UNIVERSITY OF EDINBURGH
Edinburgh Regional Computing Centre

User Note 31

(May 1985)

Title:

EMAS 2900 Program Loader

Author: Colin McCallum

Contact: Advisory service

Software Support Category: See Note 15

Synopsis

This Note, which is an extract of the complete loader manual, describes the EMAS 2900 program loader interface to the user. It is intended for those users who wish to know just a little more of the loading process than is given in the EMAS 2900 User's Guide.

Keywords

ALIASENTRY, CALL, CURRENTREFS, DATASPACE, EXECUTE, Errors, LOADEDENTRIES, LOADEDFILES, LOADPARM, PRELOAD, RESETLOADER, RUN, USEFOR, Warnings.

Note 1. Loader search order.

When the loader is asked to find an entry point, the search is done in the following order:

1. System entries i.e. subsystem and Director system call list
2. Privately loaded entries
3. Current active directory
4. Subsystem base directory
5. Privately nominated directories i.e. SEARCHDIR list

The subsystem base directory contains pointers to heavily used software such as the standard compilers and mathematical libraries, MAIL, VIEW, SETMODE etc. Object files found via the subsystem base directory are always added to the 'permanently loaded' loader table and remain loaded until the end of the current session or a call of RESETLOADER.

Searching is always linear down the search list, never circular.

Note 2. Searching for entries at command level.

If the loader is asked to load a particular command and it could not be found after a full search then the following action is taken: The loader assumes that the command name is really a file name and tries to connect it. If the connect fails or it succeeds but the file is neither an object file nor a character file then the load has failed and the message *Load fails - ENTRY not found* is output. If the file is an object file then it is examined for a main entry point. If found then the object file is run otherwise the load fails.

If the file is a character file then a call is made to OBEYJOB to obey the contents of the file on the assumption that it contains a series of commands.

Note that for an object file with a main entry then

Command: PROGY
Command: RUN PROGY
Command: EXECUTE PROGY

have identical effects at command level.

Note 3. Return code from non-trappable events in user programs.

Just before the loader passes control to a piece of user software, it sets a trap to catch catastrophic failures in that software such as 'unassigned variable', 'address error', etc. If indeed some such contingency does occur in the user program then control passes to the trap. Almost the first thing the loader does is check the return code. If this is still zero, i.e. 'success' - which is quite likely since the failure was presumably unexpected - then the loader itself will set a standard return code of 103050709 before calling the diagnostics package and returning to whatever initiated the unsuccessful call.

In many cases the initiating software will be the subsystem itself i.e. a user command typed at command level, but some other routines/commands available in the subsystem and callable from within

user programs themselves set up loader traps. Examples of these are RUN, CALL, EXECUTE (all described below), EMASFC, FCALL and PCALL. If a user program calls on one of these utilities then control will return to the program even after a catastrophic failure. If the program always checks the return code after calling one of these utilities then at least it can detect that a non-trappable failure has occurred and take appropriate action.

User Interface

%EXTERNALROUTINE LOADPARM(%STRING(255) S)

This routine allows loader run-time options to be set. It is the loader equivalent of PARM which sets compiler options. The default is FULL which requests full cascade loading, i.e. all non-dynamic references must be satisfied for the load to succeed. Failure is reported if any unsatisfied references remain. LOADPARM MIN suppresses cascade loading and the loader will only load the file which contains the entry point being looked for. Any common areas required are created and all unsatisfied references are made dynamic. LOADPARM LET makes unsatisfied references 'unresolved' after a full cascade load so that execution can begin.

If LOADPARMs MIN and LET are both set then unsatisfied references will always be made dynamic. Although it might appear that under these circumstances LET is ignored, it does affect the handling of mismatching lengths for a data entry and its references (see 'Errors and warnings associated with loading', below).

If LOADPARM ZERO is requested then any common areas created by the loader will be zeroed otherwise they are filled with the unassigned pattern.

