Edinburgh Regional Computing Centre

User Note 32

(April 1983)

Title:

Efficient Loading on EMAS 2900

Author: Colin McCallum

Contact: Advisory Service

Software Support Category: A

Synopsis

This note, which is an extract of the complete loader manual, offers guidance on how to improve the efficiency of loading object files.

Keywords

Directory searching, entry names, LOADPARM, object files, PRELOAD

Introduction

The advice given in this note is general in nature since it is not possible to be completely categorical as to what must or must not be done in any given set of circumstances. Some of the advice is directed at you the user and other parts at you the implementer of software. You are invited to consider the various points and recommendations as they apply to your problem and make judgements accordingly. At the outset it should be pointed out that you should only be worrying about efficient loading after you are sure that the software you wish to run has passed the development phase and is ready for production running.

In the following text CODE means the specific area of an object file which contains executable code.

Loading Strategies

If the software you wish to run refers to several external files and you know that not all of them will be called then LOADPARM(MIN), which will only load files actually called, should be considered. In a very small number of cases there may be a problem with dynamic data references, but in general the more external files, the bigger the saving with LOADPARM(MIN). Note that a similar effect can be achieved if it's your own software you are running by compiling with PARM(DYNAMIC).

However, if all the external files do get called in the run then LOADPARM(MIN) becomes less efficient than a full load since each dynamic call will incur separate loader overheads. When in doubt, use a full load as it is always annoying as well as wasteful when a program fails after using a considerable amount of cpu because an item called dynamically could not be found. A full load (the default) would have revealed this immediately.

Directory Searching

The first point to be made is that searching a directory is much faster than loading a file so any savings in cpu time achieved in reordering searchdir lists are liable to be small. However the savings in unnecessary paging from directory to directory can be significant and it is recommended that searchdir lists be kept no longer than is necessary. If one particular directory contains frequently required entry points then it should be moved to the top of the search list.

Organization of Groups of Object Files

Several different ways of organizing a group of object files can be envisaged: leaving them as individual files, LINKing them to form one, larger, object file or holding them as members of a pd file.

In general, holding a group of object files as pd file members is to be avoided unless

a) You want to change individual members frequently or
b) the number of entry points is very large and you know that only a few members will be needed in any one load or
c) you need to gather several files together and they are not all object files.

In most cases apart from those above, the best way to hold a collection of object files is to LINK them into one file. Although this file may be large only those pages of the file actually required will come into main store. Additionally, the techniques for optimising the loading characteristics of object files described below can be applied which can lead to substantial improvements in efficiency.

The main objection to pd files is that each member requires a separate call on the loader to load it. If there is cross referencing between members then overheads increase. An even worse problem can arise if the pd file grows to >256K and the CODE area of one of the members crosses the segment boundary. Before such a file can be run, a temporary file has to be created and the shareable areas copied into it - both expensive operations.

Note that bound object files should NEVER be collected into pd files as this negates the whole point of binding. (See 'Binding Object Files' section for further explanation.) Furthermore it is not possible to LINK object files which have been bound, so binding should always be one of the last operations in any collection sequence.

PRELOADing Object Files

This is a powerful technique to cut down loading overheads on object files which will be required a number of times during a session. After an object file has been PRELOADed then all its entry points are added to the 'permanently' loaded entries table and it remains loaded until the end of the current session or an explicit call of RESETLOADER. No further loading overheads are incurred regardless of how many times the file is called subsequently while the file remains permanently loaded. Another use of PRELOADing is to override entry points which would otherwise be loaded as a result of normal searching. This can be extremely useful, for instance, to test a new version of a routine without disturbing the original object file. An entry which is already loaded will always be used to satisfy references to that name in an object file which is being loaded even if there is an entry point of the same name in the file. Two entries of the same name and type are not allowed in the loader tables at the same time. If this situation should arise then the second occurrence of the name and type are ignored and a warning printed. If you use this facility a lot then you will generate a lot of (unsuppressable) warning messages since the loader has no way of knowing that you are doing it deliberately.

