cadence™

irun User Guide

Product Version 14.2

January 2015

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

SystemC/HDL mixed-language simulation patent 7424703 was issued on 9/9/2008

Contents

1

Overview

The *irun* utility lets you run the simulator by specifying all input files and command-line options on a single command line. The utility simplifies the invocation process by letting you use one tool to invoke the simulator instead of invoking multiple tools separately to piece together a snapshot that can be simulated.

irun takes files from different simulation languages, such as Verilog, SystemVerilog, VHDL, Verilog AMS, VHDL AMS, Specman ***e***, and files written in general programming languages like C and C++, and compiles them using the appropriate compilers. After the input files have been compiled, *irun* automatically invokes *ncelab* to elaborate the design and then invokes the *ncsim* simulator.

The most basic way to use *irun* is to list the files that are to comprise the simulation on the command line, along with all command-line options that *irun* will pass to the appropriate compiler, the elaborator, and the simulator. For example:

```
% irun -ieee1364 -v93 -access +r -gui verify.e top.v middle.vhd sub.v
```

In this example:

- The files `top.v` and `sub.v` are recognized as Verilog files and are compiled by the Verilog parser *ncvlog*. The `-ieee1364` option is passed to the *ncvlog* compiler.
- The file `middle.vhd` is recognized as a VHDL file and is compiled by the VHDL parser *ncvhdl*. The `-v93` option is passed to the *ncvhdl* compiler.
- The file `verify.e` is recognized as a Specman ***e*** file and is compiled using `sn_compile.sh`.
- After compiling the files, *irun* then calls *ncelab* to elaborate the design. The `-access` option is passed to the elaborator to provide read access to simulation objects.
- After the elaborator has generated a simulation snapshot, *ncsim* is invoked with both the SimVision and Specview graphical user interfaces.

Use the *irun* help system to get information on tool-specific command-line options. The help options are described in Chapter 2, “Getting Help on irun”.

Two options are particularly useful for getting information on tool-specific options: `-helpshowsubject`, which displays a list of executables and other topics for which help is available, and `-helpsubject`, which displays a list of options for a selected subject. For example:

```
% irun -helpshowsubject
...
Arguments for -helpsubject

  ncvlog
  ncvhdl
  ncelab
  ncsim
  sn_compile.sh
  ncsc_run
  hal
  irun
...
...

% irun -helpsubject ncvlog
Options for requested subject:  ncvlog
  -ams                    Enable Verilog-AMS compilation
  -assert                 Enable PSL language features
  -cd_lexpragma           Process preprocessor directive before lex pragmas
...
...
```

In addition to tool-specific options, you can also include options that modify the behavior of the *irun* utility. These *irun*-specific options are described in Chapter 3, “The irun Command”.

irun Support

In the current release, you can include the following file types on the *irun* command line:

- Verilog
- SystemVerilog
- VHDL
- Specman ***e***
- Specman ***e***-library

- SystemC
- Verilog AMS
- VHDL AMS
- C or C++
- Compiled object files (`.o`), compiled archives (`.a`), and dynamic libraries (`.so`, `.sl`)
- SPICE files

The *irun* command-line interface has been enhanced to compile a design for simulation acceleration using IXCOM tools. This interface is similar to the *irun* use model for compiling a design for IES-based simulation, with additional command-line options to enable simulation acceleration. Thus, the *irun* utility provides a smooth transition from the pure IES-based software simulation mode to simulation acceleration mode.

How irun Works

This section summarizes how *irun* works and what happens by default.

The first time you run the simulator with the `irun` command, it:

1. Creates a directory called `INCA_libs`.
2. Creates a subdirectory under the `INCA_libs` directory called

   `irun.<`*`platform | platform`*`.64>.<`*`irun_version`*`>.nc`

   For example:

   `irun.lnx86.09.20.nc/`

   irun creates files and directories under this subdirectory to support tool operations.

   As a convenience, a symbolic link named `irun.nc` is created that points to the *irun* scratch subdirectory.
3. Parses the command line.
4. Invokes the appropriate compiler for each file specified on the command line.

   Design units contained in HDL design files are compiled into the work library (`worklib`).

   Verilog design units specified in `-y` libraries or `-v` library files are compiled into libraries that have the same names. These libraries are stored in subdirectories of `irun.nc/xllibs`. For example, the following command compiles `top.v` into `worklib` (`INCA_libs/worklib`). Design units in `./libs` are compiled into a library called `libs`

(`INCA_libs/irun.nc/xllibs/libs`), and design units in `./models` are compiled into a library called `models` (`INCA_libs/irun.nc/xllibs/models`).

```
% irun top.v -y ./libs -y ./models +libext+.v
```

The output from the Specman ***e*** compiler, `sn_compile.sh`, is stored in subdirectories under the `irun.nc` directory.

Note: When using the `-y` option, library directory files are not scanned unless they have the same name as a module or user-defined primitive (UDP) that has been instantiated within the normal source text but has not been resolved.

5. Invokes the elaborator (*ncelab*) to elaborate the design and generate a simulation snapshot.
6. Invokes the simulator (*ncsim*) to simulate the snapshot.

The output of all tools is written to a common log file called `irun.log` in the directory in which *irun* was invoked. You can change the name of the log file with the `-l` option. For example:

```
% irun -l run1.log ....
```

Recompilation and Re-Elaboration

When *irun* is invoked again (that is, using an already existing `INCA_libs` scratch directory), *irun* determines if changes on the command line require any files to be recompiled (and then re-elaborated) or if the design needs to be re-elaborated.

irun will go directly to simulation if:

- The content of the input files has not changed since the last time they were compiled. This includes the content of Verilog `-v` library files and `-y` directories.
- The order of the input files on the command line is the same, including the order of Verilog `-v` library files and `-y` directories.
- Command-line options are the same, or, if they are different, the changes do not affect the output of the different language compilers or *ncelab*.

  For example, some options, such as `+gui` and `-s`, affect only run-time behavior. Adding or removing them from the command line does not cause recompilation or re-elaboration. However, adding, removing, or changing other options that can affect compilation or elaboration (for example, removing the `-notimingchecks` option or changing `-access +r` to `-access +rw`) will force recompilation or re-elaboration.

IP Protection

The *ncprotect* utility lets you protect proprietary model information for both Verilog and VHDL. You can protect entire Verilog modules or UDPs and VHDL design units, or you can protect specific language constructs, such as declarations, expressions, assignments, instantiation statements, Verilog tasks and functions and specify blocks, VHDL subprograms and processes, and so on.

See IP Protection for details on *ncprotect*.

You can automatically invoke *ncprotect* to encrypt the Verilog and VHDL source files specified on the command line before *irun* compiles, elaborates, and simulates. This provides a convenient single-step way to verify that the protected files will compile, and that the design will then elaborate and simulate correctly. You can also verify that the design information you intended to protect is, in fact, protected.

There are two ways to automatically invoke *ncprotect* with *irun*:

- Include the `-autoprotect` option on the `irun` command line. This option invokes *ncprotect*, which encrypts the entire source file(s).
- Use the `-ncprotect_file` option to specify a file that contains *ncprotect* options. The arguments file can contain any valid *ncprotect* option.
  - The arguments file can contain the `-autoprotect` option, in which case *ncprotect* encrypts the entire source file(s).
  - If the `-autoprotect` option is not included in the file (or on the command line), *ncprotect* encrypts the regions marked for encryption with protection pragmas in the source files.

Running irun in Prep Mode

When run in prep mode, *irun* does not run the executables to simulate the design. Prep mode creates a multi-step invocation output script (and other required files) and exits.

Include the `-prep` option to run in prep mode. When you include the `-prep` option on the `irun` command line, *irun* processes all source files and command-line options, and then generates a script that contains all of the commands necessary to run each executable needed to simulate the design.

Irun prep mode provides a very easy way to automatically generate a multi-step script. The output script can be run to compile, elaborate, and simulate the design in multi-step mode.

In addition, the output script documents all the steps needed to simulate the design. By clearly showing how each executable is run, the script can provide invaluable help in debugging problems with an *irun* invocation.

See the description of the -prep option for more information.

Support for Compressed Files

Compressed files work just like uncompressed files, but save storage space when managing larger files. With *irun*, you can use compressed files when managing source files and when managing libraries with the -y option. When managing source files, each compressed file should contain only one source file. Archives containing directories with multiple files or subdirectories are not supported at this time.

Currently, *irun* recognizes the following archive formats:

- Gnu zip compression (.gz)
- Standard compression (.Z)

Note: Support is limited to Verilog, System Verilog, VHDL, Verilog AMS, and VHDL AMS.

Use the following syntax when naming compressed source files:

<filename>.<default_file_extension>.<compressed_file_extension>

For example:

```
dut.v.gz
```

Additionally, when using compressed source files, *irun* supports changing the file extension for supported file types. For example, use -vlog_ext to change the default Verilog file extension to .vvv as shown:

```
% irun test.vvv.gz -vlog_ext .vvv
```

However, changing the archive extension using *irun* is not supported. See Changing the Default Set of File Extensions on page 97 for more details on *irun* extension options.

irun also supports compressed files with user defined plus options, and multiple compressed files are supported. If multiple compressed files are processed together, then the software will extract each file into a /tmp directory. Once every file is decompressed and the simulation is complete, the /tmp directory is deleted. For example:

```
% irun +TESTFILE=VER.FULLSCAN.test.verilog.gz VER.FULLSCAN.test.sim.v.gz \
       chip.v.gz smc.v.gz -64bit +access+rwc
```

2

Getting Help on irun

irun is integrated with the *nchelp* utility, which you can use to display extended help on tool messages.

irun also includes an extensive help system with many options that let you display a list of all valid command-line options, recognized file types and their default file extensions, all options related to a particular subject or executable, aliases for options, the minimum characters that must be entered for an option, and so on.

Use the `-helphelp` option to display a list of all options that control help.

```
% irun -helphelp
```

Getting Help on Tool Messages

Use the *nchelp* utility to display extended help on the messages generated by *irun*, the compiler, elaborator, and simulator.

Syntax:

```
% nchelp [options] tool_name message_code
```

You can enter the *message_code* argument in lowercase or in uppercase.

Examples:

```
% nchelp irun BDOPT
% nchelp irun bdopt
% nchelp ncvlog NOBIND
% nchelp ncelab cuvwsp
% nchelp ncsim NOSNAP
```

Getting Help on Command-Line Options

irun is a single executable that lets you use one command to invoke various compilers to compile different types of files specified on the command line, elaborate the design, and

simulate a snapshot. Because *irun* can invoke several tools, each of which has its own set of command-line options, the number of options that you can use on the `irun` command is large.

irun includes several help options that let you display the options in various ways. These help options are described below.

If you need more information on an option displayed by an *irun* help option, use the search facility in the online help system.

irun Help Options

The following options can be used to get help on *irun* options.

-h

Display a minimal list of `irun` command options.

-helpalias

Display the different ways to enter an option.

The various options you can use to display a list of command-line options display the dash option by default (for example, `-nocopyright`, `-notimingchecks`, `-nolog`). Most command-line options have a corresponding plus option that were used when running the simulator with the `ncverilog` command (for example, `+nocopyright`, `+notimingchecks`, `+nolog`). Use the `-helpalias` option to display the aliases for options. For example:

```
% irun -helpall -helpalias
...
  +ncnolog
  +nolog
  -nolog              Suppress generation of the default logfile
...
```

-helpall

Display a list of every supported option.

-helpargs

Display a list of source files specified on the command line, with their type, and a list of the command-line options being used.

The `-helpargs` option displays the information and then exits. No simulation is performed.

Example:

```
% irun -nocopyright -ieee1364 -v93 -access +r sub.v top.v middle.vhd -helpargs
Files on the command line and their determined type
  sub.v                     verilog
  top.v                     verilog
  middle.vhd                vhdl

  -nocopyright              Suppress printing of copyright banner
  -ieee1364                 Report errors according to IEEE 1364 standards
  -v93                      Enable VHDL93 features
  -access <+/-rwc>          Turn on read, write and/or connectivity access
...
...
```

-helpfileext

Display all source file types, the default file extensions defined for each type, and the command-line option you can use to change the set of defined file extensions.

```
% irun -helpfileext
      unknown  The file type is unknown
      verilog  Verilog HDL           -vlog_ext .v,.vp,.vs,.V,.VP,.VS
      vcnf     Verilog config        -vcfg_ext
    verilog95  Verilog 1995 HDL      -vlog95_ext .v95,.v95p,.V95,.V95P
         vhdl  VHDL HDL              -vhdl_ext .vhd,.vhdp,.vhdl,.vhdlp,
                                               .VHDL,.VHDLP,.VHD,.VHDP
        vhcfg  VHDL config           -vhcfg_ext .vhcfg,.vhcfgp
            e  Specman e             -e_ext .e,.E
systemverilog  SystemVerilog HDL     -sysv_ext .sv,.svp,.SV,.SVP,.svi,.svh,
                                               .vlib,.VLIB
        amsva  Verilog-A file        -amsva_ext
  verilog-ams  Verilog-AMS HDL       -amsvlog_ext .vams,.VAMS
     vhdl-ams  VHDL-AMS HDL          -amsvhdl_ext .vha,.VHA,.vhams,.VHAMS,
                                                  .vhms,.VHMS
     psl_vlog  PSL file for Verilog  -propvlog_ext .pslvlog
```

```
psl_vhdl  PSL file for VHDL     -propvhdl_ext .pslvhdl
 psl_sc   PSL file for SystemC  -propsc_ext .pslsc
       c  C                     -c_ext .c
     cpp  C++                   -cpp_ext .cpp,.cc
assembly  Assembly              -as_ext .s
       o  Compiled object       -o_ext .o
       a  Compiled archive      -a_ext .a
      so  Dynamic library       -dynlib_ext .so,.sl
     scs  SPICE file            -spice_ext .scs,.sp
```

The output of the `-helpfileext` option provides two important pieces of information:

- The file types are shown in the first column. These are the arguments that you can use with the `-default_ext` option to specify the file type for files with unrecognized file extensions.
- The third column shows you the default file extensions for each file type, and the option that can be used to override them or to add to the set of extensions.

  For example, the default file extensions for Verilog files are `.v`, `.vp`, `.vs`, `.V`, `.VP`, and `.VS`. A source file with a `.vlog` or `.vg` extension will not be recognized as a Verilog file. You can add `.vlog` and `.vg` to the list of defined Verilog file extensions by using the `-vlog_ext` option. See "Changing the Default Set of File Extensions" on page 97 for more information.

-helphelp

Display all options that control help for *irun*.

-helpncverilog

Show the *ncverilog* form of the options.

This option is useful for users transitioning from *ncverilog* to *irun*. It displays a list of `irun` command options, but options that can be used with *ncverilog* are shown as plus options.

```
% irun -helpall -helpncverilog
```

See "ncverilog" on page 111 for more information.

-helpshowmin

Show the minimum characters required for dash options.

Command-line options can be abbreviated to the shortest unique string. The `-helpshowmin` option displays this string for dash options. For example:

```
% irun -helpall -helpshowmin
  ...
  -64[bit]                  Invoke 64bit version
  -a_[ext] <ext>            Override extensions for archive files
  -acce[ss] <+/-rwc>        Turn on read, write and/or connectivity access
  -af[ile] <file>           Specify an access file to be used
  ...
```

-helpshowsubject

Display a list of all subjects that can be used with the `-helpsubject` option.

```
% irun -helpshowsubject
```

-helpsubject *subject*

Display a list of options for a specified subject.

Use the `-helpshowsubject` option to display a list of subjects.

```
% irun -helpshowsubject
```

Then use the `-helpsubject` option to display help for a subject. For example:

```
% irun -helpsubject timing
% irun -helpsubject systemverilog
% irun -helpsubject ncvlog
```

You can display help for multiple subjects by providing a comma-separated list of subjects. Do not include a space after the comma.

```
% irun -helpsubject sn_compile.sh,specman
```

Note: For an executable, such as *sn_compile.sh* or HAL, the `-helpsubject` option displays a list of options that can be passed to that executable when invoked by *irun*. This list of options may be different from the list of options displayed when using the help system for the executable itself. For example, the following two commands may not display the same list of options:

```
% irun -helpsubject hal
% hal -help
```

-helpusage

Display a list of every supported option, with each option followed by two fields:

- `flags:`

  The flags displayed in this field can be:

  - `ARG`–This option requires an argument.
  - `HARD`–In a subsequent *irun* invocation, adding or removing this option, or changing the argument to this option, will cause some (or all) executables to be rerun.
  - `NOMULT`–This option can be specified once on the command line.
  - `PSC`–This option can have a list of arguments separated by a comma.
  - `PSP`–This option can have a list of arguments separated by a `+`.
  - `PSE`–This option can have a list of arguments separated by a `=`.
  - `PSFS`–This option can have a list of arguments separated by a `/`.
  - `PSSP`–This option can have a list of arguments enclosed in quotes. For example:

    `"a b c"`
  - `SCMUL`–This option can have a list of arguments.

- `execs:`

  This field lists the executables to which the option applies.

Example:

```
% irun -helpusage
...
-vlogext          flags:ARG execs:irun
-vpicompat        flags:ARG,HARD execs:ncelab,ncsim
-work             flags:NOMULT,ARG,HARD execs:ncvlog,ncvhdl,ncelab,ncsc_run
-xlstyle_units    flags: execs:ncsim
```

-helpverbose

Display options with verbose help text.

```
% irun -helpall -helpverbose
% irun -helpsubject ncvlog -helpverbose
```

-helpwidth *width*

Set the maximum width for help messages.

By default, help messages are displayed in a format that is 89 characters wide. Use the `-helpwidth` option to specify a different number of characters. The minimum width you can specify is 58.

Using the Online Help

Most `irun` command-line options are described in the product manuals shipped with the release. If you need more detailed information on a particular option than what is provided by the various help options, use the online search facility in the online help.

When using Search, enter the name of the option in the Search box, including the dash. The dash must be escaped with a backslash character. For example, enter `\-profile`.

3

The irun Command

This chapter describes the `irun` command and the command-line options that are specific to `irun`.

irun Command Syntax

The `irun` command has the following syntax:

```
irun [options] files
```

Managing Your irun Command

With *irun*, you specify all input files and all command-line options on one command line. As a result, an `irun` command can be complex and quite lengthy. It is recommended that you use the following features to help simplify the command line.

Using the IRUNOPTS Variable

The `hdl.var` file is an optional configuration file, written in ASCII text, which contains configuration variables that determine how your environment is configured, that allow you to define command-line options and arguments, and that specify the locations of support files and invocation scripts.

See The hdl.var File in *Setting Up Your Environment* for more details.

With *irun*, you specify command-line options in an `hdl.var` file using the `IRUNOPTS` variable. For example:

```
# hdl.var
DEFINE IRUNOPTS -ieee1364 -notimingchecks -access +rw
```

You can also define custom file extensions with the `IRUNOPTS` variable in an `hdl.var` file:

```
#hdl.var file
DEFINE IRUNOPTS -vlog_ext .v,.vg,.rtl -ieee1364 -notimingchecks -access +rw
```

Note: When using *irun*, `DEFINE IRUNOPTS -vlog_ext` is required when specifying custom file types to use with the Verilog parser. The `VERILOG_SUFFIX` variable is not compatible with *irun*, See Customizing irun for more details on changing default file extensions.

Additionally, you can specify project-wide options in a project `hdl.var` file. To do so, link the project `hdl.var` file to your own `hdl.var` file with the `INCLUDE` statement. For example:

```
# hdl.var
INCLUDE path_to_project_hdlvar
DEFINE IRUNOPTS -ieee1364 -notimingchecks -access +rw
```

Note: Because some actions take place before the `hdl.var` file is read, not all command-line options can be included in the definition of the `IRUNOPTS` variable. For example, because the log file is opened before the `hdl.var` file is read, the `-l` option, which is used to change the name of the log file, is ignored if you include it in the definition of `IRUNOPTS`. The following options are ignored if included in an `hdl.var` file:

`-64bit`

`-l` *`logfile_name`*

`-append_log`

`-cdslib` *`cdslib_file`*

`-hdlvar` *`hdlvar_file`*

`-version`

`-nocopyright`

Using an Arguments File

An arguments file can specify input files and command-line options. For example:

```
./vhdl_src/file.vhd
./vlog_src/file.v
./psl/file.pslvlog
-ieee1364
-notimingchecks
-access +rw
```

irun includes two options that you can use to specify an arguments file: `-f` and `-F`.

```
% irun -f irun.args         // Scans for files relative to the
                            // irun invocation directory.
```

```
% irun -F ./args/irun.args  // Scans for files relative to the location of
                            // the arguments file irun.args.
```

See the description of the -F option for details on the difference between these options and for more information on using an arguments file.

Input Files

Input files specified on the command line must have a full, local, or relative path.

The type of a file is determined by its file extension. The following table shows the default mapping of file extensions to file type.

File Type	File Extensions
Verilog	`.v`, `.vp`, `.vs`, `.V`, `.VP`, `.VS`
Verilog 1995	`.v95`, `.V95`, `.v95p`, `.V95P`
SystemVerilog	`.sv`, `.SV`, `.svp`, `.SVP`, `.svi`, `.svh`, `.vlib`, `.VLIB`
VHDL	`.vhd`, `.vhdp`, `.vhdl`, `.vhdlp`, `.VHD`, `.VHDP`, `.VHDL`, `.VHDLP`
VHDL configuration	`.vhcfg`
Specman ***e***	`.e`, `.E`
Compiled ***e***-library	`.elib`
Verilog-AMS	`.vams`, `.VAMS`
VHDL-AMS	`.vha`, `.VHA`, `.vhams`, `.VHAMS`, `.vhms`, `.VHMS`
VIP Library	`.viplib`
PSL file for Verilog	`.pslvlog`
PSL file for VHDL	`.pslvhdl`
PSL file for SystemC	`.pslsc`
C	`.c`
C++	`.cpp`, `.cc`
Assembly	`.s`
Compiled object	`.o`
Compiled archive	`.a`
Dynamic library	`.so`, `.sl`
SPICE file	`.scs`, `.sp`

Specman Input Files

When specifying Specman input files on the command line, one or more precompiled ***e***-library files may be specified given the following rules:

- `.elib` files must be compiled. If the `-nosncomp` option is combined with an ***e***-library file on the command line, then *irun* will generate an error.
- `.elib` files are only supported with the default generation engine, IntelliGen. Attempting to compile or link ***e***-library files with the `-pgen` option will result in an error.

Note: *irun* only supports the compilation of ***e*** files on top of previously compiled ***e***-library files. It does not support the creation of ***e***-library files.

If the `SPECMAN_PATH` environment variable has been set, *irun* scans the specified directories to find ***e*** and ***e***-library files. You can also use the `-snpath` option to list directories to scan when loading ***e*** files or compiling ***e*** and ***e***-library files. Any path you specify with this command-line option is prefixed to paths defined in the `SPECMAN_PATH` environment variable.

Custom File Extensions for Input Files

For each file type, there is a command-line option that you can use to change, or add to, the list of defined file extensions. For example, you can change the list of valid extensions for Verilog files by using the `-vlog_ext` option.

irun generates an error if it encounters a file with an extension it does not recognize. You can use the `-default_ext` option to specify the file type for files with unrecognized extensions.

See "Changing the Default Set of File Extensions" on page 97 for details on using these options.

Command-Line Options

Use the `irun -helpall` command to display a list of all valid `irun` command options.

You can enter command-line options in lowercase or uppercase. For example `-nolog` and `-NOLOG` are both valid.

Command-line options can be abbreviated to the shortest unique string. For example, the shortest unique string for the `-nowarn` option is `-now`. Use the `irun -helpshowmin` command to display the minimum characters required for an option.

This manual describes only the command-line options that are unique to the *irun* executable. Most options that you can include on the `irun` command are options that are passed to the executables that *irun* invokes: the various compilers (*ncvlog*, *ncvhdl*, *sn_compile.sh*, the C or C++ compilers, and so on), the elaborator (*ncelab*), the simulator (*ncsim*), and other tools, such as HAL. To get help on these command-line options, you can:

- Use the various `irun` command help options. See "Getting Help on Command-Line Options" on page 13 for details on the *irun* help system.
- Use the online documentation to get a more detailed description of a specific option. You can:
  - Use the search facility in the online help system. When using Search, enter the name of the option, including the dash. The dash must be escaped with a backslash character. For example, enter `\-profile`.
  - Refer directly to a manual. The following table provides details on where options are described.

