General Directives [3]

The loader directives identify relocatable object files to be loaded, select various control options, and declare the segmentation structure. When using the `segldr` command, you can specify files of `segldr` directives with the `-i` option or you can specify directives themselves with the `-D` option.

The loader recognizes the following groups of directives, which should be specified in the indicated order:

1. Global directives identify relocatable object files to be loaded and select various options that control the load process. Most of the global directives are described in this section; global and segment directives are also discussed in sections 5, 8, 9, 10, 11, and 12. Global directives can be entered in any order, but all global directives must precede all other directives.

2. Segment tree definition directives should follow the global directives and are described in, "Segment tree definition directives," page 65.

3. Segmentation directives specify the structure of segmented programs, should follow tree-definition directives, and are described in, "Segment description directives," page 66.

Most loader directives have *KEYWORD=value* syntax. Exceptions are stated in individual directive descriptions. The following describes the conventions used in representing loader directives:

- You can enter directives and keywords in uppercase or lowercase, but not in mixed case. Files, modules, entry points, and common blocks can be specified in uppercase, lowercase, or mixed case; however, under the UNICOS operating system, the loader treats file names and module names of different cases as different names. You can use the CASE directive to change the way in which the loader interprets lowercase directives.

- Comments can appear anywhere in the input directives. Each comment must be preceded with an asterisk (`*`), and all characters to the right of the asterisk are not processed.

- Terminate directives with a semicolon (`;`), an asterisk (`*`), or an end-of-line character.
- More than one directive can appear on a single line, but you must separate multiple directives on a single line with a semicolon.
- A directive cannot be longer than 256 characters.
- Separate elements in a list with commas.
- The loader ignores null directives (for example, two successive semicolons or a blank line).
- Some loader directives can consist of more than one line. These directives have a comma as the last nonblank character before the end-of-line character. See individual directive descriptions for more detail.
- The loader normally uses such special characters as semicolons (`;`), commas (`,`), and others as delimiting characters when processing directives. If you want to use any of these characters (except semicolons) in the names of files, entry points, common blocks, or modules, place the complete name within single or double quotation marks. For example: `bin='`*abc*`:`*def*`.o'`
- Because semicolons are used to separate directives, they cannot be included in literal strings (strings enclosed in quotation marks).

Including directives files

3.1

The `INCLUDE` and `LINCLUDE` directives allow you to specify the names of files that contain directives for the loader to process. When an `INCLUDE` or `LINCLUDE` directive is encountered, the loader stops reading the current directives files and begins reading the file specified with the `INCLUDE` or `LINCLUDE` directive. When the end of the included directives file is reached, the loader resumes processing the original file, using the directive that follows the `INCLUDE` or `LINCLUDE` directive. `INCLUDE` or `LINCLUDE` directives can appear in included files, up to a maximum of 10 nesting levels.

INCLUDE *directive*
3.1.1

The `INCLUDE` directive specifies a file that should be included in the load process.

Format:

```
INCLUDE=file
```

Example:

```
MAP=stat
INCLUDE=dirfile1
DUPLOAD=caution
```

In this example, the loader processes the `MAP=stat` directive, then it processes the directives found in `dirfile1`, and lastly it processes the `DUPLOAD=caution` directive.

LINCLUDE *directive*
3.1.2

The `LINCLUDE` directive specifies a file that should be included in the load process. Only the file name component should be specified. The loader scans the list of search directories to locate the file. (See "`LIBDIR` directive," page 111, for information on user directory search lists.)

Format:

```
LINCLUDE=file
```

Example:

```
LIBDIR=/mydir/lib
LINCLUDE=dirfile2
```

In this example, the loader searches for file `/mydir/lib/dirfile2`. If it is found, the directives in `dirfile2` is processed. Otherwise, the loader looks for `/lib/dirfile2`, then `/usr/lib/dirfile2`. It uses the first of these files it finds.

Including object modules

3.2

The BIN, LBIN, LIB, and LLIB directives let you identify the relocatable modules that you want the loader to include in your program. The DUPORDER directive lets you determine how to select the modules to be retrieved from libraries. The NODEFLIB and OMIT directives provide control over the system default libraries. The FORCE, MODULES, and COMMONS directives provide you with additional control over the loading process.

