,Green'`

Looking back at the Event Processing Loop on page 4 shows that an entity type of "TrafficLight" induces a call to the subroutine TrafficLight. The rest of the Event Queue data members are passed as parameters in the call statement.

```
call TrafficLight(Event_Code,Entity_ID,Entity_ID_2, Time_Stamp,Event_Text,
                  nTrafficLights,rc);
```

The first parameter, Event_Code, is set to the method to be performed (TL_ADD in our example.) When its value is TL_ADD, the SELECT statement, about half way down the code segment on page 7, directs us into the TL_ADD case. As part of TL_ADD's secondary purpose (to add a future event to change the status of the light), it will change the value of Event_Code to "TL_Change_Light". By the time the TrafficLight subroutine has completed its execution, each of the changeable parameters passed to the subroutine will be reassigned values that are consistent with the creation of a new event on the event queue. The first order of business for the event processing loop when we

return from TL_ADD is to place a new entry into the event queue hash table (not shown in the code segment on page 4.) The reassignment of Event_Code is about half way through TL_ADD code segment on page 8.

The second parameter, Entity_ID, contains the value of a hash key for an existing TrafficLight object. Notice that its value was NULL in our example because we are creating a new object. For an event like TL_Change_Light, we would use the Entity_ID parameter to initialize the hash key (TL_ID) for the TRAFFICLIGHT hash table. Once TL_ID is set, the hash FIND method can be used to locate the data members associated with the particular traffic light of interest. For TL_ADD, which is creating a new traffic light, the value of Entity_ID passed into the subroutine is irrelevant. However, once TL_ID is set for the new light, we set Entity_ID equal to TL_ID (line 3 in the TL_ADD code). As Entity_ID is listed in the OUTARGS statement, it can be returned to the event-processing loop for use in creating the TL_Change_Light event described in the previous paragraph.

Entity_ID_2 is normally missing and not used; however, there are times when a method requires a pointer to multiple objects (e.g. one to the traffic light and one to a vehicle sitting at the traffic light.) TL_ADD does not use Entity_ID_2.

Time_Stamp is the time of the current event that is being processed (in seconds from the start of the simulation). It is also the hash key for the Event Queue. In the select statement at the bottom of TL_ADD, you can see how the time is set for when the light should change (i.e. the new Time_Stamp for the TL_Change_Light event that is being created).

nTrafficLights is the current count of TrafficLights stored in the traffic light hash table. TL_ADD will bump this parameter by 1 and return it to the outside world. It will also be used to establish the hash key for the new traffic light. See first two lines of TL_ADD.

RC is used to return error codes to the main event processing loop.

Each of these parameters has essentially the same meaning and usage in the other two other classes (VEHICLE and ROADWAY), although they have nVehicles and nRoadways parameters respectively in place of nTrafficLights.

Event_Text, on the other hand, is context driven and the meaning can change from Class to Class and Method to Method within a class. Although the content may be different, the structure is always the same. Event_Text is always built as a comma delimited string. The string below gives the meaning of each position for TL_Add. They provide for an alpha name for the traffic light, length of time it takes the light to cycle through the three states, percent of time it stays green, percent of time it stays red, operation mode (which is currently not used), and the initial state (i.e. Green-Yellow-Red). The scan function, as can be seen in the upper third of the TL_Add code segment, is used to unpack the string and store values in the appropriate hash variables. This may seem a little onerous, but remember, we are defining an class which we intend to user over and over again. Also, we are emulating what happens in an object-oriented code. In defining member functions in a language like C++, we would be writing functions and defining their parameters. I would argue that the Event_Text serves the same purpose. The only difference is that we have to unpack the parameters ourselves – but we knew going into this work that we were going to have to take on some compiler responsibilities.

```
"Alpha_Name,Cycle_Time,Pct_Green,Pct_Red,Operating Mode,Current_State"
```

A closer look at the Select statement inside of the Traffic Light subroutine shows that it currently has 12 different behaviors. Although there clearly isn't room here to show all of the code, I will at least provide a quick overview of each one's functionality.

We have already discussed "TL_Create" and "TL_ADD" which are used to create the traffic light hash table and add new traffic light objects to the hash table. We touched briefly on "TL_Change_Light" which does exactly as one might think – changes the light from green to yellow, yellow to red, etc. The "TL_Change_Light_Cycle_Time" allows you to change how long it takes to cycle through each of the states. Parameters to this event include what portion of time the light spends in each color. The idea was to allow a light to stay green longer on the main roadway late at night. TL_Get_Light_Status can be called by another entity to determine a lights status at any time. Although I did not include it, one could certainly see the need to turn a light from regular status to blinking at certain times of the day.

