DYNAMIC LOADING IN THE SAS/C™ COMPILER, THEORY AND PRACTICE

Stuart N. Newton, a.l.p Systems

ABSTRACT
This paper explores some of the practical uses of the loadm and loadd functions of the SAS/C compiler under CMS. Both benefits and drawbacks are discussed, along with "gotcha's" which accompany some uses of these functions.

INTRODUCTION
Dynamic loading of both executable code and data as provided by the SAS/C runtime library can be used in a number of valuable ways to augment the capabilities of CMS, and to make up for some of the UNIX functionality which is missing in the CMS environment.

Three general reasons for using dynamic loading will be discussed in this paper. They are:
- modularity of code and data
- emulation of UNIX fork
- providing user exits

DYNAMIC LOADING, IN THEORY
Generally C programs will be linked together as a single load module. However, some cases exist in which this may not be desirable. In these cases it is more desirable to have a base module which loads and runs other load modules as needed.

Although with dynamic loading it is possible to load entire load modules and run them, its use is not confined to load modules. As is the case with the SAS/C runtime library, it is sometimes desirable to provide a library of runtime routines. These routines are then loaded as needed, and only when needed, by the calling program.

In addition to allowing load modules or runtime functions to be loaded dynamically, the concept behind dynamic loading also allows data such as tables to be loaded as needed.

In theory then, dynamic loading is the concept that the entire program need not be linked together as one entity. Rather, parts of the program may be loaded or unloaded as needed. These parts of the program may be either executable code or some other form of data.

USING DYNAMIC LOADING
Dynamic loading and unloading is accomplished using seven functions. These are: addsrch, delsrch, buildm, loadm, loadd, unloadm, and unloadd.

The first two, addsrch and delsrch are used to create a search order for the LOADLIB's or DCSS's that you desire to load a module from.

buildm can be used to create a function pointer from a character pointer which points to a load module entry point.

loadm is used to load executable code into storage. The function returns a function pointer which may then be used to invoke the module just loaded. With loadm, you can load functions or entire programs into storage.

loadd is used to load data (non-executable code) into storage. The function returns a character pointer (char *) to the first byte of the data.

unloadm and unloadd are used to unload modules or data when they are no longer needed. Unloading something after it is no longer needed helps keep execution size of the module within reasonable limits.

Either of the two load functions may load their modules or data from a LOADLIB, from a Discontiguous Saved Segment (DCSS), or from a NUCLEUS EXTENSION.

If the loading is done from a LOADLIB, a complete copy of the module or data is loaded into free storage. Each virtual machine wherein some program has loaded the module or data will have its own copy of this data or module.

If the loading is done using the NUCLEUS EXTENSION option, it is expected that the module being called has already been loaded from CMS, using the NUCXLOAD command. In this case, as with the LOADLIB option, a private copy of the module will exist in every virtual machine that uses it.
If the loading is done from a DCSS, a much more efficient process takes place. Multiple copies are not brought into storage. Rather, the DCSS is attached to the virtual machine, and a pointer is returned to the one copy of the data or module in real storage. All users share the same copy, thus reducing system overhead in areas such as paging, storage contention, and real storage usage.

DATA AND CODE MODULARITY

As has already been mentioned, it is not always wise to create a single large load module. There exist reasons which are both compelling and exciting which will lead the prudent programmer to consider dynamic loading of modular code or data in the design or implementation of any program product.

Three cases which might compel one to investigate dynamic loading further are as follows:

- If the software product includes peripheral utilities or functions that are called infrequently, it may not be desirable to load these every time the product is invoked. Greater efficiency can be achieved by loading only that portion of the product that is needed at any given time.

- If the software product is very large, requiring unnecessarily large amounts of storage to load and run it, breaking the product into smaller modules can reduce overhead and improve performance.

- If the software product is extremely large, it may not be possible to link it together as a single load module. In this case the programmer is forced to break his program into smaller pieces.

Many advantages can be realized by breaking a program into smaller pieces and loading each one as needed. These advantages range from improved performance to improved maintainability. Assuming that the program has been modularized, the following benefits could come as a direct result of the dynamic loading process:

- Since not all of the program needs to be loaded at once this can decrease the time necessary to invoke the program.

- Since the program is now essentially smaller, it requires less virtual and real storage to run. This will increase performance for both the

user running the program and for the entire system.

- Changes made in one part of the product need not affect the rest of the product, thus increasing maintainability of the product.

- If the loading is done from a DCSS, everyone using the program can use the same copy in real storage, further reducing storage requirements and improving overall performance.

- If many programs use a given routine or table, loading the routine or table dynamically will eliminate the many copies which would otherwise be created as part of the link process. Having a single copy of a routine or table makes it much easier to change or maintain.