Files specified in BIN or LBIN directives or specified as command-line arguments by the loader are all considered to be bin files. Segmented object files specified as arguments on the ld command line are also considered to be bin files. By convention, bin files should be the portion of your program that you have written. Files specified in LIB, LLIB or DEFLIB directives, or specified with the -l option on the segldr or ld command line, are all lib files. Library files built by bld and specified as arguments on the ld command line are also considered lib files. By convention, lib files are libraries of previously written routines that the loader includes in your program as needed. The loader processes bin and lib files in a very similar manner: it scans all modules from both bin and lib files, and it establishes and retains the calling relationships between all modules. After processing all files in this way, the loader determines which modules must be loaded. It begins at the module containing the transfer entry address and scans the calling relationships, retaining all modules that are called and deleting all others. Exceptions and differences between bin and lib file processing are as follows:

- All bin files are processed before all lib files. If modules containing duplicate entry points are discovered, the loader uses the first occurrence. See "DUPORDER directive," page 32.
- The FORCE directive causes the loader to include all modules from bin files, even if they are not referenced. FORCE does not affect modules from lib files. See "FORCE directive," page 30.
- The BRIEF option to the MAP directive limits load maps to modules derived from bin files. Modules from lib files are not listed. See "MAP directive," page 37.
- The DUPORDER directive affects the selection of modules from library files. See "DUPORDER directive," page 32.
- The DUPENTRY directive controls messages concerning duplicate definitions of the same entry point. It differentiates between entry points from bin files and those from lib files. See "DUPENTRY directive," page 41.

- Fortran `BLOCKDATA` subprograms encountered in `bin` files are always included in the program. `BLOCKDATA` subprograms encountered in `lib` files are included only if they are referenced.
- The loader always includes a module written in C and encountered in a `bin` file if the module initializes global data. C modules from `lib` files that initialize global data are included only if they are referenced.

In addition to the files you provide, the loader also scans a set of default system libraries. You can use the `NODEFLIB` directive to inhibit default library processing.

The default libraries that `segldr` and `ld` scan and the order of scanning are specified in the default directives files. The default directives files as released by Cray Research specify processing the libraries in the order listed:

```
libc.a
libu.a
libm.a
libf.a
libfi.a
libsci.a
libp.a
```

Some of the default libraries listed above may be released separately from the UNICOS operating system; therefore, they may not be present on your system. Missing libraries are silently ignored.

The loader uses a directory search algorithm to locate each default `lib` file. If you have provided a list of search directory names by using the `-L` option or `LIBDIR` directive, the loader searches the directories specified to locate the default libraries. If the libraries cannot be located in those directories, the loader searches the directories specified in the default directory search list. (See "`DEFDIR` directive," page 109, for information on the default directory search list.)

`BIN` *directive*
3.2.1

The `BIN` directive names the relocatable object input files to be searched. Multiple `BIN` directives have a cumulative effect. If you specify multiple files with `BIN`, the loader processes them in the order specified.

If a module is present in more than one file, the loader loads the first module encountered. However, if you use the `MODULES` directive and specify a particular file, this rule may not apply.

Format:

```
BIN=file1[,file2,file3,...,filen]
```

file$_i$ Names of relocatable input files to be included. If no `bin` files are specified, the default is `a.o`.

If you continue this directive beyond one line, end each continued line with a comma.

Examples:

```
bin=myfile,../group/ourfile.o,
     ../sue/herfile.a,/u/steve/anyfile.o

bin=newfile.a,oldfile.a
```

Modules contained in global `BIN` files (as opposed to segmented `BIN` files) are not assumed to be in any particular segment, unless the module is specified in a segmented `MODULES` directive.

Command-line equivalent: *objfiles* argument

`LBIN` *directive*
3.2.2

The `LBIN` directive, in a manner similar to the `BIN` directive, names relocatable object input files for the loader to search. With `LBIN`, however, only the file name component is specified. The `LIBDIR` directory search applies to names on the `LBIN` directive. Each `LIBDIR` directory is searched for the files specified. The first file found is included in the program.

Format:

```
LBIN=file1[,file2,file3,...,filen]
```

file$_i$ Names of the files you provide.

`LIB` *directive*
3.2.3

The `LIB` directive names the relocatable object library files for the loader to search when the loader is trying to find entry points that are referenced in `BIN` files but are not defined in any `BIN` files or previously searched `LIB` files.

Use the `LIB` directive to specify `lib` files in addition to those in the loader's default list of libraries. Library files specified with the `LIB` directive are searched in the order specified and before any default libraries.

The effect of multiple `LIB` directives is cumulative.

If you continue this directive beyond one line, end each continued line with a comma. The `LIBDIR` directory search does not apply to files specified in `LIB` directives. Each name should be a complete path name.

Format:

```
LIB=lib1[,lib2,lib3,...,libn]
```

lib$_i$ Names of the libraries you provide.

Examples:

```
lib=/u/lib/lib7.a,/u/lib/libarf.a,
     /lib/lib3A.a,mytmplib.a,mylibY.a

lib=mylibs.o,/lib/libc.a
```

These examples each specify seven libraries that the loader should search before searching the default libraries. The libraries are searched in the order given.

`LLIB` *directive*
3.2.4

The `LLIB` directive, in a manner similar to the `LIB` directive, specifies relocatable object libraries for the loader to search. With `LLIB`, however, only the file name component is specified. The `LIBDIR` directory search applies to what is specified on the `LLIB` directive. Each `LIBDIR` directory is searched for the files specified. The first file found is included in the program.