If you want information on ...	Look in ...
Verilog parser (*ncvlog*) options	Verilog Compilation Command-Line Options
VHDL parser (*ncvhdl*) options	VHDL Compilation Command-Line Options
Elaborator (*ncelab*) options	Elaborator Command-Line Options
Simulator (*ncsim*) options	Simulation Command-Line Options
ncsc_run options	Section "Simulating SystemC Models Using ncsc_run" in the chapter "Simulating SystemC Models" in the *SystemC Simulation User Guide*.

If you want information on ...	Look in ...
AMS-related options	The “Elaborating”, “Simulating”, and “Tcl-Based Debugging” chapters in the *Virtuoso AMS Designer Simulator User Guide*.
IXCOM options	“IXCOM Command-Line Options” on page 68
OVM-related options	“OVM Command-Line Options” on page 70
Specman-related options	“Specman Command-Line Options” on page 75.
Coverage options	“Generating Coverage Data” in the *ICC User Guide*.
HAL options	“Performing HDL Analysis” in the *HAL User Guide*.
UVM-related options	“UVM Command-Line Options” on page 87

Executable Options Not Defined in irun

Executables may have options that are not defined in *irun*. You can pass these options to an executable with one of the following options:

- `-ncvlogargs "list_of_options"`

  Pass arguments to the Verilog parser

- `-ncvlog_args,option[,option]`

  Pass arguments to the Verilog parser (*ncsc_run* compatibility)

- `-ncvhdlargs "list_of_options"`

  Pass arguments to the VHDL parser

- `-ncvhdl_args,option[,option]`

  Pass arguments to the VHDL parser (*ncsc_run* compatibility)

- `-ncelabargs "list_of_options"`

  Pass arguments to the elaborator

- `-ncelab_args,option[,option]`

  Pass arguments to the elaborator (*ncsc_run* compatibility)

- `-ncsimargs "list_of_options"`

  Pass arguments to the simulator

- `-ncsim_args, option[,option]`

  Pass arguments to the simulator (*ncsc_run* compatibility)

- `-ncsc_runargs "list_of_options"`

  Pass arguments to *ncsc_run*

- `-halargs hal_options`

  Pass options to Incisive HDL Analysis (HAL)

- `-sncompargs "list_of_options"`

  Pass arguments to the ***e*** compiler

These options provide a workaround for when options are missing in *irun*. The specified options are passed directly to the executable. Do not use these options to pass options that are defined in *irun*. Supported options should be entered directly on the command line.

Some options that are not defined in *irun* and that are passed to executables using the options listed above may conflict with other options specified on the command line or may cause unexpected behavior. *irun* scans the contents of the options and generates a warning if it detects an option that might cause unexpected behavior.

irun-Specific Command-Line Options

This section describes options that are specific to the `irun` command.

Options shown below in lowercase can also be entered in uppercase. For example, both `-makelib` and `-MAKELIB` are valid.

Note: Several Specman-related command-line options have been implemented. See "Specman Command-Line Options" on page 75 for details on these options.

-64bit

Invoke the 64-bit version of *irun*.

Besides including the `-64bit` command-line option when you invoke *irun*, you can also run the 64-bit version by:

- Setting up your `PATH` environment variable to point to the 64-bit version.
- Setting the `INCA_64BIT` or `CDS_AUTO_64BIT` environment variable.

- The INCA_64BIT variable is treated as boolean. You can set this variable to any value, or to a null string.

  ```
  setenv INCA_64BIT
  ```

- The CDS_AUTO_64BIT variable is set to INCLUDE:INCA.

  ```
  setenv CDS_AUTO_64BIT INCLUDE:INCA
  ```

See Configuring Your Environment for the 64-Bit Version for more information on running in 64-bit mode.

*_ext [+].extension[,.extension...]

Irun includes several command-line options that let you override the built-in file extensions used to recognize the file type of source files included on the command line. These options include:

- -a_ext
- -amsvhdl_ext
- -amsvlog_ext
- -as_ext
- -c_ext
- -cpp_ext
- -dynlib_ext
- -e_ext
- -o_ext
- -spice_ext
- -sysv_ext
- -vhcfg_ext
- -vhdl_ext

irun uses the file extensions of the input files specified on the command line to determine their file type. Each recognized file type has a built-in, predefined set of file extensions. For each file type, there is a command-line option that you can use to change, or add to, the list of file extensions mapped to a given language.

For example, the default file extensions for Verilog files are `.v`, `.vp`, `.vs`, `.V`, `.VP`, and `.VS`. If you have Verilog files with other extensions (for example, `.rtl` and `.vg`), you must specify that these are valid extensions for Verilog files. You can do this with the `-vlog_ext` option. You can:

- Replace the list of built-in, predefined extensions with a new list. For example, the following option specifies that the valid extensions for Verilog files are `.v`, `.rtl`, and `.vg`:

  ```
  -vlog_ext .v,.rtl,.vg
  ```

- Add extensions to the list of built-in, predefined extensions by using a plus sign (`+`) before the list of extensions to add. For example, the following option adds `.rtl` and `.vg`.

  ```
  -vlog_ext +.rtl,.vg
  ```

  This is the same as:

  ```
  -vlog_ext .v,.vp,.vs,.V,.VP,.VS,.rtl,.vg
  ```

See "Changing the Default Set of File Extensions" on page 97 for details on using an extension option to override the default set of file extensions.

-allowredefinition

When compiling multiple source files on the command line, allow the module or unit that is defined in one file to be redefined again if it is also defined in other files.

The `-allowredefinition` option works with the following file types:

- Verilog: By default, *irun* generates an error if a module is defined in multiple Verilog input files specified on the command line. Use the `-allowredefinition` option to override the default and allow modules to be redefined. The *irun* command will generate a warning to inform you that the module is being redefined, and the last definition that is compiled will be used in the design.
- VHDL: By default, *irun* generates a warning noting that the most-recently analyzed architecture is being elaborated If duplicate VHDL design units are encountered when parsing multiple files on the command line. However, if the `-smartorder`, `-smartlib`, or `-smartscript` option is also included, then the compilation order is determined by the tool and an error is generated. Instead of editing the source files to make sure that there are no duplicate design units, you can use the `-allowredefinition` option.

-append_log

Append log information from multiple *irun* runs into one log file. By default, the log file is overwritten each time you run *irun*.

-armfm argument

Used with Cadence Virtual System Platform, this option allows simulation of a virtual prototype with the given Fast Model from ARM in the design. The option invokes ARM model-specific capability.

The *argument* parameter is either:

- A packaged ARM CPU core model name (for example, ARM926CT)
- One of several reserved values

For more information on using this option with Virtual System Platform, see the *Virtual System Platform User Guide*.

-autoprotect

Encrypt the Verilog and VHDL source files with *ncprotect* before compiling, elaborating, and simulating.

The -autoprotect option enables the automatic protection of source files using *ncprotect*. The Verilog and VHDL source files specified on the irun command line are protected using the -autoprotect option of *ncprotect*. This option encrypts the entire source file(s). For example, the following command:

```
% irun -autoprotect -v93 top.v middle.vhd sub.v
```

invokes *ncprotect* to encrypt the Verilog files, and then invokes it again to encrypt the VHDL file.

```
ncprotect -autoprotect -lang verilog top.v sub.v
ncprotect -autoprotect -append_log -lang vhdl middle.vhd
```

After encrypting the files, *irun* compiles the protected files (top.vp, sub.vp, and middle.vhdp), elaborates the design, and simulates the snapshot.

You can pass options to *ncprotect*, including the -autoprotect option, by including the options in an arguments file, which is specified with the -ncprotect_file *filename* option. For example,

```
% irun -ncprotect_file ncprot.args -v93 top.v middle.vhd sub.v
```

See IP Protection for details on *ncprotect*.

By default, the *ncprotect* output is written to `irun.log`. Use the `-log_ncprotect` *`filename`* option to redirect *ncprotect* output to a specified log file.

```
% irun -autoprotect -log_ncprotect ncprot.log -v93 top.v middle.vhd sub.v
```

-c

This option is the same as `-elaborate`.

-cdn_vip_root directory

Specify the location of the VIPCAT installation.

By default, *irun* uses this information to treat VIP models as native simulation components. *irun* automatically compiles the model and loads the necessary VIP shared objects, making it so that all you need to do is specify the HDL model wrappers and the VIP HDL packages that are required.

See the *Cadence CDN_AXI VIP User Guide* for details.

-cdn_vip_svlib

Integrate the new SystemVerilog VIP library.

This option requires that you use the `-cdn_vip_root` option to specify the location of the VIPCAT installation. When both options are specified on the command line, *irun* will automatically load the VIP library `libcdnsv.so`.

See the *Cadence CDN_AXI VIP User Guide* for details.

-cdn_viplib

Enable the new method of integrating VIP models.

This option requires that you use the `-cdn_vip_root` option to specify the location of the VIPCAT installation. When both options are specified on the command line, *irun* will link the available VIP models rather than compiling them. This new method should save time during simulation.

See the *Cadence CDN_AXI VIP User Guide* for details.

-checkargs

Check command-line arguments for validity.

Use the `-checkargs` option to verify that the options you have included on the `irun` command line are valid options. For example:

```
% irun -nocopyright -ieee1364 -v93 -access +r \
xor_verify.e xor.v xor_specman.vhd -top xor_top -checkargs

Checking arguments. Following are the command line arguments recognized by irun:
   Minus ("-") options:
        -nocopyright -ieee1364 -v93 -checkargs
   Paired minus ("-") options:
        -access +r
        -top xor_top

   Source file arguments:
        xor_verify.e
        xor.v
        xor_specman.vhd
%
```

-clean

Delete the `INCA_libs` directory before executing.

-cleanlib

Delete the `INCA_libs` directory and all `.pak` files found by searching `cds.lib` before executing *irun*. The search includes the `cds.lib` file from the current working directory or a file specified with the `-cdslib` option on the command line.

When removing files, *irun* will search the entire `cds.lib` structure. If you assign `TmpRootDir` in `cds.lib` to provide temporary storage for libraries in a design, then *irun* will search for `.pak` files in the specified path and delete them.

You can also use the `-cds_implicit_tmpdir` option together with the `-cleanlib` option to specify a particular directory to search at the command line. For example:

```
% irun -cleanlib -cds_implicit_tmpdir ./mytmp top.v
```

Note: Read only `.pak` files and any files in the Cadence installation area will not be removed.

-cleanlibscript

Create a script in the working directory called `cleanlibscript.sh` that you can use to delete the `INCA_libs` directory and all `.pak` files found by searching `cds.lib`. The search includes the `cds.lib` file from the current working directory or a file specified with the `-cdslib` option on the command line.

When creating the script, *irun* will search the entire `cds.lib` structure. If you assign `TmpRootDir` in `cds.lib` to provide temporary storage for libraries in a design, then *irun* will search for `.pak` files in the specified path and add them to the script as well.

After creating the script, *irun* exits.

-cleanlibverify

Display a list of all `.pak` files found by searching `cds.lib` and then request confirmation before deleting them and executing *irun*. For example:

```
% irun -cleanlibverify -v93 top.v middle.vhd sub.v

The following pak files have been found in libraries defined in cds.lib:
    ./INCA_libs/vhdlspice_skeletons/inca.lnx86.20.pak
    ./INCA_libs/worklib/inca.lnx86.010.pak
    ./INCA_libs/worklib/inca.lnx86.020.pak
Would you like to remove these files? y/n
```

If you choose to preserve all `.pak` files, *irun* will cancel the operation and exit.

-compile

Parse/compile the source files, but do not elaborate.

-cpost filename [filename...] [-end]

Compile the specified C or C++ files after elaboration.

In some cases, it is necessary to compile C or C++ files after elaboration. For example, SystemVerilog DPI files that depend on an export file created by *ncelab* must be compiled after elaboration.

The files to be compiled after elaboration must be specified on the command line, following the `-cpost` option. The list of files is terminated by:

- An `-end` option
- Another option that specifies a collection of files to be processed together.

  These options are:

  - Another `-cpost` option
  - `-makelib`
  - `-snstage`

Note: When using `-cpost` with either the `-ifv` or the `-iev` option, *irun* will pass the compiled C or C++ files to the formal verifier instead of the simulator.

-date

Print the date and time when each engine is invoked.

-debug

Set read and write access to all simulation objects in the design.

This option is the same as `-access +rw`, which turns on read and write access to all objects.

If Specman ***e*** files are present, they will be interpreted and then saved into a save file instead of being compiled. This is similar to using the `-nosncomp` option.

API files (C or C++) and SystemC components will be compiled for debug.

-debugscript filename

Generate a script that captures the environment variables and commands for the executables that were invoked by *irun*. The generated script can be useful in debugging *irun* problems.

If you include the `-debugscript` option on the command line, *irun* generates two files:

- A script with the specified filename

  This script contains the commands that you can execute to run the executables that were invoked by *irun*. Each executable is invoked with the `-file` option, which specifies the argument file created by *irun* when that executable was invoked. The argument file contains the command-line arguments that were passed to the executable. When the command is executed, the executable automatically sets or modifies all environment variables that *irun* set or modified when it invoked the executable.

- A file called *filename*.env

  This file can be sourced to reproduce the same environment as the *irun* run. The debug script file contains the required `source` command. This command is commented out in the script, and you can uncomment the line before running the script.

Note: Using the `-debugscript` option causes all source files to be recompiled.

Example

```
% irun -v93 top.v sub.v middle.vhd -debugscript debug.script
...
...
% cat debug.script
#!/bin/csh -f
#
# File Created by irun to ease debug process
#
# Uncomment next line if attempt to reproduce same environment as the test run
before.
#source debug.script.env
#
specman  -version -commands exit
setenv NCRUNMODE "irun:./INCA_libs/irun.sun4v.09.20.nc"
setenv IRUNBATCH "TRUE"
ncvlog -file /home/joe/irun/vlog_vhdl/INCA_libs/irun.sun4v.09.20.nc/ncvlog.args
unsetenv NCRUNMODE
unsetenv IRUNBATCH
#
setenv NCRUNMODE "irun:./INCA_libs/irun.sun4v.09.20.nc"
setenv IRUNBATCH "TRUE"
ncvhdl -file /home/joe/irun/vlog_vhdl/INCA_libs/irun.sun4v.09.20.nc/ncvhdl.args
unsetenv NCRUNMODE
unsetenv IRUNBATCH
#
setenv NCRUNMODE "irun:./INCA_libs/irun.sun4v.09.20.nc"
setenv IRUNBATCH "TRUE"
ncelab -file /home/joe/irun/vlog_vhdl/INCA_libs/irun.sun4v.09.20.nc/ncelab.args
unsetenv NCRUNMODE
unsetenv IRUNBATCH
#
setenv NCRUNMODE "irun:./INCA_libs/irun.sun4v.09.20.nc"
setenv IRUNBATCH "FALSE"
```

```
ncsim -file
/home/joe/irun/vlog_vhdl/INCA_libs/irun.sun4v.09.20.nc/bird_3792/ncsim.args
unsetenv NCRUNMODE
unsetenv IRUNBATCH
#
```

Note: *irun* also has a -prep option, which generates a script containing the commands to invoke each executable that *irun* would invoke to simulate the design. The primary difference between the scripts is that the script generated by `-debugscript` contains internal environment variables and options used by *irun* when it invokes an executable. These internal variables and options are not included in the *irun* prep mode script.

-default_ext file_type

Override the default file type to file extension mapping.

irun generates an error if it encounters a file with an extension that it does not recognize. Use the `-default_ext` option to specify the file type for files with unrecognized file extensions. For example, the following option specifies that all files with unrecognized extensions are to be treated as Verilog files:

```
-default_ext verilog
```

See "Changing the Default Set of File Extensions" on page 97 for more information.

-discapf

Disable the capital F (`-F`) behavior of loading input files relative to the directory containing the arguments file.

In addition to the `-f` option, which scans the arguments file for files relative to the *irun* invocation directory, *irun* also includes the `-F` option, which scans for files relative to the location of the arguments file. If a file is not found relative to the location of the arguments file, *irun* rescans relative to the *irun* invocation directory.

Use the `-discapf` option to disable the `-F` behavior. Because `-F` options might exist in arguments files that are now nested by other files specified by either `-F` or `-f`, this option provides a convenient way to disable the behavior from the command line.

-e

Enable command-line editing, including the following features:

- The right-arrow and left-arrow keys move the cursor right or left one character on the command line. You can insert new characters at the cursor position.
- The up-arrow and down-arrow keys scroll backward or forward in the command-line history buffer.
- The Tab key provides filename completion functionality.
- The Backspace (or Delete) key deletes characters.
- `Control-a` moves the cursor to the beginning of the command line.
- `Control-e` moves the cursor to the end of the command line.

Command-line editing works only for interactive mode. It is not compatible with batch or GUI mode. Do not use the `-e` option if you are using a debugger.

Note: In the current release, the command line editing functionality is part of the Specman environment. You must have Specman installed, and your `path` variable must include:

specman_install_directory`/tools.$ARCH/bin`

-elaborate

Parse/compile the source files, elaborate the design, and generate a simulation snapshot, but do not simulate.

If source files have already been compiled, `-elaborate` will recompile any changed design units before re-elaborating the design. If you want to re-elaborate without compiling any source files, include the `-noupdate` option on the command line. For example:

```
% irun -elaborate -noupdate [other_options] -top lib.cell:view
```

-end

Terminate the list of files specified after a collection option. A collection option is an option used to specify a list of files to be processed as a group. The collection options are:

- `-cpost`
- `-makelib`

  A `-makelib` list can be terminated with `-end` or `-endlib`.
- `-snstage`

  A `-snstage` list can be terminated with `-end` or `-endsnstage`.

The list of files specified with one of these options is also terminated by another collection option. For example, the following two commands are identical:

```
% irun -makelib lib1 file1.vhd file2.vhd -end -makelib lib2 file3.vhd -end ....

% irun -makelib lib1 file1.vhd file2.vhd -makelib lib2 file3.vhd -end ....
```

-endlib

Terminate the list of files to be compiled into a library.

This option is used with the `-makelib` option, which is used for compiling specified files into a specified library. The syntax is as follows:

`-makelib` *`path_to_library`*`[:`*`logical_name`*`]` *`source_files`* `[-endlib]`

All files on the command line following the `-makelib` option are compiled into the specified library. The list of files for an active `-makelib` option is terminated by the next use of a `-makelib` option or by a `-endlib` or `-end` option.

For example:

```
% irun -makelib lib1 file1.vhd file2.vhd -makelib lib2 file3.vhd -endlib file4.vhd
```

See "Compiling into Multiple Libraries" on page 101 for details on using the `-makelib` option.

-F arguments_file
-f arguments_file

Use the command-line arguments contained in the specified arguments file.

irun includes two options that you can use to specify an arguments file: `-f` and `-F`. These two options differ in the following ways:

- The `-f` option scans the arguments file for files relative to the *irun* invocation directory.
- The `-F` option first scans for files relative to the location of the arguments file. If a file is not found relative to the location of the arguments file, *irun* rescans relative to the *irun* invocation directory.

irun also allows nested `-f` or `-F` options. A `-f` arguments file can contain other `-f` or `-F` options. Likewise, a `-F` arguments file can contain other `-f` or `-F` options. The arguments in an included `-f` file will be processed relative to the *irun* invocation directory. The arguments in an included `-F` file will be processed relative to the search path of that specified `-F` option.

Examples:

```
% irun -f irun.args   // Scans for files relative to the irun invocation directory.
```

```
% irun -F ./args/irun.args  // Scans for files relative to the location of
                             // the arguments file irun.args.
```

You can use the wildcard character when specifying input files. For example:

```
/vlog/*
../rtl/*.v
../rtl/count*.vhd
../rtl/*count.vhd
../rtl/c*nt.vhd
/usr1/libs/rtl*54*stl/*.v
```

Environment variables can be used in an arguments file. The syntax is `${env_var}`. For example, if you set the environment variable `SRC` to point to a directory that contains source files, the arguments file can contain the variable `${SRC}`, as in the following example:

```
${SRC}/source1.v
${SRC}/source2.v
${SRC}/*.sv
```

Note: Single-line comments beginning with a pound sign (`#`), `//`, or `--` are supported in an arguments file. For example:

```
// File: irun.args
-- This is a comment
# This is another comment
-ieee1364
source.v
```

Multi-line comments are not supported.

-filemap source_file [source_file ...] list_of_options -endfilemap

Start and end a filemap collection.

By default, all options specified on the command line apply to all source files on the command line. The `-filemap` and `-endfilemap` options let you attach certain Verilog compiler (*ncvlog*) or VHDL compiler (*ncvhdl*) options to a particular source file. *irun* uses this option to write the `FILE_OPT_MAP` variable to the `hdl.var` file, which is read by *ncvlog* and *ncvhdl* when they are invoked.

Example:

```
% irun file1.v -filemap file2.v -view view1 -v1995 -endfilemap -assert
```

In this example, `file1.v` will be processed with `-assert` only. The file `file2.v` will be processed with `-assert`, `-view`, and `-v1995`. The following `FILE_OPT_MAP` variable is written to the `hdl.var` file:

```
DEFINE FILE_OPT_MAP (\
        file2.v => -VIEW view1 -v1995, \
)
```

Each filemap collection can contain an unlimited number of Verilog or VHDL source files and options. For example:

```
% irun file1.v -filemap file2.v file3.v file4.v -assert -work foo -endfilemap ....
```

Each filemap collection must contain at least one file and at least one option.

You can use multiple `-filemap` options on the command line. If multiple `-filemap` options are used, a new `-filemap` option closes the previous filemap collection and starts a new collection. For example, the following two command lines are equivalent:

```
% irun file1.v -filemap file2.v -view view1 \
-filemap file3.v -view view2 -endfilemap

% irun file1.v -filemap file2.v -view view1 -endfilemap \
-filemap file3.v -view view2 -endfilemap
```

You can also control the compilation of different source files with different options by defining the `FILE_OPT_MAP` variable in an `hdl.var` file. See Compiling Source Files with Specific Options for details. If you include your own `hdl.var` which defines the `FILE_OPT_MAP` variable, the user-defined `FILE_OPT_MAP` will be included in the *irun*-generated variable.

Note: Not all parser options can be used with the `-filemap`/`-endfilemap` options. The following options are not supported:

`-ams`	`-neverwarn`
`-cdslib`	`-nocopyright`
`-cds_implicit_tmpdir`	`-nolink`
`-cds_implicit_tmponly`	`-nowarn`
`-cmdfile`	`-ovl`
`-define`	`-smartorder`
`-design_top`	`-smartscript`
`-errormax`	`-specificunit`

`-hdlvar`	`-status`
`-ial`	`-sv`
`-ncerror`	`-zlib`
`-ncfatal`	`-zparse`

-forceelab

Force *ncelab* to execute.

-gdb

Run *ncsim* under gdb.

-gdbelab

Run *ncelab* under gdb.

-gdbpath path_to_gdb

Use the specified gdb instead of the gdb shipped with the product.

-gdbsh

Force gdb to run under sh instead of under the user SHELL.

-norundbg

When attaching gdb to the executable, do not execute run.

-hal

Run Incisive HDL Analysis (HAL) on the generated snapshot.

If you include the `-hal` option on the command line, the source files are compiled and the design is elaborated. *irun* then invokes HAL (instead of the simulator) on the snapshot.

By default, a summary report of the checks is printed to STDOUT and a verbose report, which reports all checks, is printed to the *irun* log file (`irun.log`, by default). Include the `-messages` option if you want verbose output printed to the screen.

You can use the `-log_hal` *`filename`* option to redirect HAL output to a specified file. For example:

```
% irun -hal -log_hal hal.log [other_options] files
```

If you include the `-gui` option with `-hal`, *irun* invokes the NCBrowse GUI rather than the SimVision GUI. See the *NCBrowse User Guide* for details on using NCBrowse.

All HAL command-line options are supported and can be included directly on the `irun` command line.

-h and help*

irun includes an extensive help system with many help options that let you display a list of all valid command-line options (`-helpall`), recognized file types and their default file extensions (`-helpfileext`), all options related to a particular subject or executable (`-helpsubject`), aliases for options (`-helpalias`), the minimum characters that must be entered for an option (`-helpshowmin`), and so on.

Use the `-helphelp` option to display a list of all options that control help.

```
% irun -helphelp
```

See "Getting Help on Command-Line Options" on page 13 for a description of all options related to help.