%EXTERNALROUTINE DATASPACE(%STRING(255) S)

This command allows the user to set up a data area in a file and use it to satisfy data references which occur during loading. The file can be a data file created by NEWSMFILE or a character file. DATASPACE permits the caller to add data entries to the 'permanent entries' loader table. Each entry is associated with a specific area within a file. Two entries are not allowed to overlap and any entry must be wholly contained within the file in which it occurs.

The command takes 5 parameters of which 2 are optional:

ENTRY - name of the new data entry.
FILE - name of the file which contains the area of store to be used by ENTRY.
LENGTH - length of data area in BYTES.
OFFSET(optional) - offset of the start of ENTRY from the start of the file in BYTES. This parameter defaults to 0 i.e. the first available byte in the file is the first byte of the data area.

ACCESS(optional) - Type of access required to the data area.
The permitted values are:

R - Read and Read Shared
W - Unshared Write
WS - Write Shared

The default is W for a file, R for a pd file member. Indeed pd file members can ONLY be used in R access mode. This is to prevent problems arising when object files and DATASPACE areas are members of the same pd file. The loader connects object files in R mode to load them so if an object file which was a pd file member were loaded then the whole pd file would be connected in this mode regardless of any previous use of the pd file or any of its members.

Whatever the access type requested, the appropriate permission must have been given. W and WS permissions are allowed on another user's file. Write shared access can be tricky, e.g. what if two of you are trying to write to the same area at the same time, and should not be used unless you are sure you know what you are doing.

This command will be particularly useful to users of programming languages other than IMP since it allows the powerful EMAS facility of store mapping to be used from any language. The only requirement is that the language implementation allows external data areas. IMP defines %extrinsic variables which have this property while in FORTRAN the equivalent is the common area. If an external data area is used as a common area it should always be a multiple of 8 bytes. Normally the loader will create common areas at load time by assigning space from the user's gla file (see User Note 33 for a definition of gla (general linkage areas)). DATASPACE allows the user to set up the data area independently of the loader in a location of his own choosing. When the loader encounters the data reference it will find that there is a data entry of the correct name and type already loaded and use it to satisfy the reference. By using common areas for input and output all conventional (and expensive) READ and WRITE operations can be avoided. There is the further advantage that more of the user's gla file is available for programs and the 'user gla full' error will occur less often.

Note. This is not a language provided facility but a system supplied one and users should bear this in mind when making use of it. DATASPACE definitions remain in force until the end of the current session or an explicit call of RESETLOADER (q.v.).

Example of the use of DATASPACE.

Consider the following two programs:

```
%BEGIN
%EXTRINSICINTEGERARRAY STAR(1:10)
%INTEGER I
%FOR I=1,1,10 %CYCLE
   STAR(I)=STAR(I)+I
   WRITE(STAR(I),6)
%REPEAT
%ENDOFPROGRAM
```

```
      COMMON /STAR/ISTAR(10)
      DO 100 I=1,10
      ISTAR(I)=ISTAR(I)+I
  100 CONTINUE
      WRITE(6,600)ISTAR
  600 FORMAT(1H ,10I7)
      STOP
      END
```

Both refer to an external data area called STAR 40 bytes long which is to be regarded as 10 integers. In the IMP program the extrinsic array STAR generates the data reference whereas in the FORTRAN program the array ISTAR is contained in the common block STAR.

Both programs increment each array element by the array subscript before outputting the value of each element.

Each program requires access in W mode to a data area 40 bytes long and we can provide this using the command DATASPACE. First the file to hold the area is created by, say,

```
NEWSMFILE(DAREA,80)
```

This command creates a zeroed file 80 bytes long called DAREA and we assign the first 40 bytes of it to a data entry called STAR by the command DATASPACE STAR,DAREA,40

If either program is run then when the loader tries to satisfy the data reference to STAR it will find an entry of the correct name, type and length already loaded.

Running either program would give the result

```
  1 2 3 4    5  6  7  8  9 10
```

STAR remains loaded after the run so a second run would give the result

```
  2 4 6 8 10 12 14 16 18 20
```

and so on.