The TL_Set_Roadway and TL_FindByName methods are used in conjunction with methods in other entity classes to link specific roads, traffic lights, and vehicles in the network. TL_Put (a single light) and TL_Dump (all lights) can be used to dump the contents of the traffic light hash table to the SAS Log, primarily used for debugging purposes. Similarly, TL_Remove (a single light) and TL_Clear (all lights) can be used to remove lights from the hash table. Finally, TL_Destroy is used to remove the hash table completely.

This is hardly the complete set of behaviors one would want to do a serious traffic light study. On the other hand, it does provide a fairly rich set of operators to test whether or not we can emulate object-oriented functionality. Behaviors for our other two entities, Vehicles and Roadways, can be found in Appendices B and C respectively.

It is my belief, that the structure laid out in this section satisfies 3 of Lippman's 4 attributes for a Class - data members, data functions, and scope limitation. Although we have not discussed scope limitation, one could effectively hide the inner workings of the class library by distributing compiled code from FCMP. In the next sections, we will focus on the last attribute – tag names and how to use our new classes.

```
proc fcmp  outlib=work.funcs.test;
subroutine TrafficLight(Event_Code $,Entity_ID, Entity_ID_2,Time_Stamp,
           Event_Text $,nTrafficLights,rc);
 outargs Event_Code,Entity_ID, Entity_ID_2,Time_Stamp,Event_Text,nTrafficLights,rc;
 length  Event_Code $ 20;
 length  Event_Text $ 80;
 length  Entity_ID Entity_ID_2 Time_Stamp rc 8;
 length  nTrafficLights 8;
 length  ID_Found 8;
 length  ID_Name $ 16;
 length  Roadway_Direction $ 1;

 * declares for Traffic Light Hash Table;
 length  TL_ID  TL_Cycle_Time  TL_Pct_Green  TL_Pct_Yellow  TL_Pct_Red
         TL_Green_Secs  TL_Yellow_Secs  TL_Red_Secs 8;
 length  TL_Name $ 16;
 length  TL_Operation $ 16;
 length  TL_Curr_State TL_Curr_State_N TL_Curr_State_S TL_Curr_State_E
         TL_Curr_State_W $ 16;
 length  TL_Roadway_N  TL_Roadway_S  TL_Roadway_E  TL_Roadway_W  8;
 declare hash TrafficLight(ordered:'a');

select (Event_Code);
  when ("TL_Create")            do; * create hash table for traffic lights     end;
  when ("TL_Add")               do; * add traffic light to hash table;         end;
  when ("TL_Change_Cycle_Time") do; * change light cycle time;                 end;
  when("TL_Change_Light")       do; * change the color of the light;           end;
  when ("TL_Get_Light_Status")  do; * return the current light color;          end;
  when ("TL_Remove")            do; * remove a traffic light from hash table;  end;
  when ("TL_Set_Roadway")       do; * assign roadway to a traffic light;       end;
  when ("TL_Put")               do; * dump data for a traffic light to log;    end;
  when ("TL_Dump")              do; * dump data for all traffic lights to log; end;
  when ("TL_FindByName")        do; * given a light name, return its hash ID;  end;
  when ("TL_Clear")             do; * remove all lights from hash table;       end;