-ial

Specifies that you are using the Incisive Assertion Library (IAL) components in your design. Also specifies one of the following configuration parameters to use: Property Specification Language assertions (PSL) or SystemVerilog Assertions (SVA).

- `PSL`–To specify the PSL version of the IAL components
- `SVA`–To enable SVA assertions
- `PSL:COVERAGE`–To enable coverage gathering
- `PSL:DETAILED_COVERAGE`–To enable detailed coverage gathering

For example:

```
irun -ial PSL [other_options] ...
```

```
irun -ial SVA [other_options] ...
irun -ial PSL:COVERAGE +sv [other_options] ...
irun -ial PSL:DETAILED_COVERAGE +sv [other_options] ...
```

See Single-Step Mode in the *Incisive Assertion Library Reference* for details.

-ida

Enable the Debug Analyzer environment through *irun*.

The Incisive Debug Analyzer is a post-process, graphical debugger. You run the simulation once to gather data about both the DUT and the testbench. After simulation is complete, you can load the data into the Debug Analyzer for analysis. Because the simulation data is stored in a database, you do not need to rerun the simulation in order to debug the test environment. The `-ida` option enables the settings needed to allow the debug analyzer to record from the *irun* command line.

See Using Incisive Debug Analyzer for more information.

-l filename

Use the specified name for the log file instead of the default name `irun.log`. For example:

```
% irun -l mylog.log [other_options] source_files
```

Note: `-logfile` is an alias for `-l`.

-layout name

Launch SimVision with a built-in layout.

In the current release, the only built-in layout is the CDebug layout, which is useful for debugging C source code. The `-layout cdebug` option arranges the windows with the CDebug layout, and launches the GUI if needed.

```
% irun -sysc -layout cdebug -access r mem.sv main.cpp
```

Note: The `-layout cdebug` option is the same as the `-scgui` option in previous releases.

You can also specify `-layout none`, which deselects the previous layout style.

See "SimVision CDebug Layout" in the *SystemC Simulation User Guide* for more information.

-liborder

Scan for definitions of unresolved instances in library files and library directories as they follow on the command line. If an instance is still unresolved, scan the remaining library files and directories on the command line and then wrap around to preceding libraries not yet visited.

When *irun* finds an instance of a module or user-defined primitive (UDP) that cannot be resolved in the source description files, it scans for a definition in the library files, specified with `-v` on the command line, or library directory files, specified with `-y` on the command line. Once *irun* finds a definition, it resolves the reference and ignores all subsequent definitions of the module or UDP that it encounters in library files or library directories.

The default scan precedence that *irun* uses to search for definitions in library files or directories is as follows:

- If the unresolved instance is in a source file, *irun* scans library files or directories in the order in which they are entered on the command line. It begins with the left-most library file or directory, no matter where the source file appears on the command line. After *irun* scans the left-most library, it scans the others in the order in which they appear on the command line.

  Consider the following command-line example:

  ```
  irun src1.v -y /usr/lib/NMOS \
  src2.v -v usr/lib/TTL/ttl.v -y /usr/lib/CMOS
  ```

  In this example, if source file `src2.v` instantiates a module that is not defined in source files `src1.v` or `src2.v`, *irun* scans for a definition first in `/usr/lib/NMOS`, then in `/usr/lib/TTL/ttl.v`, and finally in `/usr/lib/CMOS`.

- If the unresolved instance is in a library file or library directory file, *irun* scans in the following order:

  - It checks the library file or directory that contains the unresolved instance.
  - If the instance remains unresolved, it then scans the remaining libraries, beginning with the one that follows the library containing the unresolved instance. It continues in a circular manner, scanning the libraries as they follow on the command line and then wraps around to the left-most library until it visits each one.

  ```
  irun src1.v -y /usr/lib/NMOS \
  src2.v -v usr/lib/TTL/ttl.v -y /usr/lib/CMOS
  ```

  If the module is neither instantiated nor defined in either source file, but is instead instantiated in the library file `/usr/lib/TTL/ttl.v`, then *irun* looks for a definition first in `/usr/lib/TTL/ttl.v`, then in `/usr/lib/CMOS`, and finally in `/usr/lib/NMOS`.

The `-liborder` option lets you order the library search according to where the first instance of an unresolved module is detected.

- If the unresolved instance is in a source file, `-liborder` directs *irun* to start searching in the library file or directory immediately following that source file.

  ```
  irun src1.v -y /usr/lib/NMOS \
  src2.v -v /usr/lib/TTL/ttl.v -y /usr/lib/CMOS -liborder
  ```

  Suppose that the compiler detects an instance of the unresolved module `dff` in the source description `src2.v`. To resolve the module, *irun* first searches for a description of `dff` in `/usr/lib/TTL/ttl.v`. If the module remains unresolved, the search continues in `/usr/lib/CMOS`, then in `/usr/lib/NMOS`.

- If the module instance is detected in a library file, `-liborder` initiates the search in that library. If the module remains unresolved, *irun* scans the remaining library files and directories in a circular order as they follow on the command line and then wraps around to preceding libraries that it has not yet visited.

  ```
  irun src1.v -y /usr/lib/NMOS \
  src2.v -v /usr/lib/TTL/ttl.v -y /usr/lib/CMOS -liborder
  ```

  Suppose that the compiler finds an unresolved module `ttl` in library `/usr/lib/TTL/ttl.v`. The `-liborder` option directs the search for a description of `ttl` in the following manner:

  a. Scan library file `/usr/lib/TTL/ttl.v`.

  b. If `ttl` remains unresolved, scan library files in the directory `/usr/lib/CMOS`.

  c. If `ttl` is still unresolved, scan library files in the directory `/usr/lib/NMOS`.

Note: You cannot use the `-liborder` option with the `-librescan` option.

Note: The behavior of the `irun -liborder` option is the same as the Verilog-XL `+liborder` option. See "Library Scan Precedence: The Former Scheme" in the *Verilog-XL User Guide* for more details on this option.

-librescan

Scan for definitions of unresolved module and UDP instances in library files (`-v`) and library directories (`-y`), beginning with the left-most library specified on the command line, and continuing through the remaining libraries from left to right.

The behavior of `-librescan` depends on the location of the undefined instance. That is, it depends on whether the instance is located in a source file, a library file, or a file within a library directory.

- When the undefined instance is located in a source file, `-librescan` acts the same as the default scan precedence. That is, scanning begins with the left-most library on the command line and continues through the remaining libraries from left to right.
- When the undefined instance is located in a library file, and `-librescan` is in effect, *irun* does not continue to search the library file for the matching definition; instead, it begins to scan through the left-most library on the command line, and then scans the remaining libraries in the order in which they appear.
- When the undefined instance is located in a library directory file, and `-librescan` is in effect, *irun* scans the library directory file first to try to resolve the instance. If the instance remains unresolved, it begins to scan the left-most library on the command line, followed by the remaining libraries in the order that they appear.

The following example includes three library files: `lib.orig.v`, `lib.revised.v`, and `lib.latest.v`. Library `lib.orig.v` contains the original versions of all the modules. Library `lib.revised.v` contains revised versions of many of the modules from `lib.orig.v`. The library `lib.latest.v` contains the latest revisions of just a few of the modules. To resolve all undefined instances with the most up-to-date modules, use the following command line:

```
irun source.v \
-v lib.latest.v \
-v lib.revised.v \
-v lib.orig.v -librescan
```

In this example, if `lib.orig.v` instantiates a module that is defined in that library, *irun* looks for a definition of the module instance first in `lib.latest.v`, then in `lib.revised.v`, and finally in `lib.orig.v`.

Note: You cannot use the `-librescan` option with the `-liborder` option.

Note: The behavior of the `irun -librescan` option is the same as the Verilog-XL `+librescan` option. See "Library Scan Precedence: The Former Scheme" in the *Verilog-XL User Guide* for more details on this option.

-location

Print the location of the installation and exit. For example:

```
% irun -location
Location of installation: /project/ius61/install
```

-log_* filename

Redirect log information for the specified executable from `irun.log` to a logfile with the specified name.

By default, all log information is written to `irun.log`. In some cases, you might want to redirect the output of some executable to its own log file. The following options are available:

`-log_amsspice` *`filename`*	Redirect amsspice output to the specified log file.
`-log_hal` *`filename`*	Redirect HAL output to the specified log file.
`-log_iev` *`filename`*	Redirect IEV output to the specified log file.
`-log_ifv` *`filename`*	Redirect IFV output to the specified log file.
`-log_ncelab` *`filename`*	Redirect *ncelab* output to the specified log file.
`-log_ncprotect` *`filename`*	Redirect *ncprotect* output to the specified log file.
`-log_ncsc_run` *`filename`*	Redirect *ncsc_run* output to the specified log file.
`-log_ncsim` *`filename`*	Redirect *ncsim* output to the specified log file.
`-log_ncvhdl` *`filename`*	Redirect *ncvhdl* output to the specified log file.
`-log_ncvlog` *`filename`*	Redirect *ncvlog* output to the specified log file.
`-log_svpp` *`filename`*	Redirect *svpp* output to the specified log file.

For example, the following command includes the `-log_ncvlog` and `-log_ncvhdl` options. The output of *ncvlog* will be written to `ncvlog.log`, and the output of *ncvhdl* will be written to `ncvhdl.log`.

```
% irun -v93 -log_ncvlog ncvlog.log -log_ncvhdl ncvhdl.log top.v sub.v middle.vhd
```

-makelib path_to_library[:logical_name] source_files [-endlib | -end]

Compile the files that follow the option into the specified library.

By default, all design units in HDL source files are compiled into the default `worklib` library (located within the `INCA_libs` directory tree). Use the `-makelib` option if you want to compile different files into different libraries.

The `-makelib` option precompiles design units in the specified files into a reference library. When top-level design files are compiled, the reference library is scanned for components instantiated in the design.

The list of files to be compiled into a specified library is terminated by:

- A `-end` or `-endlib` option
- Another option that specifies a collection of files to be processed together. These options are:
  - Another `-makelib` option
  - `-cpost`
  - `-snstage`

The files to be compiled into a specified library can be listed directly on the command line following the `-makelib` option, or they can be listed in a file specified with the `-f` or `-F` option. For example:

```
% irun -compile -v93 -makelib mylib file1.vhd file2.vhd -endlib ....

% irun -compile -v93 -makelib mylib -f mylib_files.txt -endlib ....
```

You can specify command-line options within a `-makelib` collection. The specified options are applied only to the source files within the collection.

See "Compiling into Multiple Libraries" on page 101 for details on using the `-makelib` option.

-mltypemap_input input_file -mltypemap_tcl

Generate data types in a target language based on data types in a source language. These data types can then be used as data types in ML UVM transactions using ML UVM TLM ports.

For more information, see the "mltypemap Utility" chapter of the *UVM Multi-Language Reference* document.

-ncdebug

Turn on read access to all objects. This is equivalent to `-access +r`.

-ncelabexe path_to_ncelab

Invoke the specified elaborator when spawning *ncelab*. Use this option when an elaborator with statically linked PLI must be used.

-ncerror warning_code[:warning_code ...]

Increase the severity level of the specified warning message from warning to error. The *warning_code* argument is the message code (mnemonic) that appears in the warning message following the severity code.

Example:

```
% irun -ncerror ABCDEF ....
```

You can increase the severity level of multiple warning messages either by using multiple `-ncerror` options or by using one `-ncerror` option and separating the *warning_code* arguments with a colon. For example,

```
% irun -ncerror ABCDEF -ncerror HIJKLM ....
% irun -ncerror ABCDEF:HIJKLM ....
```

-ncfatal {warning_code | error_code}[:{warning_code | error_code} ...]

Increase the severity level of the specified warning message or error message from warning or error to fatal. The *warning_code* or *error_code* argument is the message code (mnemonic) that appears in the message following the severity code.

Example:

```
% irun -ncfatal ABCDEF ....
```

You can increase the severity level of multiple warning messages or error messages to fatal either by using multiple `-ncfatal` options or by using one `-ncfatal` option and separating the *warning_code* or *error_code* arguments with a colon. For example,

```
% irun -ncfatal ABCDEF -ncfatal HIJKLM ....
% irun -ncfatal ABCDEF:HIJKLM ....
```

-nclibdirpath path

Specify the relative path for new libraries.

* irun flexible directory configuration

* nclibdirname

* -R recompilation

* Anything else?

By default, irun creates a scratch directory called INCA_libs.

The `-R` option lets you simulate the same snapshot multiple times using different simulator command-line options or Tcl command input files. You can prebuild a snapshot (by using the `-c` or `-elaborate` option) and then use the `-R` option to simulate that snapshot multiple times. For example:

```
% irun -R -input file1.tcl

% irun -R -input file2.tcl
```

-ncprotect_file filename

Pass the arguments in the specified file to *ncprotect*.

You can automatically invoke *ncprotect* to encrypt the Verilog and VHDL source files specified on the command line in two ways:

- Include the `-autoprotect` option on the `irun` command line. This option invokes *ncprotect*, which encrypts the entire source file(s).
- Use the `-ncprotect_file` option to specify a file that contains *ncprotect* options.
  - The arguments file can contain the `-autoprotect` option, in which case *ncprotect* encrypts the entire source file(s).
  - If the `-autoprotect` option is not included in the file (or on the command line), *ncprotect* encrypts the regions marked for encryption with protection pragmas in the source files.

The arguments file can contain any valid *ncprotect* option except for:

- `-decrypt with eif`. This option specifies an EIF (encryption information file) that is used to convert the specified encrypted files back to clear text files.
- `-outname`. This option generates a single encrypted file with the specified name.

For example, the following arguments file, `ncprot.args`, contains four *ncprotect* options, including `-autoprotect`:

```
# File ncprot.args
-autoprotect
-messages
-overwrite
-outdir ./encip

% irun -ncprotect_file ncprot_args -v93 top.v middle.vhd sub.v
```

This command invokes *ncprotect* to encrypt the files using the following commands:

```
ncprotect -autoprotect -messages -overwrite -outdir ./encip -lang vlog sub.v top.v
ncprotect -autoprotect -messages -overwrite -outdir ./encip -append_log -lang vhdl
middle.vhd
```

After encrypting the files, *irun* compiles the protected files (`./encip/top.vp`, `./encip/sub.vp`, and `./encip/middle.vhdp`), elaborates the design, and simulates the snapshot.

The following arguments file does not include `-autoprotect`. *irun* will invoke *ncprotect*, which encrypts the protected regions in the source files.

```
# File ncprot.args
-messages
-overwrite
-outdir ./encip
```

```
% irun -ncprotect_file ncprot_args -v93 top.v middle.vhd sub.v
```

See IP Protection for details on *ncprotect*.

-ncsimexe path_to_ncsim

Invoke the specified simulator when spawning *ncsim*. Use this option when a simulator with statically linked PLI must be used.

-ncuid ncuid_name

Use the specified unique ID name to identify the current run.

The `-ncuid` option lets you run, either sequentially or in parallel, multiple simulations using the same intermediate objects and using the same storage location so that you can save disk space as well as compilation and elaboration time.

During regression testing, in which there are typically many testbench modules that all instantiate the same design, you can assign a unique ID name to each run using the `-ncuid` option and then run these jobs in parallel or in sequence.

The *`ncuid_name`* argument must consist of only case-sensitive alphanumeric characters and the underscore character.

For example, suppose that you have three testbenches that test the same design. You can invoke the simulations with the following commands:

```
% irun tbench1.v -y ./libs -y ./models +libext+.v -ncuid test1
% irun tbench2.v -y ./libs -y ./models +libext+.v -ncuid test2
```

```
% irun tbench3.v -y ./libs -y ./models +libext+.v -ncuid test3
```

When you run *irun* (in parallel or in sequence), each run reuses existing intermediate objects if these objects are the objects that the process needs. New objects are generated if they are required but do not exist. If an object exists, but is not the one that the current process requires, *irun* automatically detects that the new object will overwrite pre-existing data and renames that object using the *ncuid_name* so that the new object does not overwrite existing data.

This functionality only affects data that is both written and read by the tools for the purpose of compilation, elaboration, or simulation. This data includes:

- The contents of the library system.
- The contents of the INCA_libs directory, including the INCA_libs/irun.nc invocation information.
- The compiled SDF file.

Note: Output data, such as log files and waveform databases, are not affected by the -ncuid option. *irun* does not detect that these files will be overwritten, and does not rename these files. You must use mechanisms, such as -l *logfile*, $test$plusarg(), and mc_scan_plusargs(), to rename these files to ensure that output files are uniquely identified to avoid any data collision from multiple invocations.

The -ncuid functionality creates new objects and renames them if necessary to avoid overwriting existing data. *irun* renames different objects in different ways. For example:

- The INCA_libs/irun.nc invocation information directory is always renamed to INCA_libs/*ncuid_name*.nc because each invocation must have a unique invocation directory. For example, the following command generates INCA_libs/test1.nc:

  ```
  % irun tbench1.v -ncuid test1
  ```

- Snapshots are always named lib.cell:*ncuid_name* because *irun* must generate a unique snapshot for each unique user-supplied name. For example, the following command generates a snapshot called worklib.top:test1 (assuming that the top-level module is called top):

  ```
  % irun tbench1.v -ncuid test1
  ```

- *irun*'s automatic SDF annotation requires a compiled SDF file. The compiled SDF file generated by the first run is called *sdf_filename*.X. In subsequent runs, *irun* uses this file if it is the file that the current process needs. If the current process requires a different SDF file, that SDF file is compiled and is renamed *sdf_filename.ncuid_name*.X (for example, dcache.sdf.test1.X).

You can use the `-R` option to rerun a simulation with a unique ID. For example, suppose that you have run two simulations on the same design using different testbenches, as follows:

```
% irun source.v -ncuid test1
% irun source.v -ncuid test2
```

irun has generated two snapshots in the `INCA_libs/worklib` directory: `worklib.top:test1` and `worklib.top:test2`. You can rerun the second simulation using the following command:

```
% irun -R -ncuid test2 [different_simulator_options]
```

-noelab

Do not invoke the elaborator (*ncelab*).

If you include this option, source files specified on the command line are compiled if necessary, elaboration is skipped, and the simulator is invoked if a snapshot exists.

This option is useful when updating SystemC, ***e***, DPI, VPI, PLI, and VHPI code where invoking the elaborator may not be required. Files specified on the command line are recompiled and you can proceed directly to simulation to debug the non-HDL updates.

If you only want to compile files use the `-compile` option.

If you want to compile files and invoke *ncelab* to generate a snapshot, use the `-c` or `-elaborate` option.

-nolibcell

Do not tag modules extracted from libraries (from `-y`, `-v`, or `` `uselib ``) as cells.

By default, *irun* tags modules extracted from libraries (from `-y`, `-v`, or `` `uselib ``) as cells, as if the modules were enclosed with `` `celldefine ``/`` `endcelldefine `` directives. Use the `irun -nolibcell` option to override this behavior.

Note: Modules surrounded with `` `celldefine ``/`` `endcelldefine `` are still treated as cell modules.

-nolog

Do not generate a log file.

-noprimupdate

Do not recompile or re-elaborate primary snapshots in this design that were built with separate `irun` commands.

By default, *irun* will look at primaries built with other irun commands and re-run those commands to bring the primary snapshots up-to-date if necessary before re-elaborating the simulation snapshot. Use this option to stop this default behavior. In cases where these primaries are used in other simulation snapshots, it may not be desirable to update them.

For example:

```
% irun -v93 top.vhd -top top -primname dut -noprimupdate
```

-noremovescratch

Do not remove the scratch directory for simulation.

irun creates a scratch directory to hold the simulation arguments that are passed to *ncsim*. By default, this directory is deleted when *irun* has completed. Use the `-noremovescratch` option to prevent the directory from being deleted. An informational message is printed telling you the name of the scratch directory.

-o0, -o1, -o2, -o3

These options map to the same options on the different C compilers that are used to compile any C/C++ code on the *irun* command line. For example, the following command instructs GCC to use the -O3 optimization when compiling the file `test.c`.

```
% irun test.v test.c -plimapfile test.pmf -O3
```

-ovl

Specifies that you are using the Open Verification Library (OVL) version of the Incisive Assertion Library components.

To use the OVL components, use the `irun` command and specify the `-ovl` and `+sv` options. Also specify one of the following SVA configuration parameters:

- `SVA` to enable SystemVerilog assertions
- `SVA:COVERAGE` to enable coverage gathering
- `SVA:DETAILED_COVERAGE` to enable detailed coverage gathering

For example:

```
irun -ovl SVA +sv [other_options] ...
irun -ovl SVA:COVERAGE +sv [other_options] ...
irun -ovl SVA:DETAILED_COVERAGE +sv [other_options] ...
```

See “Using the IAL Library for OVL Users” in the *Incisive Assertion Library Reference* for details.

-perfstat

Generate a report of performance metrics.

Use the `-perfstat` option if you want to produce a summary of performance statistics using *irun*. The `-perfstat` option creates a single log file (`ncperfstat.out`) in the current working directory. This file includes the following details:

- Compilation Time and Memory Usage
- Elaboration Time and Memory Usage
- Simulation Time and Memory Usage
  - License Acquisition Time During Simulation
  - Snapshot Loading Time and Memory Usage
  - Coverage Dumping Time (If Enabled)

Example:

```
% irun -perfstat board.v counter.v clock.v ff.v
```

-prefix_ncsim

Execute the specified argument before *ncsim* when using *irun*.

The argument given to `-prefix_ncsim` must be a valid executable to run the simulator. For example:

```
% irun race.v -prefix_ncsim collect
```

In this example, the argument `collect` generates additional debug information after compiling and elaborating the design, and then starts *ncsim*. However, if the argument is not a valid executable, then *irun* will halt after elaboration and generate a warning.

-prep

Run *irun* in *prep* mode.

When you include the `-prep` option on the `irun` command line, *irun* processes all source files and command-line options, and then generates a script that contains all of the commands necessary to run each executable needed to simulate the design.

When run in prep mode, *irun* does not run the executables to simulate the design. Prep mode creates the output script (and other required files) and exits.

Irun prep mode provides a very easy way to automatically generate a multi-step script. The output script can be run to compile, elaborate, and simulate the design in multi-step mode.

In addition, the output script documents all the steps needed to simulate the design. By clearly showing how each executable is run, the script can provide invaluable help in debugging problems with an *irun* invocation.

`irun -prep` generates the following files:

- `cds.lib`
- `hdl.var`
- *`executable`*`.args` files

  An arguments file is created for every executable that *irun* would need to run in normal mode.

- `RUN_NC`

  The output script contains all information required by each executable. Internal options that are used by *irun* when run in normal mode are not included in the script.

  For each executable, the script contains:

  - Commands to set any required environmental variables.
  - The executable invocation command. This command invokes the executable with the `-file` option to include the options and arguments in the relevant `.args` file. For example:

    ```
    ncelab -file ncelab.args
    ```

  - A check to ensure the executable finished without errors. If the executable exited on an error, the script will exit as well.

    ```
    if ($status != 0) then
      echo "Error detected while executing ncelab. Now exiting."
    ```

```
  exit
endif
```

- Commands to unset any environmental variables.

By default, all generated files are written to the current working directory. You can use the `-redirect` option to write them to a different location. For example:

```
irun -v93 -top worklib.top -prep -redirect ./multi_step \
sub.v middle.vhd top.v
```

The default name and location of the script can be changed by using the `-prep_name` option.

irun will not overwrite an existing `cds.lib`, `hdl.var`, `.args` file, or script. Include the `+overwrite` option to overwrite existing files.

Note: *irun* also has a `-debugscript` option, which generates a script that captures the environment variables and commands for the executables that were invoked by *irun*. The primary difference between the scripts is that the script generated by `-debugscript` contains internal environment variables and options used by *irun* when it invokes an executable. These internal variables and options are not included in the *irun* prep mode script.

-prep_name script_name

Use the specified name for the output script generated by the `irun -prep` option.

By default, the `-prep` option generates a script called `RUN_NC` in the current working directory. Use the `-prep_name` option to give the script a different name or to write the script to a different directory.

The following command generates a script called `RUN_NC` in the current working directory.

```
irun -v93 -top worklib.top -prep \
sub.v middle.vhd top.v
```

The following command generates a script called `RUN_SIM` in the `./multi_step` directory.

```
irun -v93 -top worklib.top -prep -prep_name ./multi_step/RUN_SIM \
sub.v middle.vhd top.v
```

The following command uses the `-redirect` option to write all generated files to a directory called `multi_step`. The output script is called `RUN_SIM`.

```
irun -v93 -top worklib.top -prep -prep_name RUN_SIM -redirect ./multi_step \
sub.v middle.vhd top.v
```

-propfile_sc filename
-propfile_vhdl filename
-propfile_vlog filename

Use the specified file containing SystemC/VHDL/Verilog PSL/Covergroup verification code.

