Name ld– link-editor for object files

Synopsis

```
ld [-64] [-a | -r] [-b] [-Bdirect | nodirect]
[-B dynamic | static] [-B eliminate] [-B group] [-B local]
[-B reduce] [-B symbolic] [-c name] [-C] [-d y | n]
[-D token,...] [-e epsym] [-f name | -F name] [-G] [-h name]
[-i] [-I name] [-l x] [-L path] [-m] [-M mapfile]
[-N string] [-o outfile] [-p auditlib] [-P auditlib]
[-Q y | n] [-R path] [-s] [-S supportlib] [-t]
[-u symname] [-V] [-Y P,dirlist] [-z absexec]
[-z allextract | defaultextract | weakextract ]
[-z altexec64] [-z combreloc | nocombreloc ] [-z defs | nodefs]
[-z direct | nodirect] [-z endfiltee] [-z finiarray=function]
[-z globalaudit] [-z groupperm | nogroupperm] [-z help ]
[-z ignore | record] [-z initarray=function] [-z initfirst]
[-z interpose] [-z lazyload | nolazyload]
[-z ld32=arg1,arg2,...] [-z ld64=arg1,arg2,...]
[-z loadfltr] [-z muldefs] [-z nocompstrtab] [-z nodefaultlib]
[-z nodelete] [-z nodlopen] [-z nodump] [-z noldynsym]
[-z nopartial] [-z noversion] [-z now] [-z origin]
[-z preinitarray=function] [-z redlocsym] [-z relaxreloc]
[-z rescan] [-z target=sparc|x86] [-z text | textwarn | textoff]
[-z verbose] filename...
```

Description The link-editor, `ld`, combines relocatable object files by resolving symbol references to symbol definitions, together with performing relocations. `ld` operates in two modes, static or dynamic, as governed by the `-d` option. In all cases, the output of `ld` is left in the file `a.out` by default. See NOTES.

In dynamic mode, `-dy`, the default, relocatable object files that are provided as arguments are combined to produce an executable object file. This file is linked at execution with any shared object files that are provided as arguments. If the `-G` option is specified, relocatable object files are combined to produce a shared object. Without the `-G` option, a dynamic executable is created.

In static mode, `-dn`, relocatable object files that are provided as arguments are combined to produce a static executable file. If the `-r` option is specified, relocatable object files are combined to produce one relocatable object file. See `Static Executables`.

Dynamic linking is the most common model for combining relocatable objects, and the eventual creation of processes within Solaris. This environment tightly couples the work of the link-editor and the runtime linker, `ld.so.1`(1). Both of these utilities, together with their related technologies and utilities, are extensively documented in the Linker and Libraries Guide.

If any argument is a library, `ld` by default searches the library exactly once at the point the library is encountered on the argument list. The library can be either a shared object or relocatable archive. See `ar.h`(3HEAD)).

A shared object consists of an indivisible, whole unit that has been generated by a previous link-edit of one or more input files. When the link-editor processes a shared object, the entire contents of the shared object become a logical part of the resulting output file image. The shared object is not physically copied during the link-edit as its actual inclusion is deferred until process execution. This logical inclusion means that all symbol entries defined in the shared object are made available to the link-editing process. See Chapter 4, "Shared Objects," in *Linker and Libraries Guide*.

For an archive library, `ld` loads only those routines that define an unresolved external reference. `ld` searches the symbol table of the archive library sequentially to resolve external references that can be satisfied by library members. This search is repeated until no external references can be resolved by the archive. Thus, the order of members in the library is functionally unimportant, unless multiple library members exist that define the same external symbol. Archive libraries that have interdependencies can require multiple command line definitions, or use of the `-z rescan` option. See "Archive Processing" in *Linker and Libraries Guide*.

`ld` is a cross link-editor, able to link 32–bit objects or 64–bit objects, for SPARC or x86 targets. `ld` uses the ELF class and machine type of the first relocatable object on the command line to govern the mode in which to operate. The mixing of 32–bit objects and 64–bit objects is not permitted. Similarly, only objects of a single machine type are allowed. See the `-64` and `-z target` options, and the `LD_NOEXEC_64` environment variable.

Static Executables

The creation of static executables has been discouraged for many releases. In fact, 64–bit system archive libraries have never been provided. Because a static executable is built against system archive libraries, the executable contains system implementation details. This self-containment has a number of drawbacks.