The final values are always preserved between calls and the definition of STAR will remain in force until log off or a call of RESETLOADER. DAREA can support as many other DATASPACE definitions as we wish provided that the areas defined are wholly within the file, no two areas overlap and there is no conflict in access mode. For example if we have STAR set up as above then the following attempts to set up another DATASPACE area would fail:

```
DATASPACE PLANET,DAREA,40,60      => Not wholly contained in DAREA
DATASPACE ASTEROID,DAREA,40,20    => Overlaps STAR
DATASPACE COMET,DAREA,40,40,R     => DAREA already connected in W mode
                                     for STAR
```

whereas

DATASPACE RIGEL,DAREA,20,40 would set up a new data entry 20 bytes long from byte 41 to 60 in DAREA

while

DATASPACE CASTOR,DAREA,10,60 and DATASPACE POLLUX,DAREA,10,70 would assign the remaining 20 bytes in the file to data entries CASTOR and POLLUX each 10 bytes long.

The important point to remember is that the DATASPACE area length is always given in BYTES.

WARNING. IMP programmers should not use SMADDR on a file which has active DATASPACE definitions. This is because SMADDR can change the access mode to a file without the loader knowing about it. For example a file with DATASPACE entries connected in READ access mode could have this switched to WRITE mode by a call of SMADDR with consequences best left to the imagination! If you must access a file currently in use by DATASPACE then you should seek advice.

%EXTERNALROUTINE ALIASENTRY(%STRING(255) S)

This command allows a user to add an alias to a system or permanently loaded entry point directly to the loader tables for the duration of the session or the next call on RESETLOADER (q.v.). The alias is added to the 'permanent entries' loader table with a copy of the type and descriptor of the original name.

The command takes two parameters:

ENTRY - Name of a currently loaded entry point
ALIAS - Name to be added to the permanently loaded entry table

This method of aliasing differs in several ways from the command ALIAS. ALIAS works by adding an entry of the form ALIAS=ENTRY to the current active directory. For example ALIAS ANALYSE,A would enter A=ANALYSE in the active directory. The alias is permanent and can only be removed by another call on ALIAS. A call on A would cause the loader to search the currently loaded material. An entry called A would not be found so the loader would now search the active directory where it would find A=ANALYSE. The loader would remember that it started off looking for A in case this branch proves fruitless then start to look for ANALYSE. ANALYSE would be found among the currently loaded entries and the loader would return the descriptor to enter the command.

If the alias had been set up using ALIASENTRY then a call on A would have found A among the currently loaded entries and returned the descriptor immediately.

ALIASENTRY is more efficient than alias but ALIASENTRY definitions only remain in force for the current session or until the next call on RESETLOADER.

%EXTERNALROUTINE RUN(%STRING(255) PROG)

This command is as described in the EMAS 2900 User's Guide. Note that the first action is to increment the loadlevel before starting any loading operations. In essence this means that the loader stores away its current state before commencing the load. After the RUN has terminated then everything loaded at the new loadlevel is unloaded and the loadlevel decremented before proceeding. In consequence the loader is left in the state it was in when the load started. By implication, anything loaded by a call on RUN will be unloaded again after the RUN. A routine which calls RUN will not have access to any temporarily loaded code or any temporary data area created by the RUN.

If RUN is called from within a program and the file RUN fails catastrophically then a standard return code of 103050709 will be set unless the file itself had already set a non-zero return code.

At command level RUN and EXECUTE have identical effects.

%EXTERNALROUTINE EXECUTE(%STRING(255) PROG)

This command operates in the same way as RUN except that the loadlevel is unaltered. Any code loaded or data area created by the call of EXECUTE can be used by the routine or program calling EXECUTE. Unlike RUN, EXECUTE does not unload.

If EXECUTE is called from within a program and the file EXECUTEd fails catastrophically then a standard return code of 103050709 will be set unless the file itself had already set a non-zero return code.

At command level EXECUTE and RUN have identical effects.

%EXTERNALROUTINE CALL(%STRING(31) COMMAND, %STRING(255) PARAM)

This routine is as described in the EMAS 2900 User's Guide with the difference that if the load fails then CALL will set a return code and return to the routine which called it, rather than returning directly to command level.