  when ("TL_Destroy")           do; * delete the traffic light hash table;     end;
  otherwise                     do; * error code for illegal Event code;       end;
 end;
endsub;


* Code for TL_CREATE Case extracted from the SELECT statement in above code;
  when ("TL_Create")
    do;
      nTrafficLights=0;
      rc=TrafficLight.defineKey('TL_ID');
      if rc=0 then
          rc=TrafficLight.defineData('TL_ID','TL_Name','TL_Cycle_Time',
             'TL_Pct_Green','TL_Pct_Red','TL_Green_Secs','TL_Yellow_Secs',
             'TL_Red_Secs','TL_Operation','TL_Curr_State','TL_Curr_State_N',
             'TL_Curr_State_S','TL_Curr_State_E','TL_Curr_State_W',
             'TL_Roadway_N','TL_Roadway_S','TL_Roadway_E','TL_Roadway_W');
      if rc=0 then rc=TrafficLight.defineDone();
      call missing(TL_ID,TL_Name,TL_Cycle_Time,TL_Pct_Green,TL_Pct_Red,
                   TL_Green_Secs,TL_Yellow_Secs,TL_Red_Secs,TL_Operation,
                   TL_Curr_State,TL_Curr_State_N,TL_Curr_State_S,TL_Curr_State_E,
                   TL_Curr_State_W,TL_Roadway_N,TL_Roadway_S,
                   TL_Roadway_E,TL_Roadway_W);
      if rc > 0 then
         do;
           Event_Text="Error Creating the Traffic Light Hash Table";
           rc=9999;
           return;
         end;
      return;
    end;
```

```
* Code for TL_ADD Case extracted from the SELECT statement in above code;
when ("TL_Add")
  do;
    nTrafficLights=nTrafficLights+1;
    TL_ID=nTrafficLights;
    Entity_ID=TL_ID;
    TL_Name=Scan(event_text,1,",");
    TL_Cycle_Time=scan(event_text,2,",");
    TL_Pct_Green=scan(event_text,3,",");
    TL_Pct_Red=scan(event_text,4,",");
    TL_Operation= scan(event_text,5,",");
    TL_Curr_State=scan(event_text,6,",");
    TL_Curr_State_N=TL_Curr_State;
    TL_Curr_State_S=TL_Curr_State;
    if TL_Curr_State="Red" then
       do;
         TL_Curr_State_E="Green";
         TL_Curr_State_W="Green";
       end;
    else
       do;
         TL_Curr_State_E="Red";
         TL_Curr_State_W="Red";
       end;
    TL_Green_Secs=min(round(TL_Cycle_Time*TL_Pct_Green,1),TL_Cycle_Time);
    TL_Red_Secs=min(round(TL_Cycle_Time*TL_Pct_Red,1),
                  TL_Cycle_Time-TL_Green_Secs);
    TL_Yellow_Secs=TL_Cycle_Time-TL_Green_Secs-TL_Red_Secs;
    TL_Roadway_N=0;
    TL_Roadway_S=0;
    TL_Roadway_E=0;
    TL_Roadway_W=0;
    rc=TrafficLight.Add();
    if rc > 0 then
      do;
        Event_Text="Fatal Error: Problem adding new Traffic Light to Hash Table";
        rc=9999;
        return;
      end;
    Event_Code="TL_Change_Light";
    Entity_ID=TL_ID;
    put TL_Curr_State;
    select(TL_Curr_State);
      when ("Green")
        do;
          Time_Stamp=Time_Stamp+TL_Green_Secs;
          Event_Text="Green to Yellow";
        end;
      when ("Yellow")
       do;
         Time_Stamp=Time_Stamp+TL_Yellow_Secs;
         Event_Text="Yellow to Red";
       end;
     when ("Red")
       do;
         Time_Stamp=Time_Stamp+TL_Red_Secs;
         Event_Text="Red to Green";
       end;
     otherwise
       do;
         Event_Text="Fatal Error: Curr State for LT invalid";
         rc=9999;
       end;
    end;
    return;
  end;
```