- The executable is immune to the benefits of system patches delivered as shared objects. The executable therefore, must be rebuilt to take advantage of many system improvements.
- The ability of the executable to run on future releases can be compromised.
- The duplication of system implementation details negatively affects system performance.

With Solaris 10, 32–bit system archive libraries are no longer provided. Without these libraries, specifically `libc.a`, the creation of static executables is no longer achievable without specialized system knowledge. However, the capability of `ld` to process static linking options, and the processing of archive libraries, remains unchanged.

Options

The following options are supported.

`-64`

Creates a 64–bit object. By default, the class of the object being generated is determined from the first ELF object processed from the command line. This option is useful when creating an object directly with `ld` whose input is solely from an archive library or a `mapfile`. See "The 32–bit link-editor and 64–bit link-editor" in *Linker and Libraries Guide*.

`-a`
In static mode only, produces an executable object file. Undefined references are not permitted. This option is the default behavior for static mode. The `-a` option can not be used with the `-r` option. See `Static Executables` under DESCRIPTION.

`-b`
In dynamic mode only, provides no special processing for dynamic executable relocations that reference symbols in shared objects. Without the `-b` option, the link-editor applies techniques within a dynamic executable so that the text segment can remain read-only. One technique is the creation of special position-independent relocations for references to functions that are defined in shared objects. Another technique arranges for data objects that are defined in shared objects to be copied into the memory image of an executable at runtime.

The `-b` option is intended for specialized dynamic objects and is not recommended for general use. Its use suppresses all specialized processing required to ensure an object's shareability, and can even prevent the relocation of 64–bit executables.

`-B direct | nodirect`
These options govern direct binding. `-B direct` establishes direct binding information by recording the relationship between each symbol reference together with the dependency that provides the definition. In addition, direct binding information is established between each symbol reference and an associated definition within the object being created. The runtime linker uses this information to search directly for a symbol in the associated object rather than to carry out a default symbol search.

Direct binding information can only be established to dependencies specified with the link-edit. Thus, you should use the `-z defs` option. Objects that wish to interpose on symbols in a direct binding environment should identify themselves as interposers with the `-z interpose` option. The use of `-B direct` enables `-z lazyload` for all dependencies.

The `-B nodirect` option prevents any direct binding to the interfaces offered by the object being created. The object being created can continue to directly bind to external interfaces by specifying the `-z direct` option. See Appendix D, “Direct Bindings,” in *Linker and Libraries Guide*.

`-B dynamic | static`
Options governing library inclusion. `-B dynamic` is valid in dynamic mode only. These options can be specified any number of times on the command line as toggles: if the `-B static` option is given, no shared objects are accepted until `-B dynamic` is seen. See the `-l` option.

`-B eliminate`
Causes any global symbols, not assigned to a version definition, to be eliminated from the symbol table. Version definitions can be supplied by means of a `mapfile` to indicate the global symbols that should remain visible in the generated object. This option achieves the same symbol elimination as the *auto-elimination* directive that is available as part of a

mapfile version definition. This option can be useful when combining versioned and non-versioned relocatable objects. See also the -B local option and the -B reduce option. See "Defining Additional Symbols with a mapfile" in *Linker and Libraries Guide*.

-B group
Establishes a shared object and its dependencies as a group. Objects within the group are bound to other members of the group at runtime. This mode is similar to adding the object to the process by using dlopen(3C) with the RTLD_GROUP mode. An object that has an explicit dependency on a object identified as a group, becomes a member of the group.

As the group must be self contained, use of the -B group option also asserts the -z defs option.

-B local
Causes any global symbols, not assigned to a version definition, to be reduced to local. Version definitions can be supplied by means of a mapfile to indicate the global symbols that should remain visible in the generated object. This option achieves the same symbol reduction as the *auto-reduction* directive that is available as part of a mapfile version definition. This option can be useful when combining versioned and non-versioned relocatable objects. See also the -B eliminate option and the -B reduce option. See "Defining Additional Symbols with a mapfile" in *Linker and Libraries Guide*.

-B reduce
When generating a relocatable object, causes the reduction of symbolic information defined by any version definitions. Version definitions can be supplied by means of a mapfile to indicate the global symbols that should remain visible in the generated object. By default, when a relocatable object is generated, version definitions are only recorded in the output image. The actual reduction of symbolic information is carried out when the object is used in the construction of a dynamic executable or shared object. The -B reduce option is applied automatically when a dynamic executable or shared object is created.