Note that the remarks pertaining to loadlevel and loading/unloading made in the description of RUN also apply to CALL (and its equivalents in other programming languages - FCALL (FORTRAN77), EMASFC (FORTE) and PCALL (PASCAL)).

If the code CALLed fails catastrophically then a standard return code of 103050709 will be set by the loader unless a non-zero return code has already been set.

%SYSTEMINTEGERFN USEFOR(%ROUTINENAME MYNAME, %STRING(31) EXTERNALNAME)

This function can be used to call any external routine or function from within a program where the routine or function need not be specified until run time. The nearest equivalent is CALL, but USEFOR is a much more powerful tool than CALL since it is not restricted to commands with a single %STRING(255) parameter.

Code loaded via the USEFOR mechanism is not unloaded after it has been run, unlike the CALL mechanism. This is particularly useful in situations where constant loading and unloading could cause inefficiency e.g. CALL inside a loop.

To use USEFOR you must first declare a dummy dynamic routine or function which has the same specification as the target routine(s) or function(s). A call to USEFOR at run time will then make a call to the dummy routine or function equivalent to a call on the desired code.

USEFOR takes two parameters:

MYNAME, which is the name of the dummy dynamic routine or function, and EXTERNALNAME, which is the name of the external routine or function which you actually want to call at run-time, and will return a result of zero if successful.

For example here is an extract from a program showing how USEFOR could be used in place of CALL:

```
%BEGIN
%DYNAMICROUTINESPEC ANYTHING(%STRING(255) S)
%SYSTEMINTEGERFNSPEC USEFOR(%ROUTINENAME DUM(%STRING(255) S),
      %STRING(31) NEX)
%ROUTINESPEC QUERY(%STRING(31) PROMPT, %STRINGNAME ANSWER)
      .
      .
      .
%CYCLE
   ! Get next command
   QUERY("Next command: ",NCOM)
   %EXIT IF NCOM=".END"
   FLAG=USEFOR(ANYTHING,NCOM); ! Make call on ANYTHING into call on
   NCOM %RETURN %IF FLAG£0; ! USEFOR failed for some reason.
   ! What params do we want to give NCOM?
   QUERY("Param? ", PARM)
   ! Now call NCOM by calling ANYTHING
   ANYTHING(PARM)
   ! Check return code
   %IF RETURN CODE£0 %THEN %START
            .
            .
   %FINISH
         .
         .
         .
%REPEAT
      .
      .
%ENDOFPROGRAM
```

By contrast here is another program fragment which shows USEFOR being used to allow different integer functions to be selected:

```
%BEGIN
%DYNAMICINTEGERFNSPEC ANYTHING(%INTEGERNAME I,J,K)
%SYSTEMINTEGERFNSPEC USEFOR(%INTEGERFNNAME DUM(%INTEGERNAME I,J,K),
      %STRING(31) EXTERNALNAME)
%ROUTINESPEC QUERY(%STRING(31) PROMPT, %STRINGNAME ANSWER)
            .
            .
            .
!  Solve loop
%CYCLE
   !  Get name of next solve function
   QUERY("Solve function? ", NEXTFN)
   %RETURN %IF NEXTFN=".END"
   FLAG=USEFOR(ANYTHING,NEXTFN); !  Make call on ANYTHING
                                 !  into call on NEXTFN
   %RETURN %IF FLAG£0; !  Abandon if error
   TESTFLAG=0
   I=NEXTPRIME
   J=I*2
   %WHILE TESTFLAG=0 %CYCLE
      TESTFLAG=ANYTHING(I,J,K)
      %EXIT %IF K<0
      I=K//2
      J=K*2
   %REPEAT
   %IF TESTFLAG£0 %THEN PRINTSTRING(NEXTFN." is diverging")
   PRINTSTRING("Final values are: ")
   WRITE(I,8)
   WRITE(J,8)
   WRITE(K,8)
   NEWLINE
%REPEAT
            .
            .
            .
%ENDOFPROGRAM
```

USEFOR will fail if

a) MYNAME is not declared as dynamic
b) MYNAME had been satisfied before USEFOR was first called e.g.
   %DYNAMICROUTINESPEC FILES(%STRING(255) S) would be satisfied by the subsystem command FILES at load time.
c) the load of EXTERNALNAME failed.