SIMULATION STRUCTURE

The goal of this work is to demonstrate that it is possible to implement an object-oriented Application Framework inside base SAS, using the simulation of a simple traffic signal as the backdrop. The following pseudo code segment shows the structure of what such a simulation might look like. First PROC FCMP is used to define each class of objects to be used in the simulation. Our class library/application framework is essentially a set of subroutines housed inside of PROC FCMP. This is the static portion of the simulation.

Then a large DATA Step is developed. The top of the DATA Step includes the declaration of local variables and the Event Queue hash table described in earlier sections. This part of the code is also static. The middle section of the DATA Step is made up of declarations for objects (entities like traffic lights, roadways, etc.) that will be used in the simulation as well as key initialization events that are placed into the event queue to get the simulation up and running (e.g. the time bookkeeping events described in the previous section). This is the dynamic portion of the code and will change from simulation to simulation. The bottom of the DATA Step is the Event Queue processing loop discussed previously, which is also static. The remainder of this paper will describe how we dynamically build the middle portion of the data step to create a unique simulation program.

The power in the structure laid out below is that about 90% of the code is reusable from simulation to simulation. Unless you need to extend the number of types of objects available in the class library or change one or more of their behaviors, only the dynamic portion of the code below must be rewritten to perform a new simulation. Many of us tend to work in the same space over a long period of time. For me it has been order fulfillment and logistics. Manufacturing plants, trains, trucks, roadways, warehouses, etc. etc. always seem to be involved in these types of modeling efforts. Unless they are the specific are of study, the data and behaviors of most of these entities do not change from modeling effort to modeling effort. What normally changes is how we connect the dots. Perhaps we are changing routes. Or maybe we want to experiment with extending the boundary line between whether we use truck or rail for shipping? To answer these types of questions with the structure laid out below, one would only need to change the dynamic portion of the code to build a new simulation.

Another strength of this structure is that the static portions of the code can be developed in a central group and distributed to an end-user community who only needs to understand the class interface to use the functionality in their own code. Think about the SAS hash tables we have been discussing throughout this paper. Clearly these have been developed by SAS as classes and we, the end-users, access their functionality through the member functions (object.defineKey, object.defineData, object.find, etc.) Unfortunately for us, we cannot define our own new behaviors. (If we could, I wouldn't be writing this paper.)

```
*  Structure of SAS code created using the Application Framework;
proc fcmp  outlib=work.funcs.test;
 subroutine TrafficLight(. . .);
   /*  Traffic Light member data and common code */
 endsub;
subroutine Roadway(. . .);
   /*  Roadway member data and common code */
 endsub;
subroutine Vehicle(. . .);
   /*  Vehicle member data and common code */
 endsub;
quit;
run;

Data _NULL_;
  /* Local declarations */
  /* Event Queue hash table declaration */
  /* Dynamically generated Object declarations */
  /* Initialization Events loaded into the event queue */
  /* Event Queue processing loop */
run;
```

CLASS DEFINITION MACROS:

Remember that the TRAFFICLIGHT class developed in the previous section was just a user defined data type. It is simply an extension of the SAS language sitting on the shelf waiting to be used. We need a mechanism to be able to create a new variable of type TRAFFICLIGHT. The SAS Macro TrafficLight_Create, shown below, provides this capability. Notice how the parameters for the macro match up with the comma delimited parameters in the Event_Text we examined in a previous section. The macro uses them to create 7 lines of SAS code that are stored in a text file. When executed, these 7 lines of code will create a TL_ADD event and place it at its appropriate time slot in the Event Queue. When the appropriate time occurs in the simulation, a new variable of type TRAFFIC light will be created and available for use in the simulation. Since data member storage for the TRAFFICLIGHT class was implemented as a hash table, multiple calls to the TRAFFICLIGHT macro will result in the creation of multiple traffic lights for use in the simulation. Similar macros are written for each entity. A call to this macro gives us the ability to emulate Lippman's 4th attribute: a tag name for the class. We now have a set of 3 classes (TRAFFICLIGHT, VEHICLES, and ROADWAYS) designed to extend the capability of running traffic light simulations in SAS and the means to create variables that are of those classes – an Application Framework. We will see how to use this framework to write and execute simulations in the next section.

```
%macro TrafficLight_Create(Time_Stamp,LightName,CycleTime,
                           Green_Pct,Red_Pct,Operation,Initial_Color);
data _null_;
 file Outf mod;
 put @2 "Time_Stamp=&Time_Stamp.;";
 put @2 "Entity_Type='TrafficLight';";
 put @2 "Event_Code='TL_Add';";
 put @2 "Entity_ID=.;";
 put @2 "Entity_ID_2=.;";
 put @2 "Event_Text='&LightName.,&CycleTime.,&Green_Pct.,&Red_Pct.,&Operation.,&Initial_Color.';";
 put @2 "if Time_Stamp<maxTime then rc=EventQue.Add();";
run;
%mend;
```

WRITING SIMULATIONS USING THE APPLICATION FRAMEWORK:

The first code segment shown on the next page, coupled with the class definition macros described in the previous section, resolves to a complete SAS program that will build and execute a specific traffic light simulation. The code starts out with a simple DATA step that makes one call to the TRAFFICLIGHT_CREATE macro and four calls to the ROADWAY_CREATE macro. These 5 macro calls generate the second code segment shown on the next page. The code is stored in a text file on the local hard drive (CODESTUB.SAS). As you examine this code notice one thing in particular, there are no DATA STEP or RUN statements. It is an open block of SAS code.

