ARMing SAS® to Answer Your Questions
Fabian Robinson, PJM Interconnection, Norristown, PA

ABSTRACT
With the advent of more complex and distributed systems, businesses and enterprises have not been able to truly characterize system and/or network consumption on a detailed scale such as a specific application or workload. Businesses were not able to answer the following questions easily (if at all):

- Are transactions completing successfully?
- Where are the bottlenecks?
- What are my response times?
- What are the applications doing with the information?
- What servers are being used?
- How is the application component performance?

Several vendors (Tivoli, HP, SAS etc) united to create a vendor neutral open API called Application Response Measurement (ARM). ARM allowed capacity planners, business people and system administrators to answer the questions asked globally, but also at the application level by business need.

INTRODUCTION
This paper discusses what ARM is and how to access the ARM API from within SAS. Though ARM is vendor neutral, the scope of this paper will focus mostly on the SAS perspective and routines, with references to the ARM API objects, classes and routines.

WHY IS ARM SO DIFFERENT?
Monitoring of an application is usually from the application perspective, which gives the business the freedom to decide what is important for measurement and monitoring. The ARM API defines transactions into units of work, which can then be collected and be made into useful information by an application. By calling the API at the beginning and the end of the transactions one can measure and monitor these transactions. Previously assumptions had to be made as to what is a transaction, when it begins, when it ends, and also if it ended successfully. Though useful, these were guesses and were not always accurate, hence the need for ARM.

ARM VERSIONS
ARM 1.0 was first introduced in SAS version 8.2 in 1997. Subsequently, version 2 of ARM was shipped with SAS version 9 in 2002. One main benefit of ARM version 2 is that it not only allows one to identify transaction issues on one system but also external transactions that interact with the system. ARM version 4 is currently being developed.

WILL ARM AFFECT YOUR APPLICATION?
Yes, but only minimally: ARM agents are designed to quickly extract information needed and return control to the calling SAS application. Processing of the information gathered does not have to happen immediately but can be deferred to another time when more resources are available.

GETTING STARTED
Before anything else there must be a clear understanding and definition of what business transactions in a specific application need to be measured. Usually these are the “loved ones”: the transactions that the user/business community are usually most familiar with and also have the greatest business impact. Once identified, the application can be ARMed. At that point, ARMers can be strategic about how to modify the code because they have implicit knowledge of where to embed the necessary ARM macros to capture the beginning and the end of the identified transactions.
HOW DOES ARM WORK WITHIN SAS?
SAS has developed SAS macros to interface with the ARM API functions. These macros interact with an executable module within SAS known as the ARM Shared Library Module (ASLM). It is important to note that the ASLM is what defines the ARM function calls and parameters according to the ARM API specification. The ASLM passes the performance data (results) to a Measurement Agent Module (MAM) that is responsible for logging the information to any of the two places below:

1. ARMLOG.LOG in version 9 or the SAS Log in version 8.
2. Redirected Path via the ARMLOG fileref in the current SAS session (both versions of SAS) or (in version 9) the ARMLOC option which sets the output location.

SAS also supports other vendor’s ASLMs. An example of this is HP’s Measureware, which has its own MAM for logging the results to its performance database.

At its very core the ARM API is made up of three object classes:

- applications
- transaction classes
- transaction instances

APPLICATIONS
This defines the scope of the system you are ARMing. The SAS interface with ARM provides totals on a per application basis, which is a major consideration when implementing the ARM macros in your code. Associated with this object class is a unique numeric identifier or ID called the application ID.

TRANSACTION CLASSES
This is the unit of work or “loved one”. These are the transactions that have the most visibility to the end user and business impact within the application. Similar to Applications above, associated with this object class is a unique numeric ID called the transaction class ID.

TRANSACTION INSTANCES
Specification of the actual start time for a “loved one”: transaction instances have response time information that is associated with them. To be consistent with the previous classes, associated with this object class is a unique numeric ID called start handles for transaction instances.

IDs are signed integers and it is the MAM that usually assigns them. Depending on the vendor, their MAM may allow you to pre-assign IDs.

ARM API CALLS AND CORRESPONDING SAS MACROS
The following are the Arm API calls used (can be found on the SAS Support website):

- **Arm_init**
  Initializes the ARM environment and names the application and optionally the user of the application. An application id is returned to the calling program. This ARM API call is typically coded once during application initialization.