Note that in the spec of USEFOR given above, which is how it appears in the code of the loader, the %ROUTINENAME MYNAME parameter is simply a pointer to a code item external to the loader. The loader knows nothing about the spec of the external object. It could be a routine, a function, have many or few parameters; it doesn't matter because USEFOR does not call it.

In the program or routine which calls USEFOR, however, the MYNAME spec must be identical to the spec of the dummy routine or function as the above examples illustrate. Once this has been done then any routine or function with the same spec as the dummy can be called.

The one restriction on USEFOR is that it can only be used with one dummy routine or function per program.

Technically, what USEFOR does is to work its way back from its own stack to the location in the gla which contains the escape descriptor corresponding to the dummy routine. An attempt is then made to load EXTERNALNAME and if this is successful the escape descriptor in the gla is overwritten by the descriptor to EXTERNALNAME. A call on the dummy routine in the user program is then equivalent to a call on EXTERNALNAME. If we want to call EXTERNALNAME many times then we only have to load it once. If desired then we can give the program different EXTERNALNAMEs in the same run.

IMPORTANT

Care must be taken when using USEFOR over the problem of serial re-entrancy. This is discussed at greater length in the 'PRELOADing Object Files' section of User Note 32 but in essence the problem is that global variables and common areas are only initialised by the loader when a file is loaded. USEFOR can call a routine or function many times but it is only loaded once. The nth call of the routine has as its starting values for global variables the n-1th final values.

Not all routines have global variables or common blocks and in these there is no ambiguity but even in files which are not serially re-entrant the fact that global variables and common block values are preserved between runs can be turned to advantage but initially it is vital to be aware of the potential problems.

%EXTERNALROUTINE PRELOAD(%STRING(255) FILE)

This command causes object file FILE to be 'permanently loaded' i.e. until the end of the current session or an explicit call on RESETLOADER (q.v.). Any references which remain unsatisfied after the file is loaded are made dynamic. Note that this includes ALL data references except common areas. Common areas are set up by claiming space from the base gla. Preloading is generally used to 'permanently load' files which are going to be frequently used during a session. Loading overheads are only incurred once.

IMPORTANT

PRELOAD should not be used until the 'PRELOADing Object Files' section of User Note 32 has been read. In particular the implications of loading an object file once but running it several times must be understood if the command is to be used safely.

%EXTERNALROUTINE RESETLOADER(%STRING(255) S)

This command will unload any user files which are currently loaded. Any DATASPACE or ALIASENTRY definitions will also be lost.
This command can ONLY be issued at command level, attempts to call it from a program will fail.

Current load status

LOADEDENTRIES, LOADEDFILES, CURRENTREFS

%EXTERNALROUTINE LOADEDENTRIES(%STRING(255) S)

Prints a list of entries which have been loaded by the caller i.e. no subsystem or system call table entries.

%EXTERNALROUTINE LOADEDFILES(%STRING(255) S)

Prints a list of currently loaded files.

%EXTERNALROUTINE CURRENTREFS(%STRING(255) S)

Prints a list of currently active references. An active reference is one which will trigger off a loader search if encountered during a load (unsatisfied reference) or program execution (dynamic reference).

Loader Monitoring

£MONLOAD is a command used to control the amount of loader diagnostic information generated during loading operations. The command takes two parameters - monitor level and output. The first is mandatory and is a bit significant integer. Currently the lowest 5 bits are meaningful:

```
2**0 - requests minimal loading information and some important but
       non-critical warnings.
2**1 - requests information on object files which are being loaded and
       unloaded. Also information on the location and layout of areas
       in loaded files and the module source language.
2**2 - requests information on names and locations of code and data
       entry
       points as they are loaded. Also information on common areas
       set up
       by the loader.
2**3 - requests information from the loader search module on which
       entry
       points are being sought, which directories are being searched,
       how aliases are handled, etc.
2**4 - requests information on which unsatisfied references are being
       made
       dynamic when LOADPARM MIN is set.
```

If the second, output, parameter is not specified then diagnostic information will appear on the output terminal (or job journal if it's a batch job) otherwise the parameter should specify an own filename which will be created if it does not exist or overwritten if it does. In this second case loader monitoring will go directly into the file. Should the file be filled and incapable of further extension then the monitoring will switch automatically to the terminal.