Format:

```
LLIB=name[,name,...]
```

Command-line equivalent: `-l` option

Example:

```
LIBDIR=/lib/xlib
LLIB=libscan.a
```

First, the loader looks for file `/lib/xlib/libscan.a`, then `/lib/libscan.a`, and finally `/usr/lib/libscan.a`. It uses the first of these files it finds.

`NODEFLIB` *directive*
3.2.5

The `segldr` default directives file contains a set of `DEFLIB` directives. `NODEFLIB` instructs the loader to ignore some or all of the libraries that have been specified by `DEFLIB` directives. If all default libraries are to be ignored, only modules found in files declared as `BIN` or `LIB` files are considered for loading.

Format:

```
NODEFLIB
  NODEFLIB=deflib1[,deflib2,...,deflibn]
```

If the first format is used, all default libraries are ignored. If the second format is used, only the specified default libraries are ignored.

Note: For a segmented load, you must specify the library containing the loader run-time routine `$SEGRES`.

Example:

```
NODEFLIB
LIB=/lib/libio.a,/lib/libc.a
```

The preceding example tells the loader to search the libraries specified by the `LIB` directive (in the order specified) for unsatisfied externals. The loader does not search the default libraries for entry points not found in the specified libraries.

Example:

```
NODEFLIB=libp.a
```

This example directs the loader to ignore the Pascal library, and to process the other default libraries as usual.

Command-line equivalent: `-N` option

DEFLIB *directive*
3.2.6

The `DEFLIB` directive instructs the loader to add libraries to its list of default libraries. Each library specified in the `DEFLIB` directive is added to the end of the list of default libraries. If `DEFLIB` specifies a library that is already part of the default library list, the loader moves that library name to the end of the list. You may use `NODEFLIB` and `DEFLIB` together to replace some or all of the default system libraries (See "Including object modules," page 24). All libraries specified by the `DEFLIB` directive are processed after all libraries that are specified by the `LIB` directive are processed.

Format:

`DEFLIB=`*deflib*$_1$`[,`*deflib*$_2$`,...,`*deflib*$_n$`]`

deflib$_i$ Name of one library to add.

Example:

```
DEFLIB=libmylib.a
```

This example directs the loader to add the user's library to the end of the default library list.

Example:

```
NODEFLIB; DEFLIB=libuser.a
```

This example suppresses all normal default system libraries, replacing them with one user library.

FORCE *directive* 3.2.7

The loader gathers all modules in all files specified with global BIN and LIB directives. It then discards all modules with entry points that are never called. FORCE specifies that subprograms not called by other subprograms are to be loaded anyway (force-loaded). This can be helpful in debugging, letting you force-load a debug routine not actually called by the program.

Format:

```
FORCE=ON|OFF
```

ON — Enables force-loading; when FORCE=ON, the loader loads all modules specified in MODULES directives and in all bin files.

OFF — Disables force-loading; when FORCE=OFF, the loader discards modules to which no references have been made (except the XFER directive's module and the BLOCKDATA subprograms found in bin files) (default).

Command-line equivalent: -F option

MODULES *and* SMODULES *directives* 3.2.8

The MODULES and SMODULES directives specify modules to load. Normally, if more than one module with a particular name exists, the loader chooses the first such module it encounters. If modules of the same name are encountered in different files, you can use the MODULES directive to specify the files from which the modules are obtained.

Additionally, you can use MODULES to specify the loading order in a nonsegmented load. Loading order can be affected by other considerations such as the current memory ordering algorithm. See "Executable program organization," page 96. If you use the MODULES directive, an error message will be issued if the modules specified cannot be located in any included file. Error messages are not issued if SMODULES is used.

Format:

```
MODULES=modname1[:file1][,modname2[:file2],...,modnamen[:filen]]
SMODULES=modname1[:file1][,modname2[:file2],...,modnamen[:filen]]
```

modname$_i$ Name of the module to be loaded.

file$_i$ Name of the file from which to obtain the module.

Example:

```
MODULES=SUBA,SUBB:myfile.o,SUBC
MODULES=SUBD:lib1.a
```

In the preceding example, the MODULES directive tells the loader to obtain SUBB from file myfile.o and to obtain SUBD from file lib1.a; modules SUBA and SUBC are obtained from the first file in which each is encountered.

COMMONS *and* SCOMMONS *directives*
3.2.9

In an unsegmented program, COMMONS and SCOMMONS cause the listed common blocks to be loaded in the indicated order. In a segmented load, however, the COMMONS directive serves only to order and/or set the size of common blocks. Loading order can be affected by other considerations such as the current memory ordering algorithm. See "Executable program organization," page 96.

