

JanusHDL

User manual

Everything JanusHDL does, and how.

Version 0.8.0

Contents

1. Welcome
2. Installing
3. Quick start
4. Running simulations from the command line
5. Projects
6. Libraries
7. Diagnostics, rules and compatibility mode
8. The GUI
9. The waveform viewer
10. Insight panels
11. The code editor
12. External editors and the language server
13. Running existing scripts
14. Determinism and races
15. Command reference
16. Troubleshooting and known limits
17. Glossary

Welcome

JanusHDL is an HDL simulator with a graphical workbench. It analyzes, elaborates and runs VHDL designs and their testbenches, from VHDL-1993 to VHDL-2019, with PSL, and it shows what happened: every signal's waveform, and why a signal has the value it has. Verilog and SystemVerilog come next.

JanusHDL is free of charge, for anyone and for any use, commercial work included. You may share unchanged copies of the download: with colleagues, on a shared drive, or in a CI image.

This manual is for FPGA and ASIC engineers who write VHDL and run testbenches, with UVVM, OSVVM, VUnit or a setup of their own, on Windows or Linux. It assumes you know VHDL and simulation; it does not assume you know JanusHDL.

What is in the download

One download holds three programs and a set of small front ends:

Program	What it is for
<code>jhd1</code>	The command line: analyze, elaborate and run a design; record its waveforms. Made for scripts and continuous integration.
<code>jhd1w</code>	The window: a waveform viewer, a design browser, a code editor, projects, and the insight panels that explain values. It runs simulations too.
<code>jhd1-lsp</code>	A language server: JanusHDL's own analyzer in the editor you already use.
<code>compat/</code>	Small programs named after the commands that existing simulation scripts call, so those scripts can run on JanusHDL unchanged. See Running existing scripts .

There is nothing else to install: the standard packages (`std` , and `ieee` from IEEE's official sources) are built in, and so are the legacy arithmetic packages that many designs still use. Verification libraries such as UVVM, OSVVM and VUnit are not part of the download; you get them from their own projects. You analyze their sources with your design, or compile UVVM and OSVVM once with `jhd1 lib compile` , after which a design that names their libraries finds them, as [Libraries](#) shows.

What JanusHDL does

- **Runs VHDL from source.** One command analyzes your files in order, elaborates the top level and runs it. Reports and failed assertions print with their time, delta cycle and source location, and the exit code tells a script whether the run passed.
- **Checks against the standard.** Every analysis message names its rule and the section of the IEEE 1076 standard (the LRM) it comes from, and says how to fix the code. Code that the standard forbids but whose meaning is clear can still be run, in compatibility mode.

- **Gives the same result every time.** The same design and inputs produce the same output on every run and every machine. You can also run the processes of each simulation cycle in another fixed order, to find testbenches that pass only by luck.
- **Records everything.** A run can record every signal into a waveform database, written as the simulation runs. The database also holds the design's structure (which processes drive and read each signal, and where in the source) and the delta cycles of every time step.
- **Shows waveforms of any size.** The viewer reads a recording a block at a time, as the screen needs it. A recording of 1.8 billion value changes (a 5.2 GB file) opened in under 4 seconds, in less than 1 GB of memory (measured).
- **Explains values.** Click a signal at a point in time and the **Why this value?** panel shows the processes that drive it, the signals they read and which of those changed last. Follow an **x** back to where it came from, step by step.
- **Edits code.** The window has a code editor with the analyzer's errors and warnings as you type, find and replace, folding and several cursors.
- **Keeps a design together.** A project file holds the sources with their libraries and VHDL versions, the top level, generics, stop time and options, in a text file made for version control.

Platforms

Platform	Download
Windows 10 and 11, x86-64	A zip file
Linux, x86-64	A tar.gz file
Linux, 64-bit ARM (tested on a Raspberry Pi 5 with Raspberry Pi OS, 64-bit)	A tar.gz file

The programs are the same on every platform, and so are their results. [Installing](#) covers each.

JanusHDL is a preview

The current builds are previews. They are complete enough for real testbenches, and they are still changing: options, file formats and the window may change between versions. Project files and wave setup files carry a version number, so a newer build reads what an older one wrote, and a build tells you when a file is newer than it understands. Where something is not supported yet, JanusHDL says so in the message, by name, rather than failing silently. [Troubleshooting and known limits](#) lists what is missing today.

How this manual is organized

Not sure which way of working fits you: the window with or without a project, the command line, your existing scripts, or your own editor? [Ways to work with JanusHDL](#), on the web site, sums them up and says where each is explained.

Chapter	Read it when you want to
Installing	Install JanusHDL on Windows or Linux

Chapter	Read it when you want to
Quick start	Run a first testbench and look at its waveforms in five minutes
Running simulations from the command line	Use <code>jhd1</code> in scripts and CI: libraries, generics, VHDL versions, exit codes, recording
Projects	Keep a design's sources and settings in one file
Libraries	Use the built-in packages, your own libraries, UVVM, OSVVM or VUnit
Diagnostics, rules and compatibility mode	Read an error, change how a rule is reported, run code the standard forbids
The GUI	Find your way around the window and run simulations from it
The waveform viewer	Add signals, set radices, measure, find values, save wave setups
Insight panels	Find out why a signal has its value
The code editor	Edit VHDL in the window
External editors and the language server	Get JanusHDL's diagnostics in your own editor
Running existing scripts	Keep your current simulation scripts, or run VUnit
Determinism and races	Understand process order and find races
Command reference	Look up an option, a command or a key
Troubleshooting and known limits	Solve a problem, or check whether something is supported
Glossary	Look up a term

Conventions

- `Monospace` marks commands, options, file names and VHDL code. Words in capitals in a command, such as `FILE.vhd` or `NAME`, stand for what you fill in.
- **Bold** marks the names of menus, buttons, tabs and other parts of the window. **File** → **Open project...** means the **Open project...** item of the **File** menu.
- Keys are written as you press them: `Ctrl+Shift+P` means holding Ctrl and Shift while pressing P.
- Command examples are written for a Linux shell (or Git Bash on Windows); a single `jhd1` command works unchanged in PowerShell too. Where Windows needs something different, a PowerShell example follows. Paths use forward slashes, which JanusHDL accepts on Windows too.

Installing

JanusHDL needs no installer, no administrator rights and no license server. You unpack the download into a folder of your choice and, if you like, put its `bin` folder on your `PATH`. To remove it, you delete the folder.

Windows

JanusHDL runs on 64-bit Windows 10 and 11.

1. Download the zip file for Windows x86-64 from the download page.
2. Unpack it anywhere, for example to `C:\Tools`. You get a folder named after the version and platform, such as `C:\Tools\janushdl-VERSION-windows-x86_64`.
3. Add its `bin` folder to your `PATH` (see [Putting JanusHDL on the PATH](#)), or run the programs from that folder.
4. Open a **new** terminal and check that it works:

```
powershell jhdl --version
```

It prints `jhdl` and the version number.

Start the window from a terminal with `jhdlw`, or make a shortcut to `bin\jhdlw.exe` on your desktop or in the Start menu. `jhdlw.exe` is a window program: it opens no console window of its own.

To open waveform databases (`.jhw`) and project files (`.jhdlproj`) with a double click, associate those file types with `bin\jhdlw.exe`: right-click such a file in Explorer, choose **Open with** → **Choose another app**, browse to `jhdlw.exe` and tick **Always use this app**.

Linux

JanusHDL runs on 64-bit Linux on x86-64 and on 64-bit ARM, with glibc 2.28 or later: for example RHEL 8 and 9, Ubuntu 20.04 and later, and Debian 10 and later. The ARM build is tested on a Raspberry Pi 5 with Raspberry Pi OS (64-bit).

1. Download the tar.gz file for your processor: `linux-x86_64` for 64-bit x86 (PC) processors, `linux-aarch64` for 64-bit ARM.
2. Unpack it, for example into `~/tools`:

```
bash mkdir -p ~/tools tar -xzf janushdl-VERSION-linux-x86_64.tar.gz -C ~/tools
```

3. Add its `bin` folder to your `PATH`, in `~/.bashrc` for example:

```
bash export PATH=~/tools/janushdl-VERSION-linux-x86_64/bin:$PATH
```

4. Open a new shell and check:

```
bash jhdl --version
```

The command line, `jhd1`, needs nothing else. The window, `jhdlw`, needs a desktop session (X11 or Wayland) and draws with the graphics card through Vulkan or OpenGL. Without a usable graphics driver, in a virtual machine or a remote session for example, it draws in software instead (see [The window does not start](#)).

On a minimal system the window may need a system library or two that is not installed, such as `libxkbcommon-x11`. If one is missing, `jhdlw` says which, and how to install it, and stops:

```
jhdlw: the window needs the system library libxkbcommon-x11.so, which is not installed.
Install it and start jhdlw again:
  Debian, Ubuntu:  sudo apt install libxkbcommon-x11-0
  RHEL, Fedora:    sudo dnf install libxkbcommon-x11
(The command line, jhd1, does not need it.)
```

On the Raspberry Pi the window uses the Pi's own graphics, and native code (see [Native code](#)) works on ARM as on x86-64.

Putting JanusHDL on the PATH

With `bin` on your `PATH`, you can run `jhd1`, `jhdlw` and `jhd1-lsp` from any folder, and editors find the language server.

Windows, for your user only (PowerShell; open a new terminal afterwards):

```
$bin = "C:\Tools\janushdl-VERSION-windows-x86_64\bin"
[Environment]::SetEnvironmentVariable("Path", [Environment]::GetEnvironmentVariable("Path", "User") +
";$bin", "User")
```

Or use **Settings** → **System** → **About** → **Advanced system settings** → **Environment Variables**, select **Path** under your user variables, choose **Edit** and add the `bin` folder.

Linux: add the `export PATH=...` line shown above to `~/.bashrc` (or your shell's start-up file).

To run existing simulation scripts unchanged, you put one of the folders under `compat` on your `PATH` instead of `bin`, or before it; each of those folders has its own `jhd1` too. [Running existing scripts](#) explains which folder to choose.

What is in the folder

```
janushdl-VERSION-PLATFORM/
├─ bin/                      the programs
│   └─ jhd1                  the command line (jhd1.exe on Windows)
│   └─ jhdlw                 the window
│   └─ jhd1-lsp              the language server, started by editors
├─ compat/                  front ends for existing scripts, one folder
│                             per family of commands (chapter 13)
└─ vhd1/                    the sources of the built-in VHDL libraries, to read
```

```

|   |   | std/          STD: standard, textio, env, reflection
|   |   | ieee/        IEEE's packages, from IEEE 1076-2019
|   |   | ieee93/      std_logic_1164, numeric_std and numeric_bit
|   |   |              for VHDL-1993 and VHDL-2002 designs
|   |   | legacy/      the legacy arithmetic packages
|   |   | README.txt
|   |   | LICENSE-ieee.txt
|   |   | └─ AUTHORS-ieee.md
|   | └─ README.txt
|   └─ LICENSE.txt      the terms of JanusHDL
|   └─ THIRD_PARTY_LICENSES.txt  the licenses of the components JanusHDL contains

```

The VHDL sources folder

The `vhd1` folder holds the source of every built-in library, byte for byte as the programs use it. It is there to read: to look up a function's exact declaration, or to see the line a message points to. Messages that come from a built-in package cite these paths, for example `ieee/numeric_std-body.vhd1:120`; in the window, clicking such a location opens the file read-only.

Editing these files changes nothing: the programs carry their own copies. If the folder is lost, `jhd1 --write-libraries DIR` writes the same files into `DIR`.

[Libraries](#) describes what each package is for.

Where JanusHDL keeps your settings

The window keeps its settings between sessions: the theme, the window's size and place, the layout of the tabs, the recent files and runs, and the editor's settings. They are in one file:

System	File
Windows	<code>%APPDATA%\JanusHDL\data\app.ron</code>
Linux	<code>~/.local/share/janushdl/app.ron</code>

A project keeps its own layout and view beside the project file, in a folder named `.jhd1` (see [Projects](#)).

`jhd1 lib compile` keeps its record of the libraries you compile (UVVM, OSVVM) in a folder of its own, outside the JanusHDL folder: `%LOCALAPPDATA%\JanusHDL\libraries` on Windows, `~/.cache/janushdl/libraries` on Linux ([UVVM and OSVVM, compiled once](#)).

While the window runs a simulation, it records the waveform into a file in your temporary folder, under `janushdl`. The window removes its own files when it exits, and files left behind by a window that did not exit normally are removed after a day.

Updating and removing

To update, unpack the new version next to the old one and change your `PATH` to the new `bin` folder. Versions share nothing but the settings file and the record of compiled libraries above, so you can keep an older version for a while and go back to it.

Files your designs write keep working across versions, with one exception: typed files (`file of T` , other than text files and files of `character`) written by versions before the current file layout are not read; write them again ([Files the design reads and writes](#)). Analysis caches (`--cache`) are simply made again by a new version.

To remove JanusHDL, delete its folder, take it off your `PATH` , and delete the settings file and the libraries folder if you want to.

Quick start

This chapter runs a small counter and its testbench, first from the command line, then in the window. It assumes JanusHDL is installed and `jhd1` is on your `PATH` ([Installing](#)).

1. Write a design and a testbench

Make a folder and save this as `counter.vhd`:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
  generic (WIDTH : positive := 8);
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
    count : out unsigned(WIDTH - 1 downto 0)
  );
end entity;

architecture rtl of counter is
  signal n : unsigned(WIDTH - 1 downto 0) := (others => '0');
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        n <= (others => '0');
      else
        n <= n + 1;
      end if;
    end if;
  end process;
  count <= n;
end architecture;
```

And this as `counter_tb.vhd`:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.env.all;

entity counter_tb is
end entity;

architecture sim of counter_tb is
  signal clk : std_logic := '0';
```

```

signal rst    : std_logic := '1';
signal count : unsigned(7 downto 0);
begin
    clk <= not clk after 5 ns;

    dut : entity work.counter
        generic map (WIDTH => 8)
        port map (clk => clk, rst => rst, count => count);

    stimulus : process
    begin
        wait for 20 ns;
        rst <= '0';
        wait until count = 10;
        report "count reached " & to_string(to_integer(count));
        assert count = 10 severity error;
        finish;
    end process;
end architecture;

```

2. Run it

In that folder:

```
jhd1 counter.vhd counter_tb.vhd
```

JanusHDL analyzes the two files in the order given, into library `work`, elaborates the last entity it analyzed (`counter_tb`) and runs it. It prints:

```

115 ns (delta 2) counter_tb.vhd:25:12: note: count reached 10
simulation of work.counter_tb ended at 115 ns (stopped); analysis 210 ms, elaboration 6 ms, run 2 ms

```

The first line is the testbench's `report`: the simulation time, the delta cycle within that time, the place in the source, the severity and the text. The last line is JanusHDL's summary: where the run ended and why (`stopped`, because the testbench called `finish`), and how long each phase took on your machine.

The exit code is 0. Had the assertion failed, or any report of severity `error` or `failure` come up, it would be 1, so a script or a CI job can rely on it:

```
jhd1 counter.vhd counter_tb.vhd && echo "passed"
```

3. Record and look at the waveforms

Run again, recording every signal into a waveform database:

```
jhd1 counter.vhd counter_tb.vhd --wave counter.jhw
```

The window opens with the recording:

1. The **Hierarchy** tab, at the top left, shows the design: `counter_tb`, and inside it the instance `dut`; under them, **Files** lists the recording's source files. Click `counter_tb`. (Double-click `dut` to open its source, `counter.vhd`, in the code editor.)
2. The **Signals** tab, below it, lists that scope's signals with their types. Double-click `clk`, `rst` and `count` to add them to the **Waves** tab, or click the button beside the filter box to add every signal listed.
3. Press `F` to fit the whole run in view. Click in the waves to place the cursor: the **Value** column shows each signal's value there. `count` shows in hexadecimal; right-click its name and choose **Unsigned decimal** to see 0 to 10.
4. Hold `Ctrl` and turn the mouse wheel over the waves to zoom; drag to pan.

The [waveform viewer](#) covers everything else: markers, finding values, analog rows, groups and saving the view.

4. Run it in the window

The window can run the simulation itself, and keeps the design loaded so you can run it again, pause it or restart it.

1. Save a file list, `counter.f`, in the same folder. It names the files to analyze, in order:

```
text library work counter.vhd counter_tb.vhd
```

2. Start `jhd1w` (or use the window you have open) and choose **Simulate → Run simulation...** (`Ctrl+Shift+R`).
3. Type the path of `counter.f`, or press **Browse...** to pick it, and press **Run**. (The dialog also takes a top entity, a stop time, the VHDL version, generics and compatibility mode; the defaults suit this example, and what you choose is remembered for the file.)

The **Terminal** tab shows the analysis, the report and how the run ended; the waveform opens as the simulation runs, with the top level's signals shown. Press `F5` to run it again from time 0, and `Ctrl+Shift+F5` to go back to time 0 and wait there. If you change a source file, the next run analyzes it again.

You can also drop `counter.f` onto the window: it runs straight away.

5. Ask why a signal has its value

Select `count` in the waves, place the cursor anywhere after the reset, and choose **Signals → Why this value?**. The panel shows the value at the cursor, since when it has it, and the process that drives it, with a link to the line in `counter.vhd` and the values of the signals that process reads. Click the link to open the line in the code editor. [Insight panels](#) explains the three panels.

6. Keep it as a project

For anything bigger than this, keep the sources and settings in a project file. Choose **File** → **Import a file list (.f) as a project...**, pick `counter.f`, and save the project as `counter.jhdlproj`. The project's settings open: on the **Top level** page set the top to `work.counter_tb`, and on **Simulation** a stop time if you want one; press **Save**. `F5` now runs the project, and on the command line:

```
jhdl run counter.jhdlproj
```

[Projects](#) describes the project file and project mode.

Where to go next

- [Running simulations from the command line](#): libraries, generics, VHDL versions, exit codes and recording.
- [Libraries](#): how to use UVVM, OSVVM or VUnit with JanusHDL.
- [The GUI](#): the window, its tabs and its simulation controls.
- [Ways to work with JanusHDL](#), on the web site: the window with or without a project, the command line, your existing scripts, your own editor, and where each is explained.

Running simulations from the command line

`jhd1` is the command line of JanusHDL. One command analyzes your source files, elaborates the top level and runs the simulation, printing what the design reports as it goes. It is made for scripts and for continuous integration: its output is plain text, and its exit code says whether the run passed.

```
jhd1 [options] FILE.vhd... [-f FILES.f]...
jhd1 run PROJECT.jhdlproj [options]
jhd1 lib compile uvvm|osvvm FOLDER | lib list | lib remove uvvm|osvvm
```

The second form runs a project file; see [Projects](#). The third compiles UVVM or OSVVM once from your copy ([Libraries compiled once](#)). Everything in this chapter applies to both: options given after a project take precedence over the project's own settings. [Command reference](#) lists every option in one table.

The basic run

```
jhd1 counter.vhd counter_tb.vhd
```

does three things:

1. **Analysis.** The files are analyzed in the order given, into library `work`. The standard libraries (`std` and `ieee`) are built in and need no analysis.
2. **Elaboration.** The top level is elaborated: by default the entity analyzed last, here `counter_tb`.
3. **Simulation.** The design runs until nothing is left to do, until the stop time, or until the testbench stops it (with `std.env.stop` or `std.env.finish`, or a failed assertion of severity `failure`).

At the end, `jhd1` prints a summary line:

```
simulation of work.counter_tb ended at 115 ns (stopped); analysis 210 ms, elaboration 6 ms, run 2 ms
```

The reason in parentheses is one of:

Reason	Meaning
<code>nothing left to simulate</code>	No process will resume and no signal will change. The time is that of the last thing that happened.
<code>stop time reached</code>	The run reached the time given with <code>--stop-time</code> ; the time shown is the stop time, even when the last change came earlier.
<code>stopped</code>	The design stopped the run: <code>std.env.stop</code> or <code>finish</code> was called, a report reached the stop level (see Severity levels), or a run-time error occurred.
<code>stopped with status N</code>	<code>std.env.stop</code> or <code>finish</code> was called with a status other than 0. The run fails: the exit code is 1.

Options and file names can come in any order on the command line, with two exceptions: `--work` and `--file-std` apply to the files that follow them.

Analysis order

Files are analyzed in the order you give them, and a design unit must be analyzed before the units that use it: a package before the files that `use` it, an entity before its architecture. When the order is wrong, the error says what to analyze first:

```
error[undeclared]: `my_pkg` is not declared in `work`
--> tb.vhd:4:5
|
4 | use work.my_pkg.all;
|   ^^^^^^^^^^^^^
= help: library `work` has no design units yet: analyze its sources into it first, before the files
that use it
= LRM 8.3
```

If you have a list of files in no particular order, the language server can sort it for you. Write the files into a file list (see [File lists](#)) and run:

```
jhdl-lsp --order unsorted.f > sorted.f
```

It prints the same list with the files of each library in an order their dependencies allow.

Libraries

By default every file goes into library `work`. There are two ways to put files into other libraries.

`--work NAME` sets the library for the files that come after it on the command line:

```
jhdl --work util src/fifo_pkg.vhd src/fifo.vhd --work work tb/fifo_tb.vhd
```

Here `fifo_pkg.vhd` and `fifo.vhd` go into library `util`, and `fifo_tb.vhd` into `work`. The testbench then uses them with `library util; use util.fifo_pkg.all;`.

A file list (`-f`) names the files with their libraries; for anything with more than a couple of libraries it is the better choice.

File lists

A file list is a text file, usually named with the extension `.f`, that names the files to analyze in order:

```
# The FIFO and its testbench
library util
src/fifo_pkg.vhd
src/fifo.vhd
```

```
library work
tb/fifo_tb.vhd
```

- A line `library NAME` chooses the library for the files after it, up to the next `library` line. Files before any `library` line go into `work`.
- Every other line is the path of a file, relative to the folder of the file list (or absolute). The whole line is the path, spaces included; do not put quotes around it.
- Blank lines, and lines that start with `#`, are ignored.

Run it with:

```
jhdl -f fifo.f --top work.fifo_tb
```

`-f` can be given several times, and mixed with file names; everything is analyzed in the order given. This makes it easy to keep a verification library's files in one list and your design's in another:

```
jhdl --compat -f uvvm_util.f -f my_design.f --top work.my_tb
```

The same file lists work in the window (**Simulate** → **Run simulation...**), with the language server, and as the starting point of a project (**File** → **Import a file list (.f) as a project...**).

