An Overview of MVA — A Philosophy of Hosting

John A. Toebes, VIII, SAS Institute Inc.

ABSTRACT

Version 6 of the SAS® System has been successfully hosted on a wide variety of architectures. In order to accomplish this feat, SAS Institute has established a general philosophy of development along with some specific guidelines to ease certain problems that have occurred during the development effort.

This paper describes the basic approach taken in this effort. Specific topics to be addressed include limitations of C compilers and the C language, exploiting machine features, multi-machine product configuration, and a general overview of the system layering. Special emphasis is given to how the SAS System avoids a 'least common denominator' approach to hosting and instead is able to exploit the features of each individual host.

HISTORY

Until the advent of the SAS System for Minicomputers, the SAS System was traditionally written in a combination of 370 Assembler and IBM PL/I. The supervisor being the most heavily used component was coded in Assembler for speed and size while the procs were coded in PL/I to improve maintainability.

With the introduction of the SAS System for Minicomputers and support of non-IBM compatible machines, it became clear that it was no longer possible to code the supervisor in 370 Assembler. A compatible supervisor was written in PL/I-Subset G for the mini-computers while maintaining an Assembler version for the IBM systems. This choice of having two separate supervisors led to several consistency problems which became apparent to the end user. Recoding the supervisor was a first step, but the process of hosting the SAS System on another machine was still significant, utilizing time that would be better spent expanding the features of the SAS system for our current users. This resulted in delays between the same version of the product shipping on various machines. One significant reason for the delays was the necessity for each host to know in-depth what pieces to put together in order to construct the final system.

Another major problem was the lack of a standard version of PL/I for all the machines. Even though there is an ANSI specified PL/I Subset G that we attempted to follow, many of the compilers lacked certain features or implemented a feature differently. In addition, the PL/I language is inherently machine dependent in its specification of data types based on numbers of bits. Lastly, many new machines were being introduced that did not have a PL/I compiler of any form, effectively limiting the scope of machines on which we could host the SAS System.

Even if the problem of locating a compiler for the target machine was solved, another problem started to become apparent. With each new machine came a new operating system with its own limits and peculiarities. Because we evolved the code for each new machine instead of an initial design encompassing a wide range of architectures, these limitations often crept into the resultant user visible feature set. One example of this is the naming of SAS data sets which closely mimicked the MVS JCL data set naming conventions. In order to take advantage of native naming conventions on the target hosts, the libname and filename statements were introduced.

VERSION 6 DESIGN GOALS

With the experiences of the previous versions of the SAS System behind us, we set out to solve these problems with a new architecture - Version 6. Initially targeted for the IBM/PC, this system was
completely rewritten. In doing so, it was our intention to make the SAS System available on as many architectures as possible, while improving the productivity of the programming staff both in terms of producing new features and in rehosting the system on new machines. This is the fundamental concept of MVA, our Multi-Vendor Architecture.

With Version 6, we also sought to maximize our ability to exploit the advantages of each particular host environment instead of taking a least common denominator approach. In so doing we hoped to improve our ability to adapt to and even take advantage of new technologies as they became available - particularly in the area of user interface.

Above all, our goal was to improve productivity by eliminating the need for applications developers to become involved with the issues associated with a particular host. This would provide faster turnaround for enhancements and reduce maintenance time by eliminating host dependencies.

VERSION 6 APPROACH

In order to accomplish these goals, we first needed to choose a programming language. At this point in time, PL/I was on the way out and Pascal, Modula-2 and C were vying for public attention. Pascal was not a good candidate, because in standard ANSI it is not usable for a large system and the extensions to Pascal on any given compiler were as varied as the machines themselves. While Modula-2 overcame some of the Pascal deficiencies, it was not available on the range of machines to which we needed to move the SAS System. Only C was flexible enough for so large a project and at the same time was available on almost every machine.

Choosing a language was only a small part of the Version 6 approach. We also made many decisions that affect the fundamental design of the SAS System and how we organize the work. One of the most important and far-reaching decisions was the decomposition of the System into three basic layers:

- The Applications
- The Core Supervisor
- The Host Supervisor

In order to deal with the ever growing volume of source code that goes into the SAS System, we instituted a source code management system and spent time investigating portable methods of tying all the individual pieces together.

THE HOST SUPERVISOR

By careful layering, we intended to isolate the machine dependencies at the Host layer while providing a robust set of services at the Core layer. To enforce this layering, each layer was only allowed to call the layer directly beneath it and no layer was allowed to call a layer above it. In this manner we could develop a completely independent Host supervisor suitable for putting any system on top of it.

This Host supervisor was designed to provide a virtual operating system interface to the Core layer with a specification for the services that make a well rounded function set. These services, comprising approximately 10% of the total system code include the following fundamental operations:

Task Management - The fundamental organizational aspect of the host supervisor that ties together all resource allocations. Each task (or thread as it is often referred to) is a separate entity with automatic cleanup of all allocated resources when the task terminates.