If you continue this directive beyond one line, end each continued line with a comma.

If you use the COMMONS directive, an error message will be issued if the indicated common blocks cannot be located in any included file. No error messages are issued if SCOMMONS is used.

Format:

```
COMMONS=blkname1[:size1][,blkname2[:size2 ],...,blknamen[:sizen]]
SCOMMONS=blkname1[:size1][,blkname2[:size2 ],...,blknamen[:sizen]]
```

blkname$_i$ Name of the common block to be loaded.

size$_i$ Decimal number indicating the size of the common block. If present, it overrides any common block sizes declared in your code. If the size specified is 0, the first common block size encountered in your code (for this common block) is used. By default, the loader uses the longest common block definition it encounters in those modules of your code that are actually referenced and loaded.

DUPORDER *directive*
3.2.10

The DUPORDER directive selects the method the loader uses to process duplicated entry points found in libraries. When processing BIN files, the loader always chooses the first occurrence of a duplicated entry point. If the duplicated symbol appears in both a BIN and a LIB file, the loader always chooses the one in the BIN file. If the duplicated symbol appears only in library files, the loader has two methods of selecting the occurrence of the symbol to use: if the DUPORDER directive is not enabled (OFF, default for `segldr`), the loader uses the first occurrence of the symbol. If the DUPORDER directive is enabled (ON, default for `ld`), the loader uses *ordered duplicate selection*, which means that the loader locates the first module that references the duplicated symbol and then looks for a definition of the symbol in succeeding modules. The first definition found in a succeeding module is the one used. If the loader finds no succeeding definition, the first definition encountered anywhere is used.

Format:

```
DUPORDER=ON|OFF
```

`ON` The loader uses *ordered duplicate selection* to choose the entry point to use (default for `ld`).

`OFF` The loader uses the first occurrence of the duplicated entry point (default for `segldr`).

The following example, in which a partial Fortran program is loaded, contrasts the `ON` and `OFF` settings of the `DUPORDER` directive.

```
module 1:  PROGRAM DUPEXAMP
             .  .  .
           CALL REFMOD
             .  .  .
           END

module 2:  SUBROUTINE DUPLICAT
             .  .  .
           END

module 3:  SUBROUTINE REFMOD
           CALL DUPLICAT
           END

module 4:  SUBROUTINE DUPLICAT
             .  .  .
           END
```

Module 1 contains the main program and is included in the load in a binary file. Modules 2, 3, and 4 occur, in the order shown, in library files. If the `DUPORDER` directive is disabled (`OFF`, or not used), the loader selects the `DUPLICAT` symbol in module 2 to satisfy the reference in module 3, because it is the first occurrence of the symbol. If the `DUPORDER` directive is enabled (`ON`), the loader selects the `DUPLICAT` symbol from module 4 because this is the first definition for `DUPLICAT` that occurs after the reference in module 3.

`OMIT` *directive*
3.2.11

The `OMIT` directive specifies modules that should be bypassed by the loader when processing object or library files.

Format:

```
OMIT=module1[:file1][,module2[:file2],...]
```

Modules specified on the `OMIT` directive are not included in the program, even if referenced from other modules. If *file* is not present, all modules with the indicated name are omitted, regardless of the file in which they are found. If *file* is specified, only *module* from that file is omitted. Modules with the same name, but in different files, are included.

If a module is omitted, and the program makes references to symbols within that module (that cannot be satisfied by any other module), the reference is treated in the same manner as any other unsatisfied reference.

Example:

```
omit=printf$c:/lib/libc.a,mymodule
```

In this example, the `printf$c` module, from file `/lib/libc.a` and from any module with the name `mymodule`, is bypassed in the load process.

The executable program
3.3

The `ABS` and `TRIAL` directives give you a measure of control over the executable program that the loader produces. You can tell the loader where to write the executable file, or whether the loader should produce the executable file or only a load map and error messages.

ABS ***directive***
3.3.1

The ABS directive specifies the file to receive the executable program constructed by the loader.

Format:

```
ABS=file
```

file The *file* parameter specifies the file to receive the executable program. The default is `a.out`.

Command-line equivalent: `-o` option

TRIAL ***directive***
3.3.2

The `TRIAL` directive lets you make a sample of the loader run without creating any executable output. You can therefore print a load map and most error messages without using a lot of memory to build the executable output. Making test runs with `TRIAL` also lets you determine optimal memory use for data areas or identify total memory requirements for a particular application.

Format:

```
TRIAL
```

Command-line equivalent: `-t` option

Load map control
3.4

The `ECHO`, `COMMENT`, `MAP`, and `TITLE` directives control the information that the loader writes to the listing output file. The default listing output file is `stdout`. You can change these defaults by using the `-M` option.

ECHO ***directive*** 3.4.1

The ECHO directive resumes or suppresses the printing of input directives.