This list of uses for dynamic loading is by no means comprehensive. It represents only an introduction to the concept of data modularity as it relates to dynamic loading. Since each application is unique, a careful study of the benefits and possible drawbacks should be taken before implementing dynamic loading.

CREATING UNIX-LIKE PROGRAMS

One of the most common reasons for selecting C as a programming language for many companies is its portability. This portability to various kinds of hardware appeals to marketing departments and to management alike. However, not all C's are created equal.

The programmer who is involved in converting software to run on a new piece of hardware soon finds himself trying to make up for missing features or for new and different restraints and limitations. Invariably, large amounts of creativity are required to complete such a project. This is especially true when converting UNIX software to run under CMS.

Some of the most common UNIX functions that can be emulated with dynamic loading will be discussed in the next section.

The UNIX fork

One of the features of UNIX which is commonly used is fork or vfork. The use of this function allows one program to invoke another program, transferring control to the "child" process, and having control return to the "parent" process once the child has terminated. In some applications both the child and the parent process can be allowed to run simultaneously.
Obviously, since CMS is not a multi-tasking operating system, one cannot expect to allow both the child and parent to run simultaneously in the same virtual machine. However, to provide an equivalent of the UNIX fork, in the case where the parent process waits until the child process has terminated before continuing, is a perfect use of the SAS/C loadm function.

When loadm is called, the designated function is loaded into storage if necessary, and a function pointer is returned which can be used to call the now loaded function or program. Control can then be passed to this new function or program which runs within its own address space, independent of the parent process. When it has finished, the child returns control to the parent process.

With very few modifications then, programs which have been in the habit of calling fork and one of the exec functions (execvp, exec1, etc.) can be retrained to call loadm and then use the function pointer returned to invoke the child process.

A sample of just such a retraining is shown in Figure 1.

The Child Process

Once the parent process has been modified, a careful look must be taken at the child process. We will examine the most important concerns that should be in mind when converting a UNIX program to run as a dynamically loaded module under CMS.

exit

Under normal circumstances when a C program desires to terminate, a call is made to the exit function. Under UNIX this returns control to the parent process. The same thing must happen in CMS. When exit is called under UNIX, it returns control to whoever invoked this one.

Under CMS exit does the same thing. It returns control to whoever invoked this program. However, since the main routine and all others that have been loaded dynamically are considered to be one program, calling exit will return us to CMS.

So if the child is left unmodified, it will finish its work and call exit, but rather than returning control to the parent it will commit patricide and terminated its parent as well.

Obviously we must prevent our child from calling exit. One of the ways to do this is to have them call a generic exit (ourexit, noexit, etc.), which behaves differently on different operating systems. In the CMS environment this new exit routine can take care of any housecleaning that might be necessary and then longjmp to some prearranged place in the "dynam" portion of our program. This "dynamic main" function can then issue a return call and return control to the parent process.

malloc

It is extremely common for UNIX programs to use the malloc family of routines to obtain storage (memory) when needed. These functions also exist in SAS/C and therefore the child process is able to allocate storage whenever it needs. But what happens when it encounters an error and aborts. The applications programmer who wrote this for a UNIX operating system assumed that UNIX would clean up the memory that was allocated by malloc or calloc, and he didn’t worry about it. This problem not only happens when an error occurs, but very often memory which is allocated is never free-ed. After all, under UNIX, the system will take care of it.

We can assume that CMS will take care of it when we exit our program, and it will. But remember that we aren’t going to exit our program right away. We are going to return control to the C function which called us. So CMS doesn’t know about us yet. Now what will happen if we use loadm to invoke another program, or if we invoke the same program again without ever freeing the storage that was allocated? Somewhere down the line somebody is going to get a nasty "DMSFRE159T INSUFFICIENT STORAGE AVAILABLE..." message.

So we need to prevent the calling of another standard routine. One of the easiest ways is to do like we did with exit; invent our own routine to do the malloc’s. If all our children call our malloc routine, then we can save pointers to the storage that they allocate. Then since they will be going through our exit routine that we implemented above, we can free up all the storage that they didn’t free.

If we are trapping malloc’s and calloc’s and saving pointers, then we must also trap free’s and cfree’s so that we can release those pointers. After all, if the programmer went to such unusual courtesy as to free the storage that he allocated, then we ought to at least oblige him.
open, fopen, freopen

Another one of those things that UNIX does for you is close your files when you exit. So in the case of an error that causes our child to abort, or if the programmer was sloppy, he won't have bothered to close the files that he opened either.