Problems in PRELOADing - Serial re-entrancy and filling the base gla

There are, however, three pitfalls in PRELOADing which lurk for the unwary user. The first is whether or not the object file to be PRELOADed is serially re-entrant i.e. do you get the same result if you run a PRELOADed file twice with the same data. The problem is not as trivial as it may appear since global variables and common areas are only initialised when the file is loaded. If a file is PRELOADed then the second time the object file is run there is no loading to be done and hence no initialisation. All the global variables and variables in common areas will hold the final values from the previous run of the object file. The effect can easily be overlooked since it is likely that the program will run without any obvious error even though the final results may be quite wrong! Of course it is possible to take advantage of the fact that global variables are preserved from run to

run, but first it is necessary to be aware of the effect. The second pitfall is most likely to be encountered by FORTRAN users who use PRELOADing. The problem in this instance is that when a file is permanently loaded it takes the gla it requires from the base gla. This file is 256K in length but only about 200000 bytes of this are actually available for permanently loaded object files to use. It may be recalled that the space for common areas is created on the gla by default (see User Note 31). It is very easy with FORTRAN programs which have even moderate sized common blocks to request more space than is actually available and the failure 'BASE GLA full' will occur.

There are two possible ways round this. It may be that you have a number of permanently loaded files which are no longer required, which, if unloaded, would release enough space for the PRELOAD to succeed. A call of RESETLOADER will unload all currently loaded files. A better solution however is not to allow the common areas to be created on the base gla at all by setting them up with the command DATASPACE in a file specifically created for the purpose.

The third pitfall is again more likely to be encountered by FORTRAN users and relates to the use of DATASPACE with PRELOAD as suggested above. You must ensure that any areas you set up with DATASPACE are large enough to cope with the largest occurrence of the named area in different modules. The biggest problem is usually with blank common (always called F#BLCMN) whose length may vary from routine to routine, module to module. You may, for instance, decide to PRELOAD an object file which you see requires 1000 bytes of blank common, so you use DATASPACE to set up an entry called F#BLCMN of length 1000 bytes and PRELOAD the file. You then run the file only to find it calls another module which actually needs 1500 bytes of blank common. The result will be a catastrophic failure. The moral of the tale is that it is not sufficient only to consider the lengths of data areas in the file you want to PRELOAD, you must also take into account the lengths of identically named areas in other modules which might be called. The DATASPACE areas must be big enough to cover all possibilities.

Optimising the Loading Characteristics of Object Files

This section is a discussion of some of the facilities of the EMAS object file editor, MODIFY, which can be used to reduce the cost of loading object files. Separate documentation is available which gives a complete description of the facilities (User Note 4) and this section should be read in conjunction with it.

The MODIFY operations of interest are: SUPPRESS, SUPPRESS DATA, SATISFY REFS, SATISFY DATA, COMMON ENTRY, FUSE CODE, FUSE GLA and BIND. Note that SATISFY REFS, SATISFY DATA and COMMON ENTRY are all performed as side effects of BIND.

FUSE CODE and FUSE GLA - Action before LINKing object files

An EMAS object file, for the purposes of loading, can be considered as comprising seven sections: two which are generally shareable (CODE and SST), four unshareable areas which are grouped together in the gla (PLT, GLA, UST, INITCMN) and the seventh, also unshareable which is copied to the user stack (INITSTK). In its simplest form these areas are laid out as

```
CODE
SST
PLT
GLA
UST
INITCMN
INITSTK
```

If two object files, 1 and 2, are LINKed then the resultant object file will look like

```
CODE1
CODE2
SST1
SST2
PLT1
PLT2
GLA1
GLA2
UST1
UST2
INITCMN1
INITCMN2
INITSTK1
INITSTK2
```

and similarly if more than two files are LINKed.

It will be observed that as object files are linked together the individual areas of any given object file become more and more separated. For example, in the above case CODE1 and SST1 are now separated by CODE2. Since CODE1 refers to SST1 then if these areas end up in different parts of the store, every reference by one to the other will cause a page fault and efficiency will be seriously reduced. The FUSE CODE operation on an object file causes the SST area to be permanently appended to the CODE area so that subsequent LINKing will not force them apart. Similar considerations apply to the unshareable areas copied into the gla and the FUSE GLA operation is used to achieve a similar result.