Format:

```
ECHO=ON|OFF
```

`ON` Resumes the listing of input directives.

`OFF` Suppresses directive listing. If ECHO=OFF, the loader automatically echoes erroneous directive lines, followed by the error message (default).

Comments 3.4.2

Comments annotate the loader directives. They are echoed to the listing file but are otherwise ignored. All characters to the right of the asterisk are considered part of the comment string.

The asterisk character begins a comment. You can use comments in either the global or the segment description directives, but you cannot embed comments within directives.

Format:

```
* comment string
```

Example:

```
TITLE=GLOBAL DIRECTIVES
****************************************
* Global directives
****************************************
BIN=X
TITLE=TREE DIRECTIVES
****************************************
*Tree directives
****************************************
TREE
   ROOT(A,B)
ENDTREE
TITLE=SEG.DESCR.DIR.
****************************************
SEGMENT=ROOT
* Segment Description Directives follow
```

MAP *directive* 3.4.3

The MAP directive controls the loader map output generation. Besides memory mapping, MAP provides the time and date of load, the length of the longest branch and the last segment, and the transfer address. Map output is written to the listing file. See "Examples," page 123, for more information on map output.

Format:

```
MAP=[keyword1, ..., keywordn]
```

NONE	Writes no map output to the listing file (default).
STAT	Writes statistics for the load (such as date and time), length of longest branch, last segment, transfer entry point, and stack and heap information.
ALPHA	Writes the STAT information plus the block map for each segment, listing the modules in alphabetical order.
ADDRESS	Writes the ALPHA information, but it lists modules by ascending load address.
PART	Writes both ALPHA and ADDRESS information.

`EPXRF`	Writes the `STAT` information plus the Entry Point Cross-reference table.
`CBXRF`	Writes the `STAT` information plus the Common Block Cross-reference table.
`FULL`	Writes all `PART`, `EPXRF`, and `CBXRF` information.
`BRIEF`	Limits information in the `ADDRESS` and `ALPHA` output to modules from `bin` files.

The effects of multiple keywords are cumulative.

Command-line equivalents: `-m` and `-M` options.

`TITLE` *directive*
3.4.4

The `TITLE` directive places an arbitrary, user-defined character string in the second line of each page header. `TITLE` forces a page eject and then writes the header lines at the top of the new page.

The title line is initially clear, and it can be reset by `TITLE` directives in either the global or the segment description directives portion of the input. An end-of-line or a semicolon (;) signals the end of the `TITLE` string.

Format:

```
TITLE[=title string]
```

title string	User-defined character string; maximum length is 74 characters. If no *title string* is specified, the title line is cleared.

Example:

```
TITLE=Place this in the page header, please.
```

This `TITLE` directive copies the string "`Place this in the page header, please.`" to the page header.

Controlling error messages
3.5

The MLEVEL, USX, REDEF, DUPENTRY, DUPLOAD, NODUPMSG, NOUSXMSG and MSGLEVEL directives let you control the printing of error messages. Error messages are written to stderr by default, although you can redirect them to other files by using standard I/O redirection or with the loader -k command-line option.

MLEVEL *directive*
3.5.1

The MLEVEL directive controls the loader messages on the listing output. The keyword indicates the lowest-priority message to be printed. If you do not use the MLEVEL directive, MLEVEL=CAUTION is assumed.

Format:

```
MLEVEL=keyword
```

FATAL — Prints only FATAL-level messages. When a message with this severity level is issued, the loader is terminated immediately, and no executable output is written.

WARNING — Prints FATAL- and WARNING-level messages. A WARNING-level message indicates that the executable output may not be written; if the output is written, it is not executable. Processing continues so that additional messages may be printed.

CAUTION — Prints FATAL-, WARNING-, and CAUTION-level messages. A CAUTION-level message indicates that an error may have occurred, but it is not severe enough to prohibit generation of executable output (default).

NOTE — Prints FATAL-, WARNING-, CAUTION-, and NOTE-level messages. A NOTE-level message indicates that the loader may have been misused or used inefficiently; it has no effect on execution validity.

COMMENT — Prints all levels of messages. A COMMENT-level message does not affect execution.

USX *directive*
3.5.2

The USX directive lets you determine the severity level of unsatisfied external symbols. CAUTION is the default.

Format:

```
USX=keyword
```

FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

IGNORE This is the same as COMMENT.

REDEF *directive*
3.5.3

The loader generates an error message if you redefine common blocks with varying lengths in different modules.

REDEF lets you control the severity level of the loader's messages when common blocks are defined with varying sizes. The loader always takes the longest definition, regardless of the REDEF value. The severity level you select applies to cases in which the common block is redefined with a larger size. The severity level is one level lower for cases in which the common block is redefined with a smaller size.