-propsc_ext propfile_extension
-propvhdl_ext propfile_extension
-propvlog_ext propfile_extension

Use property files with the specified file extension.

This option (and the `-propdir` option) makes specifying specific property files easier if you are working with a design that uses a large number of property files.

The following command specifies that the compiler is to search the directory `./prop_files` for VHDL property files that have a file extension of `.prop`:

```
% irun -propdir ./prop_files -propvhdl_ext .prop other_options source_files
```

If a file extension is not specified with an option, the default is `.psl`.

You can use multiple property file extension options. The compiler will search for all of the specified extensions, in all of the directories specified with `-propdir`.

-q and -Q

Suppress informational messages.

irun prints messages from the compiler, elaborator, and simulator by default, including a list of command-line options used in the run. By default, messages are printed to a log file called `irun.log`. Use the `-l` option to rename the log file. Use `-nolog` if you do not want a log file.

Messages are also printed to the screen by default, except for the command-line options. Use `-nostdout` if you want to suppress printing to the screen.

You can use the -q or -Q option to suppress certain messages. The following table summarizes the differences between `-q` and `-Q`.

	STDOUT	Log File
Default (no `-q` or `-Q`)	Display tool banner and summary messages from tools. Do not display command-line arguments.	Print tool banner, command-line arguments, and summary messages from tools.
`-q`	Do not display tool banner, command-line arguments, or summary messages from tools.	Do not print tool banner, command-line arguments, or summary messages from tools.
`-Q`	Display tool banner. Do not display command-line arguments or summary messages from tools.	Print tool banner and command-line arguments. Do not print summary messages from tools.

-R

Simulate the last snapshot generated by an *irun* command.

The `-R` option lets you simulate the same snapshot multiple times using different simulator command-line options or Tcl command input files. You can prebuild a snapshot (by using the `-c` or `-elaborate` option) and then use the `-R` option to simulate that snapshot multiple times. For example:

```
% irun -R -input file1.tcl
% irun -R -input file2.tcl
```

Note: The snapshot must be generated by an `irun` command. You cannot use `irun -R` to simulate a snapshot created by a Tcl `save` command. Use `irun -r` *`snapshot_name`* to simulate a saved snapshot. You cannot use `irun -R` to simulate a snapshot generated by compiling the source files with *ncvlog* or *ncvhdl* and then elaborating the design with *ncelab*.

Note: If you run *irun* with the `-snapshot` option (or its synonym `-name`) to specify a name for the simulation snapshot, you must include the original `-snapshot` option when you invoke *irun* with `irun -R`. For example:

```
% irun -access +r -snapshot mysnapshot file.vhd file.v
```

```
% irun -R -snapshot mysnapshot
```

If `-snapshot` was not used in the original run, `-snapshot` should not be specified with `-R`. The following sequence generates an error:

```
% irun -access +r file.vhd file.v
% irun -R -snapshot default_snapshot_name
```

-r snapshot_name

Load and simulate the specified snapshot.

One common use for the `-r` option is to load a snapshot that you have saved with the Tcl `save` command. *irun* does not perform any kind of source file checking. The snapshot is simply loaded and simulated.

Note: The snapshot must be generated by *irun*. You cannot use `irun -r snapshot_name` to simulate a snapshot generated by compiling the source files with *ncvlog* or *ncvhdl* and then elaborating the design with *ncelab*.

Note: If you run *irun* with the `-snapshot` option (or its synonym `-name`) to specify a name for the simulation snapshot, you must include the original `-snapshot` option when you invoke *irun* with `irun -r snapshot_name`. For example:

```
% irun -tcl -snapshot mysnapshot file.vhd file.v
...
Writing initial simulation snapshot: worklib.mysnapshot:v
Loading snapshot worklib.mysnapshot:v ..................... Done
ncsim> run 500 ns
Ran until 500 NS + 0
ncsim> save sss:1
Saved snapshot worklib.sss:1
ncsim> run 500 ns
Ran until 1 US + 0
ncsim> save sss:2
Saved snapshot worklib.sss:2
ncsim> exit
% irun -snapshot mysnapshot -r sss:1
Loading snapshot worklib.sss:1 ..................... Done
```

-rebuild

Rebuild the executables for *ncelab* and *ncsim*.

The executables shipped with a given release are stripped down. By rebuilding them using the `-rebuild` option, you will have more symbol information available for debugging. After processing, the new rebuilt executables are used for elaboration and simulation.

-reflib path_to_library[:logical_name]

Add the specified library to the list of libraries to be searched.

You can precompile files into libraries with the `-makelib` option. For example:

```
% irun -v93 -compile -makelib ./libs/plib bv_images_p.vhd bv_images_pb.vhd
```

Use the `-reflib` option to reference the libraries when the entire design is being elaborated. The `-reflib` option adds the specified libraries to the list of libraries to scan.

The argument to `-reflib` is the path to the library as specified with the `-makelib` option. For example:

```
% irun -v93 -compile -makelib somearea/plib bv_images_p.vhd bv_images_pb.vhd
% irun -v93 -reflib somearea/plib inverter.vhd counter_4bit.vhd counter_32bit.vhd
counter_32bit_tb.vhd -top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

See "Compiling into Multiple Libraries" on page 101 for more information on using the `-reflib` option.

-saveenv

Save all shell environment variables used by *irun*.

This option saves the entire environment to a file that is automatically loaded by *ncelab* or *ncsim* when under a debugger. The option does not affect the simulation run.

For example, the following script sets the value of `VRST_HOME`, sources the setup script, sets some variables used by a VPI application, and then runs *irun*. By including the `-saveenv` option on the command line, you can go directly to a debugging session as shown below. You do not have to set `VRST_HOME` or any other variable that the `env.csh` script modifies.

```
#!/bin/csh -f
setenv VRST_HOME /some/install/path/specman
source $VRST_HOME/env.csh
# Set environment variables for VPI application
# ...
# ...
% irun -loadvpi my_pli_app:app_boot file.v file.vhd file.e -saveenv
...
```

```
...
% gdb ncsim
gdb> run -f INCA_libs/irun.nc/ncsim.args
```

Note: The value of the library path environment variable (`LD_LIBRARY_PATH`) is not updated. If the value of this variable does not match the value saved by *irun*, *ncelab* and *ncsim* generate an error telling you that the current value of the variable does not match the expected value. You can set the variable to the expected value at the `gdb>` prompt by cutting and pasting the output from the error message.

-sctop top_level_unit

Specify the top-level SystemC unit.

This option does not disable the automatic calculation of Verilog top-level units. When the `-sctop` option is used, and there are Verilog files on the command line, *irun* calculates the Verilog top-level modules from any compiled module definitions that are not instantiated in the design.

-seed seed

Pass a seed value to Specman and SystemVerilog.

The recommended options for specifying different seed values explicitly are as follows:

- Specman `-snseed`
- SystemVerilog/OVM/UVM `-svseed`

When working in a multi-language environment, you can use the `-seed` option for passing a global seed value to both `-svseed` and `-snseed`. If both the `-seed` and `-snseed` options are specified on the same command line, then the value of `-snseed` overrides the value of `-seed`. Similarly, the value of `-svseed` overrides the value of `-seed`.

See the `-snseed` option for information on passing seed values explicitly to Specman.

For details on `-svseed`, see *Random Constraints* in the System Verilog Reference.

-snapshot snapshot_name

Generate a snapshot with the specified name.

Note: Both `-snapshot` and `-name` are synonyms for the same option.

See "Specifying a Snapshot Name" on page 107 for more information.

-stdout_options

Print the specified *irun* options to the screen.

By default, *irun* captures all specified invocation options in the standard log file. You can use `-stdout_options` to print all options specified on the command line to the screen, as well as to the log file, which may be useful for debugging purposes.

For example:

```
% irun test.v -f args.f -stdout_options
irun: 14.10-p001: (c) Copyright 1995-2014 Cadence Design Systems, Inc.
...
irun
      test.v
      -f args.f
           -top top
           -sv
           Caching library 'worklib' ....... Done
...
```

If both the `-nostdout` option and `-stdout_options` are used on the command line, then `-nostdout` will take precedence and *irun* will not print any output to the screen. Similarly, if the `-quiet` option and `-stdout_options` are used on the command line, then *irun* will suppress the output of its invocation options to the log file as well as to the screen.

-sysc

Specify that SystemC is part of the design.

When simulating a design that includes SystemC, you must include the `-sysc` option on the `irun` command line.

-top [lib.]cell[:view]

Specify the top-level unit.

Use the `-top` option to specify the top-level HDL design unit to be elaborated and simulated. You can use multiple `-top` options to specify multiple top-level units.

If no `-top` options are specified, top-level design unit(s) are determined automatically from the Verilog and SystemVerilog sources. If the design has one or more VHDL top-level entities, or a SystemC top-level unit, these units must be specified with one of the following options:

- `-top`

  You can use `-top` to specify VHDL or SystemC top-level units. However, when you use the `-top` option, auto-detection of top-level Verilog modules is disabled, and multiple `-top` options must be used to specify all of the top-level units.

- `-vhdltop`

  Specifies the VHDL top-level unit to be elaborated and simulated. This option does not disable the automatic calculation of Verilog top-level units.

  Note: If there are multiple VHDL top-level entities, you must use multiple `-top` options to specify these tops. You cannot specify multiple VHDL top-level units with multiple `-vhdltop` options. See Elaborating a Design with Multiple Top-Level VHDL Design Units for information on elaborating a design with multiple VHDL top-levels.

- `-sctop`

  Specifies the SystemC top-level unit. This option does not disable the automatic calculation of Verilog top-level units.

For Verilog and VHDL AMS, the `-top` option must be used to specify connect modules and cds_globals.

-unbuffered

Do not buffer the output.

-v <filename>

Specify the name of the library file to use. The `-v` command-line option enables *irun* to scan a file for module and UDP definitions that are unresolved in the specified source file on the command line. For example:

```
irun source1.v -v libfile.v
```

See Chapter 6, “Library Management,” for more information.

-version

Display the version of *irun* and exit.

-vhdltop [lib.]cell[:view]

Specify the top-level VHDL unit to be bound to `:`. This option does not disable the automatic calculation of Verilog top-level units. When the `-vhdltop` option is used, and there are Verilog files on the command line, *irun* calculates the Verilog top-level modules from any compiled module definitions that are not instantiated in the design.

-write_metrics

Write run data into a verification session output file (VSOF) that can be loaded into Enterprise Manager for analysis. The `-write_metrics` option dumps vsof only in irun flow i.e. you use a single irun command to compile, elaborate and simulate the design.

The `VSOF_DIR` option specifies the directory location where the vsof file gets dumped,

In case there are multiple log files i.e. separate ones for compilation, elaboration and simulation, then the single vsof file will have following filter specifications;

```
filter "ncvlog" {
  filter_files : <text>ius.flt ovm_sv_lib.flt</text>;
  filter_id : <text>ncvlog</text>;
  log_file : <text>$ATTR(session_dir)/test/compile.log</text>;
        }
        filter "ncelab" {
  filter_files : <text>ius.flt ovm_sv_lib.flt</text>;
  filter_id : <text>ncelab</text>;
  log_file : <text>$ATTR(session_dir)/test/elab.log</text>;
        }
        filter "ncsim" {
  filter_files : <text>ius.flt ovm_sv_lib.flt</text>;
  filter_id : <text>ncsim</text>;
  log_file : <text>$ATTR(session_dir)/test/sim.log</text>;
```

-y <directory>

Specify the name of the library directory. When using the `-y` command-line option, you can name a directory that contains a single module or UDP definition file, or a set of complete Verilog HDL hierarchies.The following example shows how to use the `-y` option with *irun*:

```
irun source1.v -y /usr/me/proj/lib/cmos
```

You can also specify more than one library directory on the command line with multiple `-y` statements, as follows:

```
irun source1.v -y /usr/me/prij/lib/cmos -y /usr/you/proj/lib/cmos
```

If the files are hierarchical, then the top level of the hierarchy should be the first module declared in the file, and the other modules and UDPs should follow. All entries must conform to the ?library directory format?. See Chapter 6, “Library Management,” for more information.

-zlib compression_level

Compress the `.pak` file.

When you compile, elaborate, and simulate a design, the tools create or modify intermediate objects. All intermediate objects that are required by the NC tools are stored in a single database file in a library directory. This library database file is called `inca.`*`architecture.lib_version`*`.pak`. For example, the name of the library database file is similar to the following:

```
inca.sun4v.132.pak
```

For a large design, the `.pak` file can consume a significant amount of disk space. Use the `-zlib` option to compress the `.pak` file before it is written to disk.

The `-zlib` option is supported for the following tools:

- Verilog and VHDL parsers (*ncvlog* and *ncvhdl*)
- The SystemC *ncsc* utility
- The elaborator (*ncelab*)
- The simulator (*ncsim*)

With *irun*, the `-zlib` option is automatically passed to all appropriate tools.

The level of compression can be set from 1 to 9. For example:

```
% irun -zlib 1 ....
% irun -zlib 7 ....
```

A higher number results in a more highly compressed file, but performance can decrease because the tools must uncompress the file before reading it.

If no compression level is specified, a warning is issued and level 1 is used.

IXCOM Command-Line Options

The *irun* utility includes several options to compile a design for simulation acceleration and simulate hardware-accelerated snapshots.

Use the following commands to view a list of IXCOM command-line options:

Command	Description
`irun -helpsubject vlan`	Displays a list of `vlan` options and their description
`irun -helpsubject vhan`	Displays a list of `vhan` options and their description
`irun -helpsubject ixcom`	Displays a list of `ixcom` options and their description
`irun -helpsubject hwaccel`	Displays a list of all options supported by IXCOM (`vlan`, `vhan`, and `ixcom`) tools and their description

For more information about simulation acceleration using IXCOM tools, refer to *Compiling and Running Simulation Acceleration Designs with irun* in the *UXE User Guide*.

-compile_sw

Pass options and files specified between this option and the `-end` marker to the invoked IES tools.

-hw

Use IXCOM to compile and process the design for simulation acceleration.

You can use the `-hw` option on the command line with a limited number of Incisive tool options to compile the design, control messaging and other capabilities. Options that are ignored will generate a single warning message.

For example:

```
% irun -hw -linedebug -status -access +rwc counter.v
...
The following options are not used by vlan, vhan, ixcom
or IES compile process to generate the software model:
        -linedebug
        -status
        -access +rwc
```

-host <name>

Specify the name of the emulator to be used. The default value is '.', indicating the current machine.

-ixcomargs<string>

Dispatch options indicated by <string> directly to IXCOM, without any parsing by irun.

-vhanargs<string>

Dispatch options indicated by <string> directly to vhan, without any parsing by irun.

-vlanargs<string>

Dispatch options indicated by <string> directly to vlan, without any parsing by irun.

-log_ixcom <logfile>

Generate the `ixcom` output in the specified log file.

-log_vhan <logfile>

Generate the `vhan` output in the specified log file.

-log_vlan <logfile>

Generate the `vlan` output in the specified log file.

-xedebug

Use `xeDebug` as the run-time processor for the compiled snapshot.

-xedebugargs <string>

Supply arguments to xeDebug.

Note: All IXCOM (`vlan`, `vhan` and `ixcom`) options are recognized by irun command-line as new options. For a complete list of IXCOM options, refer to the IXCOM user guide.

OVM Command-Line Options

irun includes several options that support the Open Verification Methodology (OVM).

Use the following command to view a list of OVM command-line options:

```
% irun -helpsubject OVM
```

-enableoig

Enable the automatic processing of OVM Interface Generator (OIG) code.

Note: OIG is currently under deprecation and will be fully deprecated in a future release.

Cadence provides tools that facilitate easy reuse of e-OVC (Open Verification Components) in a SystemVerilog verification environment. OIG is a tool that automates much of the process of creating a SystemVerilog API to an existing e-OVC. With the SystemVerilog API in place, a SystemVerilog user can use the e-OVC as if it were a native SystemVerilog OVC.

The `-enableoig` option enables the capability to automatically compile the code generated by OVM Interface Generator.

The processing of OIG code by *irun* has a performance impact, and OIG code is not processed automatically by default. *irun* performs an additional checking of user SystemVerilog code, and if OIG code is encountered, it exits with the following Fatal error:

```
ncvlog: *F,ENAOIG: OIG code has been detected. Add -enableoig to the irun
commandline to enable OIG processing.
```

-ml_ovm

Enable the multi-language (ML-OVM) features.

Use this option to enable ML-OVM when no `-ovmtop` switch is used. This is useful in cases when you have a top ***e*** layer where the ***e*** test file is being loaded by other means (for example through a Tcl command) and a SystemVerilog OVC that is instantiated by other means than an `-ovmtop` switch.

The following rules apply to the `-ml_ovm` option:

- If you are using SystemVerilog components, you must have at least one import statement to import the `ml_ovm` package in your SystemVerilog source:

  ```
  import ml_ovm::*; // Required for ML-OVM
  ```

- When you specify the `-ml_ovm` option, the `-ovm` switch is implicitly added, causing *irun* to compile the OVM package that is included with your IES release.

For additional usage and methodology guidance, see the *OVM Multi-Language Reference*, available with the Incisive Verification Kits.

-ovm

Enable support for the Open Verification Methodology (OVM).

The `-ovm` option automates the use of OVM with the IUS install.

- Automatically loads an OVM Tcl interface
- Automatically loads an OVM GUI interface
- Automatically loads the definition of OVM-specific system tasks
- Automatically adds `-incdir <OVMHOME>/src`
- Runs *svpp* on SystemVerilog files, if needed
- Suppresses a specified set of warning messages
- Adds the definition of an OVM macro `URM_SV_ENABLE`

-ovmhome directory | string

Specify the location of the OVM installation.

By default, OVM is installed in <install_directory>/tools/methodology. Use the -ovmhome option to name a different location. There are two possible arguments:

- A valid path (relative or absolute) to a directory
- A character string that *irun* will use in an attempt to match with an existing directory in the installation hierarchy

For example, consider an installation with the following directories:

<*install_directory*>/tools/methodology/ovm/2.1
<*install_directory*>/tools/methodology/ovm/2.0
<*install_directory*>/tools/methodology/ovm/1.1

You can use `-ovmhome` with a direct path to set the location as shown:

```
% irun -ovmhome 'ncroot'/tools/methodology/ovm/1.1
```

Similarly, `-ovmhome` recognizes directory locations from prior releases. Use the following to select <*install_directory*>/tools/methodology/ovm/2.1:

```
% irun -ovmhome 'ncroot/tools/ovm-2.1
```

You can also use strings. Use the following to have *irun* match the 2.0 directory in the installation hierarchy:

```
% irun -ovmhome 2.0
```

-ovmlinedebug

Enable single-stepping through OVM functions and tasks.

By default, `-ovm -linedebug` does not apply the `-linedebug` option to the compilation of the OVM package. This means that when you are debugging your code, single-stepping through the OVM content is not enabled.

Include the `-ovmlinedebug` option if you want to single-step through the OVM content to debug OVM code.

```
% irun -ovm -linedebug -ovmlinedebug [other_options] source_files
```

-ovmnoautocompile

Disable automatic compilation of the OVM packages.

When support for OVM is enabled with the `-ovm` option, the OVM packages are, by default, automatically compiled in a separate step before any HDL files specified on the command line. Use the `-ovmnoautocompile` option to prevent compilation of the OVM packages.

For example, suppose that you have created a local copy of the `ovm_pkg.sv` package because you wanted to modify the file in some way (to include Cadence transaction recording, to include defines, and so on). You can then include the local copy on the command line, and use `-ovmnoautocompile` to turn off the automatic compilation of the package that is in the OVM installation hierarchy.

```
% irun –ovmnoautocompile ./ovm_pkg.sv –ovmhome /local/tools/ovm-2.1.1 myfiles.sv
```

-ovmtest [language:]entity_name ...

Enable multi-language OVM (ML-OVM) and declare a top test entity in a multi-language verification environment.

Use this option to declare the test, which is the logical root of the entire multi-language testbench hierarchy. You can specify only one entity with the `-ovmtest` option, and it can be in any one of the supported languages.

Use this option to specify either your ***e*** test file name, or SystemVerilog/SystemC test class. If your root is a SystemVerilog test class, an instance of it is created with the name `ovm_test_top`, as in a pure SystemVerilog testbench.

The `-ovmtest` option takes an entity name consisting of a language identifier and an entity name. The language identifier marks the language domain of the top entity. It can be one of the following:

- `sv` (or `SV`) for SystemVerilog
- `e` (or `E`) for ***e***
- `sc` (or `SC`) for SystemC

The *`entity_name`* can be one of the following:

- ***e*** test file name
- OVM SystemVerilog class name, representing a test (or another type derived from ovm_component)
- SystemC class name

The following rules apply to the `-ovmtest` option:

- Can be used only once in a command.
- You can omit the `e:` language identifier if you specify an ***e*** test file name with a `.e` suffix. If you omit the language identifier and the name has no `.e` suffix, it is assumed to be a SystemVerilog class name. You cannot omit the `SC` language identifier.

  Cadence recommends that you always include a language identifier, for clarity and readability.
- The `-ovmtest` option replaces the `+OVM_TESTNAME` switch in multi-language environments. It is an error to use both `-ovmtest` and `+OVM_TESTNAME` in the same irun command line.
- You cannot specify an ***e*** unit type name with `-ovmtest`. Only ***e*** test files are supported as top entities in ***e***.
- If you specify `-ovmtest`, the `-ovm` and `-ml_ovm` command line options are turned on and need not be specified explicitly.

Examples

Naming an ***e*** test file as a top entity:

```
% irun -ovmtest e:my_test.e ...
```

Naming an OVM SystemVerilog test class (similar to the pure OVM SystemVerilog `+OVM_TESTNAME` switch) as a top entity:

```
% irun -ovmtest SV:my_test_class ...
```

Naming two top entities, an ***e*** test and an OVM SystemVerilog env class:

```
% irun -ovmtest e:my_test.e \
       -ovmtop SV:my_ethernet_env ...
```

For additional usage and methodology guidance, see the *OVM Multi-Language Reference*, available with the Incisive Verification Kits.

-ovmtop [language:]entity_name ...

Enable multi-language OVM (ML-OVM) and declare a top entity in a multi-language verification environment. Use this option to name entities which are top entities in their language domain, because their parent entity is in another language domain.

Cadence recommends that you use the `-ovmtest` option and not `-ovmtop` to name the test, the root of the logical testbench hierarchy.

Use this option to declare a top entity in one of the language domains. To specify several top entities, use the option multiple times, each naming one of the entities. Each of the entities you name has a parent entity in another language domain, and therefore is a top entity (no parent) in its own language domain. This option should not be used for the root entity of the logical testbench hierarchy, where `-ovmtest` should be used instead.

The `-ovmtop` option takes an entity name consisting of a language identifier and an entity name. See the description of `-ovmtest` for a description of the argument.

The following rules apply to the `-ovmtop` option:

- The order of top entities as declared on the command line determines the order in which they are created and built. Therefore, make sure to name the test as the very first top entity.
- You can omit the `e:` language identifier if you specify an ***e*** test file name with a `.e` suffix. If you omit the language identifier and the name has no `.e` suffix, it is assumed to be a SystemVerilog class name. You cannot omit the SC language identifier.

  Cadence recommends that you always include a language identifier, for clarity and readability.
- You cannot specify an ***e*** unit type name with `-ovmtop`. Only ***e*** test files are supported as top entities in ***e***.

- If you specify `-ovmtop`, the `-ovm` and `-ml ovm` command line options are turned on and need not be specified explicitly.

Example

Naming two top entities, an ***e*** test root and an OVM SystemVerilog env class:

```
% irun -ovmtest e:my_test.e \
       -ovmtop SV:my_ethernet_env ...
```

For additional usage and methodology guidance, see the *OVM Multi-Language Reference*, available with the Incisive Verification Kits.

Specman Command-Line Options

irun includes several options to control Specman compilation, loading, and simulation.

Use the following command to view a list of Specman command-line options:

```
% irun -helpsubject specman
```

-e_ext [+].extension[,.extension...]

Override the built-in file extensions used to recognize the **e** source files on the command line. For example, the default file extensions for ***e*** files is `.e` or `.E`. If you have ***e*** files with other extensions, you must specify that these are valid extensions for ***e*** files.

With the `-e_ext` option, you can:

- Replace the list of built-in, predefined extensions with a new list. For example, the following option specifies that the valid extensions for ***e*** files are `.e and .spmn`:

  ```
  -e_ext .e,.spmn
  ```

- Add extensions to the list of built-in, predefined extensions by using a plus sign (`+`) before the list of extensions to add. For example, the following option adds `.spmn` to the list of default ***e*** extensions:

  ```
  -e_ext +.spmn
  ```

  This produces the same list of default extensions as the previous example.

See "Changing the Default Set of File Extensions" on page 97 for more information on using an extension option to override the default set of file extensions.

-end

Mark the end of a list of files. You can use this option in place of `-endsnstage`.

-intelligen

Configure the generator to use IntelliGen.

-nosncomp

Do not compile Specman input files specified on the command line.

If this option is used, the ***e*** files on the command line are treated as if they were prefixed with the `-snload` option. This option is useful when you want to debug the results of an earlier simulation, because you can add it to the end of the command line you used to invoke the earlier simulation without having to edit the command line to add `-snload`.

For example:

```
irun tb.v test_1.e
irun tb.v test_1.e -gui -nosncomp
```

-pgen

Configure the generator to use Pgen. The default generation engine is IntelliGen. Use the `-pgen` option when you must run the Pgen generation engine instead.

Note: The Pgen generation engine does not support precompiled ***e***-library files (`.elib`). Attempting to specify an `.elib` file with the `-pgen` option on the command line will result in an error.

-snchecknames

Generate a warning at elaboration time if Specman attempts to access a wrong path of a HDL entity.

Specman, being a testbench tool of a design written in HDL, includes predefined interfaces to access the needed HDL entities in the design. Various checks are performed to ensure that the paths of the HDL entities that Specman is trying to access are correct. By default, these HDL access checks take place at simulation time, after the sometimes lengthy Specman initialization phase and other Specman phases have completed, and when both Specman and the simulator are running. If Specman is trying to access a wrong path of an HDL entity, an error is generated.

If you use the `-sncheckknames` option, the HDL access checks are performed at elaboration time instead of at simulation time. The elaborator generates warnings if mismatches are found.

-sncompargs string

Pass arguments to the ***e*** compilation script, `sn_compile.sh`. This option is used for incremental compilation to pass the name and location of the extended Specman executable that was created in a previous step.

For example:

```
sn_compile.sh -t ./sn_tmp -pic -o my_macros my_macros.e
irun tb.v test_1.e -sncompargs "-s ./my_macros"
```

-sndynload e_file [e_file ...]

Load the specified ***e*** file(s) after loading a saved snapshot.

Note: This option can be used only when loading a saved snapshot with the `-r` option.

The `-sndynload` option lets you:

1. Run your test in a specific configuration until the DUT has been initialized or until some other point of interest has been reached.
2. Save the current state of the simulation.
3. Restart the saved snapshot in multiple runs with additional ***e*** files. Each run continues from the saved state, but with a different scenario. The code in the specified ***e*** files will only affect the simulation from this point onwards. For example, new constraints will not affect already-generated structs.

Note: The additional ***e*** files specified with `-sndynload` are called *Dynamic Loadable Files* (DLF). These files can contain only a subset of ***e***.

Note: The `-sndynload` option requires a Specman_Advanced_Option license, and works only when the Specman generator is set to Intelligen.

When restarting a simulation with a saved snapshot, you can also use the `-snseed` option to change the seed value for the simulation.

In the following example, the simulation is run for 500 ns and then a snapshot called `snap1` is saved. *irun* is then invoked with the saved snapshot. The DLF file `test.e` is loaded and the seed value is set to `random`.

```
% irun -tcl xor.v -snset "config gen -default_generator=IntelliGen" base.e
ncsim> run 500ns
ncsim> save snap1
ncsim> exit
%
% irun -r snap1 -sndynload test.e -snseed random
```

-sndynnow

(Rarely used) Resolve all undefined symbols when loading the library (RTLD_NOW). By default, undefined symbols are resolved at runtime (RTLD_LAZY) to give better performance.

-snfaststub on | off

Enable one-step generation of the Specman stubs file. The default value is `off`.

Use this option to choose between the old and new behavior for stubs file generation.

When running *irun* using the old behavior, the software calls Specman twice to generate the stubs file. The first call builds the stubs file using an `agent` request in the ***e*** code. If no `agent` request is found, then *irun* will generate Verilog and VHDL stubs using the second call.

To improve performance in complex verification environments, use the new method of Sepcman stubs file generation. This method calls Specman just once. For example:

`irun [`*`other_options`*`] -snfaststub on tb.v test.e`

Note: In a future release, Cadence plans to change the default value of the `-snfaststub` option to `on`. After that, this option will be in deprecation and *irun* will automatically perform one-step generation of the stubs file.

-snfliheader filename

Create a header file with the specified name to be used with the Specman C function-level interface. After creating the header file, *irun* compiles and links the C and the ***e*** code into a shared library.

You must enclose this option in the same stage as the C and ***e*** source files, by using the `-snstage` and `-endsnstage` options. For example:

```
irun -snstage stage1 test.c top.e -snfliheader top.h -endsnstage
```

In this example, `irun` creates a C function-level interface header file named `top.h`. It then compiles `test.c` and `top.e` into a shared library, and invokes Specman with that library.

For more information, see Using the C Function-Level Interface in the *Specman Integrator's Guide*.

-snglobalcompargs string

Pass arguments to the ***e*** compiler. The arguments are applied globally, that is, to all modules in the compilation and across all stages.

For example, in the following `irun` command, the `-snglobalcompargs` option passes the `-D` option to the ***e*** compiler, which applies the option to both `t1.e` and `t2.e`, regardless of the fact that they are in different compilation stages:

```
% irun -snstage stage1 t1.e -endstage t2.e -snglobalcompargs "-D ERR"
```

For more information, see "Passing Arguments to sn_compile.sh from irun" in *Running Specman with Incisive Simulator*.

-snheader filename

Create a header file with the specified name and then compile the C and ***e*** code.

For a Specman ***e***/C-interface design, this option automatically generates a header file from the ***e*** files and then compiles the C code, the header, and the ***e*** files.

The `-snheader` option is equivalent to:

```
sn_compile.sh packet.e -h_only -o packet.h
```

This command is executed before the compilation of the C files.

The ***e*** files, C files, and the `-snheader` option must be specified inside a `-snstage` collection. For example:

```
% irun -snstage cinterface packet.e packet.c -snheader ./packet.h -endstage
```

When stage `cinterface` is executed, the header file `packet.h` is generated and then the C code and header are compiled and used with the compilation of `packet.e`.

-snini initialization_file

Specify a Specman initialization file. This option is useful when you want to use a customized initialization file with a name other than "`.specman`" or a location other than `$HOME` or the current working directory.

For example:

```
irun tb.v test_1.e -snini $SETUP_FILES/system.specman
```

-snload file_list

Specify a comma-separated or space-separated list of ***e*** files to load before HDL access generation. Because these files are not compiled, you have full debugging capabilities for this code.

Alternatively, you can load files using the -snprerun option. However, because these files are loaded after HDL access generation, any new HDL objects referenced in these files are not accessible during the simulation run.

For example:

```
irun tb.v -snload test_1.e my_macros.e
irun tb.v -snload test_1.e,my_macros.e
```

-snlogappend

Append to the Specman log file any log files previously saved with the -snwithlogs option. For more information, see Creating a Single Log File for Multiple Simulations, in *Running Specman with Incisive Simulator*.

-snnoauto

Do not generate a stubs file. When you use this option, you must supply the stubs file on the command line.

-snpath path_list

Specify a colon-separated list of directories to scan when loading ***e*** files or compiling ***e*** and ***e***-library files. Any path you specify is prefixed to paths defined in the SPECMAN_PATH environment variable.

For example:

```
setenv SPECMAN_PATH /regressions/fixes
irun tb.v -snload test_1.e bug_bypass.e \
     -snpath ./my_fixes:$HOME/my_fixes
```

In this example, Specman searches the three directories in the following order:

1. ./my_fixes

2. `$HOME/my_fixes`
3. `/regressions/fixes`

-snprerun commands

Specify a list of Specman commands, separated by a semi-colon, to be executed before simulation. The list must be enclosed in quotes.

The default is `"test"`. If you use `-snprerun`, the default is overwritten, and you must include the `test` command with the other commands or execute it prior to the start of simulation.

Use `-snprerun "notest"` to disable the automatic execution of `"test"`.

You can load files using the `-snprerun` option. However, because these files are loaded after HDL access generation, any new HDL objects referenced in these files are not accessible during the simulation run.

The following command executes the commands in the ecom file and then executes the `test` command.

```
irun tb.v -snprerun "@prerun.ecom; test" test_1.e
```

-snprofileargs argument

Pass the specified flags to the Specman profiler.

The arguments to `-snprofileargs` are the options to the Specman `set profile` command. See the *Specman Command Reference* for details on this command.

See “Using the Specman CPU and Memory Profilers” in *Specman Usage and Concepts Guide for e Testbenches* for details on the profilers and profiler reports.

-snprofilecpu

Enable Specman CPU profiling.

Specman provides a CPU Profiler and a Memory Profiler to help you identify performance problems. Use the `-snprofilecpu` option to run the CPU Profiler or the `-snprofilemem` option to run the Memory Profiler. You cannot use both options on the command line.

See “Using the Specman CPU and Memory Profilers” in *Specman Usage and Concepts Guide for e Testbenches* for details on the profilers and profiler reports.

-snprofilemem

Enable Specman memory profiling.

Specman provides a CPU Profiler and a Memory Profiler to help you identify performance problems. Use the `-snprofilecpu` option to run the CPU Profiler or the `-snprofilemem` option to run the Memory Profiler. You cannot use both options on the command line.

See "Using the Specman CPU and Memory Profilers" in *Specman Usage and Concepts Guide for e Testbenches* for details on the profilers and profiler reports.

-snquiet

Suppress messages from `sn_compile.sh` during the compilation of ***e*** files. All other messages from Specman, such as messages issued during loading, are displayed.

```
irun tb.v test_1.e -snquiet
```

-snrebuild

Force the recompilation of the ***e*** input files. This option is useful if you have made changes to the environment that *irun* cannot detect, such as modifying environment variables. For example:

```
unset SPECMAN_PATH
irun tb.v test_1.e -snrebuild
```

If you use the `-snrebuild` option on the command line with either `-r` or `-R`, then *irun* will recompile the ***e*** input files and generate a new specman `.esv` file (if necessary) before simulating the snapshot.

See the `-r` and `-R` options for more information on loading the simulation snapshot.

-snrecord database

Record a simulation session that contains a DUT and an ***e*** testbench. You can later replay the session with Specman.

By default, Specman records the session in a subdirectory of the working directory, called `sn_record`. If an `sn_record` subdirectory already exists, it is overwritten. Use the database *`argument`* to specify the name that you want to give to the subdirectory.

For information about the Record/Replay feature, see Recording and Replaying a Debugging Session, in *Compiling, Linking, and Running Specman*.

-snreplay database

Replay a simulation session that you previously recorded with `-snrecord` *`database`*.

By default, Specman replays the session recorded in the default subdirectory, `sn_record`. You can use the *`database`* argument to specify another subdirectory.

For information about the Record/Replay feature, see Recording and Replaying a Debugging Session, in *Compiling, Linking, and Running Specman*.

-snseed seed

Pass a seed value to Specman. The seed value is used by Specman to generate the variables in the environment.

The value of this option overrides the global `-seed` option.

This option is useful when you want to rerun a simulation with the same generation results as a previous run. This option also lets you launch the same test multiple times with different seeds. In the example below, the second test uses the compiled ***e*** and HDL code created during the first call to *irun*.

For example:

```
irun tb.v test_1.e -snseed 123456
irun tb.v test_1.e -snseed 987654
```

You can also use the `-snseed` option when restarting a simulation with a saved snapshot (`irun -r`). In this use, the `-snseed` option lets you:

1. Run your test in a specific configuration until the DUT has been initialized or until some other point of interest has been reached.
2. Save the current state of the simulation.
3. Restart the saved snapshot in multiple runs, assigning a different value to the generation seed. Each run continues from the saved state. The different seed will cause different generation results, and thus different scenarios.

The *`seed`* argument can be a number or the word `random`.

-snset string

Specify a comma-separated list of Specman commands to be executed before compiling or loading ***e*** files. The list must be enclosed in quotes.

Use this option to configure Specman to load or compile your environment. You can specify any Specman command, but typically you specify `configure` commands, `set notify` commands, `define` commands, `set checks` commands, and so on.

For example:

```
irun -snset "configure memory -max_size=500M -absolute_max_size=800M \
-gc_threshold=200M -gc_increment=100M" ....


irun -snset "set check WARNING; set notify -severity=IGNORE DEPR_NBASYNC; \
config run -tick_max=UNDEF"....


irun -snset "@set_DEPR.ecom"
```

-snshlib shared_library_path

Use the specified precompiled ***e*** shared library. The argument to `-snshlib` must be the path to the shared library.

This option is useful when a portion of the ***e*** environment is stable and you want to avoid unnecessary recompilation and linking. See “Incrementally Compiling an ***e*** Testbench with irun” in the chapter “Running Specman with the Incisive Simulator” in *Running Specman* for more information and an example.

If you use the `-snshlib` option, you cannot specify ***e*** files for compilation. You can load ***e*** files with the `-snload` option.

For example:

```
sn_compile.sh -t ./sn_tmp -pic -shlib vr_xbus_config
irun tb.v -snshlib ./libsn_vr_xbus_config.so -snload ./test_2.e
```

-snstage stage_name filename [filename...] [-endsnstage]

Compile all ***e*** files into a Specman stage with the name specified with the *`stage_name`* argument.

The `-snstage` option is used to replicate the incremental compilation in Specman. All ***e*** files listed after the option are compiled into a stage with the specified name. Previously compiled stages are used to build the next stage, and the last stage is used to compile other ***e*** files (those not listed after a `-snstage` option) on the command line.

Each `-snstage` file list is terminated by the next `-snstage` option, or by a `-endstage` or `-end` option.

If a precompiled ***e*** library is provided with the `-snshlib` option, the library is the first stage in the list of staged compiles.

If the `-nosncomp` option is used, all ***e*** files listed after the `-snstage` option are loaded. The files are loaded in the order that they appear for the `-snstage` option and by the order in which the `-snstage` options occur on the command line.

See “Incrementally Compiling an ***e*** Testbench with irun” in the chapter “Running Specman with the Incisive Simulator” in *Running Specman* for more information and an example.

-snstubelab

A simplified port setup is available for environments that run with the Incisive Simulator, use the `irun` command to start simulation, and interface with Verilog, SystemVerilog, or VHDL designs.

If your environment meets these requirements, you can significantly reduce the number of port attributes necessary to define the environment. This is because, under these circumstances, the elaborator (*ncelab*) can query the HDL code to identify attribute data that describes the DUT and then use this data to create the Specman stub file.

As a result, you do not need to define attributes that describe the DUT only—such as agent() or verilog_wire()—for such an environment.

To use the simplified port setup described in this section, you must use the `irun -snstubelab` option when you start your simulation.

-snsvdpi

Generate a C stub file automatically for ***e*** export methods using the *irun* flow.

If your code exports ***e*** methods to SystemVerilog, you can use the `-snsvdpi` option to generate the C stub file for the ***e*** methods. When using *irun*, this option automatically generates a C file that has auxiliary C code for the exported methods. For example:

```
% irun top.sv -snload verifier.e -snsvdpi
```

In the example above, *irun* automatically finds and compiles the `.svh` file containing the Specman scope methods, and will generate the associated C stub file.

-sntimescale timeunit / precision

Set the timescale Specman uses for Verilog design access and control the precision Specman uses for VHDL design access.

The Specman timescale is used to calculate the following time expressions:

- Delays in ***e*** temporal expressions
- Delays specified in VHDL or Verilog statements
- Simulation time as shown in `sys.time`, the 64-bit integer field that stores Specman time

See the section “Setting the Specman Timescale” in the chapter “Preparing Specman for Simulation” in *Running Specman* for details on setting the Specman timescale.

-snvlog | -snvhdl | -snsv | -snsc

Specify a single agent for all ***e*** units in the verification environment, either Verilog (`-snvlog`), VHDL (`-snvhdl`), SystemVerilog (`-snsv`), or SystemC (`-snsc`).

Note: An error is generated if more than one of these options are specified on the command line.

If some units in the ***e*** environment require different agents, you cannot use these options. Instead, you must specify the `agent()` attribute for each unit explicitly in the ***e*** code.

Use this option only when:

- The DUT or the environment uses more than one language
- All ***e*** units use the same agent
- The `agent()` attribute is not specified within the ***e*** code

In the following example, if no `agent()` attribute is specified in `test_1.e`, *irun* might spend unnecessary time analyzing the code to determine the proper agent, since both Verilog and VHDL files are specified on the invocation line:

```
irun tb.v regs.vhd test_1.e
```

If, in fact, `test_1.e` accesses objects only in `tb.v`, it is more efficient to invoke *irun* as follows:

```
irun tb.v regs.vhd test_1.e -snvlog
```

This has the same effect as constraining the `agent()` of `sys` to `Verilog`:

```
keep sys.agent() == "Verilog"
```

-specview

Invoke the simulator with only the Specview graphical user interface. Do not invoke the SimVision GUI.

UVM Command-Line Options

irun includes several options that support the Universal Verification Methodology (UVM).

Use the following command to view a list of UVM command-line options:

```
% irun -helpsubject UVM
```

-ml_uvm

Enable the multi-language (ML-UVM) features.

Use this option to enable ML-UVM when no -uvmtop switch is used. This is useful in cases when you have a top ***e*** layer where the ***e*** test file is being loaded by other means (for example through a Tcl command) and a SystemVerilog OVC that is instantiated by other means than an -uvmtop switch.

The following rules apply to the -ml_uvm option:

- If you are using SystemVerilog components, you must have at least one import statement to import the ml_uvm package in your SystemVerilog source:

  ```
  import ml_uvm::*; // Required for ML-UVM
  ```

- When you specify the -ml_uvm option, the -uvm switch is implicitly added, causing *irun* to compile the UVM package that is included with your IES release.

For additional usage and methodology guidance, see either the *UVM Multi-Language Reference* or *UVM Multi-Language Methodology*, available with the Incisive Verification Kits.

-uvm

Enable support for the Universal Verification Methodology (UVM).

The -uvm option automates the use of UVM with the IES simulator.

- Compiles the UVM package
- Automatically loads a UVM Tcl interface

- Automatically loads a UVM GUI interface
- Automatically loads the definition of UVM-specific system tasks
- Automatically adds `-incdir <UVMHOME>/src` to the command line
- Adds the definition of a UVM macro `URM_SV_ENABLE`

Note: If you run into problems while simulating UVM designs with *irun*, refer to the following link from the Cadence Online Support site for possible solutions:

Various non-standard use case scenarios involving irun and UVM

-uvmhome directory | string

Specify the location of the UVM installation.

By default, UVM is installed in `<install_directory>/tools/methodology`. Use the `-uvmhome` option to name a different base installation and automatically import the latest Cadence extensions to UVM. The Cadence extensions offer the following functionality:

- Multi-language extensions
- Critical bug fixes
- Transaction Recording
- Additional Sequences for the UVM_REG package

There are two possible arguments to the `-uvmhome` option:

- A valid path (relative or absolute) to a directory
- A character string that *irun* will use in an attempt to match with an existing directory in the installation hierarchy

For example, consider an installation with the following directories:

```
<install_directory>/tools/methodology/UVM/1.1
<install_directory>/tools/methodology/UVM/1.1a-3
<install_directory>/tools/methodology/UVM/CDNS-1.1a-4
<install_directory>/tools/methodology/UVM/1.1a-5
```

You can use `-uvmhome` with a direct path to set the location as shown:

```
% irun -uvmhome 'ncroot'/tools/methodology/UVM/1.1a-3
```

Similarly, `-uvmhome` recognizes directory locations from prior releases, such as `'ncroot'/tools/uvm-1.1` and remaps them to the current base location.

You can also use strings. Use the following to have *irun* match the `1.1a` directory with the highest numeric value in the installation hierarchy (that is, `1.1a-5`):

```
% irun -uvmhome 1.1a
```

To disable the Cadence extensions to UVM, use the `-uvmnocdnsextra` option.

-uvmlinedebug

Enable single-stepping through UVM functions and tasks.

By default, `-uvm -linedebug` does not apply the `-linedebug` option to the compilation of the UVM package. This means that when you are debugging your code, single-stepping through the UVM content is not enabled.

Include the `-uvmlinedebug` option if you want to single-step through the UVM content to debug UVM code. This option forces the UVM package, and all other HDL files specified on the command line, to be compiled with line debug capabilities.

```
% irun -uvm -uvmlinedebug [other_options] source_files
```

-uvmnoautocompile

Disable automatic compilation of the UVM packages.

When support for UVM is enabled with the `-uvm` option, the UVM packages are, by default, automatically compiled in a separate step before any HDL files specified on the command line. Use the `-uvmnoautocompile` option to prevent compilation of the UVM packages.

For example, suppose that you have created a local copy of the `uvm_pkg.sv` package because you wanted to modify the file in some way (to include Cadence transaction recording, to include defines, and so on). You can then include the local copy on the command line, and use `-uvmnoautocompile` to turn off the automatic compilation of the package that is in the UVM installation hierarchy.

```
% irun –uvmnoautocompile ./uvm_pkg.sv –uvmhome /local/tools/uvm-1.0 myfiles.sv
```

-uvmnocdnsextra

Disable the Cadence extensions to UVM.

By default, when using the `-uvmhome` option to specify an alternate UVM installation on the *irun* command line, the latest Cadence extensions to UVM are included automatically in an add-on library. Use the `-uvmnocdnsextra` option to run UVM without these Cadence extensions. For example:

```
% irun -uvmhome 'ncroot'/tools/uvm-1.1 -uvmnocdnsextra
```

-uvmtest [language:]entity_name ...

Enable multi-language UVM (ML-UVM) and declare a top test entity in a multi-language verification environment.

Use this option to declare the test, which is the logical root of the entire multi-language testbench hierarchy. You can specify only one entity with the `-uvmtest` option, and it can be in any one of the supported languages.

Use this option to specify either your ***e*** test file name, or SystemVerilog/SystemC test class. If your root is a SystemVerilog test class, an instance of it is created with the name `uvm_test_top`, as in a pure SystemVerilog testbench.

The `-uvmtest` option takes an entity name consisting of a language identifier and an entity name. The language identifier marks the language domain of the top entity. It can be one of the following:

- `sv` (or `SV`) for SystemVerilog
- `e` (or `E`) for ***e***
- `sc` (or `SC`) for SystemC

The *`entity_name`* can be one of the following:

- ***e*** test file name
- UVM SystemVerilog class name, representing a test (or another type derived from ovm_component)
- SystemC class name

The following rules apply to the `-uvmtest` option:

- Can be used only once in a command.
- You can omit the `e:` language identifier if you specify an ***e*** test file name with a `.e` suffix. If you omit the language identifier and the name has no `.e` suffix, it is assumed to be a SystemVerilog class name. You cannot omit the `SC` language identifier.

  Cadence recommends that you always include a language identifier, for clarity and readability.

- The `-uvmtest` option replaces the `+UVM_TESTNAME` switch in multi-language environments. It is an error to use both `-uvmtest` and `+UVM_TESTNAME` in the same `irun` command line.
- You cannot specify an ***e*** unit type name with `-uvmtest`. Only ***e*** test files are supported as top entities in ***e***.
- If you specify `-uvmtest`, the `-uvm` and `-ml_ovm` command line options are turned on and need not be specified explicitly.

Examples

Naming an ***e*** test file as a top entity:

```
% irun -uvmtest e:my_test.e ...
```

Naming a UVM SystemVerilog test class (similar to the pure UVM SystemVerilog `+UVM_TESTNAME` switch) as a top entity:

```
% irun -uvmtest SV:my_test_class ...
```

Naming two top entities, an ***e*** test and a UVM SystemVerilog env class:

```
% irun -uvmtest e:my_test.e \
       -uvmtop SV:my_ethernet_env ...
```

For additional usage and methodology guidance, see either the *UVM Multi-Language Reference* or *UVM Multi-Language Methodology*, available with the Incisive Verification Kits.

+UVM_TESTNAME=test_name

Specify the name of the test.

After you have declared a user-defined test, you invoke the global UVM `run_test()` task in the top-level module to select a test to be simulated. Its prototype is:

```
task run_test(string test_name="");
```

The test name can be provided to `run_test()` by using the `+UVM_TESTNAME` option. If the top module calls `run_test()` with an argument, that test is used unless a test is also specified on the command line with the `+UVM_TESTNAME` option. Any test used by `run_test()` must be registered with the factory in order to be used.

Using the command-line option avoids having to hardcode the test name in the `run_test()` task. For example, in the top-level module, call `run_test()` as follows:

```
module tb_top;
// DUT, interfaces, and all non-testbench code
```

```
  initial
  run_test();
endmodule
```

To select a test of type `test_read_modify_write`, use the following command:

```
% irun +UVM_TESTNAME=test_read_modify_write [other_options] source_files
```

By using this method and only changing the `+UVM_TESTNAME` argument, you can run multiple tests without having to recompile or re-elaborate the design or testbench.

-uvmtop [language:]entity_name ...

Enable multi-language UVM (ML-UVM) and declare a top entity in a multi-language verification environment. Use this option to name entities which are top entities in their language domain, because their parent entity is in another language domain.

Cadence recommends that you use the `-uvmtest` option and not `-uvmtop` to name the test, the root of the logical testbench hierarchy.

Use this option to declare a top entity in one of the language domains. To specify several top entities, use the option multiple times, each naming one of the entities. Each of the entities you name has a parent entity in another language domain, and therefore is a top entity (no parent) in its own language domain. This option should not be used for the root entity of the logical testbench hierarchy, where `-uvmtest` should be used instead.

The `-uvmtop` option takes an entity name consisting of a language identifier and an entity name. See the description of `-uvmtest` for a description of the argument.

The following rules apply to the `-uvmtop` option:

- The order of top entities as declared on the command line determines the order in which they are created and built. Therefore, make sure to name the test as the very first top entity.
- You can omit the `e:` language identifier if you specify an ***e*** test file name with a `.e` suffix. If you omit the language identifier and the name has no `.e` suffix, it is assumed to be a SystemVerilog class name. You cannot omit the SC language identifier.

  Cadence recommends that you always include a language identifier, for clarity and readability.
- You cannot specify an ***e*** unit type name with `-uvmtop`. Only ***e*** test files are supported as top entities in ***e***.

- If you specify `-uvmtop`, the `-uvm` and `-ml uvm` command line options are turned on and need not be specified explicitly.

Example

Naming two top entities, an ***e*** test root and a UVM SystemVerilog env class:

```
% irun -uvmtest e:my_test.e \
       -uvmtop SV:my_ethernet_env ...
```

For additional usage and methodology guidance, see either the *UVM Multi-Language Reference* or *UVM Multi-Language Methodology*, available with the Incisive Verification Kits.

irun Command Examples

In the following example, *irun* recognizes the input file as a Verilog file and invokes the Verilog compiler, *ncvlog*, to compile design units in the file. All design units are compiled into the default work library `worklib`. After compilation, *irun* invokes the elaborator to elaborate the design and generate a snapshot, and then invokes the simulator to run the snapshot.

```
% irun sio85.v
```

In the following example, *irun* recognizes the file `file.v` as a Verilog file, and the file `test.sv` as a SystemVerilog file. The Verilog compiler is invoked to compile design units in `file.v` into the default work library. The SystemVerilog compiler is invoked to compile design units in `test.sv` into the default work library. The design is then elaborated and simulated.

```
% irun file.v test.sv
```

In the following example, *irun* recognizes the source files as VHDL and invokes the VHDL compiler, *ncvhdl*, to compile the design units in the files. All design units are compiled into the default work library. Because *irun* does not automatically calculate the top-level design unit for VHDL files, you must specify the top-level design unit on the command line with the `-top` or `-vhdltop` option.

```
% irun fa1.vhd test_adder.vhd -top test_adder
```

The following command compiles the Verilog files `top.v` and `sub.v` with *ncvlog*, and the VHDL file `middle.vhd` with *ncvhdl*. It is not necessary to specify the top-level unit with `-top` because the top-level design unit is Verilog. After the design is elaborated, the simulator is invoked with the SimVision GUI.

```
% irun -v93 -gui top.v middle.vhd sub.v
```

The following command calls `sn_compile.sh` to compile the ***e*** file into a shared library, loads the ***e*** file into Specman, generates the HDL stubs file, invokes the Verilog compiler to

compile the Verilog files, the elaborator, and then the simulator. The simulator automatically loads the shared library.

```
% irun test.v test.e
```

In the following example, *irun* will not be able to determine the file type of the file `counter.vlog` because the extension `.vlog` is not a recognized extension. The `-default_ext` option is included to specify that all files with unrecognized extensions are to be treated as Verilog files.

```
% irun board.v counter.vlog clock.v ff.v -default_ext verilog
```

By default, *irun* recognizes files with the following extensions as Verilog files: `.v`, `.vp`, `.V`, and `.VP`. The following command includes the `-vlog_ext` option to add `.vlog` to the list of recognized extensions.

```
% irun board.v counter.vlog clock.v ff.v -vlog_ext +.vlog
```

By default, *irun* recognizes files with the following extensions as Verilog files: `.v`, `.vp`, `.V`, and `.VP`. The following command includes the `-vlog_ext` option to change the list of recognized extensions to `.v`, `.rtl`, and `.vg`.

```
% irun board.v counter.vlog clock.v ff.v -vlog_ext .v,.rtl,.vg
```

The following command uses the `-f` option to pass an arguments file to *irun*. The arguments file contains command-line options and a list of input files. The `-f` option scans the arguments file for files relative to the *irun* invocation directory. Use the `-F` option if you want to first scan for files relative to the location of the arguments file.

```
% irun -f run.args
```

The following command includes the `-c` option. Design units in the source files are compiled, and the elaborator is invoked to elaborate the design. The elaborator generates the simulation snapshot, but the simulation is not run.

```
% irun -c -v93 top.v middle.vhd sub.v
```

You can build a simulation snapshot by using the `-c` command-line option and then using the `-R` option to simulate the snapshot multiple times with different simulator options or command files. For example:

```
% irun -c -v93 top.v middle.vhd sub.v
% irun -R -input cmds1.tcl
% irun -R -input cmds2.tcl
% irun -R -input cmds3.tcl
```

The following command includes the `-makelib` option. Design units in the source files following the `-makelib` option are compiled into the library `rtllib` (in the physical directory `/usr1/libs/rtllib`). Design units in `top.vhd` are compiled into the default work library.

```
% irun -v93 -top worklib.test top.vhd \
```

```
-makelib /usr1/libs/rtllib file1.vhd file2.vhd
```

The scope of a `-makelib` option extends to the next use of `-makelib` or to a `-endlib` option. In the following example:

- Design units in `file1.vhd` and `file2.vhd` are compiled into `lib1`.
- Design units in `file3.vhd` are compiled into `lib2`.
- Design units in `top.vhd` are compiled into the default work library.

```
% irun -top worklib.test \
-makelib /usr1/libs/lib1 file1.vhd file2.vhd \
-makelib /usr1/libs/lib2 file3.vhd -endlib \
top.vhd
```

The following command includes the `-compile` and `-makefile` options to precompile files into a specified library. The second `irun` command then references the library with the `-reflib` option. The argument to `-reflib` is the path to the library.

```
% irun -compile -makelib /usr1/libs/lib1 file1.vhd file2.vhd
% irun -top worklib.test -reflib /usr1/libs/lib1 top.vhd
```

If you have been running the Verilog simulator in single-step invocation mode with the `ncverilog` command, you can use *irun* by simply changing the `ncverilog` command to `irun`. All *ncverilog* plus and dash options, including `-v` and `-y`, can be used with the `irun` command. For example:

```
% irun -s +ncaccess+rw top.v -y ./libs -y ./models +libext+.v
```

irun includes several command-line options that are specific to Specman. To see a complete list of these options, use the following command:

```
% irun -helpsubject specman
```

The following `irun` command invokes `sn_compile.sh` to compile the ***e*** file `test1.e`. The `-snset` option specifies that the commands in `preAHDL.ecom` will be executed at startup before compiling or loading ***e*** files. The `-snprerun` option specifies a file of Specman commands to be executed before simulation. By default, the prerun command is `test`. If you use the `-snprerun` option to specify prerun commands, you must include the `test` command or execute it interactively before starting the simulation.

```
% irun tb.v test1.e \
-snset "preAHDL.ecom" \
-snprerun "@prerun.ecom; test"
```

The following command includes the `-snload` option. Specman will load the file `test.e` at elaboration time, and then save the Specman state at the end of elaboration. The ***e*** code is fully debuggable, and references into the DUT are added to the AHDL database.

```
% irun file.v file.vhd \
```

```
-top worklib.top \
-snload test.e
-snset "preAHDL.ecom" \
-snprerun "@debug.ecom"
```

In the following example, the ***e*** testbench is compiled into a shared library using `sn_compile.sh`. The `irun` command then uses the `-snshlib` option to specify the path to the shared library. The `-gui` option invokes the simulator with both the SimVision and Specview GUIs.

```
% sn_compile.sh -shlib -exe xor_verify.e
% irun -snshlib ./libsn_xor_verify.so xor.v
```

4

Customizing irun

This chapter contains information on how to override various default behaviors of *irun*.

Changing the Default Set of File Extensions

irun uses the file extensions of the input files specified on the command line to determine their file type. Each recognized file type has a built-in, predefined set of file extensions. For each file type, there is a command-line option that you can use to change, or add to, the list of file extensions mapped to a given language.

The following table shows the recognized file types, the set of built-in, predefined file extensions for each language type, and the command-line option you can use to change, or add to, the list of file extensions mapped to a given language.

File Type	Defined File Extensions	Option
Verilog	`.v`, `.V`, `.vp`, `.VP`, `.vs`, `.VS`	`-vlog_ext`
Verilog 1995	`.v95`, `.V95`, `.v95p`, `.V95P`	`-vlog95_ext`
SystemVerilog	`.sv`, `.SV`, `.svp`, `.SVP`, `.svi`, `.svh`, `.vlib`, `.VLIB`	`-sysv_ext`
VHDL	`.vhd`, `.VHD`, `.vhdl`, `.VHDL`, `.vhdp`, `.VHDP`, `.vhdlp`, `.VHDLP`	`-vhdl_ext`
VHDL configuration	`.vhcfg`	`-vhcfg_ext`
Specman ***e***	`.e`, `.E`	`-e_ext`
Verilog-AMS	`.vams`, `.VAMS`	`-amsvlog_ext`
VHDL-AMS	`.vha`, `.VHA`, `.vhams`, `.VHAMS`, `.vhms`, `.VHMS`	`-amsvhdl_ext`
PSL file for Verilog	`.pslvlog`	`-propvlog_ext`

File Type	Defined File Extensions	Option
PSL file for VHDL	`.pslvhdl`	`-propvhdl_ext`
PSL file for SystemC	`.pslsc`	`-propsc_ext`
C	`.c`	`-c_ext`
C++	`.cpp`, `.cc`	`-cpp_ext`
Assembly	`.s`	`-as_ext`
Compiled object	`.o`	`-o_ext`
Compiled archive	`.a`	`-a_ext`
Dynamic library	`.so`, `.sl`	`-dynlib_ext`
SPICE file	`.scs`, `.sp`	`-spice_ext`

For example, the default file extensions for Verilog files are `.v`, `.vp`, `.vs`, `.V`, `.VP`, and `.VS`. If you have Verilog files with other extensions (for example, `.rtl` and `.vg`), you must specify that these are valid extensions for Verilog files. You can do this with the `-vlog_ext` option. You can:

- Replace the list of built-in, predefined extensions with a new list. For example, the following option specifies that the valid extensions for Verilog files are `.v`, `.rtl`, and `.vg`:

  ```
  -vlog_ext .v,.rtl,.vg
  ```

- Add extensions to the list of built-in, predefined extensions by using a plus sign (`+`) before the list of extensions to add. For example, the following option adds `.rtl` and `.vg`.

  ```
  -vlog_ext +.rtl,.vg
  ```

  This is the same as:

  ```
  -vlog_ext .v,.vp,.vs,.V,.VP,.VS,.rtl,.vg
  ```

You can include extension options in the definition of the `IRUNOPTS` variable in an `hdl.var` file. For example:

```
#hdl.var file
DEFINE IRUNOPTS -vlog_ext .v,.vg,.rtl [other_options]
```

irun generates an error if it encounters a file with an extension it does not recognize. In addition to using one of the extension options to override the set of recognized extensions for a particular file type, you can use the `-default_ext` option to specify the file type for files with unrecognized extensions.

The argument to the `-default_ext` option is a string that represents a file type.

Argument to -default_ext	File Type
`verilog`	Verilog HDL
`vcnf`	Verilog configuration
`verilog95`	Verilog 1995 HDL
`systemverilog`	SystemVerilog HDL
`vhdl`	VHDL HDL
`vhcfg`	VHDL configuration
`e`	Specman ***e***
`verilog-ams`	Verilog-AMS HDL
`vhdl-ams`	VHDL-AMS HDL
`psl_vlog`	PSL file for Verilog
`psl_vhdl`	PSL file for VHDL
`psl_sc`	PSL file for SystemC
`c`	C file
`cpp`	C++ file
`assembly`	Assembly
`o`	Compiled object
`a`	Compiled archive
`so`	Dynamic library
`scs`	SPICE file

For example, suppose that you have Verilog files with `.v`, `.vlog`, and `.vg` file extensions. Because `.v` is a defined file extension for Verilog files, *irun* will recognize files with a `.v` extension. However, the tool will not be able to determine the file type of the files with `.vlog` or `.vg` extensions because these extensions do not map to any file type. If you use the `-default_ext verilog` option, *irun* will treat all files with these undefined extensions as Verilog files.

Changing the Name of the INCA_libs Directory

By default, *irun* creates a scratch directory called `INCA_libs`. You can change the name of this directory with the `-nclibdirname` option. For example:

```
% irun -nclibdirname IRUN_libs [other_options] input_files
```

The *`directory_name`* argument can be a relative or absolute path to the directory. For example:

```
-nclibdirname foo            // Creates ./foo

-nclibdirname ./foo          // Creates ./foo

-nclibdirname ../foo         // Creates ../foo

-nclibdirname foo/bar        // Creates ./foo/bar. Directory foo must
                             // exist. irun will create directory bar.
```

Changing the Name of the Work Library

By default, *irun* compiles all design units in HDL files into a work library called `worklib` (located within the `INCA_libs` directory tree).

You can change the name of the library with the `-work` option. For example, the following command creates a work library called `mylib` and the directory `INCA_libs/mylib`.

```
% irun -work mylib [other_options] input_files
```

The following command creates a work library called `mylib` and the directory `IRUN_libs/mylib`.

```
% irun -nclibdirname IRUN_libs -work mylib [other_options] input_files
```

You can also define the work library by defining the `WORK` variable in an `hdl.var` file. For example:

```
# hdl.var
DEFINE WORK mylib
```

Compiling into Multiple Libraries

By default, *irun* compiles all design units in HDL files into the default `worklib` library (located within the `INCA_libs` directory tree) . Use the `-makelib` option if you want to compile different files into different libraries.

The `-makelib` option precompiles design units in the specified files into a reference library. When top-level design files are compiled, the reference library is scanned for components instantiated in the design.

irun supports the `-makelib` option with the following file types:

- Verilog
- SystemVerilog
- VHDL
- Verilog-AMS
- C
- C++
- SystemC

The syntax of the `-makelib` option is below:

```
-makelib path_to_library[:logical_name] source_files [-endlib]
```

Specifying the Source Files to be Compiled

You can list a collection of files to compile into the specified library directly on the command line. These files can either follow the `-makelib` option, or they can be listed in a file specified with the `-f` or `-F` option. The list of files for a `-makelib` collection is terminated by:

- A `-end` or `-endlib` option
- Another option that specifies a collection of files to be processed together

  These options are:

  - Another `-makelib` option
  - `-cpost`
  - `-snstage`

For example:

```
% irun -compile -v93 -makelib mylib file1.vhd file2.vhd -endlib ...
```

Or:

```
% irun -compile -v93 -makelib mylib -f mylib_files.txt -endlib ...
```

Specifying the Library

By default, the logical name of the library is the terminating name in the provided path. If no directory hierarchy is specified, the library is stored inside the `INCA_libs` directory.

Example:

```
% irun -makelib lib1 file1.vhd file2.vhd -makelib lib2 file3.vhd -endlib file4.vhd
```

In this example:

- `file1.vhd` and `file2.vhd` will be compiled into the library `lib1`. The physical path of the library is `INCA_libs/lib1`.
- `file3.vhd` will be compiled into the library `lib2`. The physical path of the library is `INCA_libs/lib2`.
- `file4.vhd` will be compiled into the default library `worklib`.

If a directory hierarchy is specified, *irun* creates the directory and uses that directory for the library storage. For example, the following option will compile `file.vhd` into the library `lib1`. The physical path of the library is `./lib1`.

```
% irun -makelib ./lib1 file.vhd -endlib ...
```

The following option will compile `buf.v` and `and2.v` into the library `gates`. The physical path of the library is `/usr1/myarea/gates`.

```
% irun -makelib /usr1/myarea/gates buf.v and2.v -endlib ...
```

Note: *irun* generates an error if the root path to the specified directory (`/usr1/myarea` in the example) does not exist.

If you want to create a different logical name for the library, add `:`*`logical_name`* to the end of the path. For example, the following option creates a library called `rtllib`. The physical directory for the library is `/usr1/libs`.

```
-makelib /usr1/libs:rtllib source_files ...
```

The `-makelib` option adds libraries to the search path used when the entire design is being elaborated. If you have defined libraries in a `cds.lib` file, the libraries added by these options are added to the end of the list defined in the `cds.lib` file.

Using Command-Line Options Within a -makelib

You can specify command-line options within a `-makelib` collection. The specified options are applied only to source files within the `-makelib` collection. In general, only options that go to the compilers (*ncvlog*, *ncvhdl*, or *ncsc_run*) are allowed within a `-makelib` collection. These options can be specified following the `-makelib` option, or they can be in a file included with the `-f` or `-F` option.

Note: Compilation options that are specified on the command line outside of a makelib collection apply to all `-makelib` options. If you do not want to specify a global option for all makelib collections on the command line, then you can run separate `irun -makelib` commands for those instances in which you need to exclude the global switch.

irun does not support the following options within a `-makelib` collection:

Option	**Is valid only when using...**		
	ncvlog	**ncvhdl**	**ncsc_run**
-cds_implicit_tmponly	X	X	
-debug			X
-default_spice_oomr	X		
-errormax	X	X	
-ieee1364	X		
-ignore_spice_oomr	X		
-libcell	X		
-liborder	X		
-librescan	X		
-libverbose	X		
-ncerror	X	X	
-ncfatal	X	X	
-nclibdirname	X		
-nclibdirpath	X		
-ncshare	X		
-ncuid	X		

Option	Is valid only when using...		
	ncvlog	**ncvhdl**	**ncsc_run**
-ncvhdl_args		X	
-ncvhdlargs		X	
-ncvlog_args	X		
-ncvlogargs	X		
-neverwarn	X	X	
-nocopyright	X	X	X
-nopragmawarn	X	X	
-nostdout	X	X	
-noupdate	X	X	
-ovmlinedebug	X		
-partialdesign	X		
-pragma	X	X	
-status	X	X	X
-u	X		
-unbuffered			X
-uptodate_messages	X	X	
-uvmaccess	X		
-uvmlinedebug	X		
-uvmpackagename	X		
-v1995	X		
-v2001	X		
-zlib	X	X	

Precompiling Files and Referencing the Library

You can precompile files into a library with the `-compile` and `-makelib` options, and then reference the library with the `-reflib` option. The argument to `-reflib` is the path to the

library. The `-reflib` option adds the library to the list of libraries to scan when the design is elaborated.

Note: When reinvoking the simulation with SimVision, any modifications made to source files within precompiled libraries, or within files compiled with either the `-makelib` or `-reflib` option will not take effect.

Examples

The following `irun` command compiles two files (`bv_images_p.vhd` and `bv_images_pb.vhd`) into a library called `plib` (in the physical directory `/usr/proj/plib`). Design units in the other VHDL files are compiled into the library `worklib`.

Note: For Verilog, *irun* can automatically detect top-level units in the design. However, *irun* does not automatically detect top-level VHDL units, and you must specify the top-level unit with the -top or -vhdltop option.

```
% irun -v93 -makelib /usr/proj/plib bv_images_p.vhd bv_images_pb.vhd -endlib \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

In the following command, the two files to be compiled into library `plib` are listed in a file called `plib_files.txt`, which is specified with the `-f` option.

```
% irun -v93 -makelib /usr/proj/plib -f plib_files.txt -endlib \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

The following command includes two options within the `-makelib` collection. These options apply only to the files within the collection.

```
% irun -v93 -makelib /usr/proj/plib -novitalcheck -nobuiltin \
bv_images_p.vhd bv_images_pb.vhd -endlib \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

In the following example, the two VHDL source files are compiled into library `plib` using the `-compile` option. Library `plib` is then referenced with the `-reflib` option. The argument to `-reflib` is the path to the library.

```
% irun -v93 -compile -makelib /usr/proj/plib bv_images_p.vhd bv_images_pb.vhd

% irun -v93 -reflib /usr/proj/plib inverter.vhd counter_4bit.vhd \
counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

Note: The `-v93` option enables VHDL-93 features for the entire design, including the specified library `plib`. If you were to specify a library that does not require VHDL-93 features (for instance, a Verilog design for mixed-language use), then you could compile that library without the -v93 option and use `-reflib` to specify it instead of `plib`.

If you include a logical name for the library, the logical name must also be included with `-reflib`. For example:

```
% irun -v93 -compile -makelib /usr/proj/plib:mypack \
bv_images_p.vhd bv_images_pb.vhd
%
% irun -v93 -reflib usr/proj/plib:mypack \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

If you use the `-c` option to both compile source files and elaborate the design, top-level design files must be included on the command line. For example:

```
% irun -c -makelib lib1 counter.v clock.v ff.v -endlib top.v
% irun -c -makelib lib1 counter.vhd clock.vhd ff.vhd -endlib top.v
  -top worklib.top
```

If you have an existing `cds.lib` file that defines libraries, and have created the physical directories for the libraries, you can use the `-makelib` option to compile files into the libraries. The argument to `-makelib` must match the path in the `cds.lib` file because you cannot have two libraries with the same name. For example, suppose that you have a `cds.lib` file that defines a library called `plib` as follows:

```
# cds.lib
DEFINE plib ./libs/plib
```

To compile files into library `plib`, the argument to `-makelib` must match the path specified in the `cds.lib` file.

```
% irun -v93 -compile -makelib ./libs/plib bv_images_p.vhd bv_images_pb.vhd
```

Because the `cds.lib` file contains the library reference, it is not necessary to include the `-reflib` option to refer to the library.

```
% irun -v93 inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

Specifying a Snapshot Name

After invoking the appropriate compiler to compile the source files specified on the command line, *irun* invokes *ncelab* to elaborate the design. The elaborator generates a simulation snapshot that is the input to the simulator (*ncsim*). Simulation snapshots are given default names in lib.cell:view format.

- For Verilog, the default snapshot name is:

  library.top_level_module_name:*view*

  where *library* is the library into which the top-level module has been compiled, and *top_level_module_name* is the name of the top-level module (or the name of the first top-level module encountered when parsing the source files). The view name is the file extension of the file that contains the description of the top-level module. For example, if the name of the top-level module is `top`, and the module is described in `test.v`, and this module is compiled into the default `worklib` library, the snapshot is called `worklib.top:v`.

- For VHDL, the default snapshot name is:

  library.top_level_entity_name:*architecture*

  For example, if the top-level entity is called `counter_test` and the architecture is called `bench`, the snapshot is called `worklib.counter_test:bench`.

Use the `-snapshot` option to give different elaborations of your design unique snapshot names. For example, the following command generates a snapshot called `worklib.run1:v`.