Memory Management - Pool oriented memory allocation that allows for a lifetime model of memory management with differentiation of both short-term (task life-time) and long-term (session wide) allocations.

Program Management - System services to provide dynamic loading/unloading along with dynamic links that allow for a loosely coupled system.

Data Base Management - Machine dependent services that allow for efficient storage of SAS Data sets.

File Management - Basic I/O services to allow portable access to system specific files such as printers, the console, and native operating system files.

Machine Code Generation - In order to provide for efficient execution of data step programs, each host provides a machine dependent code generator.

Full screen Support - A series of services that
permit operation of the SAS system Display Manager in a manner that best fits the host.

Error reporting services - diagnostic services that allow for host specific error messages to be integrated with SAS System error messages.

Each of these groups, known as a host subsystem, is independently defined to reduce interactions and to improve the modularity of the system as a whole. Because of this, we have the ability to independently test and tune the subsystems.

In order to ensure consistency of these subsystems across machines, we documented them in a Host Guide. Currently this documentation covers over 750 routines in 13 major subsystems and spans two volumes. These routines are the only method in which the core supervisor may communicate with the host.

THE CORE SUPERVISOR

The core supervisor is the lowest layer that is completely portable across all host systems. Comprising approximately 20% of the total system code, it defines the operating environment for the SAS System. This layer contains routines to perform many of the fundamental SAS System operations such as macro processing, command parsing, Data Set manipulation, procedure invocation, and the Display Manager.

In addition, many of the services available through the host layer are piped in the Core interface so that the Applications Layer may utilize them. While some of these routines are simply direct calls to the host provided routine, many of the Core routines provide additional services such as additional resource management or simplified calling interfaces. This layering provides us the ability to isolate improvements to the Host Supervisor interface without having to go back and change any portable code.

THE APPLICATIONS

The remaining 70% of the code is made up of the portable procedures and support routines for these procedures. In keeping with the layered design, this code may only call services that are present in the Core Supervisor. While many of the routines in this layer are completely independent, there are also a number of shared services in the form of dynamically linkable appendages such as graphic and full screen support services.

It is this layer with which most users interact as it contains both Institute supplied and user-written procedures.

CODING CONVENTION

While this layering solves many of the host interface problems, it does nothing to isolate the developers from the individual limitations and quirks of the various C compilers used. Many new compilers do not fully implement the ANSI standard language. Given this situation, we chose to develop our own internal house standards for coding in C. This document is constantly updated as new limitations arise and old ones are lifted.

Many decisions have been made in determining our in-house standard, particularly with an eye to potential future hosts. Although we have strived to avoid a least common denominator, there are a few C language constructs that we have prohibited:

Bitfields - In our experience, this is one feature that is likely to be broken or at least problematic in any given compiler.

Static initialization of pointers - Given position independent code on some machines, this is a potential compiler or loader limitation.

enums - Although a newer ANSI feature, many compilers still have not implemented it.

There are many other features that have been problematic on individual compilers. Instead of simply disallowing their usage, we chose to hide the usage through a macro that the host defines appropriately. A good example of this is function prototypes such as:

    void myfun(int, int);

While this feature is becoming more popular, some compilers have not implemented it requiring you to declare the same function as:

    void myfun(int arg1, int arg2);
void myfun();

To allow for usage on those compilers that have it we use a macro to hide the parameters:

void myfun U_PARMS((int, int));

For those machines which do not provide function prototypes, we use:

#define U_PARMS(a) ()

For those machines that do support prototypes, we can use

#define U_PARMS(a) a

This macro approach of masking host dependencies permits us to write code that is free of host specific conditional compilation. In addition, we are able to extend the macro usage to efficiency concerns by defining host macros for common operations that the host can choose to implement more efficiently. For example, all memory movement is done through a series of M macros:

/* copy 8 bytes */
M8(targ, src);

/* copy up to 128 bytes */
MQ(targ, src, len);

/* copy up to 2^15 bytes */
MI(targ, src, len);

One specific limitation that has been strongly enforced is the prohibition of machine dependent conditional compilation. When such a requirement is encountered, such as the case for character set collation dependencies, the conditional compilation is controlled by a predefined symbol that is set by all hosts. In this way we are assured that the dependencies are oriented to a particular feature and not a given machine.

Given the set of standards, it was not enough to simply assume that everyone will fully understand and follow them. As with anything, interpretations vary. To combat the potential problem of code arriving at a host and not compiling because of a standards limitation, we have modified an in-house version of the Lattice® C compiler so that it issues diagnostics for anything that varies from the standard. This program, called LCPARSE, checks every piece of C source code before it can be placed in the master source library.