After the initial DATA step, you see five %INCLUDE statements. These simple five statements build and execute a large, single DATA step that is to be our simulation. The DATA step is made up of several static pieces that do not change from run to run and the one dynamic piece (CODESTUB.SAS) that is created by our macro calls.

The first #INCLUDE contains our class library. It is a large FCMP procedure with the subroutines that define our classes. Although we may do maintenance to the class library from time to time (adjust, remove, add classes), at simulation development time it is a static piece of code. The implication is that this library could be distributed to a wider user group who only need to understand the macro interface to make use of the library.

The second %INCLUDE file contains the DATA Step statement for the simulation run, local variable declarations, and the declaration for the Event Queue hash table. Initially, I had built the Event Queue as a class and placed it as part of the class library. Conceptually, it felt like it was the right place for it. But as hash tables in PROC FMCP do not support the MULTIDATA option, I moved the Event Queue into the main body of the data step. There was no real harm from this choice as each simulation has one, and only one, Event Queue.

The third %INCLUDE brings in our dynamic code built by the macro calls and stored in CODESTUB.SAS. Since the DATA step code included previously did not contain a RUN statement, this new code is simply appended at the end of the code already brought in but not yet compiled or executed.

The fourth %INCLUDE brings in optional code that can be used for debugging. Appendix E gives examples. One can use this code to place events into the Event Queue at key times throughout the simulation to keep track of the simulation's progress. In the sample code in Appendix E, I dump the contents of various objects to the SAS log.

The fifth %INCLUDE contains the SELECT statement which processes the Event Queue and has the RUN statement which triggers the execution of the simulation. Hence, code brought in by the first four include statements is all set-up and initialization, while the code brought in the final include executes the simulation.

Notice how the structure implied by the 5 #INCLUDE statements matches that shown in the previous section on SIMULATION STRUCTURE. Assuming we are satisfied with the current behavior of the objects available in our

class library, we can write an infinite number of simulations simply by changing macro-calls and their parameters in the initial data step. Further, with proper documentation of the required parameters, we could distribute this code to other users and they could write their own simulations simply by creating their own DATA Step of macro calls.

As I have said before, the current simulation is rather trivial. To be of real use, we would have to provide a mechanism to initialize general parameters of the simulation (e.g. how long should it run, warm-up period before collecting stats, etc.) Initially, when I intended to implement the event queue as a class, my plan was to have an event queue creation macro like the one shown previously for traffic lights. But other mechanisms could be used.

Also, if you do get the full-code, you'll quickly see that the functionality provided is indeed trivial. You can't do much more than simulate how long cars have to sit at one light. You would have to add more functionality (probably need something like a "Grid" entity and the ability for vehicles to move from the control of one light to another.) But, again, my primary purpose here was to demonstrate that one could emulate an object-enabled environment in base SAS.

```
* Mainline for Traffic Light Simulation;
data _null;
 file Outf;
 put @1 "*Start of Dynamic Object Generation;";
%TrafficLight_Create(1,CommonDequindre,60,0.75,0.20,Normal GYR,Green);
%Roadway_Create(1,CommonDequindre,Dequindre,N,45,5,Poisson,0.5,2,0.05,0.05);
%Roadway_Create(1,CommonDequindre,Dequindre,S,45,3,Poisson,0.5,2,0.05,0.05);
%Roadway_Create(1,CommonDequindre,Common,E,25,10,Poisson,0.5,2,0.6,0.3);
%Roadway_Create(1,CommonDequindre,Common,W,25,10,Poisson,0.5,2,0.3,0.6);
run;
%include "C:\SASCode\TL_Include_Part1.sas";
%include "C:\SASCode\TL_Include_Part2.sas";
%include "C:\SASCode\codestub.sas";
%include "C:\SASCode\TL_Include_Part3.sas";
%include "C:\SASCode\TL_Include_Part4.sas";
run;


*Start of Dynamic Object Generation;
 Time_Stamp=1;
 Entity_Type='TrafficLight';
 Event_Code='TL_Add';
 Entity_ID=.;
 Entity_ID_2=.;
 Event_Text='CommonDequindre,60,0.75,0.20,Normal GYR,Green';
 if Time_Stamp<maxTime then rc=EventQue.Add();
 Time_Stamp=1;
 Entity_Type='RoadWay';
 Event_Code='RW_Add';
 Entity_ID=.;
 Entity_ID_2=.;
 Event_Text='CommonDequindre,Dequindre,N,45,5,Poisson,0.5,2,0.05,0.05';
 if Time_Stamp<maxTime then rc=EventQue.Add();
 Time_Stamp=1;
 Entity_Type='RoadWay';
 Event_Code='RW_Add';
 Entity_ID=.;
 Entity_ID_2=.;
 Event_Text='CommonDequindre,Dequindre,S,45,3,Poisson,0.5,2,0.05,0.05';
 if Time_Stamp<maxTime then rc=EventQue.Add();
 Time_Stamp=1;
 Entity_Type='RoadWay';
 Event_Code='RW_Add';
 Entity_ID=.;
 Entity_ID_2=.;
 Event_Text='CommonDequindre,Common,E,25,10,Poisson,0.5,2,0.6,0.3';
 if Time_Stamp<maxTime then rc=EventQue.Add();
 Time_Stamp=1;
 Entity_Type='RoadWay';
 Event_Code='RW_Add';
 Entity_ID=.;
 Entity_ID_2=.;
 Event_Text='CommonDequindre,Common,W,25,10,Poisson,0.5,2,0.3,0.6';
 if Time_Stamp<maxTime then rc=EventQue.Add();
```