Libraries compiled once

UVVM and OSVVM can be compiled once from your copy instead of being listed in every run:

```
jhdl lib compile uvvm UVVM
jhdl lib list
```

A design that names one of their libraries then finds it without a file list, in every run. [UVVM and OSVVM, compiled once](#) describes the commands.

Choosing the top level

`--top NAME` names the design unit to elaborate:

You write	JanusHDL elaborates
<code>--top fifo_tb</code>	Entity <code>fifo_tb</code> , looked up in <code>work</code> first, then in the other libraries
<code>--top verif.fifo_tb</code>	Entity <code>fifo_tb</code> of library <code>verif</code>
<code>--top work.fifo_tb_cfg</code>	The configuration <code>fifo_tb_cfg</code> , with the architecture and bindings it gives

Without `--top`, the top is the entity analyzed last. An entity is elaborated with the architecture analyzed last for it; to choose another architecture, write a configuration and give its name.

`--elaborate-only` elaborates the top with its generics and exits without running it: a quick check that a testbench elaborates. The exit code is 0 if it does, and 1, with the errors, if not.

Generics

`-gNAME=VALUE` sets a generic of the top level:

```
jhdl -f fifo.f --top fifo_tb -gDEPTH=16 -gNAME=smoke -gVERBOSE=true
```

With a path of instance labels below the top, it sets a generic of that instance, replacing the actual its generic map gives:

```
jhdl -f fifo.f --top fifo_tb -gdut.DEPTH=32 -gdut.mem.WIDTH=8
```

Generic names are not case-sensitive. The value is written as VHDL writes a literal of the generic's type:

Type	Forms
Integer (<code>integer</code> , <code>natural</code> , ...)	<code>42</code> , <code>-7</code> , <code>1_000</code> , <code>1e3</code> , <code>16#FF#</code> , <code>2#1010#</code>
Real	<code>1.5</code> , <code>-0.25</code> , <code>2.0e-3</code> , <code>16#F.8#</code> ; a whole number (<code>2</code>), or a number with an exponent and no point (<code>1e-05</code>) too. A point needs a digit on each side
Time and other physical types	A number and a unit, with or without a space: <code>20 ns</code> , <code>20ns</code> , <code>1.5 us</code> , <code>16#A# ns</code> ; for <code>time</code> also <code>s</code> for seconds. A number without a unit is refused
Enumeration (<code>boolean</code> , <code>std_logic</code> , <code>bit</code> , your own)	A literal: <code>true</code> , <code>idle</code> (in any case), <code>'1'</code> , <code>'X'</code> . A character literal may also be written without its quotes: <code>-gRESET_LEVEL=1</code> for a <code>std_logic</code> or <code>bit</code> generic. Character literals keep their case: <code>x</code> is not <code>'X'</code>
<code>string</code> , <code>std_logic_vector</code> , <code>bit_vector</code>	The characters as they are (<code>smoke</code> , <code>0101</code>), in double quotes (<code>"0101"</code>), or a bit string literal: <code>x"A5"</code> , <code>b"1111_0000"</code> , <code>o"17"</code> , <code>8x"F"</code>
Another one-dimensional array, a record	A positional aggregate: <code>(1, 2, 3)</code>

The value then takes the generic's subtype, as an actual would: its range is checked, and an array value must have the generic's length (an unconstrained array is indexed from its index subtype's left bound, as a string literal is). On a command line, put a value with spaces or quotes in quotes for your shell: `"-gPERIOD=1.5 us"` , `'-gINIT=x"A5"'` .

A value that is not one of the type's forms is an error that names the generic and says which forms its type takes:

```
error: generic period: `20` is not a value of type TIME: give a number and a unit, such as 20 ns,
20ns or 1.5 us (the units: fs, ps, ns, us, ms, sec, min, hr)
elaboration of work.fifo_tb failed
```

Stop time

`--stop-time TIME` ends the run at that time, whatever is still scheduled:

```
jhdl -f fifo.f --stop-time 10us
jhdl -f fifo.f --stop-time "2.5 ms"
```

A time is a number and a unit: `fs`, `ps`, `ns`, `us`, `ms` or `s` (also `sec`), with or without a space between them; the number may have decimals. Without `--stop-time`, the run goes on until nothing is left to do or the design stops it. A free-running clock never runs out of things to do, so a testbench with one should end itself with `std.env.finish` (or `stop`), or be given a stop time.

VHDL versions

`--std VERSION` chooses the VHDL version: `93`, `02`, `08` or `19` (or `1993`, `2002`, `2008`, `2019`). The default is VHDL-2008.

```
jhdl --std 19 -f design.f
```

The version decides:

- **The language.** Each file is read with that version's reserved words, syntax and rules. A word reserved only in a later version can be a name in an earlier one (with a warning, `future-reserved-word`, that it will not be portable).
- **The standard packages.** VHDL-1993 and VHDL-2002 designs get the VHDL-93 forms of `std_logic_1164`, `numeric_std` and `numeric_bit`, in which `std_logic_vector`, `unsigned` and `signed` are types of their own (designs that overload on them depend on that). VHDL-2008 and VHDL-2019 designs get the current packages.
- **The width of `INTEGER` and `TIME`.** VHDL-2019 makes both 64 bits wide.
- **What is allowed.** For example, a shared variable of a type that is not protected is legal in VHDL-1993 and an error from VHDL-2002 on (a warning with `--compat`; see [Diagnostics](#)).

On the command line, `--std` applies to every file, except those that follow a `--file-std VERSION`. Those, named on the command line or in the file list of a later `-f`, are analyzed as that version; the standard packages stay those of `--std`. For example, an old VHDL-1993 IP block in a VHDL-2008 design:

```
jhdl --std 08 --file-std 93 -f legacy_ip.f --file-std 08 -f design.f
```

A project file gives a version per library or per file too (see [Projects](#)).

Conditional analysis

VHDL-2019 has conditional analysis: tool directives that choose which text is analyzed.

```
`if RUN_MODE = "fast" then
    constant C_CYCLES : natural := 100;
`else
    constant C_CYCLES : natural := 100_000;
`end if
```

`-DNAME=VALUE` defines an identifier for these directives; one that is not defined is the empty string:

```
jhdl --std 19 -f design.f -DRUN_MODE=fast
```

The standard identifiers are predefined: `VHDL_VERSION` (such as `2019`), `TOOL_TYPE` (`SIMULATION`), `TOOL_VENDOR` and `TOOL_NAME` (`JanusHDL`), `TOOL_EDITION` and `TOOL_VERSION`. Directives are VHDL-2019: in a file analyzed as an earlier version, they are an error that says to analyze the file with `--std 19`.

PSL

PSL directives (`assert`, `assume`, `restrict`, `cover`) and declarations written in VHDL-2008 or VHDL-2019 code are simulated as part of the design. To read PSL written in comments as well, the `-- psl` form, add `-- psl`:

```
-- psl default clock is rising_edge(clk);
-- psl assert always (req -> next ack);
```

```
jhdl --psl -f design.f
```

A PSL assertion that fails reports like a VHDL assertion, with its place in the source. A strong property still waiting when the simulation ends fails then.

Messages

Reports and assertions

Each `report`, and each assertion that fails, prints one line as it happens:

```
115 ns (delta 2) counter_tb.vhd:25:12: note: count reached 10
```

Part	Meaning
115 ns	The simulation time
(delta 2)	The delta cycle within that time, from 0

Part	Meaning
counter_tb.vhd:25:12	The file, line and column of the statement. The file is named by its path as you gave it, tidied: a file list's <code>lib/script/./src/a.vhd</code> shows as <code>lib/src/a.vhd</code> , with your system's separator (<code>\</code> on Windows); a project's files, and those of a library compiled with <code>jhd1 lib compile</code> , by their full path
note	The severity: <code>note</code> , <code>warning</code> , <code>error</code> or <code>failure</code>
count reached 10	The message

Text that the design writes to the standard output, with `writeline(OUTPUT, ...)` (the way UVVM and OSVVM log), prints as it is, without time or place.

Run-time errors

An error found while the simulation runs, such as an index out of range, stops the run and prints the time, the place and the process:

```
3 ns (delta 0): failure: [rte.vhd:11:5] index 5 is out of the range 0 to 3 (in process /rte/check)
simulation of work.rte ended at 3 ns (stopped); analysis 15 ms, elaboration 3 ms, run 1 ms
```

The process is named by its path in the design: here the process labelled `check` in entity `rte`.

Analysis and elaboration errors

Analysis errors and warnings print with their rule, the source line and a pointer to the place, how to fix it, and the section of the LRM:

```
error[undeclared]: `x` is not declared
--> bad.vhd:8:10
|
8 |      s <= x;
|          ^
= help: check the spelling, or add the use clause that makes it visible
= LRM 12.3
analysis failed
```

A design unit whose analysis failed is not put into its library, unless its only errors are of rules that compatibility mode accepts ([Diagnostics](#)). A unit that needs it says so, and where to look, instead of a confusing "not declared":

```
error[undeclared]: package `vvc_sb_support_pkg` is not in library `bitvis_vip_sbi`: its analysis
failed
```

[Diagnostics, rules and compatibility mode](#) explains the messages, and how to change how a rule is reported. Elaboration errors print as `error: ...` lines, followed by `elaboration of TOP failed`.

Where the output goes

Reports, assertions, the design's text output and the summary line go to the standard output. Analysis diagnostics and `jhd1`'s own errors go to the standard error. To keep everything in one log file:

```
jhd1 -f design.f > run.log 2>&1
```

Fewer messages

- `--quiet` prints only reports and assertions of severity `error` and `failure`, and run-time errors. Notes, warnings, the design's text output and the summary line are left out. Analysis diagnostics still print.
- `--ieee-warnings off` silences the assertions inside the IEEE packages, such as the warnings `numeric_std` gives for metavalues (`"NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0"`). They are on by default, as the standard's package bodies have them.

Severity levels: stopping and failing

Two levels decide what a report of each severity does:

Option	What it sets	Default
<code>--stop-level LEVEL</code>	The least severity of a report or assertion that stops the simulation	<code>failure</code>
<code>--exit-level LEVEL</code>	The least severity of a report or assertion that makes the exit code non-zero	<code>error</code>

`LEVEL` is `note`, `warning`, `error` or `failure`. With the defaults, an assertion of severity `error` lets the run go on but makes it fail, and one of severity `failure` stops it.

```
# Stop at the first error, and also fail the run on warnings:
jhd1 -f design.f --stop-level error --exit-level warning
```

A testbench that makes errors on purpose (to check that a checker catches them) can raise the exit level: `--exit-level failure`.

Exit codes

Code	Meaning
0	The run passed: analysis and elaboration succeeded, no report reached the exit level, and <code>std.env.stop</code> or <code>finish</code> gave no status other than 0. With <code>--elaborate-only</code> : analysis and elaboration succeeded.
1	Analysis or elaboration failed (also with <code>--keep-going</code> , even when the design ran), a run-time error occurred, a report reached the exit level (by default: any <code>error</code> or <code>failure</code>), <code>std.env.stop</code> or <code>finish</code> was called with a status other than 0, or the VCD file or waveform database could not be written.

Code	Meaning
2	The command line or a file it names is wrong: an unknown option, a bad value, a file, file list or project file that cannot be read.
3	The simulation used more memory than <code>--max-memory</code> allows.

`jhd1 --help` lists the codes too.

Recording waveforms

`jhd1` can record every signal of the design in two formats.

The waveform database

```
jhd1 -f design.f --wave run.jhw
jhd1w run.jhw
```

`--wave FILE` records every signal into a JanusHDL waveform database (`.jhw`), written while the simulation runs. Besides the values, it holds the design's structure: every process with its kind and place in the source, the signals it drives (with the lines that assign them) and the signals it reads. It also holds the delta cycles of each time step: which signals changed in each delta cycle, from which value to which, and which processes ran. The insight panels of the window use them ([Insight panels](#)).

The file is written in blocks, compressed, from a thread of its own, so memory does not grow with the length of the run. You can open it in `jhd1w` while the simulation is still running: the status bar says **recording**, and the view follows the file as it grows.

`--wave-deltas off` leaves the delta cycles out, for a smaller file; the waveforms are the same, and the **Delta cycles** panel then says there are none.

VCD files

```
jhd1 -f design.f --vcd run.vcd
```

`--vcd FILE` writes a Value Change Dump file, for tools that read VCD. It is written when the run ends. VCD has no place for delta cycles or for the design's structure, so those are not in it. `std_logic` and `bit` signals and vectors are written as the format defines them, with the nine `std_ulogic` values; records appear as scopes of their fields; enumerations, integers, times and strings are written as text values, and reals as reals. The time scale is 1 fs.

Which to use

	Waveform database (<code>--wave</code>)	VCD (<code>--vcd</code>)
Written	While the run goes on	When the run ends

	Waveform database (--wave)	VCD (--vcd)
Can be viewed during the run	Yes	No
Delta cycles	Yes (unless --wave-deltas off)	No
Design structure, for the insight panels	Yes	No
Size	Compressed	Plain text
Read by	jhd1w	jhd1w and other tools that read VCD

Give both to have both: `--wave run.jhw --vcd run.vcd` records the database as the run goes, and writes the VCD file from it at the end, the same file `--vcd` alone gives. A file that cannot be written is an error that says which, and the exit code is 1.

Files the design reads and writes

A design reads and writes files with `std.textio` and with file types of its own. Relative file names are relative to the folder `jhd1` runs in.

- **Text files** (file ... : text) are plain text, as `textio` writes them.
- **Files of character** hold one byte per character and nothing else: the way to read and write any binary file from VHDL.
- **Other typed files** (type `int_file` is file of integer; , files of records, of vectors) have a layout of JanusHDL's own: a short header that says what the file holds, then the values, in a byte order that is the same on every system. A file of another element type, or one written by another program, is not opened; the message names the file and says what it holds:

```
text 0 (delta 0): failure: [tf.vhd:14:5] cannot open file "data.bin": it holds values of type
INTEGER, not REAL (the element type of REAL_FILE). Read it with a file type of the element type it
was written with, or write it again with this one (in process /tf/_p0)
```

`file_open` with a status parameter gets `NAME_ERROR` instead. The header records the type's structure, not its name: a file written as a file of `integer` reads as a file of `natural`, and a record declared the same way in two packages reads the other's files.

Typed files written by versions of JanusHDL before this layout are not read: write them again. Text files and files of `character` are not affected.

`READ` comes in the form its file type declares (LRM 5.5.2): `READ(F, VALUE, LENGTH)` for a file of an array type that is not fully constrained, such as `file of string`, which gives the value read and its length; `READ(F, VALUE)` for the others. The other form is an error (`file-read-form`), which `--compat` accepts.

Native code

By default, `jhd1` compiles the parts of the design that qualify (process bodies, subprograms, methods of protected types and more) to native machine code as the simulation runs, and runs the rest in its interpreter. The results are the same either way, messages included: the interpreter is the reference, and compiled code reports its errors with the same text and places.

`--jit off` runs everything in the interpreter; `--jit on` is the default. Setting the environment variable `JHD1_JIT=off` does the same as `--jit off` for every run, including the window's.

You do not normally need to touch this. If a run with native code ever behaves differently from one with `--jit off`, that is a defect: please report it with the design that shows it.

Caching the analysis

A large design, or a verification library such as UVVM, takes a noticeable time to analyze before every run. `-cache FILE` keeps the analysis in a file and loads it on the next run, instead of analyzing again:

```
jhd1 --compat -f uvvm_uart_demo.f --top bitvis_uart_uart_vvc_demo_tb --cache uart.cache
```

The cache is used when the same build of JanusHDL is given the same files in the same order, with the same contents, each with the same VHDL version, and with the same rule levels, `-D` definitions and `--ps1`; otherwise the files are analyzed again and the cache is written anew. Give each list of files its own cache file.

An analysis with errors is not saved. One with warnings is, and a run that loads it says in one line how many warnings there were instead of showing them all again:

```
note: 1 warning from the analysis, loaded from the cache; --analyze-only shows it
```

The run that saves the cache shows them in full, and `--analyze-only` shows them again. A design that uses a library compiled with `jhd1 lib compile` is cached with the files it loaded from your copy, and analyzed again when one of them changes.

`--analyze-only` analyzes (and with `--cache` saves the analysis), then exits without elaborating. This prepares a cache once, for many runs that then each load it, for example one run per test with different generics:

```
jhd1 -f tests.f --cache tests.cache --analyze-only
jhd1 -f tests.f --cache tests.cache --top work.tb -gTEST=reset
jhd1 -f tests.f --cache tests.cache --top work.tb -gTEST=overflow
```

A project whose files have different VHDL versions is cached too, as are files given versions with `--file-std`: the saved analysis records each file's version.

Many runs can load one cache at the same time, as a test runner's parallel tests do. If they find it out of date (a source changed during the run) and save it anew at the same moment, each writes a file of its own and replaces the cache with it whole, so no run reads a half-written cache.

Running despite analysis errors

`--keep-going` reports the analysis errors and then elaborates anyway. A top level that does not depend on the design units in error runs; one that does stops with a message that says so:

```
error: work.bad cannot be elaborated: its analysis failed (the errors are above)
```

This lets you run one testbench of a library while another part of it does not analyze yet. The exit code is still 1, since the analysis failed, even when the design ran.

Memory limit

Each `jhd1` run stops with an error if it uses more than 4 GiB of memory, so that one runaway test cannot exhaust a build machine:

```
error: the simulation uses 4100 MiB, over the --max-memory limit of 4096 MiB; stopped (--max-memory 0 turns the limit off)
```

`--max-memory SIZE` changes the limit: a number of bytes, or a number with `K`, `M`, `G` or `T` (binary units), such as `512M` or `16G`. `--max-memory 0` turns it off. The exit code of a run stopped this way is 3.

Timing the phases

The summary line gives the time of analysis, elaboration and the run. To see each phase as it finishes, set `JHDL_PHASES=1`:

```
[analysis loaded in 22 ms]
[elaboration done in 2 ms]
[run done in 1 ms]
```

`loaded` means the analysis came from the cache. These lines go to the standard error. `JHDL_PEAK_MEMORY=1` prints the run's peak memory use at the end, in the same way.

In continuous integration

`jhd1` needs no license server, no network and nothing installed beyond its folder. A CI job can unpack the download (or keep it in the build image), put `bin` on the `PATH`, and run each testbench:

```
#!/usr/bin/env bash
set -e
```

```
export PATH=/opt/janushdl/bin:$PATH
jhdl -f sim/design.f --cache build/design.cache --analyze-only
for test in reset burst overflow; do
    jhdl -f sim/design.f --cache build/design.cache --top work.tb -gTEST=$test \
        --wave build/$test.jhw > build/$test.log 2>&1
done
```

A testbench that ends with `std.env.stop(1)` or `std.env.finish(1)` (any status other than 0) gets exit code 1 too. With `set -e`, the job fails at the first test whose exit code is not 0; the log and waveform of each test stay in `build/`. The same design gives the same result on every run and every machine ([Determinism and races](#)), so a failure in CI reproduces on your desk.

A job whose testbenches use UVVM or OSVVM lists their files, or compiles them first in the same job with `jhdl lib compile`; the record goes in the user's libraries folder, or in the folder the environment variable `JHDL_LIBRARIES` names.

Projects

Projects need JanusHDL 0.7.0 or later; `jhd1 --version` prints the version you have.

A project file holds what it takes to simulate a design: its source files with their libraries and VHDL versions, the top level and its generics, the stop time, and how to simulate it. It is a short text file with the extension `.jhd1proj`, made to be kept in version control next to the sources, so that everyone on a team, and the CI job, runs the design the same way.

You do not need a project to use JanusHDL: everything also works with file names or a file list on the command line. A project is worth it as soon as a design has several libraries, a library in another VHDL version, generics to set or options to remember.

Making a project

There are three ways:

- **In the window, from scratch.** **File** → **New project...** asks where to save the project file, makes an empty project there and opens its settings, where you add the sources.
- **In the window, from a file list.** **File** → **Import a file list (.f) as a project...** makes a project of a file list's files, in the same libraries and order, with their paths made relative to where you save the project. Its settings open so that you can set the top level and the rest.
- **By hand.** A project file is plain text; write it in any editor, following the format below.

The project file

Here is a complete example:

```
jhd1-project 1
# The UART and its testbench.
std 2008

library uvvm_util
  source ../uvvm/uvvm_util/src/types_pkg.vhd
  source ../uvvm/uvvm_util/src/adaptations_pkg.vhd
library legacy std=1993
  source "old ip/regs.vhd"
library work
  source src/uart.vhd
  source tb/uart_tb.vhd

top work.uart_tb
generic baud=115200
stop-time 2 ms

# UVVM needs compatibility mode (chapter 6)
compat on
psl off
```

```

deny future-reserved-word
native-code on
ieee-warnings on
stop-level failure
exit-level error

wave-setup waves/uart.wavesetup

```

The rules of the format:

- The first line that is not blank or a comment is the header, `jhd1-project 1`: the format and its version.
- Then one statement per line, its words separated by spaces. Indentation is free.
- Blank lines and lines that start with `#` are comments. A comment is always a whole line.
- A word that contains a space, such as a path, is written in double quotes; a quote inside it is doubled:
`source "my designs/top level.vhd"`.
- Paths are relative to the project file's folder, or absolute. Write them with forward slashes on every system, so the project reads the same on Windows and Linux.

Sources

Statement	Meaning
<code>std VERSION</code>	The project's VHDL version: 1993, 2002, 2008 or 2019 (93, 02, 08 and 19 are read too). It is the version of the standard packages and of every source that has no version of its own or of its library. Default 2008.
<code>library NAME</code> <code>[std=VERSION]</code>	The sources after it go into library NAME, up to the next <code>library</code> line. <code>std=</code> gives the library a VHDL version of its own. A library may have several sections; it has one version.
<code>source PATH</code> <code>[std=VERSION]</code>	A design file, analyzed into the library of its section (work before any <code>library</code> line). <code>std=</code> gives the file a version of its own.

Files are analyzed in the order of their `source` lines. Each file is read as its own VHDL version, else its library's, else the project's: with that version's reserved words, syntax and rules. So a VHDL-1993 library can use names that VHDL-2008 reserves, while the rest of the design is VHDL-2008. The standard packages, and the width of `INTEGER`, are those of the project's version.

A project need not list UVVM or OSVVM when they are compiled once with `jhd1 lib compile`: a design that names one of their libraries finds it ([UVVM and OSVVM, compiled once](#)). A project that lists a library's files uses those.

The design

Statement	Meaning
<code>top NAME</code>	The top level: <code>library.entity</code> (or <code>entity</code> , or a configuration). Without it, the entity analyzed last.