In using £MONLOAD it is generally better to use integer parameters such as 1,3,7,15,31 which have successively more bits set, rather than values such as 2,4,8,16 in which only one bit is set. This is because some information given by higher bits amplifies or expands that given at lower bit setting and the information is no longer seen in context. Note that £MONLOAD -1 will generate all possible monitoring.

Loader diagnostic monitoring settings will remain in force until another call on £MONLOAD: £MONLOAD 0 or £N will turn off monitoring.

Failure messages and some critical warnings are always generated regardless of the £MONLOAD settings.

£MONLOAD ? will give the current MONLOAD setting and, if output is being sent to a file, say how much has been written.

Suppressing loader warning messages - #LQUIET

The command £LQUIET will switch off all loader warning messages. This facility is useful if you are PRELOADing files which have a large number of references which have to be made dynamic. In normal circumstances each reference will generate a warning message that its status has changed and this can generate a lot of non-significant warnings. You should only ever use the £LQUIET facility sparingly and in well understood loading situations. £LQUIET is cancelled by a call on £N.

Errors and warnings associated with loading

Usually error and warning messages are self-explanatory but sometimes it is not possible to convey the complexity of a failure or how it arose in a short error message, let alone what to do about it. In this section some of the less obvious errors and warnings will be enlarged upon. Most of the theoretical background is in the Technical Aspects section of the complete loader manual (file SUBSYS.LOADERMAN on the 2976 and on the 2988).

Note 1. Loader action on encountering mismatching data entry/ data reference lengths.

A data reference in an object file always has a length associated with it to tell the loader how long the expected data entry should be. If there is a data entry of the correct name already loaded but whose length does not match the data references expected value then the loader may either a) do nothing, b) issue a warning or c) terminate the load with an error. The action followed in any particular case depends both on the loadparms set for the load and the source language of the object file which is being loaded when the mismatch occurs. The rules followed by the loader are tabulated below:

1. All data entries except common entries.

Loading conditions	Ref len > Ent len	REF len < Ent len
Default(Cascade)	FAIL	WARN (except FORTE)
LET	WARN	WARN (except FORTE)
MIN or call on dynamic ref with default loadparms	FAIL	WARN (except FORTE)
MIN+LET or call on dynamic ref with LOADPARM LET set	WARN	WARN (except FORTE)

2. Common entries.

Loading conditions	Ref len > Ent len	REF len < Ent len
Default(Cascade)	*	-
LET	*	-
MIN or call on dynamic ref with default loadparms	FAIL	WARN (except FORTRAN blank common)
MIN+LET or call on dynamic ref with LOADPARM LET set	WARN	WARN (except FORTRAN blank common)

* Common is created at the end of a cascade load if no data entry is found and is always made as long as the longest reference.

A. Errors.

1. *Unable to create USERSTACK -*
   Create AUXSTACK fails -
   Create USER GLA fails -

What happened: When you first run commands which are not in the subsystem, the loader will create up to 3 temporary files on your behalf as required. These are the user stack (T&USTK), the auxiliary stack (T&ASTK) and the user gla (T&UGLA). The attempt to create the file named in the error message failed.

What to do: The second half of the message should indicate what the problem is - e.g. too many files connected, too little file space, etc. - and action should be taken accordingly. You cannot run whatever it is you wanted to run until the file which couldn't be created is created.

2. *Extend USERGLA fails -*

What happened: When the user gla file T&UGLA is set up it is quite small but has the capacity to extend itself as necessary up to a certain system imposed limit. You have tried to exceed that limit.

What to do: The most likely cause of this failure is trying to load a program which demands huge amounts of space from the gla, e.g. very large arrays or common blocks. You should find out why so much space is being requested and which object files are causing the problem. You can set up common blocks in separate data files using the command DATASPACE if you are using FORTRAN. Similarly, large internal arrays can be made external (e.g. %extrinsic in IMP, named common blocks in FORTRAN) and mapped on to data files with DATASPACE.