DISCUSSION

In a previous MWSUG paper, Frick [8], I spent considerable time extolling the virtues of writing code in SAS. But until recently (Version 9 and later), programming in Data Steps in base SAS suffered due to a couple of rather large omissions – no subroutines and lack of data structures that can acquire memory at run time. Opening up PROC FCMP to the Data Step in Version 9.2 solved the subroutine issue. Introduction of the Data Step Hash Object in Version 9 (with many improvements in subsequent releases) has gone a long ways towards solving the other. These new capabilities open up a whole host of new possibilities. If you are interested in how one might implement many of the traditional data structures in SAS (e.g. stacks, linked lists, etc.), take a look at Shawn Edney's paper [9].

In Version 9.3, SAS extended the Data Step Hash Object to PROC FCMP for the Data Step. In this work, I have used this newest capability to emulate a data structure only found in object-oriented programming languages – the user defined data type, or Class. Further, I have demonstrated how one could use these new capabilities to build a class library, which results in a set of off-the-shelf objects that can quickly be mixed and matched to form an infinite number of different simulations -- i.e. object-oriented programming in base SAS.

APPENDIX A – WHY USE THE OBJECT-ORIENTED PARADIGM FOR SIMULATIONS?

Let's assume you are responsible for network design of an outbound logistics operation. You need to select routes, locate warehouses, cross-docks, etc. As the world is a dynamic place, your best solution today may not be your best solution tomorrow. Even if you have the best design, you will find yourself constantly challenged by leadership to test out new ideas that have the potential to improve efficiency. Hence, you need a fast, efficient way to model your network. In selecting a modeling environment, you really have three choices: (1) simplify the problem so that it can be solved with equations, (2) build a traditional data – code – output solution, or (3) build a solution using object-oriented design principals.

If one can solve the math, the first solution can be a very effective tool. My old operations research department at General Motors Research developed several very successful tools using this approach. Blumenfeld, et all [10] documents an application in inbound logistics in which the author participated. Even so, there are two potentially serious downsides to this approach. First, one may need to over-simplify the problem to solve the math. Second, and perhaps more serious, it may not be possible to evaluate alternative scenarios that call for even slightly different mathematical characteristics.

Many quality, high impact simulations have been developed using a procedural approach where data is "pushed through" the model. Unfortunately, separating the data from their operators tends to lead to models that are difficult to modify and reduces the chance of *quickly* reusing computer code from one modeling effort to the next.

On the other hand, an object-oriented approach to model building has the potential to greatly increase the chance of reusing computer code. The approach encourages one to think in depth about the entities that exist in the environment being modeled and how they interact with the outside world. Although they may be mixed and matched in different ways, many of these entities and their standard behaviors will always exist in any simulation for a given environment. If we return to the world of logistics, there will always be distribution centers, trains, trucks, ships, load-makeup times, inspections, etc.

Further, the approach encourages one to think about entities in groups that behave similarly. A major feature of object-oriented systems that I did not attempt to emulate in the main paper is the ability to define sub-classes of an existing class. The sub-class automatically inherits the members of the parent class. Distribution centers are a perfect example as they come in many flavors. They can be simple a pass-through where packages that come in are simply re-routed to the next destination. They can be cross-docks where packages are accumulated and held for specific destinations at prescribed shipping frequencies. They can be storage locations. The point is, a master class can be defined that contains all members that are common to all distribution centers. This code need only be written once. If it needs to be modified at a later time, it only needs to be modified once. The beauty of this approach is that new functionality can be added simply by deriving a new sub-class.