Format:

```
REDEF=keyword
```

FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

IGNORE This is the same as COMMENT.

DUPENTRY ***directive***
3.5.4

The DUPENTRY directive controls the severity of the message generated when the loader encounters a duplicated entry point; the default is CAUTION. The loader generates the duplicate entry error message with the severity level you specify. See "Program Duplication and Block Assignment," page 75, for more information on duplicated entry points.

Format:

```
DUPENTRY=keyword1[,keyword2[,keyword3]]
```

FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

IGNORE This is the same as COMMENT.

The default for `segldr` is DUPENTRY=CAUTION,CAUTION,NOTE. The default for `ld` is DUPENTRY=CAUTION,NOTE,NOTE.

The first keyword controls the severity level of messages issued for cases in which both duplicated entry points are in a `bin` file. The second keyword controls the severity level of messages issued for cases in which one duplicated entry point is in a `bin` file and the other is in a `lib` file. The third keyword controls the message severity level for cases in which both duplicated entry points occur in a `lib` file. Table 4 shows this correspondence.

Table 4. DUPENTRY keywords for duplicated entry definitions

Keyword	`bin` file	`lib` file
keyword$_1$	both entries	N/A
keyword$_2$	one entry	one entry
keyword$_3$	N/A	both entries

If the second or third keyword is not provided, the value of the last keyword present is used.

DUPLOAD *directive*
3.5.5

The DUPLOAD directive lets you control the severity of messages that the loader generates when a common block is initialized by two or more modules. The loader generates messages with the severity level you specify with DUPLOAD. This level applies to common blocks referenced by C language modules. The level of messages generated for multiple common block initialization by Fortran modules is one severity level lower than the level you specify. Subsequent initializations of a common block overwrite any preceding ones.

Format:

```
DUPLOAD=keyword
```

FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

IGNORE This is the same as COMMENT.

NODUPMSG *directive*
3.5.6

The NODUPMSG directive suppresses messages about duplicated entry points. If you know that one or more particular entry points are duplicated, and do not want the loader to issue messages about those symbols, use NODUPMSG to suppress the messages.

Format:

```
NODUPMSG=epname[,epname,...]
```

epname Name of an entry point for which no message should be issued.

NOUSXMSG *directive* 3.5.7

The NOUSXMSG directive suppresses messages concerning unsatisfied external references. If you know that one or more specific external references will not be satisfied in your program, and you do not want the loader to issue messages about those references, use NOUSXMSG to suppress the messages.

Format:

```
NOUSXMSG=epname[,epname,...]
```

epname Name of an entry point for which no unsatisfied references are present and for which no messages should be issued.

MSGLEVEL *directive* 3.5.8

The MSGLEVEL directive lets you set the severity level for any message that the loader issues. For instance, you can increase the severity for certain cases and decrease the severity for others. If you increase the severity to equal to or greater than the WARNING level for a particular error, the loader will not make your program executable if that error occurs. If you decrease the severity to equal to or below the NOTE level for a particular error, the loader will not print a message if that error occurs.

Format:

```
MSGLEVEL=number:keyword[,number:keyword...]
```

number Number of message for which the severity level should be changed.

keyword FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

Example:

```
MSGLEVEL=268:NOTE,114:WARNING
```

In this example, the severity level of message number 268 is set to NOTE; the severity level of message number 114 is set to WARNING.

Message numbers are displayed as part of every error message that is issued. They are appended to the ldr message group identifier. In the following message example, the message number is 112:

```
ldr-112 sldr:  WARNING
     File 'a.out' is not executable due to
     previous errors.
```

Controlling entry points and execution

3.6

The XFER, EQUIV, and SET directives let you control the point at which your program begins executing, and they also intercept definitions of entry points at load time.

XFER *directive*

3.6.1

The XFER directive specifies the transfer entry point for your program. Control is passed from the system start-up routine to the XFER entry point. If you do not use the XFER directive, the loader uses the first primary entry point it encounters as the transfer entry point. A primary entry point can be specified by the Fortran language PROGRAM statement, by the CAL START pseudo instruction, or by the C language procedure name of main.

The XFER directive can also be used to identify which primary entry point to use as the transfer entry when the loader encounters more than one primary entry point.

Format:

```
XFER=entry
```

entry Entry point name.

EQUIV ***directive*** 3.6.2

The EQUIV directive lets the loader substitute a call or reference to one entry point for a call or reference to another entry point.

Format:

```
EQUIV=epname(syn1[,syn2,...,synn])
```

epname Name of a target entry point.

syn_i Name of the entry point to be linked to *epname*.

If you continue this directive beyond one line, end each continued line with a comma.