Statement	Meaning
<code>generic NAME=VALUE</code>	A generic of the top level. With a path, such as <code>uut.width=8</code> , a generic of the instance below the top, replacing its actual. The value is the rest of the line, written as VHDL writes a literal of the generic's type, as <code>-g</code> takes it (Generics): <code>8</code> , <code>16#FF#</code> , <code>"text"</code> , <code>'1'</code> or <code>1</code> , <code>x"A5"</code> , <code>10 ns</code> . One line per generic.
<code>define NAME=VALUE</code>	An identifier for VHDL-2019 conditional analysis, for <code>`if NAME = "VALUE"</code> (see Conditional analysis).
<code>stop-time TIME</code>	When a run ends, such as <code>1 ms</code> or <code>250 ns</code> (the rest of the line). Without it, a run goes on until nothing is left to do.

How to simulate

Each of these is written in every project file JanusHDL saves, so a project runs the same whatever a later version's defaults are.

Statement	Meaning	Default
<code>compat on\ off</code>	Compatibility mode: code the LRM forbids but whose meaning is clear is a warning, and simulated as written (Compatibility mode)	<code>off</code>
<code>psl on\ off</code>	Read PSL in <code>-- psl</code> comments too	<code>off</code>
<code>allow RULE, warn RULE, deny RULE</code>	Report a diagnostic rule not at all, as a warning or as an error (<code>jhd1 -- rules</code> lists the rules); one line per rule	<code>none</code>
<code>native-code on\ off</code>	Compile what qualifies to native code (Native code)	<code>on</code>
<code>ieee-warnings on\ off</code>	Whether assertions in the IEEE packages report	<code>on</code>
<code>stop-level LEVEL</code>	The least severity of a report that stops the simulation: <code>note</code> , <code>warning</code> , <code>error</code> or <code>failure</code>	<code>failure</code>
<code>exit-level LEVEL</code>	The least severity of a report that makes <code>jhd1 run</code> exit with an error	<code>error</code>
<code>order-seed N</code>	Run the processes of each simulation cycle in a fixed order drawn from <code>N</code> (Determinism and races)	the default order

Waves and recording

Statement	Meaning
<code>wave-setup PATH</code>	A wave setup to use (Wave setup files); one line each. The first is shown when a run's waveform opens in the window; the Project tab lists them all.
<code>waveform PATH</code>	The waveform database <code>jhd1 run</code> records every signal into.
<code>vcd PATH</code>	A VCD file <code>jhd1 run</code> writes at the end of the run.

The window shows a run as it goes and records it into a temporary file, whatever `waveform` says: these two lines are for runs from the command line.

Errors in a project file

A project file JanusHDL cannot read is refused, with the line, what is wrong and how to fix it:

```
error: counter.jhdlproj: line 10: a stop time is a time such as `1 ms` or `250 ns`: `stop-time soon`
```

A statement given twice where it may be given once is an error: `std`, `top`, `stop-time`, the on/off switches, `stop-level`, `exit-level`, `order-seed`, `waveform` and `vcd`. So is a generic, an identifier or a rule given twice, and a library given two VHDL versions. This is on purpose: when two people's edits of a project are merged, the result should not leave a choice between them to chance.

Newer and older versions

Later versions of JanusHDL may add statements and options. A version that meets a statement or option it does not know skips that line and says so, with its line number, so that a project written by a newer version still opens:

```
warning: uart.jhdlproj: line 11: this build does not know `frobinate` (a newer JanusHDL may have written it); the line is skipped
```

A change that an older version would misread raises the number in the header; a version refuses a project file whose header is newer than it reads, and says which version it reads.

Running a project from the command line

```
jhd1 run uart.jhdlproj
```

runs the project: its sources in their libraries, each file as its VHDL version, its top level and generics, its stop time and its options, recording into its `waveform` and `vcd` files if it names them. `jhd1 uart.jhdlproj`, with the project first, does the same.

Options given after the project take precedence over the project's own:

```
jhd1 run uart.jhdlproj --stop-time 10us -gbaud=9600 --wave quick.jhw
```

runs the project for 10 μ s, with another baud rate, recording into `quick.jhw`. A generic given on the command line replaces the project's generic of the same name; the project's other generics still apply.

- Relative paths in the project are relative to its folder, wherever you run `jhd1` from.
- The exit code follows the project's `exit-level` (by default, a report of severity `error` or worse fails the run), unless `--exit-level` changes it.
- A project file must come first: `jhd1` refuses one given among other files, and says how to run it.

- `--cache` works for a project whose files have different VHDL versions too: the saved analysis records each file's version.

Project mode in the window

Opening a project

- **File** → **Open project...**,
- a project in the **Recent** list of the start page,
- a `.jhd1proj` file dropped on the window, or
- `jhd1w PROJECT.jhd1proj` on the command line.

`jhd1w --run PROJECT.jhd1proj` opens the project and runs it straight away. The window's title shows the project's name. One project is open at a time; opening another closes the first, keeping your state of it.

The Project tab

The **Project** tab, beside the **Hierarchy**, shows the open project (**View** → **Show or hide the project** brings it back if you closed it):

- At the top, three buttons: **Project settings**, **Add source files** and **Run the project**. Under them, a line with the project's VHDL version, its top level and its stop time.
- The sources, by library: each library with its own VHDL version if it has one, and each file with its own version beside it. A file that is not on disk is shown in red.
- Double-click a file to open it in the code editor. Right-click it to **Open it**, **Copy path**, or **Remove from the project** (the project is saved at once).
- Under the sources, the project's wave setups. Double-click one to show it in the wave view.

Project settings

File → **Project settings...** (`Ctrl+Shift+,`), or the cog button in the **Project** tab, opens the settings. They have four pages:

Page	What it sets
Sources	The files in analysis order, each with its library and VHDL version (its own, or its library's). Click a file to select it: Add files... adds the files you choose after it, into its library. The arrows move a file up or down; <code>x</code> removes it. Below the files, the VHDL version of each library.
Top level	The top level (<code>library.entity</code>), the project's VHDL version, the top's generics (a path such as <code>uut.width</code> sets an instance's generic), and the conditional analysis identifiers.
Simulation	The stop time, the severity that stops a run (Stop at reports of), the severity that fails a command-line run (A command-line run fails at), the process order seed, native code, compatibility mode, PSL in comments, IEEE warnings, and the diagnostic rules you want at another level than their default.
Waves	The wave setups, in order (the first is shown when a run's waveform opens), and the waveform database and VCD file that <code>jhd1 run</code> records into.

Save checks what you entered and writes the project file. If something is wrong, it says what and where, on the page concerned, and how to fix it: a top level with a space in it, a stop time that is not a time, a generic given twice, a rule listed twice. **Cancel** leaves the project as it was.

File → **Save project** writes the project file and your state of it; **File** → **Close project** closes it, keeping your state.

Running the project

With a project open, **Run** (F5), **Run for** (F10) and **Restart** (Ctrl+Shift+F5) run the project as it is saved. A run after the project or one of its sources changed analyzes it again. **Simulate** → **Run simulation...** runs it too, and a top level or stop time you type there takes precedence over the project's for that run.

The window records each run into a temporary file and shows it as it grows, whatever the project's waveform and vcd lines say; to keep a recording, run the project with `jhd1 run` (see [Recording waveforms](#)).

The first time a project's run is shown, the wave view shows the project's first wave setup. After that it shows the view you left: the signals, radices, cursor and visible time span of your last run of the project.

The editor and searches follow the project

With a project open, the code editor analyzes each of the project's files as the project does: after the files before it, into its library, as its VHDL version, with the project's compatibility mode, PSL comments, conditional analysis identifiers and rule levels, so its diagnostics are those `jhd1 run` gives ([Diagnostics](#)). When the project file changes, the open files are analyzed again. **Find in files** with **Files of the run** searches the project's sources.

The language server uses a project too, for editors other than the window's ([How the server finds your design](#)).

Each user's state

The window keeps what belongs to you, rather than to the project, apart from the project file, in a folder named `.jhd1` beside it: `.jhd1/NAME.state` for the project `NAME.jhd1proj`. It holds:

- the layout of the tabs, including tab windows and where they are on your screens;
- the files open in the code editor, with their cursors;
- the last wave view: the signals, cursor and visible time span of the last run you looked at.

It is written when you save or close the project, when the window exits, and every half minute while it runs; it is read when the project opens. A state file that cannot be read (one written by a newer version, say) is ignored, and the project opens with the window's current layout.

The state is not for version control. Add the folder to your `.gitignore`:

```
.jhd1/
```

Projects and file lists

A file list (`.f`) and a project file both name sources with their libraries, and both work on the command line and in the window. The differences:

	File list (<code>.f</code>)	Project file (<code>.jhd1proj</code>)
Sources and libraries	Yes	Yes
A VHDL version per library or file	No	Yes
Top level, generics, stop time	No (command-line options, or the Run dialog)	Yes
Simulation options and rule levels	No (command-line options)	Yes
Wave setups	No	Yes
Read by the language server	Yes	Yes
Read by <code>jhd1</code>	<code>jhd1 -f FILES.f</code>	<code>jhd1 run PROJECT.jhd1proj</code>

A file list stays useful as the shared part of several projects, or for tools that read one. **File** → **Import a file list (.f) as a project...** turns one into a project at any time.

Libraries

A VHDL design lives in libraries: `std` and `ieee` for the standard packages, `work` for your own units by default, and named libraries for anything you want to keep apart. This chapter describes what is built into JanusHDL, how to make libraries of your own, and how to use the verification libraries UVVM, OSVVM and VUnit.

How JanusHDL handles libraries

JanusHDL analyzes from source: each run analyzes the files you name, into the libraries you name, and then elaborates and runs. A library is simply the name you give a group of files, with `--work`, a `library` line in a file list, or a `library` statement in a project.

What makes this fast enough for large libraries is the cache: `--cache FILE` keeps the analysis and loads it on the next run while no file changed ([Caching the analysis](#)).

Two things keep a record of a library's files on disk. `jhd1 lib compile` does so for UVVM and OSVVM, so that a design finds them without listing their files ([UVVM and OSVVM, compiled once](#)). The front ends for existing scripts do so for each library, as the commands they stand in for are expected to ([Running existing scripts](#)).

Built-in libraries

These libraries are built in. You use them as you always would, with `library ieee;` and `use` clauses, and you never analyze them yourself. Their sources are in the `vhd1` folder of the download, to read ([The VHDL sources folder](#)).

STD

Package	Contents
<code>std.standard</code>	The predefined types and operations
<code>std.textio</code>	Text input and output, with the VHDL-2008 and VHDL-2019 additions (<code>sread</code> , <code>hwrite</code> , <code>justify</code> , <code>flush</code> , <code>tee</code> , ...)
<code>std.env</code>	<code>stop</code> , <code>finish</code> , <code>resolution_limit</code> , and the VHDL-2019 additions: environment variables, date and time, directories, the call path, file name and line, and the assertion API
<code>std.reflection</code>	VHDL-2019 type reflection (<code>'reflect</code>)

IEEE

From IEEE's official sources of the IEEE 1076-2019 packages, unmodified:

Package	Contents
<code>std_logic_1164</code>	<code>std_ulogic</code> , <code>std_logic</code> and their vectors

Package	Contents
<code>numeric_std</code> , <code>numeric_bit</code>	<code>signed</code> and <code>unsigned</code> arithmetic
<code>numeric_std_unsigned</code> , <code>numeric_bit_unsigned</code>	Arithmetic on <code>std_ulogic_vector</code> and <code>bit_vector</code> as unsigned numbers
<code>math_real</code> , <code>math_complex</code>	Real and complex mathematics
<code>fixed_pkg</code> , <code>fixed_generic_pkg</code> , <code>fixed_float_types</code>	Fixed-point types <code>sfixed</code> and <code>ufixed</code>
<code>float_pkg</code> , <code>float_generic_pkg</code>	Floating-point types
<code>std_logic_textio</code>	Kept for designs that use it; in VHDL-2008 its routines are part of <code>std_logic_1164</code>
<code>ieee_std_context</code> , <code>ieee_bit_context</code>	The standard contexts
<code>vital_timing</code> , <code>vital_primitives</code>	VITAL (IEEE 1076.4), for simulation libraries of FPGA and ASIC cells that use it

Designs analyzed as VHDL-1993 or VHDL-2002 get the VHDL-93 forms of `std_logic_1164`, `numeric_std` and `numeric_bit`, made from the same sources with the interface of those versions: in them, `std_logic_vector`, `unsigned` and `signed` are types of their own, as designs of that time expect. Their sources are in `vhdl/ieee93`.

The functions of `math_real` give exact results where the result is representable: `10.0 ** 2.0` is `100.0` and `log(1000.0, 10.0)` is `3.0`.

Legacy arithmetic packages

Many existing designs use packages that are not IEEE standards but that tools have long placed in library `ieee`. They are built in too, unchanged, under their own notice (`vhdl/legacy/NOTICE.txt`):

Package	Contents
<code>ieee.std_logic_arith</code>	<code>signed</code> and <code>unsigned</code> arithmetic of its own, <code>conv_integer</code> , <code>conv_std_logic_vector</code> , ...
<code>ieee.std_logic_unsigned</code> , <code>ieee.std_logic_signed</code>	Arithmetic on <code>std_logic_vector</code> as an unsigned or a signed number
<code>ieee.std_logic_misc</code>	Reduction functions such as <code>and_reduce</code> and <code>or_reduce</code>
<code>ieee.std_logic_textio</code>	The VHDL-93 text routines for <code>std_logic</code> (<code>read</code> , <code>hread</code> , <code>hwrite</code> , ...), for VHDL-1993 and VHDL-2002 designs

A design that uses them from library `ieee` runs as it is, in any VHDL version:

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;    -- built in: nothing to compile
```

For new code, `ieee.numeric_std` (or `numeric_std_unsigned`) is the standard way.

Your own libraries

Give a library a name in any of three ways:

- on the command line: `--work util src/util_pkg.vhd` ([Libraries](#));
- in a file list: a line `library util` before the files ([File lists](#));
- in a project: `library util`, with an optional VHDL version for the whole library ([Projects](#)).

Then use it as VHDL does:

```
library util;  
use util.util_pkg.all;
```

Inside the files of a library, `work` means that library. Library names are not case-sensitive.

If a design uses a library into which nothing was analyzed, the error says so, and what to do:

```
error[undeclared]: `util_pkg` is not declared in `util`  
--> tb.vhd:2:5  
  |  
2 | use util.util_pkg.all;  
  |      ^^^^^^^^^^^^^^^  
  = help: library `util` has no design units yet: analyze its sources into it first, before the files  
that use it  
  = LRM 8.3
```

UVVM and OSVVM, compiled once

Instead of listing UVVM's or OSVVM's files with every design, you can compile your copy once:

```
jhd1 lib compile uvvm UVVM  
jhd1 lib compile osvvm OsvvmLibraries  
jhd1 lib list  
jhd1 lib remove uvvm
```

`jhd1 lib compile` analyzes your copy and keeps a record of it in a folder of your own:

`%LOCALAPPDATA%\JanusHDL\libraries` on Windows, `~/cache/janushdl/libraries` on Linux. The environment variable `JHD1_LIBRARIES` chooses another folder. For UVVM, give the folder with its `script` folder; it is read through its compile orders. For OSVVM, give the `OsvvmLibraries` folder; it is read through its build scripts.

From then on, a design that names one of their libraries (`library uvvm_util;` , `context osvvm.OsvvmContext;`) finds it without listing its files: in every `jhd1` run, in the window and in the language server. The files of the units the design uses are analyzed as it uses them, each as its library is. UVVM is

analyzed in compatibility mode, which its shared variables need, whatever your design's mode. A testbench of your own that uses UVVM's utility library then runs on its own, without `--compat : jhd1 my_tb.vhd`.

- **Your copy stays in use.** The record points to its files, and each run reads them. Keep the copy where it is, and compile again after you update or move it; a file that can no longer be read is reported, with what to do.
- **A design that lists a library's files**, in a file list or a project, uses those instead, and for UVVM needs `--compat` as before.
- `jhd1 lib list` shows what is compiled and from where. `jhd1 lib remove uvvm` (or `osvvm`) forgets it and leaves your copy alone.
- **Neither compiled nor listed.** A design that names such a library is told the command that compiles it:

```
text error[undeclared]: `uvvm_util_context` is not declared in `uvvm_util` --> tb.vhd:2:9 | 2 |
context uvvm_util.uvvm_util_context; | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ = help: library `uvvm_util` has
no design units yet: compile UVVM once from your copy with `jhd1 lib compile uvvm FOLDER`, or
analyze its sources before the files that use it = LRM 8.3 - Analysis time. Each run analyzes the units
it uses from your copy. --cache keeps that analysis between runs and notices when a file of your copy
changes (Caching the analysis).
```

UVVM

UVVM is not part of JanusHDL: you get it from its own project, so you have the version you choose, and either compile it once ([UVVM and OSVVM, compiled once](#)) or analyze its sources with your design. All of UVVM analyzes in compatibility mode, and UVVM's demo testbenches run and pass.

UVVM needs compatibility mode. UVVM keeps a few settings in shared variables whose types are not protected, which VHDL-2002 and later forbid. Without compatibility mode each is an error, and the analysis fails; with it each is a warning, and UVVM runs as written ([What to expect](#)). Compiled once, UVVM is analyzed in compatibility mode whatever your design's mode, and your commands need no `--compat`. When you list UVVM's files with your design, as the commands below do, give `--compat`; in a project, write `compat on`.

Getting UVVM

```
git clone https://github.com/UVVM/UVVM.git
```

or download a release from the UVVM project and unpack it.

UVVM's libraries

UVVM is a set of libraries, each in a folder of its own: `uvvm_util` (the utility library: logging, alerts, checks, randomization, coverage), `uvvm_vvc_framework` (the framework for verification components), `bitvis_vip_scoreboard`, and one library per verification component, such as `bitvis_vip_uart` or `bitvis_vip_axistream`. Analyze them in this order, each with the libraries it needs before it:

1. `uvvm_util`
2. `uvvm_vvc_framework` (for verification components, VVCs)
3. `bitvis_vip_scoreboard` (the VVCs use it)
4. the verification components you use
5. your design and testbench

The file `script/component_list.txt` in the UVVM folder lists every component in an order that works.

File lists from UVVM's compile orders

Each UVVM component has a file `script/compile_order.txt` that lists its files in analysis order, relative to that `script` folder, under a first line `# library NAME`. With that first line made into a `library` line, it is a JanusHDL file list. In the UVVM folder, on Linux or in Git Bash:

```
for c in $(tr -d '\r' < script/component_list.txt | grep -v '^#'); do
    sed 's/^# *library/library/' $c/script/compile_order.txt > $c/script/$c.f
done
```

or in PowerShell:

```
foreach ($c in Get-Content script\component_list.txt) {
    (Get-Content "$c\script\compile_order.txt") -replace '^# *library', 'library' |
    Set-Content "$c\script\${c}.f"
}
```

This writes `uvvm_util/script/uvvm_util.f` and the others. A testbench that uses only the utility library then runs with:

```
jhdl --compat -f UVVM/uvvm_util/script/uvvm_util.f -f my_tb.f --top work.my_tb
```

and one that uses verification components with all the libraries it needs, in order. For example, UVVM's interrupt controller demo, from the UVVM folder (its testbench is in the component's `tb` folder, which its compile order does not list, so a one-line list adds it):

```
printf 'library bitvis_irqc\n../tb/irqc_demo_tb.vhd\n' > bitvis_irqc/script/demo.f
jhdl --compat \
    -f uvvm_util/script/uvvm_util.f \
    -f uvvm_vvc_framework/script/uvvm_vvc_framework.f \
    -f bitvis_vip_scoreboard/script/bitvis_vip_scoreboard.f \
    -f bitvis_vip_sbi/script/bitvis_vip_sbi.f \
    -f bitvis_irqc/script/bitvis_irqc.f \
    -f bitvis_irqc/script/demo.f \
    --top bitvis_irqc.irqc_demo_tb
```

It ends with UVVM's verdict, and JanusHDL's summary with the time each phase took:

```
UVVM:      >> Simulation SUCCESS: No mismatch between counted and expected serious alerts
simulation of bitvis_irqc_irqc_demo_tb ended at 3.4325 µs (stopped); analysis ...
```

These commands were tried with the UVVM release of April 2026.

The verification components' compile orders also list a few files of `uvvm_vvc_framework/src_target_dependent`: those files are analyzed into each component's own library, as UVVM intends. Keep them where the compile order puts them.

UVVM in a project

In a project, list the same files under `library` statements, before your own:

```
jhdl-project 1
std 2008

library uvvm_util
  source ../UVVM/uvvm_util/src/types_pkg.vhd
  source ../UVVM/uvvm_util/src/adaptations_pkg.vhd
  # ... the rest of uvvm_util's compile order
library work
  source src/my_design.vhd
  source tb/my_tb.vhd

top work.my_tb
compat on
```

The easy way to write it: make a file list of everything, then **File** → **Import a file list (.f) as a project...** in the window.

With UVVM compiled once, the project lists only your own files and needs no `compat on`:

```
jhdl-project 1
std 2008

library work
  source src/my_design.vhd
  source tb/my_tb.vhd

top work.my_tb
```

What to expect

- **Compatibility mode.** UVVM keeps a few settings in shared variables that are not of a protected type. VHDL-2002 and later allow only protected types there (LRM 6.4.2.4), so without `--compat` each is an error, and the help says what to do:

```
text error[unprotected-shared-variable]: a shared variable must be of a protected type, not
T_LOG_DESTINATION --> uvvm_util/src/string_methods_pkg.vhd:31:3 | 31 | shared variable
shared_default_log_destination : t_log_destination := C_DEFAULT_LOG_DESTINATION; |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ =
help: wrap the value in a protected type, or analyze the file as VHDL-93 (--std 93); or run with --
compat, which accepts it and simulates it as written = LRM 6.4.2.4
```

Each analysis is reported once, where it is, and the last line says how many there are and that `--compat` accepts them; the units that use those packages report nothing more. With `--compat`, each is a warning (42 of them for the UART demo), and the run is not affected. To leave the warnings out as well, add `--allow unprotected-shared-variable` (or `allow unprotected-shared-variable` in a project). With UVVM compiled once, neither the errors nor the warnings appear in your runs.

- **In the window and in editors.** With UVVM compiled once, the window and the language server find it and analyze it in compatibility mode themselves. When your design lists UVVM's files, tick **Compatibility mode** in the Run dialog, or set `compat` on in the project; for the language server, set its `compat` option ([Options](#)).
- **Logs.** UVVM prints its log as text output, which appears in the terminal and on the standard output, and also writes `UVVM_Log.txt` and `UVVM_Alert.txt` into the current folder. Run each testbench in a folder of its own if you run several at once.
- **Speed.** Analysis of UVVM takes a moment on every run. Use `--cache` with a file per testbench ([Caching the analysis](#)).
- **Errors on purpose.** A testbench that checks error handling may make its design report errors on purpose. UVVM then says SUCCESS, because it expected them, while `jhd1` exits with 1, because a report of severity `error` came up. Raise the exit level for such a testbench: `--exit-level failure`.

OSVVM

OSVVM is not part of JanusHDL either: get it from its own project, and compile it once ([UVVM and OSVVM, compiled once](#)) or analyze its sources with your design. OSVVM's own regression suite passes on JanusHDL: 314 of 314 tests. OSVVM needs no compatibility mode.