-B symbolic
In dynamic mode only. When building a shared object, binds references to global symbols to their definitions, if available, within the object. Normally, references to global symbols within shared objects are not bound until runtime, even if definitions are available. This model allows definitions of the same symbol in an executable or other shared object to override the object's own definition. ld issues warnings for undefined symbols unless -z defs overrides.

The -B symbolic option is intended for specialized dynamic objects and is not recommended for general use. To reduce the runtime relocation processing that is required an object, the creation of a version definition is recommended.

-c *name*
Records the configuration file *name* for use at runtime. Configuration files can be employed to alter default search paths, provide a directory cache, together with providing alternative object dependencies. See crle(1).

`-C`
Demangles C++ symbol names displayed in diagnostic messages.

`-d y | n`
When `-d y`, the default, is specified, `ld` uses dynamic linking. When `-d n` is specified, `ld` uses static linking. See `Static Executables` under DESCRIPTION, and `-B dynamic|static`.

`-D` *token*,...
Prints debugging information as specified by each *token*, to the standard error. The special token `help` indicates the full list of tokens available. See "Debugging Aids" in *Linker and Libraries Guide*.

`-e` *epsym*
`--entry` *epsym*
Sets the entry point address for the output file to be the symbol *epsym*.

`-f` *name*
`--auxiliary` *name*
Useful only when building a shared object. Specifies that the symbol table of the shared object is used as an auxiliary filter on the symbol table of the shared object specified by *name*. Multiple instances of this option are allowed. This option can not be combined with the `-F` option. See "Generating Auxiliary Filters" in *Linker and Libraries Guide*.

`-F` *name*
`--filter` *name*
Useful only when building a shared object. Specifies that the symbol table of the shared object is used as a filter on the symbol table of the shared object specified by *name*. Multiple instances of this option are allowed. This option can not be combined with the `-f` option. See "Generating Standard Filters" in *Linker and Libraries Guide*.

`-G`
`-shared`
In dynamic mode only, produces a shared object. Undefined symbols are allowed. See Chapter 4, "Shared Objects," in *Linker and Libraries Guide*.

`-h` *name*
`--soname` *name*
In dynamic mode only, when building a shared object, records *name* in the object's dynamic section. *name* is recorded in any dynamic objects that are linked with this object rather than the object's file system name. Accordingly, *name* is used by the runtime linker as the name of the shared object to search for at runtime. See "Recording a Shared Object Name" in *Linker and Libraries Guide*.

`-i`
Ignores `LD_LIBRARY_PATH`. This option is useful when an `LD_LIBRARY_PATH` setting is in effect to influence the runtime library search, which would interfere with the link-editing being performed.

`-I` *name*
`--dynamic-linker` *name*
When building an executable, uses *name* as the path name of the interpreter to be written into the program header. The default in static mode is no interpreter. In dynamic mode, the default is the name of the runtime linker, `ld.so.1`(1). Either case can be overridden by `-I` *name*. `exec`(2) loads this interpreter when the `a.out` is loaded, and passes control to the interpreter rather than to the `a.out` directly.

`-l` *x*
`--library` *x*
Searches a library `lib`*x*`.so` or `lib`*x*`.a`, the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the `-B static` option is in effect, `ld` searches each directory specified in the library search path for a `lib`*x*`.so` or `lib`*x*`.a` file. The directory search stops at the first directory containing either. `ld` chooses the file ending in `.so` if `-l`*x* expands to two files with names of the form `lib`*x*`.so` and `lib`*x*`.a`. If no `lib`*x*`.so` is found, then `ld` accepts `lib`*x*`.a`. In static mode, or when the `-B static` option is in effect, `ld` selects only the file ending in `.a`. `ld` searches a library when the library is encountered, so the placement of `-l` is significant. See "Linking With Additional Libraries" in *Linker and Libraries Guide*.

`-L` *path*
`--library-path` *path*
Adds *path* to the library search directories. `ld` searches for libraries first in any directories specified by the `-L` options and then in the standard directories. This option is useful only if the option precedes the `-l` options to which the `-L` option applies. See "Directories Searched by the Link-Editor" in *Linker and Libraries Guide*.

The environment variable `LD_LIBRARY_PATH` can be used to supplement the library search path, however the `-L` option is recommended, as the environment variable is also interpreted by the runtime environment. See `LD_LIBRARY_PATH` under ENVIRONMENT VARIABLES.