Now we find that we must keep track of the files that were opened, and in our exit routine, close these files for them. And of course with regard to these files we must remember which have been fopen'ed and which have been open'ed, etc., and dispose of them properly.

Finished At Last

Once we have overcome these few problems, we have in place a fairly effective emulation of the UNIX fork/exec process. And as long as we keep in mind that the new program has not replaced the other in storage, but rather has been loaded into free storage for the moment, we can do virtually all the things that the fork/exec allow us to do in UNIX.

USER EXITS

loadm provides an easy way to provide user exits in programs. The user's programs may be loaded from a LOADLIB or from a DCSS just as your own code may. Standard linkage conventions are followed when calling modules loaded via the loadm function, which aids greatly in providing a generic interface into the outside world. The only requirement is that the user's program be relocatable, and if you desire to put it in a DCSS, reentrant.

Since the name of the DCSS or LOADLIB need not be hardcoded within the program, there are no stringent restrictions as to the names of functions or the places that the functions reside. This feature also aids in providing generic user exits.

Implementation of a User Exit

Implementing a user exit is as easy as calling any other dynamic function. Normally you will pass a pointer of some type to the exit to allow it to read some data buffer. Optionally you may want to allow it to modify some data.

In the example shown in Figure 2, a user exit by the name of "EXIT1" will be invoked. It will be passed a pointer to a data buffer, the length of the buffer, a pointer to a work area, and the maximum length of the work area.

Debugging a User Exit

Once you have allowed the user to implement his user exit, it might be nice to let him debug it. But if he plans to use a tool such as VM/PER (Program Event Recording) he will have to know where his user exit is loaded. If his program is in a DCSS he may be able to derive its address from GENCSEG LISTING, but if it is loaded from a LOADLIB into free storage, heaven help anyone who wants to guess where it is.

Allowing some sort of verbose or debug mode such that your program includes something akin to Figure 3 will aid greatly in debugging a user exit. And, as long as you have gone to the trouble to let your users know where the routine was loaded, you might as well stop and let them get PER set up before you pass control to their routine. You can do this by adding a short line of code to your program as illustrated in Figure 4.

IN SUMMARY

Dynamic loading as provided by the SAS/C runtime library provides an excellent way to emulate the fork/exec process of UNIX. With very little modification, programs can invoke other programs just as UNIX programs do. Dynamic loading solves a number of issues relating to running UNIX software on CMS.

In addition to UNIX emulation, dynamic loading also provides a very convenient way of allowing user exits within your code. Since the program that is loaded may be written in assembler or some other language this allows for user convenience when implementing user exits.

A third use of dynamic loading is to produce code modularity. Modularity of code may refer to both the code and data accessed or used by that code.

Still a fourth reason for using dynamic loading is to improve overall performance. By dividing load modules into smaller pieces the program product may be invoked more quickly and may impose less drain on system resources.

SAS/C™ is a trademark of SAS Institute Inc., Cary, NC 27511-8000
```c
#ifdef CM3
#include <dynam.h>
#include <ctype.h>
#else
#include <signal.h>
#endif

fork_child (parms, cnt)
char *parms[];
int cnt;
{
#ifdef CM3
    int (*funcp) (), re;
    loadm (*parms, &funcp);
    if (funcp)
    {
        re = (*funcp) (ent, panns);
        unloadm (funcp);
        return (re);
    }
    else
        return -1;
#else
    int status, err, pid;
    extern int errno;
    pid = fork();
    if (pid == 0)
    {  
        err = execv (*parms, parms);
        if (err)
            _exit (errno);
    }
    else
    {
        signal(SIGINT, SIG_IGN);
        err = wait (&status);
        signal(SIGINT, SIG_DFL);
        if (status)
        {
            errno = status >> 8;
            return (-1);
        }
        else
            return (0);
    }
#endif
}

Figure 1. Emulating UNIX fork
```
call_user1 ()
{
    int (*funcp)(), rc;
    extern char *inbuf;
    char outbuf[SOMESIZE];

    loadm ("EXIT1", &funcp);
    if (funcp)
    {
        rc = (*funcp) (inbuf, strlen(inbuf), outbuf, sizeof(outbuf));
        unloadm (funcp);
        return (rc);
    }
    else return -1;
}

Figure 2. Calling a user exit

loadm ("EXIT1", &funcp);
if (debug)
    printf ("Routine 'EXIT1' loaded at X'%x'\n", *((int *)funcp));

Figure 3. User exit debug setup

loadm ("EXIT1", &funcp);
if (debug)
{
    printf ("Routine 'EXIT1' loaded at X'%x'. Press ENTER to continue.\n", *((int *)funcp));
    getchar(); /* wait for response */
}

Figure 4. Interactive debugging of a user exit