3. *Base gla full*

What happened: You have tried to 'permanently load' an object file and there is not enough space in the base gla file to satisfy the gla requirement of the file. This failure can occur if you use PRELOAD a lot.

What to do: If there are several files already permanently loaded then if you no longer require them call RESETLOADER to unload them. This may release sufficient space on the base gla. Failing that, proceed as in A.3. above.

4. *Loader tables full*

What happened: You have loaded so much software that the loader tables have completely filled up. This is an extremely unlikely failure.

What to do: Inform your local Advisory Service. If you are loading FORTRAN it may be possible to suppress many of the entry points with the object file editor MODIFY (see User Note 32).

5. *Too little space for initialised stack*

What happened: The user stack has a hardware imposed upper limit of 252K. OPTION INITSTACKSIZE=nn reserves whatever you request of this 252K to be used as initialised stack. The area is used by some languages, especially FORTRAN, to store variables between calls of routines. Your program has requested more initialised stack than you have set up.

What to do: Increase the INITSTACKSIZE. If you turn on loader monitoring or ANALYSE or OBJANAL the object files you can find out how much initialised stack is being requested if you are not sure. If the problem is being caused by cumulative requests from a number of object files then LOADPARM MIN may be worth trying if you are currently on default loadparms and you think that all the files loaded may not be called.

If you are still having problems and you are a FORTRAN user then it may be worth recompiling with PARM MINSTACK

6. *Data ref ENTRY in FILE longer than entry and LOADPARM LET not set*

What happened: You have a data entry called ENTRY already loaded. A reference to ENTRY has been found in FILE which expects the length of ENTRY to be bigger than it actually is. Since you have not specifically permitted this situation (by the command LOADPARM LET), it is treated as a fatal error. See table in Note 1.

What to do: You could set LOADPARM LET but this is potentially very dangerous since you may start overwriting other areas of store. This is one of the classic ways of getting address errors which are very difficult to trace. If you are using FORTRAN and the problem is mismatching common areas then you are probably using LOADPARM MIN or using PRELOAD. In both, the first file with a data reference to ENTRY will cause it to be created with the length specified in the file. If this is not the maximum length for ENTRY then at some point this error will occur. You should always ensure that the longest reference gets loaded first. If you are not a FORTRAN user then the best plan is to modify the software so that data entries and their references have the same length.

7. *Load initiated by dynamic call to ENTRY failed*

What happened: Your program made a dynamic call to ENTRY but the load failed.

What to do: There will be an error message immediately following this message which will give the reason for the failure. The most common is that the loader could not find a particular entry point.

8. *Attempt to call unsatisfied ref ENTRY*

What happened: A previous search for ENTRY failed but as LOADPARM LET was set the reference was made unresolved. You have tried to call it.

What to do: Depends whether you expected the failure. You could provide a dummy entry point or use one of the alias facilities if you must persist. It would be better however to provide the expected software.

9. *Inconsistent directory entry for ENTRY*

What happened: When the loader searched for ENTRY it was not loaded but a reference to it was found in one of the directories in the search list. When the loader loaded the file which the directory said contained ENTRY it found that it wasn't there at all. This can happen when an object file is recompiled with different entries but the directory in which it is inserted is not updated. (Note that if you recompile an object file which is inserted in to your active directory then the active directory is automatically updated.)

What to do: Find out which directory is inconsistent by repeating the load with monitoring turned on. If it's one of your own directories then update it, if it's a system directory e.g. ERCLIB, CONLIB, SUBSYS etc. then tell the Advisory Service who will notify the relevant person otherwise send a TELL message or MAIL to the directory owner suggesting politely that the directory requires updating!

B. Warnings.

1. *Warning - connect directory fails DIRECTORY NAME*

What happened: At log on or when the active directory or searchdir list is changed the loader builds a new list of directories it must search when looking for entries. One of the files in the list could not be connected so it is not in the current search list.