`-m`
Produces a memory map or listing of the input/output sections, together with any non-fatal multiply-defined symbols, on the standard output.

`-M` *mapfile*
`--version-script` *mapfile*
Reads *mapfile* as a text file of directives to `ld`. This option can be specified multiple times. If *mapfile* is a directory, then all regular files, as defined by `stat`(2), within the directory are processed. See Chapter 9, "Mapfile Option," in *Linker and Libraries Guide*. Example mapfiles are provided in `/usr/lib/ld`. See FILES.

`-N` *string*
This option causes a `DT_NEEDED` entry to be added to the `.dynamic` section of the object being built. The value of the `DT_NEEDED` string is the *string* that is specified on the command line. This option is position dependent, and the `DT_NEEDED .dynamic` entry is relative to the

other dynamic dependencies discovered on the link-edit line. This option is useful for specifying dependencies within device driver relocatable objects when combined with the `-dy` and `-r` options.

`-o` *outfile*
`--output` *outfile*
Produces an output object file that is named *outfile*. The name of the default object file is `a.out`.

`-p` *auditlib*
Identifies an audit library, *auditlib*. This audit library is used to audit the object being created at runtime. A shared object identified as requiring auditing with the `-p` option, has this requirement inherited by any object that specifies the shared object as a dependency. See the `-P` option. See "Runtime Linker Auditing Interface" in *Linker and Libraries Guide*.

`-P` *auditlib*
Identifies an audit library, *auditlib*. This audit library is used to audit the dependencies of the object being created at runtime. Dependency auditing can also be inherited from dependencies that are identified as requiring auditing. See the `-p` option, and the `-z globalaudit` option. See "Runtime Linker Auditing Interface" in *Linker and Libraries Guide*.

`-Q y | n`
Under `-Q y`, an `ident` string is added to the `.comment` section of the output file. This string identifies the version of the `ld` used to create the file. This results in multiple `ld idents` when there have been multiple linking steps, such as when using `ld -r`. This identification is identical with the default action of the `cc` command. `-Q n` suppresses version identification. `.comment` sections can be manipulated by the `mcs`(1) utility.

`-r`
`--relocatable`
Combines relocatable object files to produce one relocatable object file. `ld` does not complain about unresolved references. This option cannot be used with the `-a` option.

`-R` *path*
`-rpath` *path*
A colon-separated list of directories used to specify library search directories to the runtime linker. If present and not NULL, the path is recorded in the output object file and passed to the runtime linker. Multiple instances of this option are concatenated together with each *path* separated by a colon. See "Directories Searched by the Runtime Linker" in *Linker and Libraries Guide*.

The use of a runpath within an associated object is preferable to setting global search paths such as through the `LD_LIBRARY_PATH` environment variable. Only the runpaths that are necessary to find the objects dependencies should be recorded. `ldd`(1) can also be used to discover unused runpaths in dynamic objects, when used with the `-U` option.

Various tokens can also be supplied with a runpath that provide a flexible means of identifying system capabilities or an objects location. See Appendix C, "Establishing Dependencies with Dynamic String Tokens," in *Linker and Libraries Guide*. The `$ORIGIN` token is especially useful in allowing dynamic objects to be relocated to different locations in the file system.

`-s`
`--strip-all`
Strips symbolic information from the output file. Any debugging information, that is, `.line`, `.debug*`, and `.stab*` sections, and their associated relocation entries are removed. Except for relocatable files, a symbol table `SHT_SYMTAB` and its associated string table section are not created in the output object file. The elimination of a `SHT_SYMTAB` symbol table can reduce the `.stab*` debugging information that is generated using the compiler drivers `-g` option. See the `-z redlocsym` and `-z noldynsym` options.

`-S` *supportlib*
The shared object *supportlib* is loaded with `ld` and given information regarding the linking process. Shared objects that are defined by using the `-S` option can also be supplied using the `SGS_SUPPORT` environment variable. See "Link-Editor Support Interface" in *Linker and Libraries Guide*.

`-t`
Turns off the warning for multiply-defined symbols that have different sizes or different alignments.

`-u` *symname*
`--undefined` *symname*
Enters *symname* as an undefined symbol in the symbol table. This option is useful for loading entirely from an archive library. In this instance, an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant. This option must be placed before the library that defines the symbol. See "Defining Additional Symbols with the u option" in *Linker and Libraries Guide*.

`-V`
`--version`
Outputs a message giving information about the version of `ld` being used.

`-Y P,`*dirlist*
Changes the default directories used for finding libraries. *dirlist* is a colon-separated path list.

`-z absexec`
Useful only when building a dynamic executable. Specifies that references to external absolute symbols should be resolved immediately instead of being left for resolution at runtime. In very specialized circumstances, this option removes text relocations that can result in excessive swap space demands by an executable.

`-z allextract | defaultextract | weakextract`
Alters the extraction criteria of objects from any archives that follow. By default, archive members are extracted to satisfy undefined references and to promote tentative definitions with data definitions. Weak symbol references do not trigger extraction. Under `-z allextract`, all archive members are extracted from the archive. Under `-z weakextract`, weak references trigger archive extraction. `-z defaultextract` provides a means of returning to the default following use of the former extract options. See "Archive Processing" in *Linker and Libraries Guide*.

`-z altexec64`
Execute the 64–bit `ld`. The creation of very large 32–bit objects can exhaust the virtual memory that is available to the 32–bit `ld`. The `-z altexec64` option can be used to force the use of the associated 64–bit `ld`. The 64–bit `ld` provides a larger virtual address space for building 32–bit objects. See "The 32–bit link-editor and 64–bit link-editor" in *Linker and Libraries Guide*.

`-z combreloc | nocombreloc`
By default, `ld` combines multiple relocation sections when building executables or shared objects. This section combination differs from relocatable objects, in which relocation sections are maintained in a one-to-one relationship with the sections to which the relocations must be applied. The `-z nocombreloc` option disables this merging of relocation sections, and preserves the one-to-one relationship found in the original relocatable objects.

`ld` sorts the entries of data relocation sections by their symbol reference. This sorting reduces runtime symbol lookup. When multiple relocation sections are combined, this sorting produces the least possible relocation overhead when objects are loaded into memory, and speeds the runtime loading of dynamic objects.

Historically, the individual relocation sections were carried over to any executable or shared object, and the `-z combreloc` option was required to enable the relocation section merging previously described. Relocation section merging is now the default. The `-z combreloc` option is still accepted for the benefit of old build environments, but the option is unnecessary, and has no effect.

`-z defs | nodefs`
`--no-undefined`
The `-z defs` option and the `--no-undefined` option force a fatal error if any undefined symbols remain at the end of the link. This mode is the default when an executable is built. For historic reasons, this mode is *not* the default when building a shared object. Use of the `-z defs` option is recommended, as this mode assures the object being built is self-contained. A self-contained object has all symbolic references resolved internally, or to the object's immediate dependencies.

The `-z nodefs` option allows undefined symbols. For historic reasons, this mode is the default when a shared object is built. When used with executables, the behavior of references to such undefined symbols is unspecified. Use of the `-z nodefs` option is not recommended.

`-z direct | nodirect`
Enables or disables direct binding to any dependencies that follow on the command line. These options allow finer control over direct binding than the global counterpart `-B direct`. The `-z direct` option also differs from the `-B direct` option in the following areas. Direct binding information is not established between a symbol reference and an associated definition within the object being created. Lazy loading is not enabled.

`-z endfiltee`
Marks a filtee so that when processed by a filter, the filtee terminates any further filtee searches by the filter. See "Reducing Filtee Searches" in *Linker and Libraries Guide*.

`-z finiarray=`*function*
Appends an entry to the `.finiarray` section of the object being built. If no `.finiarray` section is present, a section is created. The new entry is initialized to point to *function*. See "Initialization and Termination Sections" in *Linker and Libraries Guide*.

`-z globalaudit`
This option supplements an audit library definition that has been recorded with the `-P` option. This option is only meaningful when building a dynamic executable. Audit libraries that are defined within an object with the `-P` option typically allow for the auditing of the immediate dependencies of the object. The `-z globalaudit` promotes the auditor to a global auditor, thus allowing the auditing of all dependencies. See "Invoking the Auditing Interface" in *Linker and Libraries Guide*.

An auditor established with the `-P` option and the `-z globalaudit` option, is equivalent to the auditor being established with the `LD_AUDIT` environment variable. See `ld.so.1`(1).

`-z groupperm | nogroupperm`
Assigns, or deassigns each dependency that follows to a unique group. The assignment of a dependency to a group has the same effect as if the dependency had been built using the `-B group` option.

`-z help`
`--help`
Print a summary of the command line options on the standard output and exit.

`-z ignore | record`
Ignores, or records, dynamic dependencies that are not referenced as part of the link-edit. Ignores, or records, unreferenced ELF sections from the relocatable objects that are read as part of the link-edit. By default, `-z record` is in effect.

If an ELF section is ignored, the section is eliminated from the output file being generated. A section is ignored when three conditions are true. The eliminated section must

contribute to an allocatable segment. The eliminated section must provide no global symbols. No other section from any object that contributes to the link-edit, must reference an eliminated section.

`-z initarray=`*function*
Appends an entry to the `.initarray` section of the object being built. If no `.initarray` section is present, a section is created. The new entry is initialized to point to *function*. See "Initialization and Termination Sections" in *Linker and Libraries Guide*.

`-z initfirst`
Marks the object so that its runtime initialization occurs before the runtime initialization of any other objects brought into the process at the same time. In addition, the object runtime finalization occurs after the runtime finalization of any other objects removed from the process at the same time. This option is only meaningful when building a shared object.

`-z interpose`
Marks the object as an interposer. At runtime, an object is identified as an explicit interposer if the object has been tagged using the `-z interpose` option. An explicit interposer is also established when an object is loaded using the `LD_PRELOAD` environment variable. Implicit interposition can occur because of the load order of objects, however, this implicit interposition is unknown to the runtime linker. Explicit interposition can ensure that interposition takes place regardless of the order in which objects are loaded. Explicit interposition also ensures that the runtime linker searches for symbols in any explicit interposers when direct bindings are in effect.

`-z lazyload` | `nolazyload`
Enables or disables the marking of dynamic dependencies to be lazily loaded. Dynamic dependencies which are marked `lazyload` are not loaded at initial process start-up. These dependencies are delayed until the first binding to the object is made. *Note:* Lazy loading requires the correct declaration of dependencies, together with associated runpaths for each dynamic object used within a process. See "Lazy Loading of Dynamic Dependencies" in *Linker and Libraries Guide*.

`-z ld32=`*arg1*`,`*arg2*`,...`
`-z ld64=`*arg1*`,`*arg2*`,...`
The class of the link-editor is affected by the class of the output file being created and by the capabilities of the underlying operating system. The `-z ld[32|64]` options provide a means of defining any link-editor argument. The defined argument is only interpreted, respectively, by the 32–bit class or 64–bit class of the link-editor.

For example, support libraries are class specific, so the correct class of support library can be ensured using:

```
ld ... -z ld32=-Saudit32.so.1 -z ld64=-Saudit64.so.1 ...
```

The class of link-editor that is invoked is determined from the ELF class of the first relocatable file that is seen on the command line. This determination is carried out *prior* to any `-z ld[32|64]` processing.

`-z loadfltr`
Marks a filter to indicate that filtees must be processed immediately at runtime. Normally, filter processing is delayed until a symbol reference is bound to the filter. The runtime processing of an object that contains this flag mimics that which occurs if the `LD_LOADFLTR` environment variable is in effect. See the `ld.so.1`(1).

`-z muldefs`
`--allow-multiple-definition`
Allows multiple symbol definitions. By default, multiple symbol definitions that occur between relocatable objects result in a fatal error condition. This option, suppresses the error condition, allowing the first symbol definition to be taken.

`-z nocompstrtab`
Disables the compression of ELF string tables. By default, string compression is applied to `SHT_STRTAB` sections, and to `SHT_PROGBITS` sections that have their `SHF_MERGE` and `SHF_STRINGS` section flags set.

`-z nodefaultlib`
Marks the object so that the runtime default library search path, used after any `LD_LIBRARY_PATH` or runpaths, is ignored. This option implies that all dependencies of the object can be satisfied from its runpath.

`-z nodelete`
Marks the object as non-deletable at runtime. This mode is similar to adding the object to the process by using `dlopen`(3C) with the `RTLD_NODELETE` mode.

`-z nodlopen`
Marks the object as not available to `dlopen`(3C), either as the object specified by the `dlopen()`, or as any form of dependency required by the object specified by the `dlopen()`. This option is only meaningful when building a shared object.

`-z nodump`
Marks the object as not available to `dldump`(3C).

`-z noldynsym`
Prevents the inclusion of a `.SUNW_ldynsym` section in dynamic executables or sharable libraries. The `.SUNW_ldynsym` section augments the `.dynsym` section by providing symbols for local functions. Local function symbols allow debuggers to display local function names in stack traces from stripped programs. Similarly, `dladdr`(3C) is able to supply more accurate results.

The `-z noldynsym` option also prevents the inclusion of the two symbol sort sections that are related to the `.SUNW_ldynsym` section. The `.SUNW_dynsymsort` section provides sorted access to regular function and variable symbols. The `.SUNW_dyntlssort` section provides sorted access to thread local storage (TLS) variable symbols.

The `.SUNW_ldynsym`, `.SUNW_dynsymsort`, and `.SUNW_dyntlssort` sections, which becomes part of the allocable text segment of the resulting file, cannot be removed by `strip`(1). Therefore, the `-z noldynsym` option is the only way to prevent their inclusion. See the `-s` and `-z redlocsym` options.

`-z nopartial`
Partially initialized symbols, that are defined within relocatable object files, are expanded in the output file being generated.

`-z noversion`
Does not record any versioning sections. Any version sections or associated `.dynamic` section entries are not generated in the output image.

`-z now`
Marks the object as requiring non-lazy runtime binding. This mode is similar to adding the object to the process by using `dlopen`(3C) with the `RTLD_NOW` mode. This mode is also similar to having the `LD_BIND_NOW` environment variable in effect. See `ld.so.1`(1).

`-z origin`
Marks the object as requiring immediate `$ORIGIN` processing at runtime. This option is only maintained for historic compatibility, as the runtime analysis of objects to provide for `$ORIGIN` processing is now default.

`-z preinitarray=`*function*
Appends an entry to the `.preinitarray` section of the object being built. If no `.preinitarray` section is present, a section is created. The new entry is initialized to point to *function*. See "Initialization and Termination Sections" in *Linker and Libraries Guide*.

`-z redlocsym`
Eliminates all local symbols except for the `SECT` symbols from the symbol table `SHT_SYMTAB`. All relocations that refer to local symbols are updated to refer to the corresponding `SECT` symbol. This option allows specialized objects to greatly reduce their symbol table sizes. Eliminated local symbols can reduce the `.stab*` debugging information that is generated using the compiler drivers `-g` option. See the `-s` and `-z noldynsym` options.

`-z relaxreloc`
`ld` normally issues a fatal error upon encountering a relocation using a symbol that references an eliminated COMDAT section. If `-z relaxreloc` is enabled, `ld` instead redirects such relocations to the equivalent symbol in the COMDAT section that was kept. `-z relaxreloc` is a specialized option, mainly of interest to compiler authors, and is not intended for general use.

`-z rescan`
Rescans the archive files that are provided to the link-edit. By default, archives are processed once as the archives appear on the command line. Archives are traditionally specified at the end of the command line so that their symbol definitions resolve any preceding references. However, specifying archives multiple times to satisfy their own interdependencies, can be necessary.

The `-z rescan` option causes the entire archive list to be reprocessed in an attempt to locate additional archive members that resolve symbol references. This archive rescanning continues until a pass over the archive list occurs in which no new members are extracted.

`-z target=sparc|x86`
Specifies the machine type for the output object. Supported targets are SPARC and x86. The 32–bit machine type for the specified target is used unless the `-64` option is also present, in which case the corresponding 64–bit machine type is used. By default, the machine type of the object being generated is determined from the first ELF object processed from the command line. This option is useful when creating an object directly with `ld` whose input is solely from an archive library or a `mapfile`. See the `-M` option. See "The 32–bit link-editor and 64–bit link-editor" in *Linker and Libraries Guide*.

`-z text`
In dynamic mode only, forces a fatal error if any relocations against non-writable, allocatable sections remain. For historic reasons, this mode is not the default when building an executable or shared object. However, its use is recommended to ensure that the text segment of the dynamic object being built is shareable between multiple running processes. A shared text segment incurs the least relocation overhead when loaded into memory. See "Position-Independent Code" in *Linker and Libraries Guide*.

`-z textoff`
In dynamic mode only, allows relocations against all allocatable sections, including non-writable ones. This mode is the default when building a shared object.

`-z textwarn`
In dynamic mode only, lists a warning if any relocations against non-writable, allocatable sections remain. This mode is the default when building an executable.

`-z verbose`
This option provides additional warning diagnostics during a link-edit. Presently, this option conveys suspicious use of displacement relocations. This option also conveys the restricted use of static TLS relocations when building shared objects. In future, this option might be enhanced to provide additional diagnostics that are deemed too noisy to be generated by default.