```
git clone --recursive https://github.com/OSVM/OsvmLibraries.git
```

Compiled once, `jhd1 lib compile osvvm OsvvmLibraries` reads OSVVM's build scripts itself, its verification component libraries (`osvvm_common`, `osvvm_uart`, `osvvm_axi4` and the others) included, and a testbench that names `osvvm` runs with `jhd1 my_tb.vhd`. To list the files with your design instead:

OSVVM's build scripts (`.pro` files) choose some files by simulator and VHDL version. For JanusHDL with VHDL-2008, the `osvvm` library is these files, in this order (paths relative to the `osvvmLibraries` folder):

```
library osvvm
osvvm/IfElsePkg.vhd
osvvm/OsvvmTypesPkg.vhd
osvvm/SetGetBoundedGenericPkg.vhd
osvvm/SetGetBoundedPkg_Instances.vhd
osvvm/OsvvmScriptSettingsPkg.vhd
osvvm/OsvvmScriptSettingsPkg_default.vhd
osvvm/OsvvmSettingsPkg.vhd
osvvm/OsvvmSettingsPkg_default.vhd
```

```

osvvm/TextUtilPkg.vhd
osvvm/FileUtilPkg.vhd
osvvm/ResolutionPkg.vhd
osvvm/NamePkg.vhd
osvvm/OsvvmGlobalPkg.vhd
osvvm/CoverageVendorApiPkg_default.vhd
osvvm/NameStorePkg.vhd
osvvm/TranscriptBasePkg.vhd
osvvm/deprecated/LanguageSupport2019Pkg_c.vhd
osvvm/deprecated/FileLinePathPkg_c.vhd
osvvm/deprecated/AssertApiPkg_c.vhd
osvvm/AlertLogPkg.vhd
osvvm/IdFifoPtPkg.vhd
osvvm/TbUtilPkg.vhd
osvvm/MessageListPkg.vhd
osvvm/SortListPkg_int.vhd
osvvm/RandomBasePkg.vhd
osvvm/RandomPkg.vhd
osvvm/RandomProcedurePkg.vhd
osvvm/CoveragePkg.vhd
osvvm/CoveragePtPkg.vhd
osvvm/DelayCoveragePkg.vhd
osvvm/ResizePkg.vhd
osvvm/deprecated/DynamicVectorPkg_IntV_c.vhd
osvvm/deprecated/DynamicVectorPkg_slv_c.vhd
osvvm/ScoreboardGenericPkg.vhd
osvvm/ScoreboardPkg_IntV.vhd
osvvm/ScoreboardPkg_slv.vhd
osvvm/ScoreboardPkg_int.vhd
osvvm/ScoreboardPkg_signed.vhd
osvvm/ScoreboardPkg_unsigned.vhd
osvvm/MemorySupportPkg.vhd
osvvm/MemoryGenericPkg.vhd
osvvm/MemoryPkg.vhd
osvvm/TranscriptPkg.vhd
osvvm/ReportPkg.vhd
osvvm/deprecated/RandomPkg2019_c.vhd
osvvm/ClockResetPkg.vhd
osvvm/OsvvmContext.vhd

```

Save it as `osvvm.f` in the `OsvvmLibraries` folder. The `_default` package bodies are the ones OSVVM's scripts use when they do not generate their own. The list was made from OSVVM as of October 2026; if your version has added or moved a file, its `osvvm/osvvm.pro` script shows where (and the error for a missing file names it). OSVVM's verification component libraries (`osvvm_common`, `osvvm_uart`, `osvvm_axi4` and the others) follow in the same way, each from its own `build.pro`.

A testbench then uses the context as usual:

```

library osvvm;
context osvvm.OsvvmContext;

```

```
jhdl -f OsvvmLibraries/osvvm.f -f my_tb.f --top work.my_tb
```

OSVVM writes its reports (`.ym1` files and others) into the current folder.

VUnit

VUnit is a Python test runner with VHDL libraries of its own. It runs on JanusHDL through VUnit's own package for JanusHDL, `vunit-janushdl`, which needs VUnit 5.0.0.dev16 or later. VUnit 5 is a pre-release, so install both with `--pre`:

```
pip install --pre "vunit_hdl>=5.0.0.dev16" vunit-janushdl
```

In `run.py`, add the package before the calls that depend on the simulator, and select JanusHDL with the environment variable `VUNIT_SIMULATOR=janushdl`:

```
vu.add_package("vunit-janushdl", allow_setup=True)
```

VUnit then analyzes its own libraries (`vunit_lib` and the rest) with your sources, and runs each test case in a simulation of its own; nothing needs compiling first. `--gui` records a test's waveform and opens it in `jhdlw`. A design that needs compatibility mode gets `--compat` through VUnit's option `janushdl.a_flags`; the package uses those flags for both analysis and simulation.

With VUnit 4.x, `run.py` scripts still run unchanged through one of the front ends in the `compat` folder: you put that folder first on your `PATH` and select it in VUnit as you would select a simulator.

[VUnit](#) in the chapter on existing scripts gives the exact settings.

Libraries from other tools

Simulation libraries delivered as VHDL sources, such as the cell and primitive libraries of FPGA and ASIC tools, are analyzed like any other library: name the library and list its files. Those that use VITAL find `ieee.vital_timing` and `ieee.vital_primitives` built in.

Libraries delivered only in a tool's own compiled form, or as encrypted (IEEE 1735) sources, cannot be used: JanusHDL needs the VHDL text, and does not decrypt protected code yet.

Diagnostics, rules and compatibility mode

JanusHDL checks your design against the VHDL standard, IEEE 1076 (the LRM), and reports every violation it finds: where it is, why it is wrong, how to fix it, and which section of the LRM says so. This chapter explains how to read those messages, how to change how each kind of message is reported, and how to run existing code that the standard forbids but whose meaning is clear.

The same analyzer, with the same messages, runs on the command line, in the window's code editor and in the language server.

Reading a message

```
error[undeclared]: `x` is not declared
--> bad.vhd:8:10
|
8 |      s <= x;
|      ^
= help: check the spelling, or add the use clause that makes it visible
= LRM 12.3
```

Part	Meaning
error	The severity: <code>error</code> stops the analysis of the design; <code>warning</code> does not.
[undeclared]	The rule the message belongs to: a stable name, to search for and to configure.
`x` is not declared	What is wrong.
--> bad.vhd:8:10	The file, line and column.
The source line with ^ marks	The text in question.
= help:	How to fix it.
= LRM 12.3	The section of the LRM (IEEE 1076-2019) with the rule.

When analysis finds errors, `jhd1` prints them all and ends with `analysis failed` (exit code 1). Warnings are printed and the run goes on.

Fix the first error first

A design unit with an error is not added to its library (as the LRM says). Each unit that needs it then reports, once, that its analysis failed and where its errors are:

```
error[call-before-body]: `twice` is called before its body is elaborated
--> cbb.vhd:3:27
...
error[undeclared]: package `p` is not in library `work`: its analysis failed
--> cbb.vhd:5:1
|
```

```

5 | package body p is
  | ^^^^^^^^^^^^^^^^^
  = help: fix the errors reported for `p` in cbb.vhd; this unit needs it
  = LRM 13.5

```

The second error is a consequence of the first: package `p` failed, so its body has no declaration to belong to. Fix the errors the help points to, and the errors that followed from them go away.

A unit whose only errors are of rules that compatibility mode accepts is the exception: it is added to its library as written, so the units that use it are analyzed against it and report only their own errors. Each such error is reported once, where it is, and the last line says how many there are and that `--compat` accepts them:

```

analysis failed: 16 errors, all of code the LRM forbids whose meaning is clear (unprotected-shared-
variable), which some libraries rely on; run with --compat to accept it and simulate it as written

```

Where the help points

The help line is written for the case at hand. Some examples:

Message	Help
A name used before its package is analyzed	library work has no design units yet: analyze its sources into it first, before the files that use it
A UVVM or OSVVM library that is neither compiled nor listed	library uvvm_util has no design units yet: compile UVVM once from your copy with <code>jhdl lib compile uvvm FOLDER</code> , or analyze its sources before the files that use it
A package body before its declaration	analyze the package declaration first
A unit that needs a unit whose analysis failed	fix the errors reported for <code>p</code> in <code>cbb.vhd</code> ; this unit needs it
A VHDL-2019 directive in a VHDL-2008 file	analyze the file with <code>--std 19</code> , or remove the directive
A shared variable of an unprotected type	wrap the value in a protected type, or analyze the file as VHDL-93 (<code>--std 93</code>); or run with <code>--compat</code> , ...
A number against the next word	insert a space after <code>10</code> , as in <code>10 ns</code> ; or run with <code>--compat</code> , ...
Code that compatibility mode accepts	the legal way to write it, or run with <code>--compat</code> , which accepts it and simulates it as written

The rules

Every analysis message belongs to a rule. `jhd1 --rules` lists them all with their level:

Rule	Default	What it covers
<code>syntax</code>	error	Text that is not VHDL: tokens, grammar, tool directives.
<code>future-reserved-word</code>	warning	A word used as a name that is reserved in a later VHDL version.
<code>missing-separator</code>	error; a warning with <code>--compat</code>	A number directly followed by a letter or digit, as in <code>10ns</code> (LRM 15.3).
<code>directive-warning</code>	warning	A <code>`warning</code> tool directive (LRM 24.2).
<code>directive-error</code>	error	An <code>`error</code> tool directive (LRM 24.2).
<code>unsupported</code>	error	A construct JanusHDL does not support yet.
<code>undeclared</code>	error	A name that is not declared, or not visible where it is used.
<code>redeclaration</code>	error	A name declared twice in one region (LRM 12.3).
<code>use-conflict</code>	error	A name made visible by two use clauses, so that neither is (LRM 12.4).
<code>wrong-kind</code>	error	A name that denotes the wrong kind of thing: a type where a value is needed, a signal where an entity is.
<code>type-mismatch</code>	error	A value of the wrong type.
<code>type-from-context</code>	error	A literal, aggregate, <code>null</code> or allocator whose type the context does not give.
<code>no-match</code>	error	A call that no visible subprogram matches.
<code>ambiguous</code>	error	A call or name that more than one declaration matches.
<code>assignment-target</code>	error	An assignment to something that cannot be assigned.
<code>association</code>	error	A wrong association of generics, ports or parameters.
<code>range</code>	error	A wrong range, bound or constraint.
<code>misplaced</code>	error	A statement where it is not allowed, such as <code>exit</code> outside a loop.
<code>instance</code>	error	A generic package that could not be instantiated.
<code>unprotected-shared-variable</code>	error; a warning with <code>--compat</code>	A shared variable of a type that is not protected: allowed in VHDL-1993, an error since VHDL-2002 (LRM 6.4.2.4). UVVM declares some.

Rule	Default	What it covers
<code>port-attribute-in-architecture</code>	error; a warning with <code>--compat</code>	An attribute specification for a port in an architecture instead of its entity (LRM 7.2). In compatibility mode the attribute decorates the port, as written.
<code>protected-before-body</code>	error; a warning with <code>--compat</code>	An object of a protected type created before the type's body is elaborated: in the same declarative part before the body, or in the package declaration whose body holds it (LRM 14.4.2.1).
<code>call-before-body</code>	error; a warning with <code>--compat</code>	A subprogram called while its declarative part is elaborated, before its body is: in the same declarative part before the body, or in the package declaration whose body holds it (LRM 14.4.2.1).
<code>index-out-of-range</code>	error; a warning with <code>--compat</code>	An index or slice outside its prefix's index range, both known at analysis (LRM 8.4, 8.5). Inside a generate statement it is a warning: the statement is elaborated only if its condition holds, and the error comes if the name is evaluated.
<code>file-read-form</code>	error; a warning with <code>--compat</code>	A file <code>READ</code> in the form its file type does not declare (LRM 5.5.2): with <code>LENGTH</code> for a file of a scalar, record or fully constrained array type, or without it for a file of an array type that is not fully constrained, such as <code>file of string</code> .
<code>port-subtype</code>	error; a warning with <code>--compat</code>	A signal on a scalar port whose subtype is not compatible with the port's where values move, found at elaboration (LRM 14.3.5): an <code>in</code> or <code>inout</code> port's subtype must hold the signal's range, and an <code>out</code> , <code>inout</code> or <code>buffer</code> port's range must lie in the signal's subtype. A <code>natural</code> port on an <code>integer</code> signal is not compatible.

Constructs not supported yet

A construct that JanusHDL does not support yet is reported under the rule `unsupported`, by name, at the place it is used: never as an error in your code, and never silently ignored. [Known limits](#) lists them.

Changing how a rule is reported

Three options set the level of a rule, for the whole run:

Option	Effect
<code>--allow RULE</code>	Do not report the rule's diagnostics at all.
<code>--warn RULE</code>	Report them as warnings.
<code>--deny RULE</code>	Report them as errors.

Give one option per rule; they can be repeated for several rules.

```
# UVVM in compatibility mode, without its shared-variable warnings:
jhd1 --compat -f uvvm.f -f tb.f --allow unprotected-shared-variable

# Keep code portable to newer VHDL versions:
jhd1 -f design.f --deny future-reserved-word
```

A warning can be dropped or made an error. An error can only be lowered where the design still has a meaning without the rule: those are the rules that compatibility mode accepts, marked `error (--compat: warning)` in `jhd1 --rules`. Asking for more is refused, with the reason:

```
error: `undeclared` is an error that cannot be lowered: the design has no meaning without it
```

An unknown rule name is refused with the list of rules. In a project, write the same as statements (`allow RULE`, `warn RULE`, `deny RULE`, one per line); in the window, on the **Simulation** page of **Project settings**, under **Diagnostic rules at another level than their default**.

Compatibility mode

Some code that the LRM forbids still has one clear meaning, and existing designs contain it:

- a shared variable whose type is not protected (legal only in VHDL-1993), as UVVM declares;
- an attribute of a port specified in the architecture rather than in the entity, as some FPGA flows expect;
- a number written against the next word (`10ns`);
- an index or slice outside its prefix's range, in code that may never run;
- a file `READ` in the form its file type does not declare;
- an object of a protected type created before the type's body is elaborated;
- a function called before its body is elaborated;
- a signal on a scalar port whose subtype does not hold the values that move between them, such as a `natural` port on an `integer` signal.

By default such code is an error, as the LRM makes it, and the analysis fails. The error says how to make the code legal, and that compatibility mode accepts it:

```
error[call-before-body]: `twice` is called before its body is elaborated
--> cbb.vhd:3:27
  |
3 |   constant c : integer := twice(2);
  |                               ^^^^^^^
  = help: call it from the package body, after the subprogram body, or compute the value in another
package; or run with --compat, which accepts it and simulates it as written
  = LRM 14.4.2.1
```

With `--compat`, the same code is a warning, and the design runs as written:

```
warning[call-before-body]: `twice` is called before its body is elaborated
--> cbb.vhd:3:27
|
3 |   constant c : integer := twice(2);
|                               ^^^^^^^^
= help: accepted in compatibility mode (--compat) and simulated as written; to make it legal: call
it from the package body, after the subprogram body, or compute the value in another package
= LRM 14.4.2.1
```

Turn it on with:

- `--compat` on the command line;
- `compat on` in a project file, or **Compatibility mode** on the **Simulation** page of **Project settings** in the window;
- **Compatibility mode** in the Run dialog, for a file list or VHDL file run in the window;
- the `compat` option of the language server, or `jhdl-lsp --check DIR --compat` ([Options](#)).

Of the libraries this manual covers, UVVM needs compatibility mode when you list its files with your design; compiled once with `jhdl lib compile`, it is analyzed in compatibility mode whatever your design's mode ([Libraries](#)). OSVVM and VUnit's own libraries run without it.

The rules it accepts are marked in `jhdl --rules (error (--compat: warning))`. `--allow`, `--warn` and `--deny` still set any single rule, and take precedence: `--compat --deny call-before-body` keeps that one an error.

The front ends in the `compat` folder always run in compatibility mode, so that existing scripts keep working unchanged ([Running existing scripts](#)).

Compatibility mode changes what is accepted, never what a legal design does: a design that analyzes without it simulates the same with it.

Elaboration errors

Errors found while elaborating, after analysis succeeded, print as `error:` lines with the path of the instance concerned, followed by `elaboration of TOP failed`:

```
error: /conv/u/o: an output port can be associated only with a signal name or parts of one (LRM
6.5.6.3)
elaboration of work.conv failed
```

An elaboration error of a rule that compatibility mode accepts ends with the rule's name, and `--compat` makes it a warning:

```
error: /top/u/x: the actual's subtype (-2147483648 to 2147483647) is not compatible with the subtype
of the in port (0 to 2147483647): the port does not hold every value of its actual (LRM 14.3.5; give
the port a subtype that holds the actual's range, or the actual one within the port's; or run with --
```

```
compat, which accepts it and simulates it as written) [port-subtype]
elaboration of work.top failed
```

A generic given on the command line that does not fit its type is reported the same way ([Generics](#)). A component instance that nothing binds (no entity of that name, no configuration) is left unbound with a warning, and the rest of the design runs.

Run-time errors

Errors found during the simulation, such as an index out of range or a value outside its subtype, stop the run with a message that gives the time, the place in the source and the process ([Run-time errors](#)).

Messages in the window and in editors

- In the window, analysis messages appear in the **Terminal** tab when you run a simulation; click a line with a source location to open the file there. The code editor shows the same diagnostics as you type, underlined, with the help and the LRM section when you hover over them, and lists them in the **Problems** tab ([The code editor](#)).
- In other editors, the language server shows them with their rule name and LRM section ([External editors and the language server](#)).

The GUI

`jhdlw` is JanusHDL's window. It shows waveforms, browses the design, edits code, keeps projects and runs simulations, which you can pause, continue, run for a while and restart while the design stays loaded. This chapter is about the window as a whole and about running simulations in it; the next chapters cover the [waveform viewer](#), the [insight panels](#) and the [code editor](#).

Starting the window