Example:

```
      .
      .
      .
      CALL A
      .
      .
      .
      CALL B
      .
      .
      .
```

In the preceding code sequence, the calls to A and B are linked to C by the following specification:

```
EQUIV=C(A,B)
```

The module containing entry point C is loaded, but the module or modules containing A and B might not be loaded. The module or modules containing A and B are loaded if they are needed to satisfy other references to other entry points within those modules.

In this example, EQUIV has the same effect as using a text editor to replace all occurrences of CALL A and CALL B with CALL C, except that you do not have to recompile or change the source code.

SET *directive*
3.6.3

The SET directive assigns a value to an external entry point. A SET value for the specified entry point takes precedence over a value encountered in the relocatable modules.

Format:

```
SET=epname:value[:mod]
```

epname Specifies the entry point to be given a value.

value Decimal value associated with *epname*.

mod Alignment modifier. *mod* may be one of the following:

- `W` Represents a word address (default)
- `P` Represents a parcel address
- `V` Represents a constant

UNSAT *directive*
3.6.4

The UNSAT directive specifies the names of one or more unsatisfied external references that are placed in the loader symbol tables before loading any object files. UNSAT is useful if all files to be loaded are `lib` files. Modules from `lib` files are included in the executable program only if an entry point in the module satisfies a reference to the `lib` file. With the UNSAT directive, you can specify the entry points that will cause modules to be included from library files.

Format:

```
UNSAT=epname1[,epname2...]
```

$epname_i$ Name of an unsatisfied entry point.

Example:

```
UNSAT=blocktwo
LIB=mylib.a
```

An unsatisfied external reference to `blocktwo` is generated, causing the module in `mylib.a` that contains the `blocktwo` entry point to be included in the program.

Command-line equivalent: `-u` option

Program alignment and initialization
3.7

The `ALIGN`, `PRESET`, and `ORG` directives let you initialize uninitialized data areas and control the loading of some modules or common blocks.

`ALIGN` *directive*
3.7.1

The `ALIGN` directive controls the starting locations of modules and common blocks. The loader recognizes an align bit for each relocatable module and common block containing an `ALIGN` pseudo-op. See the *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*, publication SR–3108.

Format:

```
ALIGN=keyword
```

`IGNORE` Allocates the local blocks of each module and each common block at the beginning of the word following the previous local block or common block. The align bit is ignored.

`MODULES` Allocates the local blocks of each module containing code to an instruction buffer boundary according to the instruction buffer size of the machine. The instruction buffer size is 32 words for Cray PVP systems. Common blocks are forced to instruction buffer boundaries only when the align bit is set.

NORMAL — Allocates the local blocks of each module and each common block with the align bit set to an instruction buffer boundary, according to the machine's instruction buffer size. The instruction buffer size is 32 words for Cray PVP systems.

If the align bit is not set for a local or common block, that local or common block is allocated at the word following the previous local or common block (default).

Command-line equivalent: -a option

PRESET *directive* 3.7.2

The PRESET directive specifies a value that the loader uses to preset uninitialized data areas within the object module (for example, variables in labeled Fortran common blocks with no DATA statements).

Stack-allocated data is not part of the program image. As a result, the loader cannot preset variables that reside on the stack.

Format:

```
PRESET=keyword
```

ONES — Sets uninitialized data words to -1.

ZEROS — Sets uninitialized data words to 0 (default).

INDEF — Sets uninitialized data to O'060505400000000000000. This value generates a floating-point error if used as an operand in a floating-point operation.

-INDEF — Sets uninitialized data to O'160505400000000000000. This value is the same as that of INDEF, except that it is negative.

`INDEFA` Sets uninitialized data to the sum of a logical OR operation of `O'060505400000000000000` and the address of the word being preset. This value is the same as that of `INDEF`, except the address of the word referenced appears in the low-order bits of the value.

`-INDEFA` Sets uninitialized data to the sum of a logical OR operation of `O'106050540000000000000000` and the address of the word being preset. This value is the same as that of `-INDEF`, except the address of the word referenced appears in the low-order bits of the value.

value Inserts a 16-bit, user-supplied octal value into each parcel of uninitialized data words. The *value* must be in the range `0 <=` *value* `<= O'177777`

Command-line equivalent: `-f` option

`ORG` *directive*
3.7.3

The `ORG` directive sets the initial addresses for different portions of your program. Normal programs must have `ORG` values of 0. The `ORG` directive should be used only for special-purpose programs.

Format:

```
ORG=corg:dorg:lorg
```

corg Specifies an octal value between 0 and 77777777. The default is 0, which is the initial address for the code portion of the program.

dorg Specifies an octal value between 0 and 77777777. The default is 0, which is used for initial data for the program.

lorg Specifies an octal value between 0 and 177777. The default value is 0.