```
% irun -snapshot run1 file1.v file2.v
```

The `-snapshot` option also changes the name of the scratch subdirectory under the `INCA_libs` directory. *irun* uses this directory as a scratch area to create files and pass them between tools and to track information necessary to execute *irun* multiple times. By default, this subdirectory is called

irun.<*platform* | *platform*.64>.<*irun_version*>.nc

For example:

```
irun.lnx86.09.20.nc/
```

As a convenience, a symbolic link named `irun.nc` is created that points to the *irun* scratch subdirectory.

Using the `-snapshot run1` option generates a directory called `run1.lnx86.09.20.nc/`. The symbolic link is `run1.nc`.

Note: Both `-snapshot` and `-name` are synonyms for the same option.

Compiling Source Files with Specific Options

When source files are compiled, compiler options specified on the command line apply to all files listed on the command line. For example, the following command includes four options.

```
% irun -v93 -controlrelax ARSHCH -ieee1364 -linedebug top.v middle.vhd sub.v
```

The `-v93` and `-controlrelax` options are VHDL-specific and are passed to the *ncvhdl* compiler when the file `middle.vhd` is compiled. The `-ieee1364` option is Verilog-specific and is passed to the *ncvlog* compiler when `top.v` and `sub.v` are compiled. The `-linedebug` option is common to both Verilog and VHDL and is passed to both compilers.

You can also include these options in the definition of the `IRUNOPTS` variable in an `hdl.var` file. For example:

```
# hdl.var
DEFINE IRUNOPTS -v93 -relax -ieee1364 -linedebug


% irun top.v middle.vhd sub.v
```

In some cases, it might be necessary to compile some Verilog or VHDL source files with a specific option, or set of options, and other files with different options. There are two ways to do this:

- Define the `FILE_OPT_MAP` variable in an `hdl.var` file.

  To define the `FILE_OPT_MAP` variable, specify either a filename or a directory, followed by a list of options. The format is as follows:

  ```
  DEFINE FILE_OPT_MAP ( {filename | directory} => list_of_options[, {filename
  | directory} => list_of_options ...)
  ```

  The options are separated by a space.

  If a filename is specified, the options are applied to that file. If a directory is specified, the options are applied to all files in that directory.

  The backslash character (`\`) can be used to define the variable over multiple lines.

  ```
  DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug)


  DEFINE FILE_OPT_MAP (file1.vhd => -v93 -linedebug, file1.v => -view view1
  -v1995)


  DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug, \
                       file1.v => -view view1 -v1995)
  ```

  Only one `FILE_OPT_MAP` variable can be defined in an `hdl.var` file.

  Example:

```
# File: hdl.var
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug, \
                     ./src/vhdl/file2.vhd => -controlrelax ARSHCH -v93, \
                     ./src/vhdl => -v93, \
                     ./src/vlog/file1.v => -view view1 -v1995)
```

In this example:

- `file1.vhd` is compiled with the `-v93`, `-controlrelax`, and `-linedebug` options.
- `./src/vhdl/file2.vhd` is compiled with the `-controlrelax` and `-v93` options.
- All other files in the directory `./src/vhdl` are compiled with the `-v93` option.
- `./src/vlog/file1.v` is compiled with the `-view` and `-v1995` options.

■ Use the `-filemap`/`-endfilemap` options on the `irun` command.

These options start and end a filemap collection. The syntax is as follows:

`-filemap` *`source_file`* `[`*`source_file ...`*`]` *`list_of_options`* `-endfilemap`

Example:

```
% irun file1.v -filemap file2.v -view view1 -v1995 -endfilemap -assert
```

In this example, `file1.v` will be processed with `-assert` only. The file `file2.v` will be processed with `-assert`, `-view`, and `-v1995`.

See the description of the `-filemap` option for more details on using the `-filemap` and `-endfilemap` options.

Note: Not all parser options can be used in the definition of the `FILE_OPT_MAP` variable or with the `-filemap`/`-endfilemap` options. The following options are not supported:

```
-ams                        -neverwarn
-cdslib                     -nocopyright
-cds_implicit_tmpdir        -nolink
-cds_implicit_tmponly       -nowarn
-cmdfile                    -ovl
-define                     -smartorder
-design_top                 -smartscript
-errormax                   -specificunit
-hdlvar                     -status
-ial                        -sv
-ncerror                    -zlib
-ncfatal                    -zparse
```

5

Compatibility with Existing Use Models

This chapter includes several sections that provide information to help you transition from simulating a design using existing use models to simulating the same designs using *irun*.

ncverilog

Note: Because *irun* supports all features of *ncverilog*, including its command-line options, Cadence is replacing *ncverilog* with *irun*. Beginning with the IUS 8.1 release, using the `ncverilog` command will invoke *irun*.

If you have been simulating Verilog designs with the `ncverilog` command, you can simulate with *irun* by replacing the `ncverilog` command with the `irun` command. For example:

```
% ncverilog +gui +ncaccess+rw top.v -y ./libs -y ./models +libext+.v
% irun +gui +ncaccess+rw top.v -y ./libs -y ./models +libext+.v
```

The following `ncverilog` options are not supported in *irun*:

- `+cellview`
- `+redirect`
- `-d`

Because *irun* determines the language of a file from its file extension, any extension used in the `+libext+` option must be a recognized extension that maps to a Verilog type language. If the extension is not one of the predefined extensions, you must use the `-default_ext`, `-vlog_ext`, `-sysv_ext` or `-amsvlog_ext` option to add the `+libext+` extension to the list of recognized extensions.

Note: The `+libext+`*`extension`* (or `-libext` *`extension`*) option must be used to specify the extension of the files referenced by the `-y` option. For example:

```
irun ... -y ./libs -y ./models +libext+.v
```

or:

```
irun ... -y ./libs -y ./models -libext .v
```

If the `+libext+` (`-libext`) option is not included on the command line, all files referenced by the `-y` option must not have a file extension.

When simulating SystemVerilog files with *ncverilog*, you used the `+sv` option. This option is not required with *irun* because it recognizes SystemVerilog files using the file extensions. If you include the `-sv` option on the `irun` command line, all Verilog type files are compiled as SystemVerilog.

While *ncverilog* supported multiple single-character options following a single dash (`-`) character, this is not supported in *irun*. The options must be entered separately. For example:

```
ncverilog -qc
irun -q -c
```

ncverilog and *irun* print messages from the compiler, elaborator, and simulator by default, including a list of command-line options used in the run. By default, messages are printed to a log file called `irun.log`. Use the `-l` option to rename the log file. Use `-nolog` if you do not want a log file.

Messages are also printed to the screen by default, except for the command-line options. Use `-nostdout` if you want to suppress printing to the screen.

Use the `-q` option to suppress the printing of the tool banner, command-line arguments, and summary messages from the tools. Use the `-Q` option if you want command-line arguments printed to the log file.

Compatibility with ncverilog for Mixed-Language

To simulate a mixed-language design with *ncverilog*, you:

1. Compiled the VHDL source files with *ncvhdl*.
2. Ran *ncverilog* with the `+mixedlang` option.

To simulate with *irun*, include all source files on the command line. For example:

```
% irun file1.vhd file2.vhd top.v
```

VHDL files are compiled before Verilog files.

If there is a top-level VHDL unit in the design, you must specify this top-level unit with the `-top` or `-vhdltop` option. Using `-top` disables the automatic calculation of top-level Verilog units, and these units will also have to be specified using `-top`.

Multi-Step Mode Simulation

The behavior and features that are supported with the multi-step invocation model are supported in *irun*.

Example 1

In this example, no `cds.lib` file is created to define libraries. All design units are compiled into a default work library called `worklib`.

```
% ncvlog buf.v
% ncvlog and2.v
% ncvhdl -v93 top.vhd
% ncelab worklib.top:a
% ncsim worklib.top:a
```

These steps can be replicated with the following `irun` command:

```
% irun buf.v and2.v -v93 top.vhd -top top:a
```

Example 2

In this example, some design files are compiled into a library called `lib2`. The following steps show how to simulate the design in multi-step invocation mode:

1. Create a `cds.lib` file to define the libraries and map them to their physical locations.

   ```
   # cds.lib
   INCLUDE $CDS_INST_DIR/tools/inca/files/cds.lib
   DEFINE lib2 ./lib2
   DEFINE worklib INCA_libs/worklib
   ```

2. Create the directories for the libraries.

   ```
   mkdir lib2
   mkdir INCA_libs
   mkdir INCA_libs/worklib
   ```

3. Define the work library in an `hdl.var` file.

   ```
   # hdl.var
   DEFINE WORK worklib
   ```

4. Compile `file1.vhd` and `file2.vhd` into `lib2`.

   ```
   % ncvhdl -work lib2 file1.vhd file2.vhd
   ```

5. Compile `top.vhd` into the work library, `worklib`.

```
% ncvhdl -v93 top.vhd
```

6. Elaborate the design.

```
% ncelab worklib.top:a
```

7. Invoke the simulator.

```
% ncsim top
```

With *irun*, the `cds.lib` and `hdl.var` files are not needed. The `-makelib` option can be used to compile `file1.vhd` and `file2.vhd` into `lib2`. The file `top.vhd` will be automatically compiled into the library `worklib` (`INCA_libs/worklib`). If the directories do not exist, *irun* creates them for you.

```
% irun -makelib ./lib2 file1.vhd file2.vhd -endlib top.vhd -v93 -top top:a
```

Then, in subsequent runs, you can reuse `lib2` by setting the `-reflib` option for other testbenches without using the `-makelib` option again to recompile `file1.vhd` and `file2.vhd`. For instance, using the file `top2.vhd` as a new testbench, enter the following:

```
% irun -reflib ./lib2 top2.vhd -top top2:a
```

SystemVerilog

With other invocation modes, you compile SystemVerilog files with the `ncvlog -sv` (`ncverilog +sv`) option.

```
% ncvlog -sv systemverilog_files
% ncverilog +sv systemverilog_files
```

If you include the `-sv` option on the `irun` command line, all Verilog files are compiled as SystemVerilog.

Because *irun* determines the language of a file from the file extension, it is not necessary to include `-sv` on the `irun` command line. If you use specific file extensions for SystemVerilog files, and different extensions for Verilog files, *irun* will determine the file type and invoke the appropriate compiler.

The default recognized file extensions for SystemVerilog files are: `.sv`, `.svp`, `.SV`, `.SVP`, `.vlib`, and `.VLIB`.

See "Changing the Default Set of File Extensions" on page 97 for information on how to change the list of recognized file extensions.

Use the following command to view a list of SystemVerilog command-line options:

```
% irun -helpsubject systemverilog
```

Verilog and VHDL AMS

irun determines the language of a file from the file extension.

The default recognized file extensions for Verilog AMS files are: `.vams` and `.VAMS`

The default recognized file extensions for VHDL AMS files are: `.vha`, `.VHA`, `.vhams`, `.VHAMS`, `.vhms`, and `.VHMS`.

See "Changing the Default Set of File Extensions" on page 97 for information on how to change the list of recognized file extensions.

Do not include the `-ams` option on the `irun` command line. If you include the `-ams` option, all Verilog type files and VHDL type files are compiled as AMS.

You must use the `-top` option on the command line to specify the top-level design units. This includes connect modules and cds_globals. For example, the following command includes `-top` options to specify a connect module and cds_globals:

```
% irun stim.v counter.v inv_array_v.v buf_array_v.v inv_array_bus.v \
top.vams cds_globals.vams inv.vams \
-iereport -discipline logic -timescale 1ns/1ns -propspath props.cfg \
-modelpath "models/l25u33v.scs(l25u33v_tt)" -amsfastspice -analogcontrol top.scs \
-top top -top ConnRules_5V_full -top cds_globals
```

Use the following command to view a list of AMS-related command-line options:

```
% irun -helpsubject ams
```

Regression Analysis with Desktop Manager

In releases prior to the IUS8.2-S12 release, you could take advantage of Enterprise Manager's regression analysis features only if the simulation runs were invoked using Enterprise Manager's internal runner. Enterprise Manager creates a verification session output file (VSOF) that contains information about a set of runs that are launched serially or in parallel using a Distributed Resource Manager.

Beginning with the IUS8.2-S12 release, you can perform failure and coverage analysis even if the runs were invoked outside of Enterprise Manager. In this case, the simulator creates a VSOF for each run (a "single-run" VSOF). A single-run VSOF created by the simulator contains only the results of a single run, including:

- General information about the run, such as the location of the log file, the start time, and the end time

- Coverage information, including the location of the coverage data file (UCD) and the coverage model file (UCM)
- Additional properties (attributes) of the run, such as the random seed value and the simulation time

You then collect the single-run VSOFs into a “collected” VSOF that can be loaded into Enterprise Manager or Desktop Manager for failure and coverage analysis.

Note: Regression analysis without runner integration is supported for Incisive Enterprise Simulator XL, with or without Specman, and requires either a Desktop Manager or an Enterprise Manager license.

To perform regression analysis without runner integration:

1. Install the EMGR release.
2. Add the Enterprise Manager environment to the run execution environment by adding *EMGR_install_dir*/bin to your PATH. For example, for csh:

   ```
   setenv PATH /cad/tools/INCISVE92/EMGR92/bin:${PATH}
   ```

3. Execute one or more runs while enabling the simulator to dump information from each run into a single-run VSOF. To trigger VSOF creation, do one of the following:

   - Include the -write_metrics option when you invoke the simulator (irun -write_metrics).
   - Set the VMANAGER_WRITE_METRICS variable. Setting this variable will trigger the creation of a single-run VSOF for every run.

     Note: In the current release, ALL is the only legal value for this variable.

     ```
     setenv VMANAGER_WRITE_METRICS ALL
     ```

   You can also set the VMANAGER_RUNS_DATA_DIR variable to specify the path to a directory where all single-run VSOFs are to be written. Setting this variable reduces the time spent searching a directory hierarchy for single-run VSOFs.

4. Collect the single-run VSOFs into a collected VSOF.

   To create a collected VSOF, invoke Enterprise Manager with the collect_runs command. The collected VSOF can be created in batch mode or in GUI mode.

   - Batch mode (collect the single-run VSOFs and exit). The following command creates a collected VSOF called all.vsof in the current working directory.

     ```
     emanager -b -c [-desktop] "collect_runs -vsof all.vsof [other_options]"
     ```

   - GUI mode (collect the single-run VSOFs and keep the GUI open and ready for session analysis):

```
emanager -p [-desktop] "collect_runs -vsof all.vsof [other_options]"
```

You can invoke Enterprise Manager to collect the results either while the runs are executing or after they have completed.

Note: To perform failure analysis, you must collect single-run VSOFs into a collected VSOF. However, you can load single-run VSOFs directly into Enterprise Manager in order to perform coverage analysis.

For details on the `collect_runs` command, see Appendix A, "Invocation and Command-Line Interface" in *Managing Regressions* in the EMGR online help library.

5. Perform failure and coverage analysis.

   Once a collected VSOF has been created, regardless of whether all the runs have completed, you can load it into Desktop Manager or Enterprise Manager and start to analyze failures and coverage results.

   The following command invokes Enterprise Manager and loads the VSOF:

   ```
   emanager -desktop -p "analyze_runs -vsof ncsim.vsof"
   ```

   See *Enterprise Manager Getting Started* in the EMGR online help library for details on Enterprise Manager's analysis features.

   Note: Analysis of sessions created outside of Enterprise Manager together with those created by Enterprise Manager's internal runner is supported.

See Regression Analysis with Desktop Manager for details on this feature and for a list of limitations.

Specman

The default recognized file extensions for Specman ***e*** files are `.e` and `.E`. Use the `-e_ext` option to change, or add to, the list of recognized extensions.

Several command-line options related to Specman have been implemented. See "Command-Line Options" on page 26 for details on these options.

This section uses three examples that are included with the Specman installation to illustrate how you can use *irun* to simulate the examples.

Verilog Example

The example used for this section is the XOR example, located in your installation at:

install_directory`/specman/examples/common_files/Verilog`

The two files are:

- `xor.v`–Behavioral XOR model written in Verilog. This XOR design takes two input bits and computes one output bit, which is the exclusive-or of the two inputs.
- `xor_verify.e`–Testing environment for the design, written in ***e***.

Running with Interpreted e Code

You can use *irun* to simulate this design with interpreted ***e*** code – that is, ***e*** code loaded into Specman, rather than compiled. The following command loads the ***e*** file into Specman, generates the HDL stubs file, invokes *ncvlog* to compile the Verilog files, and then invokes the elaborator and simulator.

```
% irun xor.v -snload xor_verify.e
```

You can include the `-gui` option to invoke the SimVision and Specview GUIs.

Running with Compiled e Code

You can precompile ***e*** files and then run Specman. If you use *irun*, the following command calls `sn_compile.sh` to compile the ***e*** file into a shared library, loads the ***e*** file into Specman, generates the HDL stubs file, invokes the Verilog compiler, the elaborator, and then the simulator. The simulator automatically loads the shared library.

```
% irun xor_verify.e xor.v
```

VHDL Example

The example used for this section is the XOR example, located in your installation at:

install_directory`/specman/examples/common_files/VHDL`

The two files are:

- `xor.v`hd–Behavioral XOR model written in VHDL.
- `xor_verify.e`–Testing environment for the design, written in ***e***.

To simulate this design with interpreted ***e*** code – that is, ***e*** code loaded into Specman, rather than compiled, use the following `irun` command:

```
% irun xor.vhd -snload xor_verify.e -top worklib.xor_try
```

This command generates the Specman stubs file, loads the ***e*** file, invokes *ncvhdl* to compile the VHDL files, and then invokes the elaborator and simulator. It is not necessary to

instantiate SPECMAN_REFERENCE in your VHDL code. The -top option is required because the top-level unit is VHDL.

Mixed-Language Example

The example used for this section is the XOR example, located in your installation at:

install_directory/specman/examples/common_files/mixed

The three files are:

- xor_specman.vhd–Defines the topmost VHDL entity xor_top, which instantiates xor_try (Verilog module) and the Specman foreign entity.
- xor.v–Behavioral XOR model written in Verilog.
- xor_verify.e–Testing environment for the design, written in ***e***.

The following irun command generates the Specman Verilog and VHDL stubs file, loads the ***e*** file, invokes *ncvlog* to compile the Verilog files, invokes *ncvhdl* to compile the VHDL files, and then invokes the elaborator and simulator. It is not necessary to instantiate SPECMAN_REFERENCE in your VHDL code. The -top option is required because the top-level unit is VHDL.

```
% irun xor.v xor_specman.vhd -snload xor_verify.e -top xor_top
```

C and C++ Files

irun uses the *ncsc_run* compiler interface to compile C and C++ files. When possible, these files are compiled before the design is elaborated, and the compiled files, along with any object files provided on the command line, are linked into a single dynamic library that is then automatically loaded.

In some cases, C or C++ files must be compiled after elaboration. For example, when using the Direct Programming Interface (DPI) to export functions and tasks, you must include a header file in your C code. The header file can be generated using the elaborator -dpiheader option. Files to be compiled after elaboration are specified on the irun command line with the -cpost option.

Note: Files that are to be compiled before elaboration must not have dependencies on files tagged for delayed compilation.

irun includes command-line options for passing flags to the compiler and linker. Use the following command to see a list of these options:

```
% irun -helpsubject ccomp
```

SystemC

Note: When simulating a design that includes SystemC, you must include the `-sysc` option on the `irun` command line.

irun uses the *ncsc_run* compiler interface to compile C and C++ files. Most *ncsc_run* options can be used on the `irun` command line. Use the following command to view a list of command-line options related to SystemC:

```
% irun -helpsubject systemc
```

However, some *ncsc_run* options are ignored with a warning, some options are not supported in the current release and generate errors, and the use of some options is not recommended.

With *irun*, the path provided to the `-I` or `-L` option must start with a period (`.`) or a slash (`/`). For example:

```
-I./include
-L/usr1/mylib
```

Options That Are Ignored

Some *ncsc_run* options are ignored with a warning because they have no meaning in an *irun* flow. For example, with *irun* you cannot specify the simulator to run, so the `ncsc_run -use {NCSIM | NCV}` option will be ignored.

The `ncsc_run -out` *`output_directory`* option, which is used to specify the output directory for all generated data, is another option that is ignored with a warning. An `irun` command could include C API files, SystemC files, Specman C files to be compiled in different stages, DPI files, some of which may have to be compiled after elaboration, and so on. *irun* would invoke *ncsc_run* multiple times to compile the files, and must use different locations for the generated data.

The following *ncsc_run* options are ignored with a warning:

- `-edgonly`
- `-genlib`
- `-ncscrc`
- `-out`
- `-test`
- `-use`

Options That Generate Errors

In the current release, the following *ncsc_run* options are not supported and generate errors:

- `-cxxmain`
- `-dbx`
- `-efence`
- `-gdb`
- `-genobj`
- `-nosystemclink`
- `-static`
- `-stop`

The *_args Options

ncsc_run includes several options that you use to pass arguments to the VHDL compiler, the Verilog compiler, the elaborator, and the simulator.

- `-ncvhdl_args,`*`arg1`*`[,`*`argn`*`]`
- `-ncvlog_args,`*`arg1`*`[,`*`argn`*`]`
- `-ncelab_args,`*`arg1`*`[,`*`argn`*`]`
- `-ncsim_args,`*`arg1`*`[,`*`argn`*`]`

These options are supported in *irun* for transitioning legacy designs and for passing options that are not defined in *irun*. However, their general use is not recommended. Compiler, elaborator, and simulator options should be included directly on the command line.

The `irun` command also supports a `-ncsc_runargs "`*`arg_list`*`"` option to pass options to *ncsc_run*. As with the other `*_args` options, this option should be used only for passing options to *ncsc_run* that *irun* does not understand.

Debugging SystemC Code in SimVision

With *ncsc_run*, the `-gui` option causes SystemC components to be compiled with debugging flags set and invokes the simulator with SimVision. With *irun*, the `-gui` option just invokes the simulator with SimVision. To debug SystemC code in SimVision, you must include the `-g` option on the `irun` command line.

PLI/VPI/VHPI/CIF

The simplest way to incorporate a PLI application into an *irun* simulation is to use a *PLI map file*. A PLI map file associates user-defined system tasks and system functions with functions in a PLI application. The file contains a line for each user-defined system task or system function your application needs. In each line you specify:

- The name of the system task or system function.
- Additional specifications for the system task or system function.

  For a user-defined system function, you must specify the size of the return value.

  Other, optional, specifications include the name of the call function, the name of the check function, the name of the misc function, and the data value passed as the first argument to the call, check, and misc routines.

See the section "Using a PLI/VPI Map File" in the chapter "Using VPI" in the *VPI User Guide and Reference* for details on the PLI map file.

The PLI map file can be created as a separate file, which you can include at elaboration time using the `-afile` option, or at simulation time with the `-plimapfile` option. If passed at elaboration time, the system tasks and functions defined in the file are known to both *ncelab* and *ncsim*. If passed at simulation time, the system tasks and functions defined in the file are known only to *ncsim*.

```
irun -afile plimap.map [other_options] count_args.c test.v
irun -plimapfile plimap.map [other_options] count_args.c test.v
```

You can also include the information in an *access file*. An access file is a text file that lets you specify the type of access (read, write, connectivity) that you want for particular instances and portions of the design. An access file must be included at elaboration time, so if you include the PLI map information in an access file, use the `-afile` option, as shown above.

See the section “Enabling Read, Write, or Connectivity Access to Simulation Objects” in the chapter “Elaborating the Design with ncelab” in the *NC-Verilog Simulator Help* for information on the access file.

If a PLI application has already been compiled into a dynamic shared library, user-defined bootstrap routines can be accessed with the `-load*` options.

- `-loadcfc`
- `-loadfmi`
- `-loadpli1`
- `-loadvhpi`

- `-loadvpi`

Specify only the name of the bootstrap function. The shared library name part of the argument should be omitted. For example:

```
% irun test.v test1.c test2.c -loadvpi :test1_boot -loadpli1 :test2_boot ....
```

Use the following command to display a list of `irun` command-line options related to APIs:

```
% irun -helpsubject api
```

DPI

To use *irun* with DPI to import C tasks and functions, include the SystemVerilog files and the DPI C files on the command line.

```
% irun systemverilogfile.sv dpifile.c ....
```

When using DPI to export functions and tasks, you need to include a header file in your C code. The header file can be generated using the elaborator `-dpiheader` option. Files to be compiled after elaboration are specified on the `irun` command line with the `-cpost` option.