- **Arm_getid**
  Accepts an application id as input and names a transaction class. A *transaction class id* is returned to the calling program. Several arm_getid calls are typically coded during application initialization after the arm_init call in order to describe the type of transactions that the application will utilize.

- **Arm_start**
Signals the start of a transaction instance. An input transaction class id is used to identify the transaction class that is associated with this transaction instance. A start handle is returned to the calling program. An arm_start call is inserted before every transaction you want to measure.

- **Arm_update**
  Specifies an optional function that you can use between an arm_start and arm_stop in order to supply information about the transaction that is in progress. An input start handle is used to identify the transaction instance and an output return code is returned to the calling program.

- **Arm_stop**
  Signals the end of a transaction. You can supply an optional transaction status code as input to the call that describes the success or failure of the transaction. An input start handle is used to identify the transaction instance and an output return code is returned to the calling program. An arm_stop call is placed in the code where the transaction is known to be complete.

- **Arm_end**
  Terminates the arm environment and signals the end of an application. Its primary purpose is to free any memory that is associated with the application and to log any final application statistics. An input application id is used to identify the application and an output return code is returned to the calling program.

<table>
<thead>
<tr>
<th>ARM API CALL</th>
<th>Corresponding SAS macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm_init</td>
<td>%arminit</td>
</tr>
<tr>
<td>Arm_getid</td>
<td>%armgtid</td>
</tr>
<tr>
<td>Arm_start</td>
<td>%armstrt</td>
</tr>
<tr>
<td>Arm_update</td>
<td>%armupdt</td>
</tr>
<tr>
<td>Arm_stop</td>
<td>%armstop</td>
</tr>
<tr>
<td>Arm_end</td>
<td>%armend</td>
</tr>
</tbody>
</table>

**TYPICAL SAS FLOW**

Below is a typical flow obtained from the SAS Support website. This displays the SAS only flow. The SAS code is modified with the ARM macros to measure various transactions. These calls are executed within SAS, and the ASLM then contacts the ARM API functions, which send the calls to the MAM for processing. MAM then outputs the results to the specified output area, in this case the default area ARMLOG file. These results can then be post-processed into datasets and views from which reports can be created.
Fine -- we now have macros and have a general idea of how they work, so let us get into some more detail. For these macros to work effectively they must communicate. They do this through the use of variables which pass ID’s and other information. These macros will function in a data step as well as across data steps, so these variables could be either macro or data step variables. Here are some general rules of thumb regarding these variables and the macros:

- When the ARM calls are in the same data step, temporary SAS variables are used
- For a more global situation, such as data step to data step, global SAS macro variables are used to pass information
- Last but not least, if an SCL program is used then SCL variables are used
- SAS uses the prefix “_ARM” for its variables so avoid using this prefix
- More than one macro can be used within a data step or SCL program
- None of the macros use positional parameters
- All parameters have to be valid SAS variables
- So as not to include these variables in the output dataset, drop statements are generated

Below is a table obtained from SAS Support website of the internal variables which are utilized:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Set by</th>
<th>Used as input by</th>
</tr>
</thead>
<tbody>
<tr>
<td>_armapid</td>
<td>application ID</td>
<td>%arminit</td>
<td>%armgtid %armend</td>
</tr>
<tr>
<td>_armtxid</td>
<td>transaction class ID</td>
<td>%armgtid</td>
<td>%armstart</td>
</tr>
<tr>
<td>_armshdl</td>
<td>start handle</td>
<td>%armstrt</td>
<td>%armupdt %armstop</td>
</tr>
<tr>
<td>_armrc</td>
<td>error status</td>
<td>%armupdt, %armstop,</td>
<td>%armend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%armgend</td>
<td></td>
</tr>
<tr>
<td>_armglvl</td>
<td>global level indicator</td>
<td>calling program</td>
<td>all</td>
</tr>
<tr>
<td>_armtvl</td>
<td>target level indicator</td>
<td>calling program</td>
<td>all</td>
</tr>
<tr>
<td>_armexec</td>
<td>global enablement</td>
<td>calling program</td>
<td>all</td>
</tr>
</tbody>
</table>
There are many ways to actually implement the ARM macros. However, for the purposes of this paper I will keep it brief to help the reader get started. I have broken the examples down into two subcategories: macros within data steps and macros outside of data steps.