Miscellaneous global directives
3.8

The SYMBOLS directive determines whether the debug symbol table should be constructed. The KEEPSYM and HIDESYM directives determines the visibility of externals in the relocatable module. The CASE and CPUCHECK directives control case conversions and machine characteristic checking, respectively. The COMPRESS directive controls compression of executable files. The LOGFILE directive specifies the name of the file to which the loader writes log messages. The LOGUSE directive specifies the names of the object or library files for which log messages should be generated.

SYMBOLS *directive*
3.8.1

The SYMBOLS directives determines whether the loader constructs the debug symbol table for the executable program.

Format:

```
SYMBOLS=ON|OFF
```

ON — The loader writes symbol table information to the executable file, following the executable program (default).

OFF — Instructs the loader to ignore all symbol table information.

Command-line equivalent: -g and -s options

KEEPSYM and HIDESYM *directives*
3.8.2

The KEEPSYM and HIDESYM directives determine the visibility of externals in the relocatable module. By default, global symbols are visible. The HIDESYM directive hides selected symbols. The KEEPSYM directive hides all symbols except the selected symbols. A directive is needed for each symbol affected.

The KEEPSYM and HIDESYM directives are mutually exclusive.

Format:

```
HIDESYM=symbol:type
  KEEPSYM=symbol:type
```

symbol Name of a symbol.

type The type of symbol, which is one of the following:

`C`	Common block symbol
`E`	External symbol
`B`	Both types of symbols (a C language external)

CASE *directive* 3.8.3

The CASE directive controls whether characters in the directives file are converted to uppercase before they are processed.

Format:

```
CASE=keyword
```

`UPPER`	Directs the loader to convert all module, entry point, and common block names in the directives to uppercase. Usually this is desirable when no relocatable modules with lowercase names are encountered.
`MIXED`	Specifies that no translation is done, and names must match exactly (default).

CPUCHECK *directive* 3.8.4

The CPUCHECK directive controls whether machine characteristic checking is done within the loader. Turning off checking allows a slight increase in the execution speed of the loader, but it also allows the loading and execution of modules that have incompatible characteristics.

Format:

```
CPUCHECK=keyword
```

`ON` Enables machine characteristic type checking (default).

`OFF` Disables machine characteristic type checking.

COMPRESS *directive* 3.8.5

The `COMPRESS` directive enables or disables compression of executable files, and it specifies the size of blocks the loader should consider for compression. As the loader loads your program, it scans for large areas of the program in which each word contains the same value. When it finds a block of words with the same value, it generates a compression entry rather than the actual code. The system start-up routine expands the compression entry into actual code at run time. To be eligible for compression, a block must satisfy the following requirements:

- It must contain only data
- It must contain repetitively initialized values
- It must have a block size larger than the compression threshold

Executable programs that have been compressed require less memory to link, as well as less storage space. Execution time of the system start-up routine increases for compressed programs, but file transfer time is decreased.

Format:

```
COMPRESS=keyword
```

`OFF` Disables all compression.

number Sets compression block size to *number*. The default is 1000.

`LOGFILE` *directive*
3.8.6

The `LOGFILE` directive specifies the name of the file to which the loader writes log messages. (See "`LOGUSE` directive," page 53, for information on log messages.) Normally, this directive should be used in the default `def_seg` and `def_ld` directives files to identify the log file for all users.

Format:

```
LOGFILE=file
```

file Name of a file to which the loader writes log messages.

The log file must be created prior to loader execution, and it must have `write` permission enabled for all users. On systems with multilevel security (MLS), the log file must be created in the most restrictive partition of the file system, so that all users can write to the file. The loader appends log messages to the end of the file; it does not initialize, summarize, or report on the contents of the log file. If the log file is not present, or the loader cannot write to it, the loader suppresses all log messages without issuing an error message.

Command-line equivalent: none

`LOGUSE` *directive*
3.8.7

The `LOGUSE` directive specifies the names of object or library files for which log messages should be generated. Normally, this directive should be used in the default directives files `def_seg` and `def_ld` to log the usage of specific object or library files by all users. If the specified library is not a default library (even if it is in a default search path) you should specify the full path name of the library.

Format:

```
LOGUSE=file1[,file2,...]
```

file Name of an object or library file whose usage should be logged.

Whenever the file specified on the `LOGUSE` directive is processed by the loader, a log message is appended to the log file (specified by the `LOGFILE` directive).

The generated message is in ASCII characters, and it is terminated by new-line characters ("\n"). Individual fields within the message are separated by a vertical bar ("|"). The message format is as follows:

`loguse|`*filename*`|`*date*`|`*time*`|`*uid*`|`*code*`\n`

filename Name of file.

date Date of reference to the file; format is *mm*/*dd*/*yy*.

time Time of reference to the file; format is *hh*:*mm*:*ss*.

uid User ID of user referencing the file.

Character indicating the type of reference.

- `s` The file was scanned, but no modules were included.
- `i` The file was scanned, and modules were included.

Command-line equivalent: none.