```
% irun systemverilogfile.sv -dpiheader dpiheader_file.h -cpost dpifile.c -end ....
```

See the chapter “Direct Programming Interface” in the *SystemVerilog Reference* for details on DPI.

Example

In this example:

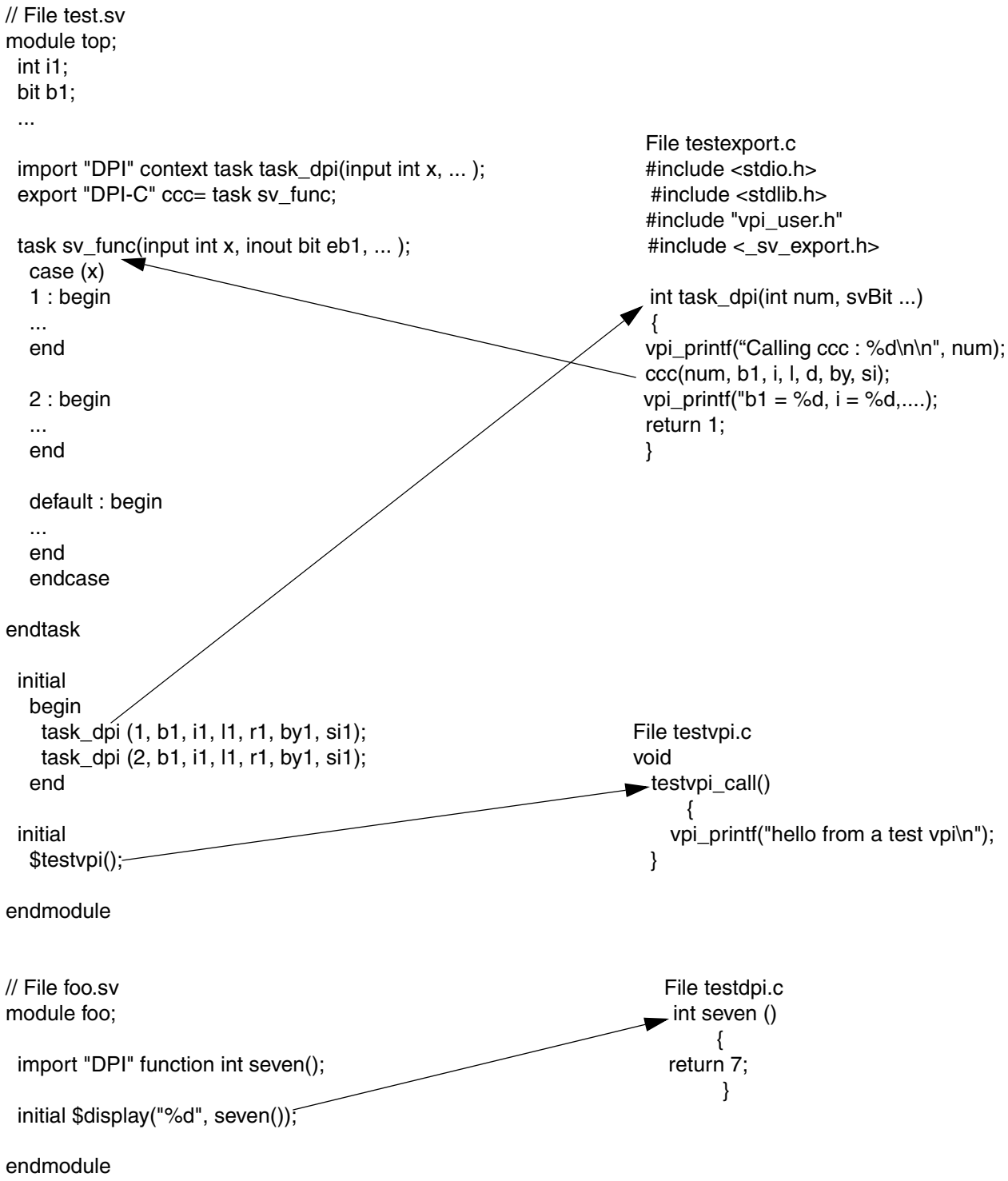
- A SystemVerilog file called `test.sv` contains a DPI `import` declaration and an `export` declaration.

  The SystemVerilog file calls a function implemented in C called `task_dpi`. This C function, in turn, calls a task implemented in SystemVerilog and called `sv_func`.

- The file `test.sv` also calls a VPI system task called `$testvpi`. A PLI map file associates this user-defined system task with a function in the VPII application. The PLI map file is called `plimap.afa`, and it contains the following line:

  ```
  $testvpi call = testvpi_call
  ```

- A second SystemVerilog file called `foo.sv` contains a DPI `import` declaration. This SystemVerilog file calls a function implemented in C called `seven`.


// File test.sv
module top;
int i1;
bit b1;
...
import "DPI" context task task_dpi(input int x, ...);
export "DPI-C" ccc= task sv_func;
task sv_func(input int x, inout bit eb1, ...);
case (x)
1 : begin
...
end
2 : begin
...
end
default : begin
...
end
endcase
endtask
initial
begin
task_dpi (1, b1, i1, l1, r1, by1, si1);
task_dpi (2, b1, i1, l1, r1, by1, si1);
end
initial
$testvpi();
endmodule
File testexport.c
#include <stdio.h>
#include <stdlib.h>
#include "vpi_user.h"
#include <_sv_export.h>
int task_dpi(int num, svBit ...)
{
vpi_printf("Calling ccc : %d\n\n", num);
ccc(num, b1, i, l, d, by, si);
vpi_printf("b1 = %d, i = %d,....);
return 1;
}
File testvpi.c
void
testvpi_call()
{
vpi_printf("hello from a test vpi\n");
}
// File foo.sv
module foo;
import "DPI" function int seven();
initial $display("%d", seven());
endmodule
File testdpi.c
int seven ()
{
return 7;
}

The design is simulated using an arguments file called `run.f`.

```
% irun -f run.f
```

The arguments file is as follows:

```
# Compile the SystemVerilog files
test.sv foo.sv

-access +rwc

# Generate a header file called _sv_export.h
-dpiheader _sv_export.h

# Delay compilation of testexport.c until after elaboration
-cpost testexport.c -end

# Compile testvpi.c and testdpi.c
testvpi.c
testdpi.c

# Include the PLI map file for the VPI application
-afile plimap.afa

# Redirect output of ncsc_run to a log file called ncsc_run.log
-log_ncsc_run ncsc_run.log
```

Incisive HDL Analysis (HAL)

You can run HAL on a snapshot generated by *irun* by including the `-hal` option on the command line.

```
% irun -hal [other_options] files
```

If you include the `-hal` option on the command line, the source files are compiled and the design is elaborated. *irun* then invokes HAL (instead of the simulator) on the snapshot.

By default, a summary report of the checks is printed to STDOUT and a verbose report, which reports all checks, is printed to the *irun* log file (`irun.log`, by default). Include the `-messages` option if you want verbose output printed to the screen.

You can use the `-log_hal` *filename* option to redirect HAL output to a specified file. For example:

```
% irun -hal -log_hal hal.log [other_options] files
```

If you include the `-gui` option with `-hal`, *irun* invokes the NCBrowse GUI rather than the SimVision GUI. See the *NCBrowse User Guide* for details on using NCBrowse.

All HAL command-line options are supported and can be included directly on the `irun` command line. Use the following command to view a list of command-line options related to HAL:

```
% irun -helpsubject hal
```

The `irun` command also supports a `-halargs` option to pass options to HAL. This option should be used only for passing options to HAL that *irun* does not understand. Do not use this option to pass supported HAL options.

Debugging HDL and e Code

As with other invocation modes, you can debug your HDL code with SimVision. Include the `-gui` option on the command line to invoke the simulator with the SimVision environment.

```
% irun -gui buf.v and2.v -v93 top.vhd -top top:a
```

See the *SimVision User Guide* for details on using the features of the SimVision environment.

If the design includes Specman ***e*** files, the `-gui` option will also invoke the Specview GUI so that you can debug your ***e*** code. For example, the following command invokes the simulator with both SimVision and Specview.

```
% irun -gui xor_verify.e xor.v xor_specman.vhd -top xor_top &
```

If you do not need to debug ***e*** code, you can invoke the simulator with only SimVision by using the `-nospecview` option.

```
% irun -gui -nospecview xor.v xor_verify.e
```

To invoke the Specview GUI without SimVision, use the `-specview` option.

```
% irun -specview xor.v xor_verify.e
```

6

Library Management

This chapter describes the following:

- The -v and -y Library Management Scheme
- Support for Compilation Units in Library Files

The -v and -y Library Management Scheme

The -v and -y library management scheme is the former scheme for binding and library management using Incisive tools. This scheme is not mentioned in any Verilog or SystemVerilog LRM; however, some legacy designs continue to depend on this method. The following sections describe using the -v and -y library management scheme with *irun*.

Note: This method is only available when using Verilog and SystemVerilog HDL designs.

Using Library Files

To use a library file, specify the `-v` command-line option with the name of the library file that you want to use. *irun* then scans this file for module and user-defined primitive (UDP) definitions that cannot be resolved in the normal source files specified.

The following example shows how to use the `-v` command-line option:

```
irun source1.v -v libfile.v
```

Using Library Directories

To use a library directory, specify the `-y` command-line option with the path to the library directory. *irun* scans this directory for files containing definitions of modules or primitives that are unresolved in the specified source file.

The following example shows how to use the `-y` command-line option:

```
irun source1.v -y /usr/me/proj/lib/cmos +libext+.v
```

Files in library directories may contain just one module or UDP definition, or they may be complete hierarchies. If they are hierarchical, then the top level of the hierarchy should be the first module declared in the file.

Library directory files are not scanned unless they have the same name as an unresolved module or UDP that has been instantiated within the normal source text.

File Extensions in Library Directories

You can specify the files in a library directory that you want *irun* to use to resolve module and UDP definitions by specifying their file extensions. If you choose to specify files with no file extensions, then each filename in a library directory must match the name of the module or UDP that they contain. To specify library directory file extensions, use the `+libext+` command-line option.

Using the +libext+ command-line option

To use this option, enter `+libext` on the command line followed immediately (no spaces) by the strings of characters that make up each extension. You must surround each extension string with two + signs. Since the + signs separate one extension string from another, the final + sign in the argument is optional.

For example, the following two command lines are equivalent:

```
irun source1.v -y /usr/me/lib/cmos +libext+.v
irun source1.v -y /usr/me/lib/cmos +libext+.v+
```

This example specifies the files named `<module_or_UDP_name>.v` in the library directory `/usr/me/lib/cmos`.

You can also specify multiple library directory file extensions. If a file that has the first extension is not found, then *irun* tries each extension that follows it on the command line until the file is found or until *irun* has tried all the listed extensions. All specified extensions must follow a single `+libext` option on the command line, as in the following example:

```
irun source1.v -y /usr/me/lib/cmos +libext+.v+.v2+
```

The extension string need not contain a period character. *irun* concatenates each string specified to the ends of the names of the modules and UDPs that need to be resolved while searching a library directory. This means that all the extensions shown below are valid:

```
irun source1.v -y /usr/me/proj/lib/cmos +libext+.v+_version_3+64+
```

Suppose the modules `NAND2`, `MUX`, and `ADDER` cannot be resolved in `source1.v` on the command line shown above. *irun* then scans the following files if they reside in the library directory `/usr/me/proj/lib/cmos`:

```
NAND2.v MUX_version_3 ADDER64
```

In some situations, you may want *irun* to scan the library directory files that have no extensions (or null extensions) along with the files that do have extensions. You can specify a null extension as two adjacent + signs, like this: `++`. Here is an example:

```
irun source1.v -y /usr/proj/lib/cmos +libext++.v+
```

This command directs *irun* to look first for files with no extension, and then for files with the extension `.v`.

Library Scan Precedence

When *irun* finds an instance of a module or UDP that cannot be resolved in the source description files, it scans for a definition in the library files or directories that you specify on

the command line. Once *irun* finds a definition, it resolves the reference and ignores all subsequent definitions of the module or UDP that it encounters.

If your library files or directories contain multiple modules or UDPs with the same name, then the scan precedence *irun* uses to find definitions becomes important.

irun has three possible methods of scan precedence:

- the default scan precedence
- the `+liborder` scan precedence
- the `+librescan` scan precedence

Default Scan Precedence

The default scan precedence that *irun* uses to search for definitions in library files or directories is as follows:

- If the unresolved instance is in a source file, *irun* scans the library files or directories in the order in which they are entered on the command line. This process begins with the left-most library file or directory, no matter where the source file appears on the command line. After *irun* scans the left-most library, it scans the others in the order in which they appear.
- If the unresolved instance is in a library file or directory, *irun* scans in the following:
  - **a.** The library file or directory that contains the unresolved instance.
  - **b.** If the instance remains unresolved, *irun* scans the remaining libraries, beginning with the one that follows the library containing the unresolved instance.

    This process continues in a circular manner—that is, irun scans the libraries as they follow on the command line and then "wraps around" to the left-most library until it visits each one.

When you use the default scan precedence, you should enter library files and directories on the command line in the order in which you want them scanned.

Consider the following example:

```
irun src1.v -y /usr/lib/NMOS src2.v -v /usr/lib/TTL/ttl.v -y /usr/lib/CMOS
```

In this example, if source file `src2.v` instantiates a module that is not defined in source files `src1.v` or `src2.v`, *irun* scans for a definition first in `/usr/lib/NMOS`, then in `/usr/lib/TTL/ttl.v`, and finally in `/usr/lib/CMOS`.

However, if the module is neither instantiated nor defined in either source file, but is instead instantiated in the library file `/usr/lib/TTL/ttl.v` then *irun* looks first for a definition in `/usr/lib/TTL/ttl.v`, then in `/usr/lib/CMOS`, and finally in `/usr/lib/NMOS`.

For instance, consider the following figure:

ttl.v

```
module ttl ();
  mod mod_inst ();
.
.
endmodule
```

src1.v

```
module s1();
  ttl ttl_inst();
.
.
endmodule
```

A module `ttl` defined in `/usr/lib/TTL/ttl.v` instantiates another module `mod`, which is neither instantiated nor defined in either source file. The software will search for `mod` in the order described above because module `s1` in source file `src1.v` instantiated `ttl`.

If you have multiple modules or UDPs with the same name in your libraries, you can control how *irun* resolves undefined instances with the default scan precedence in the following ways:

- Enter library files and directories on the command line in the order in which you want them scanned for module and UDP definitions. For example:

  ```
  irun src1.v src2.v -y /usr/lib/new -y /usr/lib/old
  ```

  In the previous example, *irun* first scans the definitions of all undefined modules and UDPs from the source files in `/usr/lib/new`. If any instances remain unresolved, *irun* scans `/usr/lib/old`. This method may not work if a library module contains an unresolved instance of another module or UDP.

  If your libraries contain multiple modules with the same name that have unresolved instances, you should use the `+librescan` option.

- Use library directory filename extensions and the `+libext+` command-line option. As described in "File Extensions in Library Directories" on page 128, *irun* searches for the extensions listed after `+libext+` from left to right in the order in which you specify them on the command line.

  To make one set of files within a library directory take precedence over another set, use different extensions on the two sets and list the extension of the dominant set immediately after the `+libext+` option. For example, a library named `/usr/lib` contains multiple modules with the same name. The newer versions of these modules

have a `.v2` extension; the older versions have a `.v1` extension. To give the modules with the `.v2` extension higher precedence, enter the following command:

```
irun src1.v src2.v -y /usr/lib +libext+.v2+.v1+
```

The module and UDP definition library files with the `.v2` extension are used to resolve undefined instantiations first because the `.v2` extension appears first after the plus option.

+liborder Scan Precedence

The `+liborder` option allows you to order a library search according to where the first instance of an unresolved module is detected. The compiler can find instances of unresolved modules in source descriptions or in library files. When the origin is a source description, `+liborder` directs the compiler to start searching in the library file or directory immediately following that source file. However, if the module instance is detected in a library file, `+liborder` initiates the search in that library.

In either case, if the module remains unresolved, `+liborder` directs *irun* to scan the remaining library files and directories in a circular order—that is, to scan libraries as they follow on the command line and then “wrap around” to preceding libraries that *irun* has not yet visited.

For example, suppose that you add the `+liborder` option to the command line, as in the following example:

```
irun source1.v -v lib1.v source2.v -v lib2.v +liborder
```

Now suppose that the compiler detects an instance of the unresolved module `dff` in the source description `source2.v`. To resolve the module, *irun* first searches for a description of `dff` in `lib2.v`. If the module remains unresolved, the search continues in `lib1.v`.

For an instance of `dff` that appears in `source1.v`, the compiler searches for the module definition first in `lib1.v`, and then in `lib2.v`.

Now consider the following command:

```
irun src1.v -y /usr/lib/NMOS \
src2.v -v /usr/lib/TTL/ttl.v -y /usr/lib/CMOS +liborder +libext+.a+.b++
```

Suppose that the compiler finds an unresolved module `ttl` in the library `/usr/lib/TTL/ttl.v`. The `+liborder` option directs the search for a description of `ttl` in the following manner:

1. Scan the library file `/usr/lib/TTL/ttl.v`.
2. Scan library files `ttl.a`, `ttl.b`, and `ttl` in the order listed in the directory `/usr/lib/CMOS`, if `ttl` remains unresolved.

3. Scan library files `ttl.a`, `ttl.b`, and `ttl` in the order listed in the directory `/usr/lib/NMOS`, if `ttl` is still unresolved.

Suppose that the description of `ttl` is detected in one of the library files in the directory `/usr/lib/CMOS`. However, in the process of resolving `ttl`, *irun* detects an instance of a new unresolved module `ttl_buf` in the description of `ttl`. The `+liborder` option directs the compiler to search for a description of `ttl_buf` in the following manner:

1. Scan the library files `ttl_buf.a`, `ttl_buf.b`, and `ttl_buf` in the order listed in the directory `/usr/lib/CMOS`.
2. Scan the library files `ttl_buf.a`, `ttl_buf.b`, and `ttl_buf` in the order listed in the directory `/usr/lib/NMOS`, if `ttl_buf` remains unresolved.
3. Scan the library file `/usr/lib/TTL/ttl.v`, if `ttl_buf` is still unresolved.

The option `+liborder` is especially useful for resolving multiple descriptions of modules or primitives that have the same name. For example, suppose you want to compare the timing performance of two modules called `ttl_fast`, each from a different library.

To make this comparison, you must create two source descriptions—`test_ttl_1.v` and `test_ttl_2.v`—that each contain instantiations of `ttl_fast`. It is critical that the compiler resolve all instances of `ttl_fast` in `test_ttl_1.v` from one library and all instances of `ttl_fast` in `test_ttl_2.v` from the other library. The following command will accomplish this task:

```
irun test_ttl_1.v -v lib1.v test_ttl_2.v -v lib2.v +liborder
```

Note: You cannot use the `+librescan` option with the `+liborder` option.

+librescan Scan Precedence

The `+librescan` option allows you to specify one order in which *irun* scans library files and directories to resolve all undefined module and UDP instances from both source files and libraries. The behavior of `+librescan` depends on the location of the undefined instance—that is, it depends on whether the instance is located in a source file, a library file, or a file within a library directory.

The Source File Scenario

When the undefined instance is located in a source file, `+librescan` acts the same as the default scan precedence—scanning begins with the left-most library on the command line and continues through the remaining libraries from left to right.

The Library File Scenario

When the undefined instance is located in a library file, and `+librescan` is in effect, *irun* does not continue to search the library file for the matching definition; instead, it begins to scan through the left-most library on the command line, and then scans the remaining libraries in the order in which they appear.

The Library Directory Scenario

Finally, when the undefined instance is located in a library directory file, and `+librescan` is in effect, *irun* scans the library directory file first to try to resolve the instance. If it remains unresolved, *irun* then begins to scan the left-most library on the command line, followed by the remaining libraries in the order that they appear.

The following example includes three library files: `lib.orig.v`, `lib.revised.v`, and `lib.latest.v`. Library `lib.orig.v` contains the original versions of all the modules. Library `lib.revised.v` contains revised versions of many of the modules from `lib.orig.v`. The library `lib.latest.v` contains the latest revisions of just a few of the modules. To resolve all undefined instances with the most up-to-date modules, use the following command:

```
irun source.v -v lib.latest.v -v lib.revised.v -v lib.orig.v +librescan
```

In this example, if `lib.orig.v` instantiates a module that is defined in that library, *irun* looks for a definition of the module instance first in `lib.latest.v`, then in `lib.revised.v`, and finally in `lib.orig.v`.

Note: You cannot use the `+librescan` option with the `+liborder` option.

Summary of Library Scan Precedence

The order that *irun* uses to search libraries for definitions of modules and UDPs depends on the location of the unresolved instance and the type of scan precedence in use. However, in all cases, all definitions in the source files specified on the command line are given first precedence. The differences between the types of scan precedence are shown in Table 6-1.

Table 6-1 Variance of Scan Precedence

Location	Scan Precedence	Library Search Order
Source file	default and `+librescan`	**1.** The left-most library file or directory on the command line **2.** The remaining libraries in the order in which they appear on the command line

Location	Scan Precedence	Library Search Order
Source file	`+liborder`	**1.** The library file or directory that follows the source files on the command line **2.** The remaining libraries in a circular order, scanning libraries as they follow on the command line, and then “wrapping around” to the left-most library and any following libraries as yet unvisited
Library file	`+librescan`	**1.** The left-most library file or directory on the command line **2.** The remaining libraries in the order in which they appear on the command line
Library file	default and `+liborder`	**1.** The library file that contains the instance **2.** The remaining libraries in a circular order, scanning libraries as they follow on the command line, and then “wrapping around” to the left-most library and any following libraries as yet unvisited
File in a Library Directory	`+librescan`	**1.** The file in a library directory that contains the instance **2.** The left-most library file or directory on the command line **3.** The remaining libraries in the order in which they appear on the command line
File in a Library Directory	default and `+liborder`	**1.** The file in the library directory that contains the instance **2.** The other files in that library directory **3.** The library file or directory that immediately follows on the command line. **4.** The remaining libraries in circular order, scanning libraries as they follow on the command line, and then “wrapping around” to the left-most library and any following libraries as yet unvisited

Support for Compilation Units in Library Files

A compilation unit is a collection of one or more source files compiled together.

SystemVerilog extends Verilog by allowing declarations outside of a module, interface, package, or program. Each compilation unit has a compilation unit scope (cu-scope), which contains all of the external declarations made across all files within the compilation unit. Unlike global declarations, which are shared by all of the modules that make up a design, cu-scope declarations are visible only to the source files that make up the compilation unit.

Prior to release 13.2, the *irun* -v and -y library management scheme did not support the compilation unit scope properly. Going forward, the defined behavior is to encapsulate all parsed elements in a compilation unit scope from the first source file parsed to the last library file parsed, and so on, until all encountered module and UDP instances are resolved.

For example, consider the following design:

File / Directory	Filename	Module Definitions	Instantiations
Source file	**top.v**	top	moda::inst_a()
lib1.v *(Library file)*	**lib1.v**	mod11 mod21	dummy::inst_dummy1() dummy::inst_dummy2()
lib2 *(Library directory)*	**moda.v** **dummy.v**	moda dummy	modb::inst_b() dummy::inst_dummy3()
lib3 *(Library directory)*	**modb.v** **dummy.v**	modb dummy	- dummy::inst_dummy5()
lib3 *(Library directory)*	**modc.v**	modc	dummy::inst_dummy6()
lib4.v *(Library file)*	**lib4.v**	mod41 mod42	dummy::inst_dummy7() dummy::inst_dummy8()

And suppose you use the following *irun* command to search the libraries for definitions of modules and UDPs that are unresolved in `top.v`:

```
irun top.v +libext+.v+ -v lib1.v -y lib2 -y lib3 -v lib4.v
```

In this example, *irun* starts by parsing the source file `top.v`, then the library file `lib1.v`, followed by the library directory file `/lib2/moda.v`, another library directory file `/lib3/modb.v`, and finally the library file `lib4.v`. The cu-scope contains all definitions outside of the module's scope using the specified files only. The other files in the library directory will not be included in the scope.

Protecting Compilation Unit Scope Elements

There are situations when the same library file may be parsed multiple times to bind unresolved instances. In order to protect cu-scope elements, designers should use unique compile-time directives as shown:

```
`ifndef DONE
    `define DONE
    bind moda checker inst_chk();
`endif
```

Index

Symbols

Numerics

A

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z