There are theoretical reasons to support this approach as well. Lewis Caroll, in his book on symbolic logic [11], defines a class as:

> "'Classification,' or the formation of Classes is a Mental Process, in which we imagine that we have put together, in a group, certain things. Such a group is called a '**Class**.'."

In the same work, Lewis Caroll addresses the concept of sub-classes and inheritance, calling this process '**Division**' and he maintains that this is also a mental process. Classification is a normal process. It is how we humans make sense out of chaos.

APPENDIX B – VEHICLE CLASS

```
subroutine Vehicle(Event_Code $,Entity_ID, Entity_ID_2,Time_Stamp,
          Event_Text $,nVehicles,rc);
 outargs Event_Code,Entity_ID, Entity_ID_2,Time_Stamp,Event_Text,nVehicles,rc;
 length  Event_Code $ 20;
 length  Event_Text $ 80;
 length  Entity_ID Entity_ID_2 Time_Stamp rc 8;
 length  nVehicles 8;
 length  avg_distance max_distance std mph seconds_per_mile
         cv r t Pct_Left Pct_Right 8;
 * declares for Vehicle Hash Table;
 length VE_ID VE_RW_ID VE_TL_ID VE_Roadway_Arrival_Time
        VE_TrafficLight_Arrival_Time VE_TrafficLight_DepartureTime  8;
 length VE_Turn_Direction VE_Status $ 8;
 length VE_RW_Direction $ 1;
 declare hash Vehicle(ordered:'a');
 select (Event_Code);
  when ("VE_Create")      do; * create the vehicle hash table;                end;
  when ("VE_Add")         do; * add a vehicle to the hash table;              end;
  when ("VE_TL_Arrival")  do; * process a vehicle arrival at a light;         end;
  when ("VE_Green_Light") do; * vehicle at a light that just turned to green; end;
  when ("VE_Put")         do; * dump a vehicle to the log;                    end;
  when ("VE_Save")        do; * dump contents of hash table to output window; end;
  when ("VE_Dump")        do; * dump all vehicles to the log;                 end;
  when ("VE_Remove")      do; * remove a vehicle from the hash table;         end;
  when ("VE_Clear")       do; * remove all vehicles from the hash table;      end;
  when ("VE_Destroy")     do; * delete the vehicle hash table;                end;
  otherwise               do; * return an error code;                         end;
 end;
endsub;
```

APPENDIX C: ROADWAY CLASS

This appendix contains the high level structure of the roadway class. The underlying structure is identical to the Traffic Light Class described in detail in the main paper. Detailed code for the methods available in the roadway class (see case statement below) can be found in the overall code listing, available from the author on request.

```
subroutine RoadWay(Event_Code $,Entity_ID, Entity_ID_2,Time_Stamp,Event_Text $,
                   nRoadways,rc);
 outargs Event_Code,Entity_ID, Entity_ID_2,Time_Stamp,Event_Text,nRoadways,rc;
 length  Event_Code $ 20;
 length  Event_Text $ 80;
 length  Entity_ID  Entity_ID_2 Time_Stamp Vehicle_Arrival_Time rc 8;
 length  nRoadways 8;
 * declares for Roadway Hash Table;
 length RW_ID  RW_TL_ID  RW_Speed RW_Arrivals RW_Average_Distance
        RW_Maximum_Distance RW_Pct_Left RW_Pct_Right  8;
 length RW_TL_Name RW_Name RW_Entry_Distribution $ 16;
 length RW_Direction $ 1;
 declare hash Roadway(ordered:'a');
 * methods for Roadway Hash Table;
 select (Event_Code);
  when ("RW_Create")              do; * create the roadway hash table ;        end;
  when ("RW_Add")                 do; * add a new roadway to the hash table ;  end;
  when ("RW_VE_ArrivalTime")      do; * set arrival time for next vehicle ;    end;
  when ("RW_Get_TL_ID")           do; * find traffic light ID for the roadway; end;
  when ("RW_Set_VE_Arrival_Time") do; * get the vehicle time between arrivals; end;
  when ("RW_Put")                 do; * dump a particular roadway to the log;  end;
  when ("RW_Dump")                do; * dump all roadways to the log;          end;
  when ("RW_Remove")              do; * Remove a roadway from the hash table;  end;
  when ("RW_Clear")               do; * Remove all roads from the hash table;  end;
  when ("RW_Destroy")             do; * Delete the roadway hash table;         end;
  otherwise                       do; * Return error code;                    end;
 end;
endsub;
```