BINDing Object Files

BINDing an object file can offer substantial savings in loading overheads and should always be considered if you want to maximize performance or you administer production versions of large packages or both. In this context, an object file can be the end result of LINKing as well as the primary output from a compiler. Bound files are not very useful in development situations.

In the binding operations the object file is processed to produce a module which can be loaded at minimal cost. Fixed sites are assumed for the shareable areas, the unshared gla areas and for the initialised stack then all the relocation requests are processed. In essence, the generality of not assuming anything about where files might be connected is traded off for considerably less processing in loading the file. The fixed sites can be nominated by the user and any number of bound files may be loaded in the course of a single load. It is important to note, however, that a bound file can still be run even if it cannot be connected at its preferred site(s). In these circumstances the site

dependent characteristics are recalculated with respect to the actual sites that had to be used at run time. You should never be put off using bound files because some of them might not be connected at their particular preferred sites.

In addition the BIND operation also performs the SATISFY REFS, SATISFY DATA and COMMON ENTRY (but see below) operations as side effects. The first two will satisfy any code or data references which can be satisfied by entry points in the same file, while COMMON ENTRY will create a data entry for a data reference marked as a COMMON reference and satisfy all references to it. In BIND, however, the entry name is NOT added to the list of data entries. Note that there is a potential problem if several files to be bound and likely to be loaded at the same time contain data common references to the same item. Each file would have its common references fixed up with the address of its own entry. At load time the loader would load all the files but any attempt to run would cause catastrophe since none of the files would have access to the same common area. At the time of writing it is not known whether these circumstances are likely to arise in practice. If this proves to be a problem then the situation will be reviewed and MODIFY changed.

The following points should be noted when using bound files:
- Never collect bound files into pd files as this negates the whole purpose of binding since the shareable areas can never be connected at their preferred sites without the loader making a private copy.
- Bound files cannot subsequently be LINKed or MODIFYed so the binding operation should always be one of the last, if not the last.
- Ensure that a series of bound files which are liable to be loaded at the same time do not try to claim the same preferred sites. (Beware of PRELOADing in this respect. Use ANALYSE or #MONLOAD to check.)
- Beware of using so many bound files that you or an unsuspecting user of your software suffers 'VM full' errors; each bound file loaded will cause one, occasionally two, extra files to be created.
- High segment number preferred sites are more likely to be free than low segment number sites. Bear this in mind when nominating fixed sites.
- Before running software which will involve loading substantial numbers of bound files, DISCONNECT(.ALL) to maximize the number of free sites.

SUPPRESSing Entry Names

The MODIFY operations SUPPRESS and SUPPRESS DATA remove from the list of code entries and data entries respectively nominated entry point names. This can be useful after LINKing or BINDing when it is known that all references to the entry point have been satisfied internally and it is not intended that any other software should have access to it. After the entry point has been SUPPRESSed then the loader will have no knowledge of it at load time so no time will be wasted adding it to the loader tables and there will be fewer entry names to search.

Note that when a MODIFY run has been completed, a new object file has been created which reflects the changes requested. If you were to perform SATISFY type operations - whether explicitly in SATISFY REF or SATISFY DATA, or implicitly as in calls of BIND - and SUPPRESS operations in the same MODIFY run, then the order in which these were done would be immaterial since MODIFY still has access to the original file. However, if in one run you SUPPRESSed a particular entry then in a subsequent run you would not be able to use it to SATISFY any internal references. This situation might arise if, for instance, you were

linking in new modules to an existing object file which contained references to an entry previously SUPPRESSed.

Of course, you might wish to do the opposite and ensure that a given reference is satisfied externally and not internally. In such cases the entry should be SUPPRESSed as soon as possible, taking care not to call an explicit or implicit SATISFY operation in the same run.

This offers another route to the testing of a new version of a routine without recompiling everything, (see PRELOADING Object Files above), although in this case the MODIFY should be done on a copy of the master object file as an entry once SUPPRESSed cannot be un-SUPPRESSed by another call of MODIFY.