```
jhdlw                # the start page
jhdlw run.jhw        # open a waveform database (or a .vcd file)
jhdlw uart.jhdlproj  # open a project
jhdlw --run uart.jhdlproj # open a project and run it
jhdlw --run design.f --top work.tb --stop-time 1ms # run a file list
```

[Command reference](#) lists every argument. Options may come before or after the file, in any order; an unknown option is an error that shows the usage.

You can also drop a file on the window: a waveform (`.vcd` , `.jhw`) opens, a project (`.jhdlproj`) opens as the project, a wave setup (`.wavesetup`) is applied to the waveform shown, and a file list (`.f`) or VHDL file (`.vhd` , `.vhd1`) is simulated straight away.

The start page

Until a waveform is open, the **Waves** tab shows the start page:

- **Start:** open a project, make a new one, run a simulation, open a waveform, open the demo waveform, simulate the built-in VHDL demo design, and show all commands, each with its key.
- **Recent:** your last projects, simulation runs and waveforms, each with its folder. Click a project or waveform to open it; click a run to open the Run dialog filled in with it.

The demo waveform and the demo design are there to try the viewer without a design of your own.

The window

The window has a menu bar, a toolbar, an area of tabs, and a status bar.

Menus. **File** (waveforms, projects, wave setups, saving the waves as an image, files in the editor, demos, quit), **Edit** (find and replace, go to line, comments, folding), **View** (the command list, which tabs show, the editor's panes, the layout, zoom, theme), **Simulate** (run and control simulations, clear the terminal), **Signals** (the wave view: adding and removing signals, dividers and expressions, changes, values, markers, the insight panels) and **Help** (keyboard shortcuts, licenses, about).

Toolbar, from the left:

Control	What it does
Open waveform	Open a <code>.vcd</code> or <code>.jhw</code> file (<code>Ctrl+O</code>)
Run simulation...	The Run dialog (<code>Ctrl+Shift+R</code>)
Run / Continue / Run again	Runs, continues a paused run, or runs again from time 0 after the end (<code>F5</code>)
Pause, Stop, Restart	Control the run (<code>F6</code> , <code>Shift+F5</code> , <code>Ctrl+Shift+F5</code>)
The time box and Run for	Run on for the time in the box (<code>F10</code>); the box takes a time such as <code>10 us</code> or <code>250 ns</code> , and turns red if it is not one
Zoom to fit, Zoom in, Zoom out	The wave view's time scale (<code>F</code> , <code>+</code> , <code>-</code>)
Previous change, Next change	Move the cursor to the selected signal's previous or next change (<code>Ctrl+←</code> , <code>Ctrl+→</code>)
Search commands	The command palette (<code>Ctrl+Shift+P</code>)
The palette button	The theme

Status bar. On the left, how the simulation stands, colored by its state: in the accent color (blue in the default themes) while it is analyzing or running, with the time reached; amber while paused, with the time; red if it failed; and how it ended once it is over, such as *Simulation of work.tb ended at 115 ns (stopped) after 0.02 s*. Then the waveform: its name, its number of signals and changes, and its end time; **recording** while it is still being written, or how long it took to load. On the right, the cursor's time, or with a second cursor both times, the time between them and its frequency.

The tabs

Everything in the window is a tab:

Tab	What it shows
Hierarchy	The design's scopes: the top level, its instances, generate blocks, and the packages folded away under Packages ; under them, the design's Files . Click a scope to list its signals.
Signals	The signals of the selected scope, with their types, and a filter box. Double-click to add a signal to the waves.
Waves	The wave view (The waveform viewer); the start page until something is open.
Terminal	What a simulation prints, and the analyzer's messages (The terminal).
Project	The open project's sources by library, and its wave setups (Projects).
A tab per file in the editor	The code editor .

Tab	What it shows
Problems and Search	The editor's errors and warnings, and find in files.
Why this value? , Dataflow , Delta cycles	The insight panels .

In the **Hierarchy**, double-click an instance, or right-click it and choose **Go to source**, to open its source in the code editor at its first process. Under the scopes, **Files** lists the design's files:

- for a run, the run's files by library (a project's, a file list's, or the VHDL file run), and before a run, those files alone;
- for a waveform opened from disk, the files its recording names.

Double-click a file to open it in the code editor. A VCD file records no sources: for one, the tab has no **Files**, and **Go to source** is greyed out.

The default layout has the **Hierarchy** above the **Signals** on the left, the **Waves** in the middle and the **Terminal** below them. Tabs you open later go where they belong: the **Project** beside the **Hierarchy**, the editor's files beside the **Waves**, the **Problems**, **Search** and insight panels beside the **Terminal**.

Arranging the tabs

- Drag a tab by its title to another group of tabs, or to the edge of one to split it there; drag the dividers between groups to resize them.
- Close a tab with its ×. The **Waves** tab cannot be closed. The commands that show a tab bring it back where it belongs: **Ctrl+B** shows or hides the **Hierarchy** and **Signals**, **Ctrl+J** the **Terminal**, **Ctrl+Shift+M** the **Problems**, **View** → **Show or hide the project** the **Project**.
- **View** → **Reset the layout** puts every tab back in the default layout.

Drag an editor tab beside the **Waves** to see code and waves side by side.

Tab windows

Any tab can have a window of its own, to put on another monitor:

- Right-click its title and choose **Move to a new window**, or drag it out of the tabs and let go where no group or split is offered.
- A window takes more tabs: right-click a tab in the main window and choose **Move to the window of ...**. Inside a window, tabs are arranged and split as in the main one.
- Right-click a tab in a window and choose **Move back to the main window** to bring it back, or close the window to bring all its tabs back, each to its place.
- **View** → **Bring every tab back into the main window** gathers them all, which helps when a window was left on a monitor that is no longer connected.

The windows, their tabs and their places on the screen are kept with the layout between sessions, and with a project's layout when a project is open. Keys and dropped files work in every window.

Commands and keys

Every action of the window is a command. **Search commands** in the toolbar, `Ctrl+Shift+P` or `F1` opens the command palette: type part of a command's name, choose with `↑` and `↓`, and press `Enter`. **Help** → **Keyboard shortcuts** shows every key and mouse action in the window. [Command reference](#) lists them all.

Keys act where they make sense:

- The wave view's keys (such as `F`, `M`, the arrows, `Delete`) act while no text field has the keyboard.
- The editor's keys act while the editor has the keyboard, and then come before the same keys elsewhere: `Ctrl+S` saves the file being edited there, and the wave setup elsewhere; `Ctrl+F` finds in the file there, and filters the signals elsewhere.
- The other keys act everywhere.

Themes

View → **Theme**, or the palette button at the right of the toolbar:

- **Follow system** (the default): **JanusHDL Dark** or **JanusHDL Light**, as your operating system is set;
- **JanusHDL Dark**, **JanusHDL Light**;
- **One Dark**, **Dracula**, **Nord** and **Solarized Light**.

The theme colors the window, the waves and the code.

What the window remembers

Between sessions, the window keeps its theme, its size and place, the layout of the tabs (tab windows included), the recent projects, files and runs, the **Run for** time, and the editor's open files, text size, minimap and find options. They are kept in your user settings ([Where JanusHDL keeps your settings](#)). A project's layout and last view are kept with the project ([Each user's state](#)). The signals shown in the wave view are kept in wave setup files and with projects, not with the session.

Running simulations

The Run dialog

Simulate → **Run simulation...** (`Ctrl+Shift+R`, or the second toolbar button) opens the Run dialog:

Field	What to enter
Project (.jhdlproj), file list (.f) or VHDL file	A path; or press Browse... . Your recent runs are listed under it: one click fills them in.
Top entity	<code>library.entity</code> . Empty: the project's top, or else the entity analyzed last.
Stop time	When the run ends, such as <code>1 ms</code> . Empty: the project's stop time, or else when nothing is left to do.

For a file list or a VHDL file, the dialog has three more fields (a project has its own, in **Project settings**):

Field	What to enter
VHDL version	1993, 2002, 2008 (the default) or 2019, for every file of the run and the standard packages
Generics	The top level's generics, one <code>NAME=VALUE</code> per line, the value as VHDL writes it (Generics): <code>width=8 , msg="hello" , period=10 ns ; uut.depth=16</code> sets the generic of an instance below the top
Compatibility mode	Code the LRM forbids but whose meaning is clear is a warning, and runs as written (Compatibility mode)

Run starts it. A file list's files are analyzed in their libraries; a single VHDL file is analyzed into `work`.

What you choose is remembered for each file: choosing the file again (typed, browsed or from the recent runs) fills in its last top entity, stop time, VHDL version, generics and compatibility mode, and dropping the file on the window runs it with them. For per-library VHDL versions, rule levels, PSL in comments or other options, make a project: a project runs with all its settings ([Projects](#)).

Other ways to start a run:

- drop a `.f`, `.vhd` or `.vhd1` file on the window: it runs straight away, with the settings remembered for it;
- `jhdlw --run FILE [--top LIB.ENTITY] [--stop-time TIME]` on the command line, with the defaults for the rest, so that a command line runs the same for everyone;
- with a project open, `F5`, **Run for** or **Restart** run the project.

Controlling the run

The simulation runs in the background; the window stays responsive. The toolbar and the **Simulate** menu control it:

Control	Keys	What it does
Run / Continue / Run again	<code>F5</code>	Runs the last simulation; continues a paused one; after the end, runs it again from time 0
Pause	<code>F6</code>	Pauses the run; the waves show everything up to the time it stands at
Run for	<code>F10</code>	Runs on for the time in the toolbar's time box (<code>10 us</code> by default), then pauses
Restart	<code>Ctrl+Shift+F5</code>	Back to time 0 with the same design, waiting there for Run or Run for ; if a source file changed, it is analyzed and elaborated again first
Stop	<code>Shift+F5</code>	Ends the simulation

The design stays loaded between runs: **Restart** and **Run again** do not analyze or elaborate again unless a source file changed. **File** → **Reload waveform** (`Ctrl+R`) on a simulation's waveform runs the simulation again.

The waveform opens while the simulation runs and grows as it goes, with the top level's signals shown (or, for a project, its wave setup or your last view). The view keeps showing the whole run as it grows, until you zoom or pan. Running the same design again, or restarting it, keeps your signal list, time scale and cursor.

The window records each run into a temporary file. To keep a recording, run the design with `jhd1 --wave` ([Recording waveforms](#)), or with `jhd1 run` for a project that names a `waveform`.

The terminal

The **Terminal** tab shows, as they happen:

- text the design writes (`writeline(OUTPUT, ...)`), which is how UVVM and OSVVM log);
- reports and failed assertions, with time, delta cycle, source location and severity;
- analysis and elaboration errors and warnings, as the command line prints them;
- the simulator's own progress lines: *Analyzing 23 files...*, *Analysis done in 410 ms*, *Elaborated work.tb in 12 ms*, how the run ended.

Above the lines:

- Five buttons show or hide lines by kind: **errors and failures** and **warnings** (each with its count), **notes**, **output** (text the design wrote) and **simulator** lines.
- **Filter**: type to show only the lines that contain the text (case does not matter).
- **Clear** empties the terminal (also **Simulate** → **Clear terminal**); the copy button copies the lines shown.

The terminal follows the newest line until you scroll up; scroll to the end to follow again. It keeps the last 200,000 lines.

A source location in a line, such as `tb.vhd:13:5` (underlined), opens the file at that line in the code editor when you click the line. A location in a built-in library, such as `ieee/numeric_std-body.vhd1:120`, opens that library's source, read-only.

`Ctrl+J` shows or hides the terminal.

The waveform viewer

The **Waves** tab shows the signals you choose, one row each, with the value at the cursor and the wave over time. It shows simulations run in the window as they run, waveform databases written by `jhd1 --wave`, and VCD files.

Opening a waveform

- Run a simulation in the window ([Running simulations](#)): its waveform opens as it runs.
- **File** → **Open waveform...** (`Ctrl+O`, or the first toolbar button) opens a waveform database (`.jhw`) or a VCD file (`.vcd`).
- Drop a `.jhw` or `.vcd` file on the window, or start the window with it: `jhd1w run.jhw`.

Files are read in the background, with a progress bar in the status bar; `Escape` cancels a load. A waveform database that `jhd1 --wave` is still writing opens too: the status bar says **recording**, and the view follows the file as the simulation adds to it.

File → **Reload waveform** (`Ctrl+R`) reads the file again, after you ran the simulation again, for example. The signals that still exist stay in the view, by their path, and the zoom and cursor are kept. For a simulation run in the window, `Ctrl+R` runs it again.

The wave view

Each row has three parts: the signal's **Name**, its **Value** at the cursor, and its wave. Drag the lines between the columns to resize them. Hover over a name to see the signal's full path.

The time axis runs along the top. The area after the last recorded time is shaded. Until you add signals, the view says how: *Double-click signals in the Signals panel to add them here.*

Adding signals

1. In the **Hierarchy** tab, click a scope: the top level, an instance, a generate block, or a package under **Packages**. When a waveform opens, the **Signals** tab starts at the first scope that has signals. A double click on an instance opens its source in the code editor; the design's files are listed under **Files**, below the scopes.
2. The **Signals** tab lists the scope's signals, records and arrays, in the order they are declared, each with its type: `std_logic`, `unsigned(7 downto 0)`, `sfixed(3 downto -4)`, `state_t`, `record bus_t`. Signals already in the wave view are highlighted.
3. Double-click a signal to add it to the wave view. The button beside the filter box adds every signal listed; so does **Signals** → **Add all signals of the selected scope**.

To find a signal in a large scope, type in the filter box (`Ctrl+F` puts the keyboard there): the list shows only the signals whose names contain what you type.

Removing, reordering and picking rows

- **Select** a row by clicking its name or its wave; **↑** and **↓** select the row above or below.
- **Pick several rows:** **Ctrl** + click names to pick them one by one, **Shift** + click to pick a run of rows from the selected one.
- **Remove** the selected or picked rows with **Delete**, or right-click a name and choose **Remove. Signals** → **Remove all signals** empties the view.
- **Reorder** rows by dragging a name up or down; a line shows where it will go. An opened row moves with its parts.

Values of every VHDL type

Signals of a simulation show as their VHDL types read:

Type	How it shows
<code>std_logic</code> , <code>std_ulogic</code> , <code>bit</code> , <code>boolean</code>	A logic wave (see the table below)
Vectors (<code>std_logic_vector</code> , <code>unsigned</code> , <code>signed</code> , <code>bit_vector</code> , <code>sfixed</code> , <code>ufixed</code> , <code>float</code> , ...)	A segment per value, in the row's radix. The name keeps the vector's own index range: <code>count[7:0]</code> , <code>fx[3:-4]</code> , <code>up[0:3]</code>
Enumerations	Their literals: <code>idle</code> , <code>busy</code>
Integers	Decimal numbers
Reals	Numbers
<code>time</code> and other physical types	With their primary unit: <code>25 ns</code>
Strings	As text
Records and arrays	One row with the whole value as an aggregate, <code>(1, A5, 7, busy)</code> , that opens into a row per field or element

How values are drawn

Signal	Drawing
One bit, <code>1</code> / <code>0</code>	A high or low line, with a light fill under high
One bit, <code>H</code> / <code>L</code> (weak)	As <code>1</code> / <code>0</code> , in a lighter color
One bit, <code>Z</code>	A line at mid height, yellow
One bit, <code>U</code> , <code>X</code> , <code>W</code> or <code>-</code>	A red block
Vector, real, string, enumeration	A segment per value with the value inside; a value too long for its segment ends with <code>...</code>
Vector with any <code>U</code> , <code>X</code> , <code>W</code> or <code>-</code> bit	A red segment

Signal	Drawing
Vector with every bit <code>z</code>	A yellow line at mid height
An <code>event</code> (from a Verilog VCD file)	An arrow at each time it was triggered
Many changes within one pixel	A solid block: zoom in to see them

In hexadecimal and octal, each group of bits shows as one digit. A group with unknown bits shows their common value (`z` when all of them are `z`), or `x` if they differ. Decimal shows `x`, or the common value, if any bit is not `0` or `1`.

Radixes

Right-click the name of a vector to choose its radix:

Radix	Shows <code>11101010</code> as
Binary	<code>11101010</code>
Octal	<code>352</code>
Hexadecimal (the default)	<code>EA</code>
Unsigned decimal	<code>234</code>
Signed decimal (two's complement)	<code>-22</code>
ASCII (a character per 8 bits)	<code>".."</code> (<code>x"4869"</code> shows as <code>"Hi"</code> ; characters that cannot be printed show as <code>."</code>)
Gray code (the number it encodes)	<code>179</code>
BCD (a decimal digit per 4 bits)	<code>??</code> (<code>x"42"</code> shows as <code>42</code>)
Fixed point	<code>-1.375</code> (signed, 4 fraction bits)
Floating point	A 32-bit vector holding <code>3.25</code> in single precision shows <code>3.25</code>

- **Fixed point** opens a menu: **Signed** or **Unsigned**, and the number of **Fraction bits** (negative: the value is scaled up).
- **Floating point** opens a menu: **Half (16 bits)**, **Single (32 bits)**, **Double (64 bits)**, or any number of **Exponent bits** (IEEE 754 format).
- Decimal works for vectors of any width.
- A VHDL-2008 `sfixed` or `ufixed` signal starts in fixed point, with the binary point its index range gives; a `float_pkg` floating-point signal starts in floating point.

Opening records, arrays and vectors

A record or array row opens into a row per field or element, and a vector into a row per bit: click the arrow before its name, double-click the name, or right-click it and choose **Open** (**Close** closes it again). Records and arrays inside open in turn. An open row moves and is removed with its parts. An array with more than 4,096 elements shows as a whole and does not open.

Navigating

Action	Mouse	Keyboard
Zoom around a point	Ctrl + wheel, or a trackpad pinch, over the waves	+ (or =) / -, around the cursor, or the middle if the cursor is off-screen
Show the whole recording		F
Pan in time	Drag the waves; Shift + wheel; a horizontal trackpad scroll	← / →
Scroll through the rows	Wheel; drag the waves up or down	
Place the cursor	Click in the waves	
Place the second cursor	Shift + click in the waves	
Move the cursor to the selected signal's previous or next change		Ctrl+← / Ctrl+→
Move the cursor to the previous or next marker (or second cursor)		Alt+← / Alt+→

The keys act while no text field (such as the filter box) has the keyboard. The toolbar has buttons for zooming and for moving between changes, and the **View** and **Signals** menus have the same commands.

A click in the waves also selects the row under the pointer, and places the cursor on that signal's nearest change if one is within a few pixels, so that measuring from edge to edge is exact.

Cursors, markers and measuring

The cursor (A) is placed by a click. The **Value** column shows every row's value at the cursor, and the status bar shows its time.

The second cursor (B, dashed) is placed by Shift + click. It is labelled with its time on the axis. The time from one cursor to the other, and the frequency it makes, show under the time axis between them, and in the status bar: A 1.2 μ s · B 1.21 μ s · Δ 10 ns (100.0000 MHz) .

Markers are named points in time, drawn in orange:

- M (or **Signals** → **Add a marker at the cursor**) adds one at the cursor, named M1, M2 and so on.

- Each marker's flag on the time axis shows its name and its distance from the cursor, such as `M1 · +30 ns`; hover over it for its time.
- Drag a flag to move the marker; it snaps to a change of the selected signal nearby.
- Right-click a flag to rename the marker (type in the **Name** field), to bring the cursor there (**Cursor here**), or to remove it (**Remove marker**). Double-click a flag to bring the cursor there.
- `Alt+←` and `Alt+→` move the cursor to the previous or next marker or second cursor.
- **Signals** → **Remove the markers and the second cursor** removes them all.

Under the time axis, a line joins each mark (the cursor, the second cursor, each marker) to the next, with the time between them on it, `Δ 30 ns`; between the two cursors, with the frequency too. A chain of markers thus measures every interval at once: set markers on successive edges and read the periods off the axis.

The second cursor and the markers are saved in wave setups.

Finding a value

`/` (or **Signals** → **Find a value of the selected signal**) opens a find bar for the selected row. Type a value and press `Enter` or `F3`: the cursor moves to the next change of that signal to the value, after the cursor. `Shift+Enter` or `Shift+F3` finds the previous one, before the cursor. `Escape` closes the bar.

What you type is read this way:

You type	It finds
A number, such as <code>A5</code> , <code>1010</code> , <code>-3</code>	That number, read in the row's radix: hexadecimal, binary, decimal
<code>0x1F</code> , <code>0b1010</code> , <code>0o17</code> , or <code>x"1F"</code> , <code>b"1010"</code> , <code>o"17"</code>	That number, in the radix it names
Digits with <code>?</code> , such as <code>1??0</code>	Any value that matches, <code>?</code> standing for any digit
<code>X</code> (or <code>U</code>)	The next unknown value: any <code>U</code> , <code>X</code> , <code>W</code> or <code>-</code>
<code>Z</code>	The next high-impedance value
<code>1</code> or <code>0</code> on a one-bit signal	The next rising or falling edge
Anything else: <code>IDLE</code> , a string, a real	A value shown as that text, ignoring case, spaces and <code>_</code>

When there is no such change, the bar says so: *No change to A5 after 1.2 μs*. Searching a very large recording goes on in the background, with its progress in the bar, and skips the parts of the recording whose values cannot match without reading them.

Analog rows

A real, or a vector read as a number in its radix (unsigned or signed decimal, fixed or floating point, Gray, BCD), can be drawn as a line: right-click its name and choose **Analog**:

- **Steps**: the value holds until the next change;

- **Lines between changes:** straight lines from one value to the next;
- **Off (digital):** back to the normal row.

An analog row is taller (3 rows to start with); set its **Height (rows)**, from 1 to 12, in the same menu. Its **Range** is **The whole signal's** (the default), **What is shown** (it follows what is in view), or **Fixed**, from and to values you set. The range shows at the left of the row.

Where many changes fall within one pixel, the line shows their range there as a bar. Values with no number (a vector with unknown bits) are shaded. Zoomed out on a large recording, the range of each part comes from the recording's summaries, without reading the changes, exact to within a pixel.

Expression rows

A row can show an expression of other signals, computed from the recording: a handshake (`valid` and `ready`), a condition (`count = 10`), a field of a bus (`addr(7 downto 4)`), a sum. Press **E** (or **Signals** → **Add an expression of signals**), type the expression in the bar that opens over the waves, and press **Enter** (or **Add**). The new row goes after the selected row, or at the end if none is selected.

The bar starts with the selected signal's name, and says as you type what the expression gives (*1 bits, 8 bits, a real*) or what is wrong with it, such as *no signal is named* `valid` . **Escape** closes it.

Expressions are written as VHDL writes them:

Write	Meaning
<code>valid</code> , <code>count</code>	A signal, by its name
<code>{/tb/dut/count[7:0]}</code>	A signal by its path, in braces: needed when several signals have the name (the bar says so)
<code>'1'</code> , <code>'X'</code> , <code>"0101"</code> , <code>x"A5"</code> , <code>o"17"</code> , <code>b"01"</code> , <code>12</code> , <code>0.5</code>	Literals: bits, bit strings, whole numbers, reals
<code>not</code> , <code>and</code> , <code>or</code> , <code>xor</code> , <code>nand</code> , <code>nor</code> , <code>xnor</code>	Element by element, with <code>std_logic</code> 's rules: <code>'0'</code> and <code>'X'</code> is <code>'0'</code>
<code>=</code> , <code>/=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	Compare numbers (unsigned); the result is one bit
<code>+</code> , <code>-</code>	Add or subtract unsigned numbers; the result is as wide as the wider operand
<code>&</code>	Join vectors
<code>x(3)</code> , <code>x(7 downto 4)</code> , <code>x[7:4]</code>	Elements of a vector, counted from the right, 0 being the rightmost
<code>()</code>	Grouping

An operand with a bit that is not `0` or `1` makes arithmetic and comparisons unknown (`X`). Vectors of different widths are widened with zeros on the left. Reals compare, add and subtract as reals.

An expression row works like a signal's row: it has a radix, a color and an analog drawing, and you can find a value in it and jump between its changes. Right-click it and choose **Change the expression...** to edit it. Expressions are saved in wave setups and computed again for each run.

Groups

Pick rows (`Ctrl` +click, `Shift` +click), right-click one of them and choose **Group the picked rows**; with one row selected, **Put in a group**. The rows, with their parts, go into a new group where the first of them was, named `Group 1` and so on.

- Click the group's arrow to close or open it. A closed group shows how many rows it holds.
- Right-click the group to rename it (type in the **Name** field) or to **Ungroup** it.
- A group moves and is removed with its rows.

Groups are saved in wave setups, closed ones too.

Dividers

A divider is a row with a line across the waves and a label, to separate groups of signals. `Ctrl+D` (or **Signals** → **Insert divider above the selected signal**, or right-click → **Insert divider above**) inserts one above the selected row. Right-click the divider and type its text in the **Label** field.

Colors and paths

Right-click a signal's name:

- **Color: Default** (the theme's), or red, orange, yellow, green, cyan, blue, purple, pink, white or gray, for its wave.
- **Copy path**: copies the signal's full path, such as `/tb/dut/count[7:0]`, for a wave setup, a script or a message.

Wave setup files

A wave setup is what the wave view shows: the signals in their order with their radix, color and analog drawing, opened records and vectors, expression rows, groups and dividers, the cursors, the markers and the visible time span. Saved as a `.wavesetup` file, it can be reused across runs, projects and designs, and kept in version control next to a testbench.

- **File** → **Save wave setup...** (`Ctrl+S`, while the editor does not have the keyboard) saves the view.
- **File** → **Open wave setup...** (`Ctrl+Shift+O`), or dropping a `.wavesetup` file on the window, applies one to the waveform shown.
- `jhd1w run.jhw --setup uart.wavesetup` opens a waveform with a setup.
- A project lists its setups (`wave-setup` lines): the first is shown when a run's waveform opens, and the **Project** tab lists them all; double-click one to show it ([Projects](#)).

A signal that the waveform does not have is kept in the view and shown as missing, not dropped. So one setup can serve several designs, and survives a renamed signal until you fix it.

The format

A wave setup is a text file of lines, which you can also write or change by hand. Blank lines and lines that start with `#` are comments. The first other line is the header, with the format's version:

```
jhdl-wave-setup 3
```

Then one statement per line, words separated by spaces:

Statement	Meaning
<code>signal PATH</code> <code>[radix=R] [color=C]</code> <code>[bit=K] [depth=D]</code> <code>[expanded]</code>	A row: the signal with full path <code>PATH</code> , as Copy path gives it (<code>/tb/dut/count[7:0]</code>), or a record or array by its path (<code>/tb/bus</code>). <code>R</code> is the radix (below; hex by default). <code>C</code> is red, orange, yellow, green, cyan, blue, purple, pink, white, gray or <code>#rrggbb</code> ; without it, the theme's color. <code>bit=K</code> makes the row one bit of the vector, counted from the left (0 is the leftmost). <code>depth=D</code> is how deep the row is inside opened rows or groups above it; <code>expanded</code> marks an opened row, whose parts are the rows after it with a larger depth.
<code>... analog=MODE</code> <code>[range=R] [height=H]</code>	Options of a <code>signal</code> line: the row is drawn as an analog line, <code>MODE</code> being <code>step</code> or <code>linear</code> ; <code>R</code> is <code>view</code> (what is shown) or <code>MIN:MAX</code> (fixed), and without it the whole signal's range; <code>H</code> is the row's height in rows, 1 to 12.
<code>expr "TEXT"</code> <code>[radix=R] [color=C]</code> <code>[depth=D]</code> <code>[analog=MODE]</code> <code>[range=R] [height=H]</code>	An expression row: the value of the expression <code>TEXT</code> (Expression rows), in double quotes (<code>\</code> for a quote, <code>\\</code> for a backslash), with a signal's options. A version of JanusHDL that does not know <code>expr</code> skips the line.
<code>divider [depth=D]</code> <code>[LABEL]</code>	A divider with <code>LABEL</code> (the rest of the line, which may be empty).
<code>group "NAME"</code> <code>[depth=D] [closed]</code>	A group: a named row holding the rows after it with a larger depth. The name is in double quotes (<code>\</code> for a quote, <code>\\</code> for a backslash). <code>closed</code> : the group is closed.
<code>cursor N fs</code>	The cursor at <code>N</code> femtoseconds.
<code>cursor2 N fs</code>	The second cursor.
<code>marker N fs NAME</code>	A marker at <code>N</code> femtoseconds, named <code>NAME</code> (the rest of the line).
<code>view A fs B fs</code>	The visible span, from <code>A</code> to <code>B</code> femtoseconds.

Times are in femtoseconds, so that a setup does not depend on a waveform's time resolution.

The radices:

R	Shows a vector as
binary	Its bits
octal	A digit per 3 bits
hex	A digit per 4 bits
unsigned, signed	A decimal number (signed: two's complement)
ascii	A character per 8 bits, in quotes
sfixed:F, ufixed:F	A fixed-point number with F fraction bits, signed or not
float:E	An IEEE 754 floating-point number with E exponent bits (5 half, 8 single, 11 double)
gray	The number a Gray code encodes
bcd	A decimal digit per 4 bits

An example:

```
jhdl-wave-setup 3
# UART transmit path
signal /tb/clk radix=binary
divider Transmitter
signal /tb/dut/tx_data[7:0] radix=hex color=cyan
signal /tb/dut/tx_busy color=#f59e3d
group "Receiver"
signal /tb/dut/rx_data[7:0] depth=1 radix=ascii
signal /tb/dut/rx_level depth=1 analog=step height=4
expr "tx_busy and not rx_busy" color=green
cursor 1500000000 fs
marker 2000000000 fs first byte
view 0 fs 2000000000 fs
```

A setup is saved in the oldest version that holds what it uses, so that older versions of JanusHDL can read it: version 1 when it uses only the radices `binary`, `hex`, `unsigned` and `signed`; version 2 for another radix; version 3 when it has a second cursor, a marker, an analog row or a group. Later versions may add statements and options: a version of JanusHDL skips those it does not know and shows what it can; it refuses a header version it does not read.

Saving the waves as an image

File → Save the waves as an image (PNG or SVG)... saves the wave view as you see it, for a report, a review or a bug report. The file's name chooses the format:

- **PNG** (`waves.png`, the default): a picture of the view, at your screen's resolution.
- **SVG** (a name ending in `.svg`): a drawing that stays sharp at any size, and whose text can be selected and searched.

The names, values, waves, cursors and markers are in the image; in an SVG drawing, the selection and the find bars are left out.

Very large recordings

Waveforms of any size open. A waveform database is read a block at a time, as the view needs it: opening reads its index, not its changes. However long the run, the viewer keeps at most 512 MB of the recording in memory, and reads again what it dropped when it is shown again. Zoomed out, it draws from summaries of the blocks kept in the file, without reading the changes, and shows what it would show had it read them all.

Measured: a recording of 1.8 billion value changes (a 5.2 GB file) opened in under 4 seconds, in less than 1 GB of memory.

A VCD file is read into a temporary waveform database as it loads (in your temporary folder; removed when the file is closed), so a large VCD file opens in little memory too. Rows of bits and records of a large recording are built over several frames, so the window stays responsive.

If part of a waveform database cannot be read (a damaged disk, say), those changes show as unknown values, and the status bar says *some changes could not be read*.

VCD files

The viewer reads 4-state VCD files as IEEE 1364 and IEEE 1800 define them, plus the nine `std_ulogic` values (`U X 0 1 Z W L H -`) that VHDL simulators write, and the text and real values `jhd1 --vcd` writes.

- When several `$var` declarations share one identifier (a port and the signal connected to it), they are one signal shown under several names.
- A file with an error is refused with what is wrong and on which line, for example: *line 42: value change for unknown identifier %: it was not declared with \$var in the header*.
- Extended VCD files (written by `$dumpports`, with port directions and strengths) are not supported yet; the viewer says so.

A VCD file holds values only: the insight panels, which need the design's structure and delta cycles, say so. Record with `jhd1 --wave` to have them.

Insight panels

A waveform shows *what* happened. The insight panels show *why*: which processes drive a signal, what those processes read, which of their inputs changed last, and in which delta cycle each change came. Following an unknown value back to where it came from takes a few clicks instead of a search through the source.

There are three panels, each a tab:

Panel	Opens with	Shows
Why this value?	Signals → Why this value?	The selected signal's value at the cursor, its drivers, and what each driver reads
Dataflow	Signals → Show the dataflow and schematic of the selected signal	The selected signal between the processes that drive and read it, as a graph; and a schematic of its instance
Delta cycles	Signals → Show the delta cycles of the time step at the cursor	The time step at the cursor, delta cycle by delta cycle

The commands are in the command palette too (`Ctrl+Shift+P` , then type *why*, *dataflow* or *delta*). The panels open beside the **Terminal**; like any tab, they can be moved, split beside the waves or put in a window of their own ([Arranging the tabs](#)).

What the panels need

The panels follow the wave view: the **selected signal** (click its name) and the **cursor** (click in the waves). As you select another signal or move the cursor, they update.

They read the design's structure and delta cycles from the recording. A simulation run in the window, and a waveform database written by `jhd1 --wave` , have them: every process with its kind and place in the source, the signals each drives (with the lines that assign them) and the signals each reads, and the delta cycles of every time step. A VCD file holds values only, and so does a recording made by an earlier version of JanusHDL: the panels say so. A recording made with `--wave-deltas off` has the structure but no delta cycles: the **Delta cycles** panel says so, the others work.

Why this value?

Select a signal, place the cursor, and the panel shows:

- **The value** at the cursor, since when the signal has it (with the delta cycle of that change), and the time of the cursor. Unknown values (`U` , `X` , `W` , `-`) show in red on a tint.
- **Driven by**: the processes that drive the signal. A resolved signal with several drivers says so, *Driven by 2 processes (resolved)*, and lists each: an `x` on a resolved signal often means two drivers drive it at once.
- For each driver: its path in the design, its kind (**process**, **concurrent assignment**, **concurrent assertion**, **concurrent procedure call**, **port connection**, **port default**, **PSL directive**), a link to its place in the source, and the lines where it assigns the signal (**assigns it at**).

- **What each driver reads:** every signal the process reads, with its value at the cursor and since when. The input that changed last is marked **changed last**: of what the process reads, it is the likeliest cause of the value. Hover over an input to see its full path, and whether it wakes the process (is in its sensitivity).

A signal that no process drives has the value it was declared with, is an input of the top level, or has a value forced on it; the panel says so. A driver that reads no signal gets its value from the process itself (a constant, a variable, a file).

Following a value back

Click an input to ask why *it* has its value: the panel moves to that signal, at the same cursor time. Following the inputs that are unknown and changed last walks an **x** back, process by process, to where it came from: an uninitialized register, a missing reset, two drivers fighting, a port left open.

The path you walked is shown at the top, `rst_n < ready < data_valid`; click an earlier name to go back to it, or press **Back** for one step.

- The **+** button beside a signal shows it in the waves: it is added if it is not there, and selected.
- A source link, such as `uart_rx.vhd:112`, opens the file at that line in the code editor.

Dataflow and schematic

Dataflow draws the selected signal in the middle; on the left, the processes that drive it, with the signals they read; on the right, the processes that read it, with the signals they drive. Each signal carries its value at the cursor, unknown values in red.

- **Click** a signal to put it in the middle. Click a process to put it in the middle: it shows what it reads and what it drives, and the processes beyond.
- **Double-click** a signal to show it in the waves, or a process to open its source in the editor.
- **Back** returns to what was in the middle before.
- **Drag**, or use the wheel, to pan; `Ctrl` + wheel or a pinch to zoom; **Fit**, or a double click on the background, shows all of it.
- Hover over a signal or process for its full path and value.

Schematic (the button beside **Dataflow**) shows the instance of what is in the middle: its processes in columns in the direction values flow, each after those it reads from; the signals between them as labelled wires with their values; and the signals from and to the rest of the design as ports on the left and right. Large instances are drawn up to a limit, and the panel says how many processes were left out.

Delta cycles

Within one time step, a VHDL simulation runs any number of delta cycles: signal updates, then the processes they wake, whose assignments take effect in the next delta cycle. The waves draw each time step's last value; the **Delta cycles** panel shows the steps in between.

Place the cursor, and the panel shows the time step at the cursor (or the last one before it, and says so), delta cycle by delta cycle:

- Each delta cycle: **Delta 0**, **Delta 1** and so on (at time 0, **Initialization** first), with how many signals changed and how many processes ran.
- The signals that changed in it, each from its old value to its new one, `0 → 1`.
- The processes that ran in it, with links to their source.

Click a signal to select it in the waves (it is added if it is not shown). The arrows at the top move the cursor to the time step before or after.

A signal that changes and changes back within one time step, a glitch of zero width, shows both changes here, though the wave, which draws only the step's last value, shows neither. This is also the place to see the order of events behind a race between processes ([Determinism and races](#)).

Messages carry their delta cycle too: `115 ns (delta 2)` in the terminal is the delta cycle that this panel numbers 2.

The code editor

The window has a code editor for VHDL, in tabs beside the waves. It colors VHDL with the simulator's own lexer, shows the analyzer's errors and warnings as you type, and has the editing tools you expect: several cursors, block editing, split view, find and replace, find in files, folding and a minimap. Every place a message points to (in the terminal, the **Problems** tab, the insight panels, the **Project** tab) opens here.

Opening, saving and closing files

- **File** → **Open file in editor...** (`Ctrl+E`) opens a file in a tab of its own. The dialog offers VHDL files (`.vhd` , `.vhd1` , `.vht` , `.vho`), and project files and scripts (`.f` , `.pro` , `.tcl` , `.do`). The first file opens beside the waves; later ones open beside the file you are editing. A file already open comes to the front.
- A click on a source location opens the file at that place: a line in the **Terminal**, a problem in **Problems**, a match in **Search**, a source link in an insight panel, a file in the **Project** tab or under **Files** in the **Hierarchy** tab (double-click), an instance in the **Hierarchy** tab (double-click, or **Go to source** in its menu: its source, at its first process).
- A dot after a tab's name means the file has changes not yet saved. Hover over the tab to see the file's path.
- **Save file** (`Ctrl+S` , while the editor has the keyboard) saves it. **Save all files** (`Ctrl+Shift+S`) saves every file with changes. **Close file** (`Ctrl+W` , or the tab's `×`) closes it.
- Closing a file with changes asks whether to **Save**, **Don't save** or **Cancel**. Quitting with changes asks whether to **Save all and quit**, **Quit without saving** or **Cancel**.

When another program changes a file that is open, the editor reads it again if you have no changes in it; your cursor stays on its line. If you do have changes, it asks whether to **Reload** the file (Undo brings your text back) or **Keep mine**.

The files open, with their cursors and panes, the text size, the minimap and the find options are kept between sessions; with a project open, its files are kept with the project ([Each user's state](#)).

The sources of the built-in libraries, opened from a message such as `ieee/numeric_std-body.vhd1:120` , open read-only; the tab says (**built in, read-only**) when you hover over it.

What you see

- **Highlighting.** VHDL files are colored by the simulator's own lexer: reserved words, comments (also `/* */` blocks), strings, character and bit string literals, numbers, attribute names after a tick, and PSL's reserved words and operators, in the colors of the theme. Other files show as plain text.
- **The gutter** shows line numbers, a dot on lines with a diagnostic, and arrows to fold regions. Click line numbers to select lines.
- The line of the cursor is highlighted, and so is the bracket matching the one at the cursor.
- **The status line** under the text shows the cursor's line and column (`Ln 12, Col 8`), the number of cursors or of characters selected (or a block's size), the counts of errors and warnings, and, on the right, the file's indentation, encoding, line ends and kind (**VHDL** or **Text**), with **Read-only** for a built-

in source.

Encodings and line ends. A file is read as UTF-8 (with or without a byte order mark), or as ISO 8859-1 if it is not valid UTF-8, and saved as it was read. Line ends stay as the file had them, LF or CRLF; a file with both is saved with the kind of its first line.

Indentation. The editor works out a file's indentation from its text: tabs, if more lines start with a tab than with spaces, else the most common step between lines. A file that does not tell gets two spaces. Tabs show four columns wide.

Editing

Keys	What they do
Enter	A new line, indented as the line before; one level more after a line that opens a block (<code>is</code> , <code>begin</code> , <code>then</code> , <code>else</code> , <code>loop</code> , <code>generate</code> , <code>record</code> , <code>(</code> , <code>when ... =></code>)
Tab / Shift+Tab	Indent or outdent the selected lines; <code>Tab</code> without a selection indents to the next stop
Ctrl+/ 	Comment out the selected lines with <code>--</code> , or take the <code>--</code> away if every line has one
Ctrl+D	Select the word at the cursor; press again to add the next place with the same text as another cursor
Alt +click	Add a cursor
Alt +drag	Select a block of text in columns (Block editing)
Alt+Shift+↑ ↓ ← →	Make a block from the cursor (or the selection), or grow it by a line or a column
Escape	Back to one cursor (also out of a block); close the find bar
Ctrl+Z / Ctrl+Y (or Ctrl+Shift+Z)	Undo / redo; typing and deleting in a row undo together
Ctrl+X / Ctrl+C / Ctrl+V	Cut, copy and paste; without a selection, the whole line (pasted above the cursor's line)
Ctrl+A	Select all
Ctrl+G	Go to a line: <code>120</code> , or <code>120:8</code> for line and column
Home , End , Ctrl+← / → , Ctrl+Home / End , Page Up / Down	Move (with <code>Shift</code> : select). <code>Home</code> goes to the first character that is not a space, then to the start of the line
Ctrl+Backspace / Ctrl+Delete	Delete the word before or after the cursor
Ctrl+↑ / ↓	Scroll a line without moving the cursor
Double click / triple click	Select a word / a line

Keys	What they do
Ctrl +wheel	Larger or smaller text

A right-click in the text offers **Cut**, **Copy**, **Paste**, **Select all**, **Toggle comment**, **Split right** and **Split down**.

Several cursors

Alt +click adds a cursor; Ctrl+D adds the next place with the same text as the selection. With several cursors, typing, deleting, Enter, Tab and pasting act at every cursor; pasting as many lines as there are cursors puts one line at each. Escape goes back to one cursor.

Block editing

Hold Alt and drag the mouse to select a rectangle of text, a block: the same columns on every line, from where you pressed to where you let go (Alt and a click without dragging still adds a cursor). From the keyboard, Alt+Shift and the arrow keys make a block from the cursor, or from the selection, and grow it a line or a character at a time. The block shows as a rectangle with a selection and a cursor on each line, and the status line gives its size: Block: 6 lines × 12 columns. Columns are counted as shown, a tab reaching the next tab stop.

- **Typing** replaces the block's text on every line and goes on in the same column. A line that ends before the block's left edge has its cursor at its end; when you type, it is first filled with spaces up to the block's edge, so that what you type lines up with the other lines.
- **Delete** and **Backspace** delete the block, or the character beside the cursor on each line. They never join lines, and leave lines that end before the block alone.
- **Tab** indents the block's text one level on each line; **Shift+Tab** takes up to one level back.
- **Copy** and **cut** take the block's lines, one per line. **Pasting** as many lines as the block has puts one on each line; other text is pasted on every line.
- Enter, a click, an arrow key without Alt+Shift, or Undo end the block and leave its cursors as several cursors; Escape ends it with one cursor.

Block editing is the quick way to line up port maps, add a prefix to a column of signal names, or fill in a column of constants.

Split view

View → **Split the editor right** (Ctrl+Shift+I) or **Split the editor down** (Ctrl+Shift+D), or **Split right** / **Split down** in the text's right-click menu, opens the file again in a pane beside (or below) the one you are in.

- Each pane has its own cursors, selection, scroll position and find bar, and they all edit the same text: what you type in one shows in the other at once, Undo takes back the last change whichever pane made it, and saving in either saves the file. Folding is shared.
- To see two files side by side, split, then open the other file (Ctrl+E, or from the **Problems** or **Search** tab): a file opens in the pane you were editing in. You can also drag a tab beside another.

- **View → Close the editor pane** (`Ctrl+Shift+W`) closes the pane you are in; its files go to another pane (or beside the waves), and a file open there too just loses its second view. Closing a pane's tab (`×`, `Ctrl+W`) closes only that view while the file is open in another pane, without asking.
- A place in a file (in **Problems**, **Search** or the **Terminal**) opens in the pane you were editing if it shows that file, else in the pane where you last edited it.

The panes are kept with the layout between sessions, and with a project's layout.

Find and replace

`Ctrl+F` opens the find bar over the text, with the selected text as what to find; typing finds the first match after the cursor.

- `Enter` or `F3` goes to the next match, `Shift+Enter` or `Shift+F3` to the previous one, round the end of the file. The bar shows which match of how many is selected: `3 of 12`.
- The buttons beside the field match case (**Aa**) and whole words only. Without **Aa**, letters match regardless of case, as VHDL's own names do (ASCII letters only).
- `Ctrl+H` opens the bar with a second field for the replacement: **Replace** (or `Enter` in that field) replaces the selected match and goes to the next; **Replace all** replaces every match, as one step to undo.
- `Escape` closes the bar.

Find in files

Edit → Find in files (`Ctrl+Shift+F`) opens the **Search** tab beside the terminal. Type what to find and press `Enter` (or **Search**). Choose where to look:

- **Files of the run:** the files of the current run: the sources of the open project, the files of a file list, or the VHDL file run;
- **Folder of the file:** the folder of the file being edited, and the folders under it: VHDL files, file lists and scripts (`.f`, `.pro`, `.tcl`, `.do`).

Match case and **Match whole word** work as in the find bar. The matches are listed by file, with their line numbers; click one to open the file there. Files open in the editor are searched as you have edited them. A search stops at the first 20,000 matches, and says so.

Folding

The arrow beside a line number folds the region that starts there: a design unit, a process, a subprogram's body, a block, a generate, an `if`, `case` or loop up to its `end`, a record or protected type, or a run of comment lines. A folded region shows `...` after its first line. **Edit → Fold all** and **Unfold all** fold or open every region. A cursor moved into a folded region, by **Go to line** or by finding, opens it.

Minimap

The right edge of the editor shows the whole file scaled down, with the part on screen marked; click or drag in it to scroll there. **View** → **Show or hide the minimap** turns it off or on.

Diagnostics

VHDL files are analyzed by the same analyzer the simulator uses, with the same messages: when they are opened and saved, and a moment after you stop typing (the text as it is, without saving it).

- Errors and warnings are underlined with a wavy line, marked in the gutter and beside the scroll bar, and counted in the status line.
- Hover over one to read the message, with its help and its LRM section.
- **View** → **Show problems** (`Ctrl+Shift+M`) opens the **Problems** tab: the problems of every VHDL file open in the editor, by file, with their line and column. Click one to go there.

A file is analyzed as the current run analyzes it (the open project, else the last simulation run), so that names declared in other files are known and the messages are those the run gives:

- A file of the run is analyzed after the run's files before it, into its own library, as its own VHDL version. For a project, that is the version of the file, of its library or of the project, with the project's compatibility mode, PSL comments, conditional analysis identifiers and rule levels. For a file list or a VHDL file, it is the VHDL version and compatibility mode chosen in the Run dialog.
- A file that is not among the run's files is analyzed after all of them, into `work`.
- With no run, a file is analyzed on its own, into `work`, as VHDL-2008.

A compiled UVVM or OSVVM (`jhd1 lib compile`, [Compiled libraries](#)) is found in every case, as in a run: a file that names one of its libraries needs none of its files in the run.

When the run changes, or its project or file list is saved, the open files are analyzed again.

External editors and the language server

`jhd1-lsp` is JanusHDL's VHDL language server. It runs JanusHDL's own analyzer on the whole design in your workspace and speaks the Language Server Protocol (LSP), so any editor with an LSP client gets:

- **Diagnostics** while you type: syntax, names and types, VHDL-1993 to VHDL-2019, each with its rule name (`undeclared` , `type-mismatch` , ...) and LRM section, exactly as `jhd1` reports them. What JanusHDL does not support yet is shown as information, not as an error in your code.
- **Go to definition**, also into the IEEE and standard packages (their sources are written to a temporary folder, read-only).
- **Hover**: the declaration of the name under the cursor, and where it is.
- **Find references** across the design.
- **Document symbols** (the outline): design units, subprograms, types, signals, constants, variables, labels.

Your editor starts the server itself. It needs to find `jhd1-lsp` : put JanusHDL's `bin` folder on your `PATH` ([Installing](#)), or give the editor the program's full path.

How the server finds your design

Names resolve across files and libraries, so the server analyzes every file of the design, in an order their dependencies allow. It finds the files, and their libraries, in the first of these ways that applies:

1. **A project or a file list.** The file the `projectFile` option names (relative to the workspace folder); else a JanusHDL project (`.jhd1proj`) in the workspace's top folder; else a file list `jhd1.f` there. - A **project** is used as `jhd1 run` and the window use it ([Projects](#)): its sources in their libraries and order, each as its own VHDL version, its library's or the project's, with the project's compatibility mode, PSL comments, conditional analysis identifiers and rule levels. With several projects in the folder, the first by name is used; the server's log says which, and that `projectFile` chooses another. - A **file list** is the format `jhd1 -f` reads ([File lists](#)): `library NAME` lines choose the library of the files after them, other lines are paths relative to the file list, `#` starts a comment.

A project that cannot be read (an error in it, or a `projectFile` that names no file) is reported in the server's log with what is wrong, and the design is found the next way until it is fixed. 2. **UVVM.** When the workspace is UVVM, or contains it, its compile orders (`COMPONENT/script/compile_order.txt` , in the order of `script/component_list.txt`). The testbenches and demos of each component follow in its library. Set the `compat` option for UVVM ([Options](#)). 3. **OSVVM.** When the workspace is OSVVM (`osvvmLibraries.pro`) or one of its libraries (`build.pro`), its build scripts. The server reads the commands that choose files (`library` , `analyze` , `include` , `build` , `ChangeWorkingDirectory` , `if` and `else`), answering OSVVM's questions about the tool as a VHDL-2008 simulator. Scripts that do more need a file list. 4. **Any folder.** Every `.vhd` and `.vhd1` file in the workspace, leaving out what `.gitignore` excludes. A file is in library `NAME` if a folder on its path is called `NAME` and some file says `library NAME`; ; otherwise it is in `work` .

A file you open that is not part of the design is analyzed into `work` . A design that names a compiled UVVM or OSVVM library (`jhd1 lib compile` , [Compiled libraries](#)) finds it as a run does, without its files in the workspace; UVVM's files are then analyzed in compatibility mode, whatever the `compat` option. A design

whose files include its own copy of the library uses that copy.

Without a project, every file is analyzed as the one VHDL version the `std` option gives, and in compatibility mode if the `compat` option asks for it.

Options

The editor sends these as initialization options:

Option	Meaning
<code>projectFile</code>	A JanusHDL project (<code>.jhd1proj</code>) or a file list (<code>.f</code>), relative to the workspace
<code>std</code>	The VHDL version: <code>93</code> , <code>02</code> , <code>08</code> (the default) or <code>19</code> . A project has its own versions, which are used instead
<code>compat</code>	<code>true</code> : compatibility mode, as <code>jhd1 --compat</code> : code the LRM forbids but whose meaning is clear is a warning, not an error (Compatibility mode). Set it for UVVM's files in the workspace, whose unprotected shared variables are errors otherwise; a compiled UVVM needs no <code>compat</code> . A project's own <code>compat</code> setting applies as well

What the server sees, without an editor