### MACROS WITHIN DATA STEP

```sas
data _null_;  
call symput('armexec','1');  
run;  
```

This step enables the ARM macros, which by default are disabled.

```sas
data _null_;  
%arminit(appname='Sample1', appuser='bogus');  
%armgtid(txnname='TRN 1', txndet='Loved one within a data step');  
%armstrt;  
<transaction code start point>  
<transaction intermediate point>  
%armupdt(data='Loved one within a data step, still alive and kicking');  
<transaction code end point>  
%armstop(status=0);  
%armend;  
run;  
```

The `%arminit` macro generates the _armapid SAS variable as stated above with the value Sample1, which represents the application ID. The `%armgtid` macro will then use the _armapid SAS variable as input to the arm_getid function and then set the transaction classes associated with the application ID Sample1. In this simplistic case we have just one “loved one” transaction, and the macro will generate the _armtxid SAS variable (the transaction class ID). Next the arm_start function is invoked through the `%armstrt` macro just before the “loved one” starts -- this is necessary to set the transaction's start handle, which is its actual starting point. The macro will utilize the _armtxid SAS variable as the transaction class ID, and the function will also return the _armshdl SAS variable as the transaction start handle.

During transaction code execution, at some intermediate point or milestone an optional function Arm_update is invoked through the `%armupdt` macro, which updates the transaction using the transaction start handle _armshdl. At the end of the “loved one's” execution the arm_stop function is invoked through the `%armstop` macro, signaling the end of the transaction. This updates information about the transaction based on the transaction handle which is in this case the _armshdl SAS variable. The optional transaction status value is set to 0, signifying success in this case.

The ARM environment is then terminated using the arm_end function via the `%armend` macro using the _armapid SAS variable, which signals the end of the application.

The one thing to note in this scenario is that no global macro variables were used to communicate, only SAS data.
step variables.

MACROS OUTSIDE THE DATA STEP

```sas
data _null_;  
call symput('_armexec','1');  
run;  
data _null_;  
%arminit(appname='Sample1', appuser='bogus');  
run;  
data _null_;  
%armgtid(txnname='TRN 1', txndet='Loved one within a data step');  
run;  
data _null_;  
%armstrt;  
<transaction code start point>
run;  
data _null_;  
%armupdt(data='Loved one within a data step, still alive and kicking');  
run;  
data _null_;  
<transaction code end point>
%armstop(status=0);  
run;  
data _null_;  
%armend;  
run;
```

In this scenario the same results will be obtained as before, with the difference being that global Macro variables are be used for communication instead of SAS data step variables.

REPORTING ON THE DATA

Now that the data has been gathered the final portion of the picture can be completed, which is reporting. Two SAS macros are available:

- `%armproc`  
  This macro collects information from the external source such as the log file and converts the data into six datasets.

- `%armjoin`  
  This macro adds the final touch. It takes the six data sets and creates data sets that contain common information about applications and transactions.

CONCLUSION

Now that we have the basics, we can see the power of ARM from macro initialization to the end reporting product. “Loved ones”, those special transactions within applications, can be treated in a more special way. Their performance can now be measured more accurately, thus influencing the next steps for the business.

We have discussed the benefits of ARM, but as always there is never a free lunch. Depending on your perspective, ARM may possess a drawback: there is a lot of effort needed to ARM applications. What do I mean? Well, users have to be keen on deciding what metrics they are looking for and then code it using the ARM API macros. If the application they have is prepackaged by some vendor, they might not have access to the code to add the ARM macros.

Nevertheless, in most cases the benefits outweigh the drawbacks. So, now that you’re ARMed and dangerous, go out there and ARM SAS to answer your questions.
REFERENCES

http://support.sas.com/rnd/scalability/tools/arm/arm.html

http://regions.cmg.org/regions/cmgarmw/marcarm.html

ACKNOWLEDGMENTS
I would like to thank the staff at SAS Support who always helped by answering any questions I may have had. Thanks to Shiva Srinivasan and Robert Jones for their support and encouragement with writing this paper, for without them this paper would not have been completed.

CONTACT INFORMATION
Contact the author at:
  Fabian Robinson
  PJM Interconnection LLC
  955 Jefferson Ave.
  Norristown, PA, 19403
  610-666-3181 (work)
  E-mail: robinf@pjm.com, frobinsn2001@yahoo.com

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 trademarks of their respective companies.