DEVELOPMENT ENVIRONMENT

The LCPARSE program is only one small piece of our development environment. At the Institute, all portable development is done on a ring of over 300 networked Apollo workstations. In order to coordinate this development effort, we have an inhouse staff of programmers who maintain and develop our own source code management system known to the developers as SDS.

With SDS, we have defined a series of LEVELS which correspond to the various stages of the SAS System. At any point in time, there are at least four levels in existence:

"old" - The previous released version for reference or patches.

RELEASE - The current stable production version of the product for porting to the other host systems.

MASTER - The current stable version of the product in preparation to move to release.

INT - An integration place to merge changes for testing.

Within these levels, the source code is divided up into development directories (DVD) corresponding to the various pieces of the SAS System. Within each DVD are separate places for each type of source including C source files, include files, grammar files and message files.

This organization is mimicked on all host machines allowing for a uniform view of the source across all hosts. In addition, this organization allows for a portable method of referring to a file. Taking a cue from the SAS System Macro language, we can specify any specific file in a level with:

`%vd(file.e)`

Where `vd` is the development directory, `file` is the name of the file and the `.e` indicates the type of file.
We use .c for a C source file, .h for an include file and .e to indicate an executable image. For example, given the name:

`.sup(xkini.c)`

For various hosts we would end up with:

**Apoll**:

```
/level/sup/src/xkini.c
```

**MVS**:

```
level.sup.src(xkini)
```

**VAX**:

```
[level.sup.src]xkini.c
```

**PC**:

```
\level\sup\src\xkini.c
```

This portable naming convention is utilized in our internally developed configuration management utility SDSBUILD which allows a developer to portably describe how to construct a procedure from the given source code.

Each build script for a procedure consists of statements that indicate the product that it belongs to and all the products in the system that it references. This information is followed by statements that indicate the pieces to be built and from which subpieces each is constructed. Through out these build scripts, this portable naming convention is used to identify the individual source files.

These scripts do not contain any actual host specific commands for compilation or linking; instead, they describe the pieces in terms of what they are using generic macros such as `procedure`, `function` and `src`. It is the responsibility of each host to define these macros to produce the appropriate commands to build the components.

In addition to the scripts to build the individual pieces, there are higher level scripts that describe successively larger collections of the pieces so that at the highest level of the system there is a single script `SAS.BLD` which when run will cause the entire SAS System to be constructed for a given host. In practice this script is never run, but the secondary scripts that it references are run individually. This is done to allow the process of building the system to be broken up into smaller units of work parcelled out to separate processes or machines. This is important because even on our IBM 3084 and 3081 machines it takes 36 CPU hours to rebuild the entire system from scratch.

**PORTING AND TESTING**

The portable build scripts play an important part in the process of moving the SAS System to each target machine. Before the system is ported to any other host, it is first built using the build scripts at the release level. This version is then subjected to a series of tests designed to verify the basic functionality of the system. Any piece that fails these low level tests must be fixed before the source is allowed to be sent to any other host.

In addition, a series of high level tests are also run to verify the statistical quality of the system by benchmarking the output from that of a previously verified run. Failure to pass these tests does not prevent us from sending the source but is simply noted so that the receiving hosts can get an idea of the state of the system.

Since these tests are stored as a part of the DVD along with the source code, the individual hosts can rerun them on the target machine after rebuilding the system to verify that the portable code came over intact. In addition, it provides for a semi-automated mechanism of verifying that improvements to the host supervisor have not adversely affected functionality.

Once the source management group is satisfied with the stability of the system, a cut is made of all the source code and sent to all hosts simultaneously. Although this occurs approximately every other week, an individual host may choose to skip an update and continue development with the existing release or even work on a production release.

When a host chooses to take a new release, the tape is loaded up into a stage area for building. Once the code has left the development machine, it is no longer in the domain of the developer. Any problems encountered such as compiler messages are handled by the individual hosts and reported back to the supporting developer so that the change can be implemented to the main repository of source code. This however is becoming less frequent as LCPARSE is made more intelligent. The host people are also responsible for tuning and performance testing. Any areas of improvement are also reported back to the developers. At this time, there are many hosts looking at the system as a whole. Because of this, there is much opportunity to locate improvements, and to globally pass the benefits on to each
host as the improvements are made.

CONCLUSION

The SAS System has evolved significantly over the years. In order to meet the demand of more features on more systems at a faster pace, we have evolved our approach to software development. In the process we have become more able to respond to new technologies by layering the system in a manner that isolates the portable aspects from the host specific aspects.

This improvement in technology has proven itself in the UNIX® marketplace, with the introduction of the SAS System for our third UNIX machine requiring only four months of development.

For our current customers this means that we are able to concentrate the majority of our efforts on improving the system and adding new features faster. This is evident as Version 6 brings such improvements as the new SCL and Display Manager to our existing environments.

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA. Lattice is a registered trademark of Lattice, Inc., Glen Ellyn, IL, USA. UNIX is a registered trademark of AT&T.