2. *Warning - Satisfying non-dynamic ref to ENTRY by entry at higher loadlevel. Ref made dynamic*

What happened: The loader satisfied a static reference to ENTRY with an entry point from an object file which was certain to be unloaded before the file containing the reference. To ensure that the reference did not point to a file which was no longer loaded, the loader changed the characteristics of the reference to be dynamic.

3. *Warning - Code-ref to ENTRY made dynamic while unloading*

Warning - Data ref to ENTRY made dynamic while unloading

What happened: The file which contained entry point ENTRY has been unloaded but a reference to ENTRY has been found in an object file which is to remain loaded. The reference has been unfixed and turned into a dynamic reference. If the dynamic reference is called later then the loader will once again carry out a full search for ENTRY. A reference which generated warning B.2. above while loading will produce this warning at unload time. Beware of dynamic data references.

4. *Code ref ENTRY made dynamic*

'Data ref ENTRY made dynamic'

What happened: You have set LOADPARM MIN. A static reference to ENTRY has been made dynamic. Beware of dynamic data references.

5. *Code ref ENTRY made unresolved*

'Data ref ENTRY made unresolved'

What happened: You have set LOADPARM LET. A static reference to ENTRY could not be satisfied after a full search so the reference has been changed to type unresolved to allow the run to proceed. If you attempt to call it you will get error A.8.

If this warning comes as an unpleasant surprise then check you have the object file containing ENTRY inserted in one of the directories in the search list. Also check for spelling inconsistencies.

6. *n data ref(s) to ENTRY in FILE LONGER than current entry*

n data ref(s) to ENTRY in FILE shorter than current entry

What happened: You have a data entry ENTRY already loaded. There are n references to this entry in the file you are currently trying to load (FILE). These references expect ENTRY to be longer or shorter - depending on which message you got - than it actually is.

What to do: This depends on the loading conditions at the time. Data references longer than the entry is by far the more serious condition (which is why 'LONGER' is output in capitals), since you may try to read from, or write to, an area of store outwith the defined scope of the data entry. It is very dangerous to proceed with a computation under these circumstances unless you are quite confident that there will be no problems. You will only get the 'LONGER' warning if you have LOADPARM LET set otherwise such a condition will cause termination of the load with error A.6. The 'shorter' condition is usually less serious. At least you won't be trampling over other areas of store but nevertheless it is always worth looking into the reason for the mismatch. While not as immediately dangerous as the 'LONGER' condition it might still mean that the run will provide erroneous results. When in doubt, try to ensure that the lengths of all data references have the same length as the entry.

C. Warnings generated in loader monitoring

1. *Warning - CODE relocated - crosses segment boundary*

What happened: You are running an object file which is a member of a pd file. Its CODE area straddles a segment (256K) boundary. The loader is creating a file called T&CODE and copying the CODE and SST areas of the object file into it so that the file can be run. This is highly inefficient and you should reorganize your pd file.

2. *Warning - CODE flagged as unshareable and relocated*

What happened: The object file has told the loader that the CODE area is unshareable so the loader is taking the action described in C.1. There is nothing you can do about this.

3. *Warning - CODE not connected at preferred site*

Warning - GLA not connected at preferred site

What happened: You are loading a bound object file but the shared (CODE) or unshared (GLA) areas respectively could not be connected at their preferred site. The site is either occupied by another file or you are trying to run a bound file which is a member of a pd file. The loader has to recalculate all the run time addresses for the named areas and plant them in the appropriate locations so there is a loss in efficiency. If you are going to run the file again, try DISCONNECT .ALL before you do to maximize the number of available sites.

4. *Warning - ISTK not connected at preferred site*

What happened: You are loading a bound object file but the initialised stack cannot occupy its preferred location. There is only one, fixed, location for the initialised stack since the user stack is connected at a fixed address. The warning was generated because you are either permanently loading a bound file or temporarily loading it and the preferred location is already in use. (See User Note 33 for further explanation of allocation of initialised stack for permanently and temporarily loaded object files). Run time addresses which refer to initialised stack are recalculated with some loss in loading efficiency. It is not always possible to do anything about this, e.g. if you are loading >1 bound file, and even when it is it may not be worth doing.