APPENDIX D: MANUALLY CREATED CODE – DATA SET "TL_INCULDE_PART4.SAS"

This appendix contains the manually created code that will place new events on the Event Queue, which will subsequently dump information to the SAS log that will be useful in debugging the simulation run/results. Note that this code is a partial block of SAS Data step statements that cannot stand on their own.

```
Time_Stamp=7;
Event_Code='RW_Dump';
Event_Text='Dumping the entire RW hashtable';
Entity_Type='RoadWay';
Entity_Id=1;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=11;
Event_Code='VE_Dump';
Event_Text='Dumping the entire VE hashtable';
Entity_Type='Vehicle';
Entity_Id=.;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=30;
Event_Code='TL_Put';
Event_Text='Test Dumping a selected TL';
Entity_Type='TrafficLight';
Entity_Id=1;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=55;
Event_Code='TL_Dump';
Event_Text='Test Dumping the entire TL Hash table';
Entity_Type='TrafficLight';
Entity_Id=.;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=155;
Event_Code='TL_Put';
Event_Text='Test Dumping a selected TL';
Entity_Type='TrafficLight';
Entity_Id=1;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=157;
Event_Code='TL_Change_Cycle_Time';
Event_Text='40,0.6,0.30';
Entity_Type='TrafficLight';
Entity_Id=1;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=159;
Event_Code='TL_Put';
Event_Text='Test Dumping a selected TL';
Entity_Type='TrafficLight';
Entity_Id=1;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Time_Stamp=maxTime-1;
Event_Code='VE_Dump';
Event_Text='Dumping the entire VE hashtable';
Entity_Type='Vehicle';
Entity_Id=.;
Entity_ID_2=.;
if Time_Stamp<maxTime then rc=EventQue.Add();
Event_Code='VE_Save';
if Time_Stamp<maxTime then rc=EventQue.Add();
```

REFERENCES

[1] Lippman, Stanley B. 1991. C++ Primer, Second Edition. Addison-Wesley Publishing Company.

[2] Excel VBA (Part of Microsoft Office Professional Edition) [computer program]. Microsoft; 2013.

[3] Secosky Jason, Bloom Janice (2007). Getting Started with the DATA Step Hash Object. Proceedings of the Eighteenth Midwest SAS Users Group Conference. paper SAS08.

[4] Ray Robert, Secosky Jason (2008). Better Hashing in SAS 9.2. SAS Global Forum 2008. paper 306-2008.

[5] Dorfman Paul, Vyverman Koen (2006).Data Step Hash Objects as Programming Tools. SUGI 31, San Franciso, CA. 2006. paper 241-31.

[6] Eberhardt Peter (2009). A Cup of Coffee and Proc FCMP: I Cannot Function Without Them. SAS Global Forum 2009. paper 147-2009.

[7] Henrick Andrew, Erdman Donald, Christian Stacey (2013). Hashing in PROC FCMP to Enhance Your Productivity. SAS Global Forum 2013. Paper 129-2013.

[8] Frick Michael (2012). Getting the Code Right the First Time. Proceedings of the Twenty-Third Midwest SAS Users Group Conference. Paper SA04-2012.

[9] Endey, Shawn (2009). Creating Common Information Structures Using List's Stored in Data Step Hash. SAS Global Forum 2009. Paper 011-2009.

[10] Blumenfeld, D.E., Burns, L.D., Daganzo, C.F., Frick, M.C., and Hall, R.W., 1987. Reducing Logistics Costs at General Motors, Interfaces, 17, 26-47.

[11] Caroll Lewis, 1958. Symbolic Logic, Part I, Elementary, New York, N.Y. Dover Publications.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Michael C. Frick

Enterprise: General Motors, Retired

Address: 30238 Underwood Drive
City, State ZIP: Warren, MI 48092
Work Phone: 586-573-0977
Fax: N/A
E-mail: mcfdaf001@yahoo.com
Web: N/A

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® Indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies