Incremental Link Editor (`ild`)

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-5693-10
Revision A, December 1996

Contents

Preface

This manual describes the incremental link editor (`ild`). `ild` replaces link editor `ld` for incremental linking. After the initial link, `ild` links can be much faster than links completed by `ld`.

`ild` runs under the Solaris™ 2.x operating environment.

Related Documentation

The following man pages contain additional information.

`cc (1)`, `CC (1)`, `ild (1)`, `ld (1)`, `exec (2)`, `elf (3E)`, `end (3C)`, `a.out (4)`, `ar (4)`, `f77 (1)`, and `f90(1)`.

Information about `ld` can be found in Solaris *Linker and Libraries Guide.*

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Incremental Link Editor (`ild`)

This manual describes incremental linking, `ild`-specific features, example messages, and `ild` options. This document is organized into the following sections:

Introduction

`ild` is an incremental version of the Link Editor `ld`, and replaces `ld` for linking programs. `ild` allows you to complete the development sequence (the edit, compile, link, and debug loop) efficiently and more quickly than by using a standard linker. You can avoid relinking entirely by using *Fix and Continue*. *Fix and Continue* (a part of `dbx`) allows you to work without relinking, but if you need to relink, the process can be faster if you use `ild`.

`ild` links incrementally so you can insert modified object code into an executable file that you created earlier, without relinking unmodified object files. The time required to relink depends upon the amount of code modified. Linking your application on every build does not require the same amount of time; small changes in code can be relinked very quickly.

On the initial link, `ild` requires about the same amount of time that `ld` requires, but subsequent `ild` links can be much faster than an `ld` link. The cost of the reduced link time is an increase in the size of the executable.

Overview of Incremental Linking

When `ild` is used in place of `ld`, the initial link causes the various text, data, bss, exception table sections, etc., to be padded with additional space for future expansion (see Figure 1 on page 3). Additionally, all relocation records and the global symbol table are saved into a new persistent state region in the executable file. On subsequent incremental links, `ild` uses timestamps to discover which object files have changed and patches the changed object code into a previously built executable. That is, previous versions of the object files are invalidated and the new object files are loaded into the space vacated, or into the pad sections of the executable when needed. All references to symbols in invalidated object files are patched to point to the correct new object files.

`ild` does not support all `ld` command options. If `ild` is passed a command option that it does not support (see ***"Notes" on page 16***), `ild` directly invokes `/usr/ccs/bin/ld` to perform the link.

How to Use `ild`

`ild` is invoked automatically by the compilation system in place of `ld` under certain conditions. When you invoke a compilation system, you are invoking a compiler driver. When you pass certain options to the driver, the driver uses `ild`. The compiler driver reads the options from the command line and executes various programs in the correct order and adds files from the list of arguments that are passed.

For example, `cc` first runs `acomp` (the front-end of the compiler), then `acomp` runs the optimizing code generator, then `cc` does the same thing for the other source files listed on the command line. The driver then generates a call to either `ild` or `ld`, depending on the options, passing it all of the files just compiled, plus other files and libraries needed to make the program complete.

Figure 1 shows an example of incremental linking.

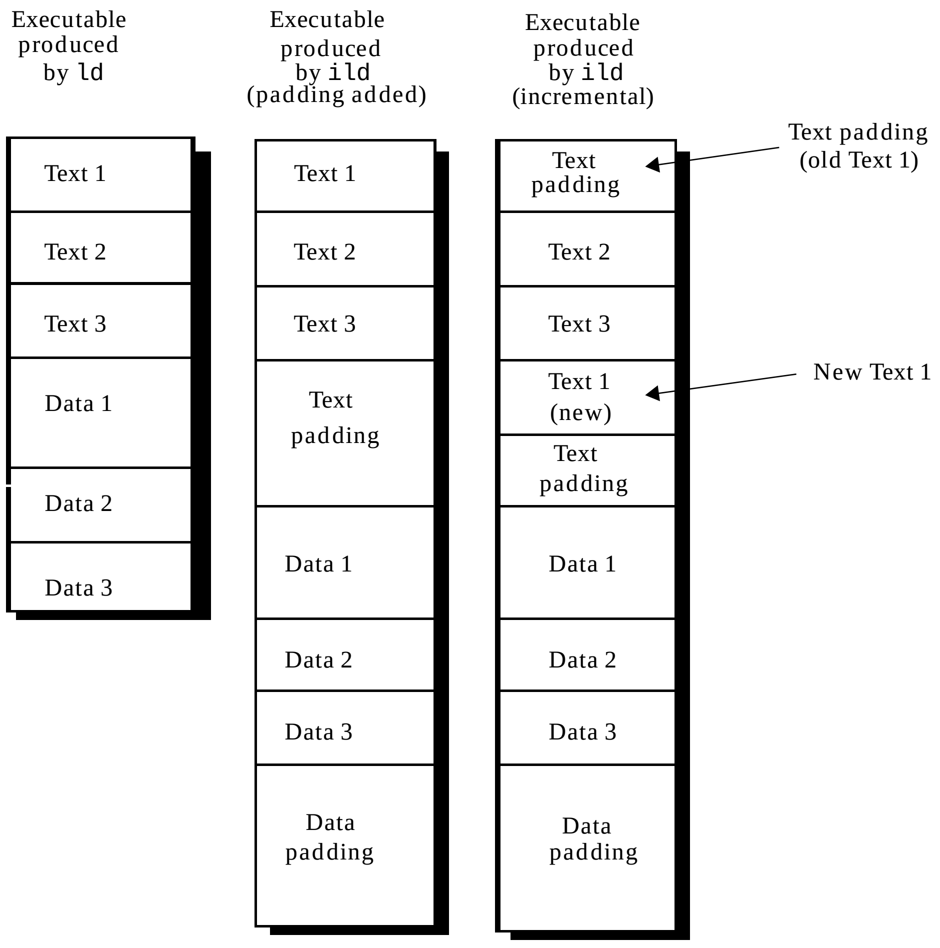

Figure 1   Example of Incremental Linking

The compilation system options that control whether a link step is performed by `ild` or `ld` are listed here:

- **`-xildon`** — Always use `ild`
- **`-xildoff`** — Always use `ld`

Note – If `-xildon` and `-xildoff` are both present, the last governs.

- **`-g`** — When neither `-xildoff` or `-G` are given, use `ild` for link-only invocations (no source files on the command line)
- **`-G`** — Prevents the `-g` option from having any effect on linker selection

Note – Both `-g` and `-G` have other meanings which are documented as part of the compilation systems.

When you use the `-g` option to invoke debugging, and you have the default Makefile structure (which includes compile-time options such as `-g` on the link command line), you use `ild` automatically when doing development.

How `ild` *Works*

On an initial link, `ild` saves information about:

- All of the object files looked at.
- The symbol table for the executable produced.
- All symbolic references not resolved at compile time.

Initial `ild` links take about as much time as an `ld` link.

On incremental links, `ild`:

- Determines which files have changed.
- Relinks the modified object files.
- Uses stored information to modify changed symbolic references in the rest of the program.

Incremental `ild` links are much faster than `ld` links.

In general, you do one initial link and all subsequent links are incremental.

For example, `ild` saves a list of all places where symbol `foo` is referenced in your code. If you do an incremental link that changes the value of `foo`, `ild` must change the value of all references to `foo`.

`ild` spreads out the components of the program and each section of the executable has padding added to it. Padding makes the executable modules larger than when they were linked by `ld`. As object files increase in size during successive incremental links, the padding can become exhausted. If this occurs, `ild` displays a message and does a complete full relink of the executable.

For example, as shown in Figure 1 on page 3, each of the three columns shows the sequence of text and data in a linked executable program. The left column shows text and data in an executable linked by `ld`. The center column shows the addition of text and data padding in an executable linked by `ild`. Assume that a change is made to the source file for Text 1 that causes the Text section to grow without affecting the size of the other sections. The right column shows that the original location of Text 1 has been replaced by Text padding (Text 1 has been invalidated). Text 1 has been moved to occupy a portion of the Text padding space.

To produce a smaller nonincremental executable, run the compiler driver (for example, `cc` or `CC`) with the `-xildoff` option, and `ld` is invoked to produce a more compact executable.

The resulting executable from `ild` can be debugged by `dbx` because `dbx`/Debugger understands the padding that `ild` inserts between programs.

For any command-line option that `ild` does not understand, `ild` invokes `ld`. `ild` is compatible with `ld` (in `/usr/ccs/bin/ld`). See “Options” on page 10, for details.

There are no special or extra files used by `ild`.

What `ild` *Cannot Do*

When `ild` is invoked to create shared objects, `ild` invokes `ld` to do the link.

Performance of `ild` may suffer greatly if you change a high percentage of object files. `ild` automatically does an full relink when it detects that a high percentage of files have been changed.

Do not use `ild` to produce the final production code for shipment. `ild` makes the file larger because parts of the program have been spread out due to padding. Because of the padding and additional time required to link, it is recommended that you do not use the `-xildon` option for production code. (Use `-xildoff` on the link line if `-g` is present.)

`ild` may not link small programs much faster, and the increase in size of the executable is greater than that for larger programs.

Third-party tools that work on executables may have unexpected results on `ild`-produced binaries.

Any program that modifies an executable, for example `strip` or `mcs`, might affect the ability of `ild` to perform an incremental link. When this happens, `ild` issues a message and performs a full relink. See *"Reasons for Full Relinks,* "Example 2: running strip" on page 8.

Reasons for Full Relinks

`ild` *Deferred-Link Messages*

The message '`ild: calling ld to finish link' . . .` means that `ild` cannot complete the link, and is deferring the link request to `ld` for completion.

By default, these messages are displayed as needed. You can suppress these messages by using the `-z i_quiet` option.

`ild: calling ld to finish link -- cannot handle shared libraries in archive` *library name*

This message is suppressed if `ild` is implicitly requested (`-g`), but is displayed if `-xildon` is on the command line. This message is displayed in all cases if you use the `-z i_verbose` option, and never displayed if you use the `-z i_quiet` option.

`ild: calling ld to finish link -- cannot handle keyword` *Keyword*

`ild: calling ld to finish link -- cannot handle -d` *Keyword*

`ild: calling ld to finish link -- cannot handle -z` *keyword*

`ild: calling ld to finish link -- cannot handle argument` *keyword*

`ild` *Relink Messages*

The message ‘`ild: (Performing full relink)`’ . . . means that for some reason `ild` cannot do an incremental link and must do a full relink. This is not an error. It is to inform you that this link will take longer than an incremental link (see “How ild Works” on page 4, for more details). `ild` messages can be controlled by `ild` options `-z i_quiet` and `-z i_verbose`. Some messages have a verbose mode with more descriptive text.

You can suppress all of these messages by using the `ild` option `-z i_quiet.` If the default message has a verbose mode, the message ends with an ellipsis (`[...]`) indicating more information is available. You can view the additional information by using the `-z i_verbose` option. Example messages are shown with the `-z i_verbose` option selected.

Example 1: internal free space exhausted

The most common of the full relink messages is the `internal free space exhausted` message:

```
$ cat test1.c
int main() { return 0; }
$ rm a.out
$ cc -xildon -c -g test1.c                      # This creates test1.o
$ cc -xildon -z i_verbose -g test1.o            # This creates a.out with
                                                # minimal debugging information.
$ cc -xildon -z i_verbose -g test1.c            # A one-line compile and link
                                                # puts all debugging information
                                                # into a.out.
ild: (Performing full relink) internal free
space in output file exhausted (sections)
$
```

These commands show that going from a one-line compile to a two-line compile causes debugging information to grow in the executable. This growth causes `ild` to run out of space and do an full relink.

Example 2: running `strip`

Another problem arises when you run `strip`. Continuing from Example 1:

# Strip a.out

# Try to do an incremental link

```
$ strip a.out
$ cc -xildon -z i_verbose -g test1.c

ild: (Performing full relink) a.out has been
altered since the last incremental link --
maybe you ran strip or mcs on it?
$
```

Example 3: `ild` *version*

When a new version of `ild` is run on an executable created by an older version of `ild`, you see the following error message:

# Assume old_executablewas created by an earlier version of ild

```
$ cc -xildon -z i_verbose foo.o -o old_executable


ild: (Performing full relink) an updated ild
has been installed since a.out was last linked
(2/16)
```

Note – The numbers (2/16) are used only for internal reporting.

Example 4: too many files changed

Sometimes `ild` determines that it will be faster to do a full relink than an incremental link. For example:

```
$ rm a.out
$ cc -xildon -z i_verbose \
     x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
$ touch x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o
$ cc -xildon -z i_verbose \
     x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
ild: (Performing full relink) too many files changed
```

Here, use of the `touch` command causes `ild` to determine that files `x0.o` through `x8.o` have changed and that a full relink will be faster than incrementally relinking all nine object files.

Example 5: full relink

There are certain conditions that can cause a full relink on the next link, as compared to the previous examples that cause a full relink on this link.

The next time you try to link that program, you see the message:

# ild detects previous error and does a full relink

```
$ cc -xildon -z i_verbose broken.o
ild: (Performing full relink) cannot do incremental relink due to
problems in the previous link
```

A full relink occurs.

Example 6: new working directory

```
                                   % cd /tmp
                                   % cat y.c
                                       int main(){ return 0;}
                                   % cc -c y.c
                                   % rm -f a.out
# initial link with cwd equal      % cc -xildon -z i_verbose y.o -o a.out
to /tmp
                                   % mkdir junk
                                   % mv y.o y.c a.out junk
                                   % cd junk
# incremental link, cwd is         % cc -xildon -z i_verbose y.o -o a.out
now /tmp/junk
                                   ild: (Performing full relink) current directory has changed
                                   from '/tmp' to '/tmp/junk'
                                   %
```

Options

Linker control options directly accepted by the compilation system and linker options that may be passed through the compilation system to `ild` are described in this section.

Options Accepted by the Compilation System

These are linker control options accepted by the compilation system:

`-i`

Ignores `LD_LIBRARY_PATH` setting. When an `LD_LIBRARY_PATH` setting is in effect, this option is useful to influence the runtime library search, which interferes with the link editing being performed.

`-s`

Strips symbolic information from the output file. Any debugging information and associated relocation entries are removed. Except for relocatable files or shared objects, the symbol table and string table sections are also removed from the output object file.

`-V`

Output a message about the version of `ild` being used.

`-B` *`dynamic`* `|` *`static`*

Options governing library inclusion. Option `-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 option “-l x (space is optional)” on page 11.

`-g`

The compilation systems invoke `ild` in place of `ld` when the `-g` option (output debugging information) is given, unless any of the following are true:

- The `-G` option (produce a shared library) is given
- The `-xildoff` option is present
- Any source files are named on the command line

`-d` *`y`* `|` *`n`*

When `-d`*y* (the default) is specified, `ild` uses dynamic linking; when `-d`*n* is specified, `ild` uses static linking. See option “-B dynamic | static” on page 11.

`-L` *`path`* (space is optional)

Adds *path* to the library search directories. `ild` searches for libraries first in any directories specified by the `-L` options, and then in the standard directories. This option is useful only if it precedes the `-l` options to which it applies on the command line. The environment variable `LD_LIBRARY_PATH` can be used to supplement the library search path (see “LD_LIBRARY_PATH” on page 15).

`-l` *`x`* (space is optional)

Searches a library `lib`x `.so` or `libx .a`, the conventional names for shared object and archive libraries, respectively. In dynamic mode, unless the `-B`*static* option is in effect, `ild` searches each directory specified in the library search path for a file `libx .so` or `libx .a`. The directory search stops at the first directory containing either `ild` chooses the file ending in `.so` if `-l` expands to two files whose names are of the form `libx .so` and `libx .a`.

If no `libx .so` is found, then `ild` accepts `libx .a`. In static mode, or when the `-B`*static* option is in effect, `ild` selects only the file ending in `.a`. A library is searched when its name is encountered, so the placement of `-l` is significant.

`-o` *`outfile`*

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

`-Q` *`y`* `|` *`n`*

Under `-Q`*y*, an *ident* string is added to the `.comment` section of the output file to identify the version of the link editor 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 is identical with the default action of the `cc` command. Option `-Q`*n* suppresses version identification.

`-R` *`path`* (space is optional)

This option gives a colon-separated list of directories that specifies library search directories to the runtime linker. If present and not null, *path* is recorded in the output object file and passed to the runtime linker. Multiple instances of this option are concatenated and separated by a colon.

`-xildoff`

Incremental linker off. Force the use of bundled `ld`. This is the default if `-g` is not being used, or `-G` is being used. You can override this default with `-xildon`.

`-xildon`

Incremental linker. Force the use of `ild` in incremental mode. This is the default if `-g` is being used. You can override this default with `-xildoff`.

`-Y P,`*`dirlist`* (space is optional)

(cc only) Changes the default directories used for finding libraries. Option *dirlist* is a colon-separated path list.

Note – The "`-z` *name*" form is used by `ild` for special options. The *i_* prefix to the `-z` options identifies those options peculiar to `ild`.

`-z defs`

Forces a fatal error if any undefined symbols remain at the end of the link. This is the default when building an executable. It is also useful when building a shared object to assure that the object is self-contained, that is, that all its symbolic references are resolved internally.

`-z i_dryrun`

(`ild` only.) Prints the list of files that would be linked by `ild` and exits.

`-z i_full`

(`ild` only.) Does a complete relink in incremental mode.

`-z i_noincr`

(`ild` only.) Runs `ild` in nonincremental mode (not recommended for customer use — used for testing only).

`-z i_quiet`

(`ild` only) Turns off all `ild` relink messages.

`-z i_verbose`

(`ild` only) Expands on default information on some `ild` relink messages.

`-z nodefs`

Allows undefined symbols. This is the default when building a shared object. When used with executables, the behavior of references to such “undefined symbols” is unspecified.

`-z weakextract`

A weak reference to a symbol will cause a file defining that sysmbol to be extracted from a static library.

Options Passed to `ild` *from the Compilation System*

The following options are accepted by ild, but you must use the form:

`-Wl,`*arg,arg* (for cc), or `-Qoption ld` *arg*`,`*arg* (for others),

to pass them to `ild` via the compilation system

`-a`

In static mode only, produces an executable object file; gives errors for undefined references. This is the default behavior for static mode. Option `-a` cannot be used with the `-r` option.

`-m`

Produces a memory map or listing of the input/output sections on the standard output.

`-t`

Turn off the warning about symbols that are defined more than once and that are not the same size.

`-e` *`epsym`*

Sets the entry point address for the output file to be that of the symbol *epsym*.

`-I` *`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, `/usr/lib/ld.so.1`. Either case can be overridden by `-I` *name*. The `exec` system call loads this interpreter when it loads the `a.out` and passes control to the interpreter rather than to the `a.out` directly.

`-u` *`symname`*

Enters *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from an archive library because, initially, the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the command line is significant; it must be placed before the library that defines the symbol.

Environment

`LD_LIBRARY_PATH`

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

dirlist1; *dirlist2*

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

`ild` ...-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, it is interpreted as *dirlist2*.

`LD_LIBRARY_PATH` is also used to specify library search directories to the runtime linker. That is, if `LD_LIBRARY_PATH` exists in the environment, the runtime linker searches the directories named in it, before its default directory, for shared objects to be linked with the program at execution.

Note – When running a `set-user-ID` or `set-group-ID` program, the runtime linker searches only for libraries in `/usr/lib`. It also searches for any full pathname specified within the executable. A full pathname is the result of a runpath being specified when the executable was constructed. Any library dependencies specified as relative pathnames are silently ignored.

`LD_OPTIONS`

A default set of options to `ild`. `LD_OPTIONS` is interpreted by `ild` as though its value had been placed on the command line immediately following the name used to invoke `ild`, as in:
`ild $LD_OPTIONS` ... *other-arguments* ...

`LD_PRELOAD`

A list of shared objects that are to be interpreted by the runtime linker. The specified shared objects are linked in after the program being executed and before any other shared objects that the program references.

Note – When running a `set-user-ID` or `set-group-ID` program, this option is silently ignored.

`LD_RUN_PATH`

An alternative mechanism for specifying a runpath to the link editor (see `-R` option). If both `LD_RUN_PATH` and the `-R` option are specified, the `-R` is used.

`LD_DEBUG`

(not supported by `ild`) Provide a list of tokens that cause the runtime linker to print debugging information to the standard error. The special token *help* indicates the full list of tokens available.

Note – Environment variable names beginning with the characters ‘`LD_` ‘are reserved for possible future enhancements to `ld`. Environment variable-names beginning with the characters ‘`ILD_` ‘ are reserved for possible future enhancements to `ild`.

Notes

If `ild` determines that a command line option is not implemented, `ild` directly invokes `/usr/css/bin/ld` to perform the link.

`ld` *Options Not Supported by* `ild`

The following options, which may be given to the compilation system, are not supported by `ild`:

`-G`

In dynamic mode only, produces a shared object. Undefined symbols are allowed.

`-B` *`symbolic`*

In dynamic mode only, when building a shared object, bind references to global symbols to their definitions within the object, if definitions are available. Normally, references to global symbols within shared objects are not bound until runtime, even if definitions are available, so that definitions of the same symbol in an executable or other shared objects can override the object's own definition. `ld` issues warnings for undefined symbols unless `-z` *defs* overrides.

`-b`

In dynamic mode only, when creating an executable, does not do special processing for relocations that reference symbols in shared objects. Without the `-b` option, the link editor creates special position-independent relocations for references to functions defined in shared objects and arranges for data objects defined in shared objects to be copied into the memory image of the executable by the runtime linker. With the `-b` option, the output code can be more efficient, but it is less sharable.

`-h` *`name`*

In dynamic mode only, when building a shared object, records *name* in the object's dynamic section. Option *name* is recorded in executables that are linked with this object rather than the object's UNIX System file name. Accordingly, *name* is used by the runtime linker as the name of the shared object to search for at runtime.

`-z muldefs`

Allows multiple symbol definitions. By default, multiple symbol definitions occurring between relocatable objects result in a fatal error condition. This option suppresses the error condition, and allows the first symbol definition to be taken.

`-z text`

In dynamic mode only, forces a fatal error if any relocations against non-writable, allocatable sections remain.

In addition, the following options that may be passed directly to `ld`, are not supported by `ild`:

`-D token,token, ...`

Prints debugging information as specified by each token, to the standard error. The special token *help* indicates the full list of tokens available.

`-F 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*.

`-M mapfile`

Reads *mapfile* as a text file of directives to `ld`. See *SunOS 5.3 Linker and Libraries Manual* for a description of mapfiles.

`-r`

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

Files Used by `ild`

`lib`*x*`.a`	libraries
`a.out`	output file
`LIBPATH`	usually `/usr/lib`

Index