Environment Variables

`LD_ALTEXEC`
An alternative link-editor path name. `ld` executes, and passes control to this alternative link-editor. This environment variable provides a generic means of overriding the default link-editor that is called from the various compiler drivers. See the `-z altexec64` option.

`LD_LIBRARY_PATH`
A list of directories in which to search for the libraries specified using the `-l` option. Multiple directories are separated by a colon. In the most general case, this environment variable contains two directory lists separated by a semicolon:

dirlist1;*dirlist2*

If `ld` is called with any number of occurrences of `-L`, as in:

ld ... -L*path1* **... -L***pathn* **...**

then the search path ordering is:

dirlist1 path1 **...** *pathn dirlist2* **LIBPATH**

When the list of directories does not contain a semicolon, the list is interpreted as *dirlist2*.

The LD_LIBRARY_PATH environment variable also affects the runtime linkers search for dynamic dependencies.

This environment variable can be specified with a _32 or _64 suffix. This makes the environment variable specific, respectively, to 32–bit or 64–bit processes and overrides any non-suffixed version of the environment variable that is in effect.

LD_NOEXEC_64
: Suppresses the automatic execution of the 64–bit link-editor. By default, the link-editor executes the 64–bit version when the ELF class of the first relocatable file identifies a 64–bit object. The 64–bit image that a 32–bit link-editor can create, has some limitations. However, some link-edits might find the use of the 32–bit link-editor faster.

LD_OPTIONS
: A default set of options to ld. LD_OPTIONS is interpreted by ld just as though its value had been placed on the command line, immediately following the name used to invoke ld, as in:

ld $LD_OPTIONS ... *other-arguments* **...**

LD_RUN_PATH
: An alternative mechanism for specifying a runpath to the link-editor. See the -R option. If both LD_RUN_PATH and the -R option are specified, -R supersedes.

SGS_SUPPORT
: Provides a colon-separated list of shared objects that are loaded with the link-editor and given information regarding the linking process. This environment variable can be specified with a _32 or _64 suffix. This makes the environment variable specific, respectively, to the 32–bit or 64–bit class of ld and overrides any non-suffixed version of the environment variable that is in effect. See the -S option.

Notice that environment variable-names that begin with the characters 'LD_' are reserved for possible future enhancements to ld and ld.so.1(1).

Files

lib*x*.so	shared object libraries.
lib*x*.a	archive libraries.
a.out	default output file.
LIBPATH	For 32–bit libraries, the default search path is /usr/ccs/lib, followed by /lib, and finally /usr/lib. For 64–bit libraries, the default search path is /lib/64, followed by /usr/lib/64.

`/usr/lib/ld` A directory containing several `mapfiles` that can be used during link-editing. These `mapfiles` provide various capabilities, such as defining memory layouts, aligning bss, and defining non-executable stacks.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtoo
Interface Stability	Committed

See Also `as`(1), `crle`(1), `gprof`(1), `ld.so.1`(1), `ldd`(1), `mcs`(1), `pvs`(1), `exec`(2), `stat`(2), `dlopen`(3C), `dldump`(3C), `elf`(3ELF), `ar.h`(3HEAD), `a.out`(4), `attributes`(5)

Linker and Libraries Guide

Notes Default options applied by `ld` are maintained for historic reasons. In today's programming environment, where dynamic objects dominate, alternative defaults would often make more sense. However, historic defaults must be maintained to ensure compatibility with existing program development environments. Historic defaults are called out wherever possible in this manual. For a description of the current recommended options, see Appendix A, "Link-Editor Quick Reference," in *Linker and Libraries Guide*.

If the file being created by `ld` already exists, the file is unlinked after all input files have been processed. A new file with the specified name is then created. This allows `ld` to create a new version of the file, while simultaneously allowing existing processes that are accessing the old file contents to continue running. If the old file has no other links, the disk space of the removed file is freed when the last process referencing the file terminates.

The behavior of `ld` when the file being created already exists was changed with `SXCE` build `43`. In older versions, the existing file was rewritten in place, an approach with the potential to corrupt any running processes that is using the file. This change has an implication for output files that have multiple hard links in the file system. Previously, all links would remain intact, with all links accessing the new file contents. The new `ld` behavior *breaks* such links, with the result that only the specified output file name references the new file. All the other links continue to reference the old file. To ensure consistent behavior, applications that rely on multiple hard links to linker output files should explicitly remove and relink the other file names.