```
jhd1-lsp --check path/to/design [--project path/to/design.jhd1proj] [--compat]
```

analyzes the folder as the server would, prints every diagnostic, and ends with a summary that says how it found the design, here from a project in the folder:

```
2 files from .\counter.jhd1proj: 0 errors, 0 warnings, 0 not supported yet (0.3 s)
```

`--project` names a project or a file list to use, as the `projectFile` option does; `--compat` analyzes in compatibility mode, as the `compat` option does.

The exit code is 1 if there are errors. It is a quick way to check that the server finds your files and libraries before you set up an editor, or to check a whole folder in CI.

`jhd1-lsp --order FILES.f` prints a file list with its files in dependency order ([Analysis order](#)).

Setting up your editor

Visual Studio Code

The **JanusHDL VHDL** extension starts `jhd1-lsp` for `.vhd`, `.vhd1`, `.vht` and `.vho` files and adds VHDL highlighting. Its settings:

Setting	Meaning
<code>jhd1.serverPath</code>	The <code>jhd1-lsp</code> program. Empty: the one that comes with the extension, else the one on the <code>PATH</code>
<code>jhd1.projectFile</code>	A project or a file list, relative to the workspace
<code>jhd1.std</code>	The VHDL version: <code>93</code> , <code>02</code> , <code>08</code> or <code>19</code> , without a project
<code>jhd1.compat</code>	Compatibility mode, as the <code>compat</code> option

The command **JanusHDL: Restart the language server** restarts it, after you change the file list, for example.

Neovim (0.11 and later)

```
vim.lsp.config('JanusHDL', {
  cmd = { 'jhd1-lsp' },
  filetypes = { 'vhd1' },
  root_markers = { 'jhd1.f', '.git' },
  init_options = { std = '08' },
})
vim.lsp.enable('JanusHDL')
```

Helix

In `languages.toml`:

```
[language-server.JanusHDL]
command = "jhd1-lsp"

[[language]]
name = "vhd1"
language-servers = ["JanusHDL"]
```

Sublime Text

In the settings of the LSP package:

```
"clients": {
  "JanusHDL": {
    "enabled": true,
```

```
"command": ["jhd1-lsp"],  
"selector": "source.vhdl"  
}  
}
```

Emacs (eglot)

```
(add-to-list 'eglot-server-programs '(vhdl-mode "jhd1-lsp"))
```

Notepad++

With the NppLspClient plugin, add a server for the VHDL language with the command `jhd1-lsp`.

Other editors

Any editor with an LSP client works: start `jhd1-lsp` (it talks over its standard input and output) for VHDL files, with the workspace's top folder as the root, and pass `projectFile` and `std` as initialization options if you need them.

How it works

When it starts, the server analyzes the whole design; for a library the size of UVVM this takes a few seconds. After that, an edit is analyzed when you pause typing (a quarter of a second), together with every file that uses what the edited file declares, and before the server answers a request about that file, so that its answers always match the text you see. Positions are exchanged in UTF-16 columns, or in bytes when the editor offers UTF-8.

The window's own editor

The JanusHDL window has a code editor of its own, with the same analyzer's diagnostics as you type ([The code editor](#)). Use whichever you prefer; many people keep their usual editor for writing code and use the window for running and debugging, where a click on a message or a driver opens the place in the window's editor.

Running existing scripts

Many teams have scripts, makefiles and regression flows that call simulation commands by name: `vcom` and `vsim`, `xvhdl` and `xelab`, `ghdl` or `nvc`. JanusHDL comes with small front ends that answer to those command names, so that such scripts run on JanusHDL as they are. This chapter explains how to use them, what each accepts, and how to run VUnit.

For new work, `jhd1` itself and project files are simpler ([Running simulations from the command line, Projects](#)). The front ends are for keeping what you have.

How the front ends work

The `compat` folder of the download has one folder per family of commands. Each holds a small program for each command name of the family, and a `jhd1` as well:

Folder	Commands	Status
<code>compat/vsim</code>	<code>vlib</code> , <code>vmap</code> , <code>vdel</code> , <code>vcom</code> , <code>vopt</code> , <code>vsim</code> (and <code>vlog</code>)	VHDL supported
<code>compat/xsim</code>	<code>xvhdl</code> , <code>xelab</code> , <code>xsim</code> (and <code>xvlog</code>)	VHDL supported
<code>compat/ghdl</code>	<code>ghdl</code>	VHDL supported
<code>compat/nvc</code>	<code>nvc</code>	VHDL supported
<code>compat/asim</code>	<code>vlib</code> , <code>vmap</code> , <code>vcom</code> , <code>vsim</code> as in <code>compat/vsim</code> ; <code>asim</code> , <code>vsimsa</code> , <code>vlog</code>	<code>asim</code> and <code>vsimsa</code> not yet
<code>compat/xrun</code> , <code>compat/vcs</code>	<code>xrun</code> , <code>xmvhdl</code> , <code>xmvlog</code> , <code>xmelab</code> , <code>xmsim</code> , <code>ncvhdl</code> , <code>ncvlog</code> , <code>ncelab</code> , <code>ncsim</code> , <code>irun</code> ; <code>vcs</code> , <code>vhdlan</code> , <code>vlogan</code> , <code>simv</code>	Not yet
<code>compat/iverilog</code> , <code>compat/verilator</code>	<code>iverilog</code> , <code>vvp</code> ; <code>verilator</code>	Not yet (Verilog)

Each program passes its command line to JanusHDL, which translates it into its own options and runs it. To use a family, put its folder first on your `PATH`, before anything else that provides those commands. Put only one family's folder on the `PATH`, so that each command name means one thing:

```
export PATH=/opt/janushdl/compat/vsim:$PATH      # Linux
```

```
$env:Path = "C:\Tools\janushdl\compat\vsim;" + $env:Path # Windows, this session
```

Since each folder has its own `jhd1`, that one folder on the `PATH` is enough to use `jhd1` too.

Commands that JanusHDL does not handle yet (the Verilog commands, and the families marked "not yet") answer with an error that says so, with exit code 2, rather than doing something else.

What they have in common

- **Libraries on disk.** As the original commands do, the front ends keep libraries between commands: each library has a record of the source files analyzed into it, in order, with each file's VHDL version, and an analysis cache beside it. Each analysis, elaboration or run step analyzes the recorded files of every library it can see (the cache makes that quick) and goes on from there. Each file is analyzed as the VHDL version its own command gave, so libraries analyzed as different versions work together; the standard packages are those of the newest version among the files. The source files must stay where they were when they were analyzed; a file that is gone is left out, with a note.
- **Compatibility mode.** The front ends always run in compatibility mode, so code that existing scripts were used to getting through, such as UVVM's unprotected shared variables, is accepted with a warning instead of the error `jhd1` gives without `--compat` ([Compatibility mode](#)). Options such as `-frelaxed`, `--relaxed`, `-relax` or `-permissive` also turn into warnings the errors JanusHDL can lower.
- **Output.** Messages are JanusHDL's own, in its format, with files named by their path relative to the current folder, as the script gave them; the summary line `jhd1` prints at the end is left out. The exit codes follow `jhd1`'s ([Exit codes](#)).
- **Options without a counterpart.** Options that tune optimization, coverage, message display, timing checks and the like are accepted and have no effect. An option that cannot be honored and would change the result is refused with an error that names it. Some accepted options print a note about what they do here.
- **Waveforms.** Requests to record waveforms record into JanusHDL's waveform database, or into a VCD file where a VCD file is asked for; a note says which file. Open them with `jhd1w`.
- **Versions.** Asked for its version, a front end prints the version number of the command language it accepts, followed by JanusHDL's version, so that tools that check the version keep working.

vcom, vsim and the rest

Command	What it does here
<code>vlib DIR</code>	Makes a library in folder <code>DIR</code> , named after the folder
<code>vmap NAME DIR</code>	Maps library <code>NAME</code> to <code>DIR</code> in the library mapping file; <code>vmap NAME</code> shows a mapping, <code>vmap</code> lists them, <code>vmap -del NAME</code> removes one
<code>vdel [-lib DIR] [-all]</code>	Deletes a library's record
<code>vcom [options] FILE...</code>	Analyzes files into <code>-work LIB</code> (default <code>work</code>); the library must exist (<code>vlib</code>)
<code>vopt TOP -o NAME</code>	Records <code>NAME</code> as that top level, with its generics (<code>-gNAME=VALUE</code>), for <code>vsim NAME</code>
<code>vsim [options] LIB.TOP</code>	Loads a design and runs the commands of <code>-do</code>

Library mapping. Library names map to folders through the `[Library]` section of a mapping file: the first `.ini` file in the current folder that has one. `vmap` writes there, and makes `jhd1-libraries.ini` if there is none. Without a mapping, library `NAME` is the folder `NAME`.

vcom options. -87 and -93 (VHDL-1993), -2002 (the default), -2008, -2019; -work LIB; -f FILE reads more arguments from a file; -psl reads PSL in comments; -permissive and -relax relax the rules that can be relaxed. Options about checks, optimization, coverage, messages and logs are accepted without effect. -pslfile is not supported yet.

vsim options. The design unit (LIB.TOP , a configuration, or a vopt name); -c, -batch or -console to run without the window; -do "COMMANDS" or -do FILE (several allowed); -lib LIB or -work LIB; -L LIB to make more libraries visible; -gNAME=VALUE and -GNAME=VALUE for generics. vsim -c -do script.do without a design runs a script that loads the design itself; vsim -c without -do reads the commands from the standard input.

Without -c, vsim opens the design in the JanusHDL window and runs it there; the -do scripts are not run in the window yet (a note says so).

Script commands. A -do script, or a file run with do, can use:

Command	What it does
run TIME, run N UNIT, run -all	Run for a time, or to the end; run alone runs 100 ns. Successive run commands add up.
restart	Back to time 0
quit, exit (-code N gives the exit code); quit -sim	End the script; quit -sim instead takes the design back to time 0
vsim ...	Load a design
vlib, vmap, vdel, vcom, vopt	As the commands above
do FILE, source FILE	Run another script
cd DIR, pwd	Change or print the folder
echo, puts	Print text
add wave ..., log ...	Record every signal, into vsim.jhw
vcd file FILE, vcd add ...	Record a VCD file (dump.vcd unless named)
onerror, onbreak, transcript, radix, view, wave, configure, WaveRestoreZoom, WaveRestoreCursors and other display commands	Accepted, without effect

Scripts are a small part of Tcl: commands separated by new lines and ;, words grouped with {...} and "...", # comments, \ at the end of a line, set NAME VALUE and \$NAME, \${NAME}, \$env(NAME). Command substitution ([...]) and control commands (if, foreach, proc, ...) are not supported yet: the script stops with an error that names them. A command that is not supported stops the script the same way.

An example, in a folder with `vlib` 's libraries:

```
vlib work
vcom -2008 src/fifo.vhd tb/fifo_tb.vhd
vsim -c work.fifo_tb -do "run -all; quit -code 0"
```

xvhdl, xelab and xsim

Command	What it does here
<code>xvhdl [options]</code> <code>FILE...</code>	Analyzes files into a library (default <code>work</code>)
<code>xelab [options]</code> <code>[LIB.]TOP</code>	Checks that the design elaborates and writes a snapshot record; with <code>-R</code> , also runs it to the end
<code>xsim SNAPSHOT</code>	Runs a snapshot: <code>-R</code> to the end, <code>-tclbatch FILE</code> with the commands there, else with commands from the standard input

Libraries are folders under `xsim.dir/` .

xvhdl options. `--2008` (Or `-2008` , `--vhd12008`), `--2019` , `--93_mode` ; the default is VHDL-1993. `-work NAME` Or `-work NAME=DIR` ; `-L NAME[=DIR]` ; `-d NAME=VALUE` defines a conditional analysis identifier; `--relax` ; `-prj FILE` reads a project file of lines `vhd1 LIB FILE...` , `vhd12008 LIB FILE...` Or `vhd12019 LIB FILE...` . `-f` argument files are not supported yet.

xelab options. The top level; `-s NAME` names the snapshot (default: the top's name); `-R` runs after elaborating; `-generic_top NAME=VALUE` ; `-prj FILE` ; and `xvhdl` 's version and library options. Options about debug, timing, SDF and optimization are accepted without effect; `-sdfmax` , `-sdfmin` , `-sdfroot` , `-sv_lib` , `-sv_root` and `-dll` are refused.

xsim options. `-R` ; `-tclbatch FILE` (several allowed); `-wdb FILE` records into JanusHDL's waveform database in `FILE` ; `-gui` runs in batch, and a note says to open the waveform with `jhd1w` .

Script commands. `run TIME` , `run N UNIT` , `run all` (`run` alone runs 1000 ns); `restart` ; `quit` or `exit` (`-code N`); `puts` ; `log_wave` , `add_wave` , `add_wave_divider` , `add_wave_group` record every signal into `xsim.jhw` ; `open_vcd [FILE]` and `log_vcd` record a VCD file; `close_vcd` , `current_scope` , `set_property` and similar commands are accepted without effect. The same part of Tcl as above.

```
xvhdl --2008 src/fifo.vhd tb/fifo_tb.vhd
xelab work.fifo_tb -s fifo
xsim fifo -R
```

ghdl

```
ghdl COMMAND [options] ...
```

Command	What it does here
<code>-a FILE...</code>	Analyze files into the work library
<code>-s FILE...</code>	Check files: analyze them, recording nothing
<code>-i FILE...</code>	Import: record files in the work library without analyzing them
<code>-e UNIT</code>	Check that the design elaborates
<code>-r UNIT [run options]</code>	Run the design
<code>--elab-run UNIT [run options]</code>	Elaborate and run
<code>-m UNIT</code>	Analyze what is recorded, then check the elaboration
<code>--remove, --clean</code>	Remove the work library's record
<code>--dir</code>	List the libraries

Options before the unit. `--std=87|93|93c|00|02|08|19` (default 93; VHDL-87 is analyzed as VHDL-93), `--work=NAME`, `--workdir=DIR` (where the library records are; default the current folder), `-PDIR` (more folders with libraries), `-frelaxed` (Or `-frelaxed-rules`, `--relaxed`), `-fpsl` (PSL in comments). Options such as `--ieee=...`, `-fexplicit`, `-W...` and `-O...` are accepted without effect (the legacy packages they would select are built in).

Run options, after the unit. `--stop-time=TIME`, `--vcd=FILE`, `--wave=FILE` (written in JanusHDL's waveform format), `--fst=FILE` and `--vcdgz=FILE` (written as a VCD file, `FILE.vcd`), `-gNAME=VALUE`, `--assert-level=note|warning|error|failure|none` (the severity that stops the run), `--ieee-asserts=enable|disable`, `--no-run`. An architecture name after the unit is noted; the architecture analyzed last is used.

```
ghdl -a --std=08 src/fifo.vhd tb/fifo_tb.vhd
ghdl --elab-run --std=08 fifo_tb --stop-time=10us --wave=fifo.jhw
```

nvc

```
nvc [global options] -a FILE... -e [options] UNIT -r [options]
```

`-a`, `-e` and `-r` can come in one command line, in that order.

- **Global options.** `--std=1993|2000|2002|2008|2019` (or the two-digit forms; default 2008), `--work=NAME` or `--work=NAME:DIR` (default: folder `./NAME`), `-L DIR` (more folders with libraries), `--map=NAME:DIR`, `--ieee-warnings=on|off`, `--init` (make the work library).
- **After `-a`.** Files; `--relaxed`; `--psl`; `-DNAME=VALUE`.
- **After `-e`.** The unit (`ENTITY` or `ENTITY-ARCH`, the architecture being noted and the last analyzed one used); `-gNAME=VALUE` (or `-g NAME=VALUE`). `-e` without `-r` only checks the elaboration.
- **After `-r`.** `--stop-time=TIME`; `--wave[=FILE]` (JanusHDL's waveform format; default `wave.jhw`); `--format=vcd` (a VCD file instead); `--exit-severity=LEVEL` (the severity that stops the run).

VHPI plug-ins (`--load` , `--vhpi...`) and the commands `-i` , `--do` , `-c` , `--make` , `--syntax` and others are not supported yet, and say so.

```
nvc --std=2008 -a src/fifo.vhd tb/fifo_tb.vhd -e fifo_tb -r --stop-time=10us
```

Limits of the front ends

- Only the Tcl described above is understood in scripts.
- Verilog, and the families marked "not yet", are not supported.

VUnit

VUnit runs on JanusHDL through VUnit's own package for it, `vunit-janushdl` , which needs VUnit 5.0.0.dev16 or later. VUnit 5 is a pre-release at the time of writing, which `pip` installs only when asked (`--pre`):

```
pip install --pre "vunit_hdl>=5.0.0.dev16" vunit-janushdl
```

In `run.py` , add the package right after making the VUnit object, before any other call that depends on the simulator, such as `add_vhdl_builtins` :

```
from vunit import VUnit

vu = VUnit.from_argv()
vu.add_package("vunit-janushdl", allow_setup=True)
vu.add_vhdl_builtins()
```

Then, with JanusHDL's `bin` folder on your `PATH` , choose JanusHDL with `VUNIT_SIMULATOR` :

```
export VUNIT_SIMULATOR=janushdl
python run.py
```

```
$env:VUNIT_SIMULATOR = "janushdl"
python run.py
```

VUnit analyzes its own libraries and your sources, then runs each test case in a simulation of its own, in parallel with `-p N` as usual. JanusHDL's own options go in VUnit's `janushdl.a_flags` (analysis) and `janushdl.sim_flags` (simulation) options; sources that need compatibility mode, such as unprotected shared variables, need `--compat` in `janushdl.a_flags` , which the package passes to the simulations too ([Compatibility mode](#)). VUnit's own test suites, 1,338 test cases, pass this way. `--gui` records the test's waveforms and opens them in `jhd1w` . Each simulation stops with an error if it uses more than 4 GiB of memory ([Memory limit](#)).

VUnit 4: through the front ends

With VUnit 4, `run.py` runs unchanged through the `ghdl` or `nvc` front end. Install VUnit:

```
pip install vunit_hdl
```

Then put one of the two front ends first on your `PATH`, and tell VUnit to use that command family with `VUNIT_SIMULATOR`:

```
export PATH=/opt/janushdl/compat/ghdl:$PATH
export VUNIT_SIMULATOR=ghdl
python run.py
```

```
$env:Path = "C:\Tools\janushdl\compat\ghdl;" + $env:Path
$env:VUNIT_SIMULATOR = "ghdl"
python run.py
```

With the `nvc` front end, use `compat/nvc` and `VUNIT_SIMULATOR=nvc`. Like every front end, these run in compatibility mode, so sources that need it (unprotected shared variables, for instance) analyze with warnings and need nothing in `run.py`. VUnit's examples (`uart`, `check`, `logging`, `array`, `composite_generics`, `generate_tests` and `run`) pass this way through either front end; in the `run` example, the tests written to fail, fail.

Through the front ends, VUnit's `--gui` option is not supported. To look at a test's waveforms, run its testbench with `jhd1 --wave` and open the recording in `jhd1w`.

Determinism and races

JanusHDL is deterministic: the same design, with the same inputs and options, gives the same output on every run and on every machine, Windows or Linux, x86-64 or ARM, with native code or without. PSL's `nondet`, `nondet_vector` and `union`, which choose values pseudo-randomly, make the same choices on every run too. A failure seen in CI reproduces on your desk.

Within that, one thing is left open by the VHDL standard: the order in which the processes of one simulation cycle run. This chapter explains what that order can and cannot change, what order JanusHDL uses, and how to use other orders to find testbenches that pass only by luck.

The simulation cycle in brief

A VHDL simulation is a sequence of cycles (LRM 14.7.5). In each cycle:

1. Time advances to the next time anything is scheduled, or stays where it is: then the cycle is a *delta cycle*.
2. Every signal with a transaction due is updated; a signal whose value changes has an event.
3. Every process waiting on a signal with an event (and whose `wait until` condition is now true), and every process whose timeout expires, resumes.
4. Each resumed process runs until it suspends again.
5. Postponed processes run last, once time is about to advance.

Signal assignments made in step 4 take effect in step 2 of a later cycle, at the earliest the next delta cycle. So within one cycle every process sees the same signal values, and no process sees another's assignments of that cycle.

What the order can change

The standard fixes which processes run in a cycle, the values they read, how resolved signals resolve, and the order of the cycles. It does not fix the order in which the processes of one cycle run, at initialization or later: every simulator chooses its own.

For signal values, the order changes nothing. A design or testbench whose processes communicate only through signals gives the same waveforms and the same verdict under any order.

It can show through anything that is not a signal and that several processes touch in the same cycle:

- shared variables and protected objects: scoreboards, queues, message boxes of verification libraries, a shared random generator; which process calls a method first;
- files and the transcript: the order of `report` messages, and of the lines several processes write to one file, within one time step;
- `std.env.stop` and `finish`, called by several processes in one cycle.

A testbench whose verdict depends on that order has a **race**: legal VHDL whose result the standard leaves to the simulator. Such races are not rare. A testbench with one can pass on one simulator and fail or hang on another, or change behaviour when an unrelated process is added. Logs can list the messages of one time step in a different order even when nothing is wrong.

JanusHDL's order

JanusHDL decides each process's place once, at elaboration, and runs the processes of every cycle in that order:

- The processes that run the same code are neighbours: the instances of one process statement, in every instance of an entity or iteration of a generate, run one after another, in elaboration order.
- The statements themselves are ordered by the names of their design unit (entity, then architecture) and, within a unit, by their place in the text. The order therefore depends only on the design's text, not on the order of the files or where they are.
- A PSL directive runs before the other processes of its cycle, so that its `'EVENT` and `'SIGNAL` are those of that cycle (LRM 16.2.7); a directive that reads another's `'EVENT` or `'SIGNAL` runs after that one. Postponed processes run last.
- A process keeps its place whatever woke it; one that wakes, finds its `wait until` condition false and waits again keeps its place too.

This order was chosen because it runs fast: processes that share code keep it in the processor's caches.

Running under another order

`--order-seed N` runs the processes in another fixed order: each process gets a pseudo-random place drawn from the seed `N`. The same seed gives the same order on every run; another seed, another order.

```
jhdl -f tb.f --top work.tb --order-seed 2
```

The same in other places:

Where	How
Command line	<code>--order-seed N</code>
Environment, for every <code>jhdl</code> run (also through the front ends and VUnit)	<code>JHDL_ORDER_SEED=N</code>
Project file	<code>order-seed N</code> (Projects)
Window	Project settings → Simulation → Process order seed

`JHDL_ORDER_SEED` takes precedence over a project's `order-seed`, so a whole suite can run under another order without changing any file. Without a seed, JanusHDL uses its default order described above.

Finding races

Running a test suite under several orders is a race detector: a test whose verdict changes from one order to another has a race.

```
for seed in 2 3 4 5 6 7 8 9; do
  JHDL_ORDER_SEED=$seed jhdl -f tb.f --top work.tb --quiet > seed_$seed.log 2>&1 \
  && echo "seed $seed: passed" || echo "seed $seed: FAILED"
done
```

```
foreach ($seed in 2..9) {
  $env:JHDL_ORDER_SEED = $seed
  jhdl -f tb.f --top work.tb --quiet * > "seed_$seed.log"
  if ($LASTEXITCODE -eq 0) { "seed ${seed}: passed" } else { "seed ${seed}: FAILED" }
}
Remove-Item Env:JHDL_ORDER_SEED
```

When a seed fails:

1. Run that seed with `--wave` and open the recording in `jhdlw`.
2. Go to the time of the first difference, and open **Signals** → **Show the delta cycles of the time step at the cursor**. The **Delta cycles** panel lists, delta cycle by delta cycle, the processes that ran and the signals that changed ([Delta cycles](#)).
3. Look for two processes in one delta cycle that both touch the same shared variable, protected object, file or `std.env` call. Comparing the panel for the default order and for the failing seed shows where the two runs part.

The cure is to make the order not matter: communicate through signals, which no process sees change within a cycle, or make one process wait a delta cycle (`wait for 0 ns;`) before it reads what another writes in the same cycle.

Comparing logs between seeds, expect messages of one time step to come in different orders; messages at different times, and the verdict, should not differ.

Delta cycles in output files

- **Messages** carry their delta cycle: `100 ns (delta 2)`.
- **The waveform database** keeps the delta cycles of every time step (unless recorded with `--wave-deltas off`): which signals changed in each, from which value to which, including changes a later delta cycle overwrote, and which processes ran. The waves draw each time step's last value; the **Delta cycles** panel shows the rest.
- **VCD files** have no place for delta cycles: changes in several delta cycles of one time step collapse into the changes at that time.

Command reference

This chapter lists every option, environment variable, window command and key in one place. The other chapters explain them.

jhdl

```
jhdl [options] FILE.vhd... [-f FILES.f]...
jhdl run PROJECT.jhdlproj [options]
jhdl PROJECT.jhdlproj [options]
jhdl lib compile uvvm|osvvm FOLDER
jhdl lib list
jhdl lib remove uvvm|osvvm
```

Analyzes the files in order, elaborates the top level and runs it. `run` runs a project file; options given after the project take precedence over its settings ([Running simulations from the command line](#)). `lib` compiles UVVM or OSVVM once ([Compiled libraries](#)).

Sources and libraries

Option	Effect
FILE.vhd	A file to analyze, into the current <code>--work</code> library
-f FILES.f	Analyze the files a file list names: <code>library NAME</code> lines choose the library, other lines are paths relative to the file list, <code>#</code> starts a comment. May be repeated
--work NAME	The library for the files that follow on the command line (default <code>work</code>)
--std 93\ 02\ 08\ 19	The VHDL version, also as <code>1993</code> , <code>2002</code> , <code>2008</code> , <code>2019</code> (default <code>08</code>)
--file-std 93\ 02\ 08\ 19	Analyze the files that follow it (on the command line, or in the file list of a later <code>-f</code>) as this VHDL version; the standard packages stay those of <code>--std</code> (VHDL versions)
-DNAME=VALUE	Define an identifier for VHDL-2019 conditional analysis (<code>`if</code>); undefined identifiers are empty
--psl	Also read PSL in <code>-- psl</code> comments

Diagnostics

Option	Effect
--compat	Compatibility mode: code the LRM forbids but whose meaning is clear is a warning, and simulated as written (the rules marked <code>error</code> (<code>--compat: warning</code>) in <code>--rules</code> ; UVVM needs it)
--allow RULE	Do not report the rule's diagnostics

Option	Effect
<code>--warn</code> RULE	Report them as warnings
<code>--deny</code> RULE	Report them as errors
<code>--rules</code>	List the diagnostic rules with their levels, and exit
<code>--keep-going</code>	After analysis errors, report them and elaborate anyway; a top level that does not depend on the units in error runs (the exit code is still 1)

Elaboration and the run

Option	Effect
<code>--top</code> NAME	The top level: <code>entity</code> , <code>library.entity</code> or a configuration (default: the entity analyzed last)
<code>--elaborate-only</code>	Analyze and elaborate the top, then exit without running it
<code>-gNAME=VALUE</code>	Set a generic of the top level; with a path (<code>-guut.sub.width=8</code>), of that instance below the top, replacing its actual. The value is written as VHDL writes a literal of the generic's type: <code>42</code> , <code>16#FF#</code> , <code>1.5</code> , <code>20ns</code> , <code>idle</code> , <code>1</code> or <code>'1'</code> , <code>0101</code> , <code>x"A5"</code> , <code>(1, 2, 3)</code> (Generics)
<code>--stop-time</code> TIME	Stop at this time, such as <code>10us</code> or <code>"2.5 ms"</code> (default: run until nothing is left to simulate)
<code>--stop-level</code> note\ warning\ error\ failure	The least severity of a report that stops the simulation (default <code>failure</code>)
<code>--exit-level</code> note\ warning\ error\ failure	The least severity of a report that makes the exit code non-zero (default <code>error</code>)
<code>--order-seed</code> N	Run the processes of each cycle in a fixed order drawn from N (default: JanusHDL's own order)
<code>--ieee-warnings</code> on\ off	Whether assertions inside the IEEE packages report (default <code>on</code>)
<code>--jit</code> on\ off	Compile what qualifies to native code (default <code>on</code>); <code>off</code> runs everything in the interpreter, with the same results
<code>--max-memory</code> SIZE	Stop with an error when the run uses more memory than SIZE : bytes, or with K, M, G, T (default <code>4G</code> ; <code>0</code> = no limit)
<code>--quiet</code>	Print only reports and assertions of severity <code>error</code> and <code>failure</code> (and run-time errors)

Recording and caching

Option	Effect
<code>--wave FILE</code>	Record every signal into a waveform database, written while the run goes on, with the design's structure and delta cycles
<code>--wave-deltas on\ off</code>	Whether the waveform database keeps each time step's delta cycles (default <code>on</code>)
<code>--vcd FILE</code>	Record every signal into a VCD file, written at the end of the run (with <code>--wave</code> too: both are written)
<code>--cache FILE</code>	Keep the analysis in <code>FILE</code> and load it next time while no file changed (each file with the same VHDL version); a run that loads it reports the analysis's warnings in one line (<code>--analyze-only</code> shows them)
<code>--analyze-only</code>	Analyze (and with <code>--cache</code> , save the analysis), then exit

Other

Option	Effect
<code>--write-libraries DIR</code>	Write the sources of the built-in libraries into <code>DIR</code> , and exit
<code>--version, -V</code>	Print the version, and exit
<code>--help, -h</code>	Print the usage, and exit

Compiled libraries

Command	Effect
<code>jhdl lib compile uvvm FOLDER</code>	Analyze your copy of UVVM (<code>FOLDER</code> holds its <code>script</code> folder; its compile orders are read) and keep a record of it: from then on a design that names one of its libraries (<code>library uvvm_util</code> ;) finds it without listing its files, in every run, the window and the language server. Its files are analyzed in compatibility mode, whatever the design's. Compile again after updating your copy
<code>jhdl lib compile osvvm FOLDER</code>	The same for OSVVM: <code>FOLDER</code> is its <code>OsvvmLibraries</code> folder, read through its build scripts
<code>jhdl lib list</code>	Show what is compiled, and from where
<code>jhdl lib remove uvvm\ osvvm</code>	Forget it

The records are kept in `%LOCALAPPDATA%\JanusHDL\libraries` on Windows and `~/.cache/janushdl/libraries` on Linux (`$XDG_CACHE_HOME/janushdl/libraries` when that is set); `JHDL_LIBRARIES` chooses another folder. A design that lists a library's files itself uses those: a library comes from the compiled copy or from the design, never from both. The files of the units a design uses are analyzed as it uses them; `--cache` keeps that analysis too.

Exit codes

Code	Meaning
0	Passed
1	Analysis or elaboration failed (also with <code>--keep-going</code>), a run-time error occurred, a report reached the exit level, <code>std.env.stop</code> or <code>finish</code> gave a status other than 0, or the VCD file or waveform database could not be written
2	A wrong command line, or a file, file list or project file that cannot be read
3	The run exceeded <code>--max-memory</code>

Environment variables

Variable	Effect
<code>JHDL_JIT=off</code>	Run everything in the interpreter, as <code>--jit off</code> , in <code>jhd1</code> and in the window
<code>JHDL_ORDER_SEED=N</code>	Run under process order seed <code>N</code> , as <code>--order-seed N</code> ; takes precedence over a project's <code>order-seed</code>
<code>JHDL_PHASES=1</code>	Print the time of analysis (or of loading it from the cache), elaboration and the run, as each finishes
<code>JHDL_PEAK_MEMORY=1</code>	Print the run's peak memory use at the end
<code>JHDL_LIBRARIES=DIR</code>	Keep the compiled libraries (<code>jhd1 lib</code>) in <code>DIR</code> , for <code>jhd1</code> , the window and the language server alike

jhdlw

```
jhdlw [FILE.jhw | FILE.vcd | PROJECT.jhdlproj] [options]
jhdlw --run PROJECT.jhdlproj|FILES.f|FILE.vhd [--top LIB.ENTITY] [--stop-time TIME] [options]
jhdlw --demo | --stress | --vhdl-demo [options]
```

Options may come before or after the file, in any order. One thing is opened: a file, a demo or `--run`. An unknown option or a missing value is an error that prints the usage (exit code 2).

Argument	Effect
<code>FILE.vcd</code> , <code>FILE.jhw</code>	Open a VCD file or a waveform database

Argument	Effect
<code>PROJECT.jhdlproj</code>	Open a project
<code>--run FILE</code>	Simulate a project, file list or VHDL file at start-up, with the output in the terminal (a project is opened too). A file list or VHDL file runs with the defaults (VHDL-2008, no generics), not with what the Run dialog remembers
<code>--top LIB.ENTITY</code>	With <code>--run</code> : the top level
<code>--stop-time TIME</code>	With <code>--run</code> : when the run ends
<code>--setup FILE.wavesetup</code>	Show this wave setup once the waveform is open
<code>--demo</code>	Open the built-in demo waveform, with all its signals shown
<code>--stress</code>	Open a generated waveform of about 29 million changes
<code>--vhdl-demo</code>	Simulate a small built-in VHDL design and show the result
<code>--renderer auto\ gpu\ software</code>	Which graphics adapter draws the window: a graphics card if there is one (<code>auto</code> , the default), a graphics card only, or software only
<code>--version , --help</code>	Print the version, or the usage

Environment variable	Effect
<code>JHDL_RENDERER=auto\ gpu\ software</code>	As <code>--renderer</code>
<code>JHDL_GUI_STATE=FILE</code>	Keep the window's settings and layout in <code>FILE</code> instead of your user settings
<code>JHDL_FRAME_STATS=1</code>	Show the time each frame takes to draw, in the status bar
<code>JHDL_JIT=off</code>	Simulations in the window run in the interpreter only

jhdl-lsp

```
jhdl-lsp
jhdl-lsp --check DIR [--project PROJECT.jhdlproj|FILES.f] [--compat]
jhdl-lsp --order FILES.f
jhdl-lsp --version
jhdl-lsp --help
```

Form	Effect
<code>jhdl-lsp</code>	The language server, over its standard input and output; editors start it (External editors and the language server)
<code>--check DIR [-- project FILE] [-- compat]</code>	Analyze the design in <code>DIR</code> as the server would (with <code>--project</code> , from that project or file list; with <code>--compat</code> , in compatibility mode), print the diagnostics and a summary; exit code 1 if there are errors

Form	Effect
<code>--order FILES.f</code>	Print the file list with its files in dependency order
<code>--version</code>	Print the version
<code>--help</code>	Print the usage

Initialization options: `projectFile` (a project or a file list, relative to the workspace), `std` (`93` , `02` , `08` or `19` ; a project's own versions take precedence) and `compat` (`true` for compatibility mode).

The front ends for existing scripts

The commands in the `compat` folders, and what each accepts, are in [Running existing scripts](#).

GUI commands

Every command is in the command palette (`Ctrl+Shift+P` or `F1`) under the name below, and most are in a menu. Keys marked *waves* act while no text field has the keyboard; keys marked *editor* act while the code editor has it, and then come before the same key elsewhere.

File

Command	Keys
Open waveform...	<code>Ctrl+O</code>
Reload waveform	<code>Ctrl+R</code>
New project...	
Open project...	
Import a file list (.f) as a project...	
Project settings...	<code>Ctrl+Shift+,</code>
Save project	
Close project	
Open wave setup...	<code>Ctrl+Shift+O</code>
Save wave setup...	<code>Ctrl+S</code>
Save the waves as an image (PNG or SVG)...	
Open file in editor...	<code>Ctrl+E</code>
Save file	<code>Ctrl+S</code> (<i>editor</i>)
Save all files	<code>Ctrl+Shift+S</code>
Close file	<code>Ctrl+W</code> (<i>editor</i>)

Command	Keys
Open demo waveform	
Open stress-test waveform (about 29M changes)	
Simulate the VHDL demo design	
Quit	Ctrl+Q

Edit

Command	Keys
Find in file	Ctrl+F (editor)
Replace in file	Ctrl+H (editor)
Find next / Find previous	F3 / Shift+F3 (editor)
Find in files	Ctrl+Shift+F
Go to line...	Ctrl+G (editor)
Toggle line comment	Ctrl+/ (editor)
Add selection to next match	Ctrl+D (editor)
Fold all / Unfold all	

View

Command	Keys
Show all commands	Ctrl+Shift+P, F1
Show or hide the hierarchy (and the signals)	Ctrl+B
Show or hide the terminal	Ctrl+J
Show problems	Ctrl+Shift+M
Show or hide the minimap	
Split the editor right	Ctrl+Shift+I (editor)
Split the editor down	Ctrl+Shift+D (editor)
Close the editor pane	Ctrl+Shift+W (editor)
Show or hide the project	
Reset the layout	

Command	Keys
Bring every tab back into the main window	
Zoom to fit	F (waves)
Zoom in	+ or = (waves)
Zoom out	- (waves)
Theme: follow system, JanusHDL Dark, JanusHDL Light, One Dark, Dracula, Nord, Solarized Light	

Simulate

Command	Keys
Run simulation...	Ctrl+Shift+R
Run / continue	F5
Pause	F6
Run for the step time	F10
Restart: back to time 0	Ctrl+Shift+F5
Stop	Shift+F5
Clear terminal	

Signals

Command	Keys
Filter signals	Ctrl+F
Add all signals of the selected scope	
Insert divider above the selected signal	Ctrl+D (waves)
Add an expression of signals	E (waves)
Cursor to the previous / next change of the selected signal	Ctrl+← / Ctrl+→ (waves)
Find a value of the selected signal	/ (waves)
Find the next / previous change to the value	F3 / Shift+F3 (waves)
Add a marker at the cursor	M (waves)
Cursor to the previous / next marker	Alt+← / Alt+→ (waves)
Remove the markers and the second cursor	
Why this value?	

Command	Keys
Show the dataflow and schematic of the selected signal	
Show the delta cycles of the time step at the cursor	
Remove selected signal	Delete (waves)
Remove all signals	

Also in the palette, without a menu: **Pan left** / **Pan right** (← / →, waves) and **Select the signal above** / **below** (↑ / ↓, waves).

Help

Command	Keys
Keyboard shortcuts	
Licenses	
About JanusHDL	

Mouse actions in the wave view

Action	Effect
Ctrl +wheel, or a pinch	Zoom around the pointer
Wheel	Scroll the rows
Shift +wheel, drag	Pan in time (dragging up or down scrolls the rows)
Click	Place the cursor (it snaps to a change of the signal under the pointer nearby) and select that row
Shift +click	Place the second cursor
Click a name; Ctrl +click; Shift +click	Select a row; pick rows one by one; pick a run of rows
Drag a name	Reorder rows
Double-click a name, or click its arrow	Open or close a record, array, vector or group
Right-click a name	Radix, analog drawing, color, copy path, divider, group, open or close, change an expression, remove
Drag a marker's flag	Move the marker; right-click it to rename or remove it; double-click it to bring the cursor there
Drop a file	Open a waveform or a project, apply a wave setup, or simulate a file list or VHDL file

Action	Effect
Right-click a tab	Move it to a window of its own, or back

The editor's keys are listed in [The code editor](#).

Files

File	What it is
<code>.vhd</code> , <code>.vhd1</code> (also <code>.vht</code> , <code>.vho</code>)	VHDL sources
<code>.f</code>	A file list: <code>library NAME</code> lines and file paths (File lists)
<code>.jhd1proj</code>	A project file (Projects)
<code>.jhd1/NAME.state</code>	Your state of project <code>NAME</code> : layout, open files, last view; not for version control
<code>.jhw</code>	A waveform database, written by <code>jhd1 --wave</code> and by the window's runs
<code>.vcd</code>	A Value Change Dump file
<code>.wavesetup</code>	A wave setup: what the wave view shows (Wave setup files)
A cache file (any name)	The saved analysis of <code>--cache</code>

Troubleshooting and known limits

Installation and start-up

The command is not found

`jhd1: command not found` (Linux) or *The term 'jhd1' is not recognized* (Windows) means the `bin` folder is not on your `PATH`, or the terminal was opened before you changed it. Check the folder ([Putting JanusHDL on the PATH](#)) and open a new terminal. On Windows, an editor's built-in terminal may need the editor itself restarted to see the change.

The window does not start

- **On Linux, a missing system library.** `jhd1w` names the library and the command that installs it ([Linux](#)). The command line, `jhd1`, does not need it.
- **No graphics card, or a remote session.** The window draws with the graphics card when there is one, and in software otherwise. If it fails to start, or draws wrongly, choose the renderer yourself: `jhd1w --renderer software` (Or set `JHDL_RENDERER=software`). `--renderer gpu` insists on a graphics card.
- **Linux without a desktop.** The window needs an X11 or Wayland session. On a build server, use `jhd1`, record with `--wave`, and open the recording on a desktop machine.
- **A strange layout or a window off-screen.** **View** → **Reset the layout** puts the tabs back; **View** → **Bring every tab back into the main window** gathers tab windows from monitors that are gone. To start from scratch, close the window and delete the settings file ([Where JanusHDL keeps your settings](#)).

Analysis

A name is not declared, or a library is empty

```
error[undeclared]: `my_pkg` is not declared in `work`  
  = help: library `work` has no design units yet: analyze its sources into it first, before the files  
that use it
```

- **The order of the files.** A unit must be analyzed before the units that use it. Put the package first, or let `jhd1-lsp --order FILES.f` sort a file list ([Analysis order](#)).
- **A library you did not analyze.** UVVM, OSVVM, VUnit and vendor libraries are not built in: list their sources before yours ([Libraries](#)), or compile UVVM or OSVVM once with `jhd1 lib compile uvvm|osvvm FOLDER` ([Compiled libraries](#)). For a UVVM or OSVVM library, the message's help names that command.
- **The wrong library.** A file analyzed into `work` is not in `util`. Check the `library` lines of your file list or project, and `--work` on the command line.

When a unit fails to analyze, each unit that needs it says so (`p ... its analysis failed`) and names the file with the errors to fix ([Fix the first error first](#)).

Code that other tools accepted is an error

JanusHDL reports what the VHDL standard forbids as an error. Read the message's help line: it says how to make the code legal and, where compatibility mode accepts the code, that `--compat` runs it as written ([Compatibility mode](#)). The usual cases:

Message	What it is
<code>error[unprotected-shared-variable]</code>	A shared variable whose type is not protected (UVVM has some)
<code>error[missing-separator]</code>	A number against the next word, such as <code>10ns</code> (write <code>10 ns</code>)
<code>error[port-attribute-in-architecture]</code>	An attribute of a port specified in the architecture instead of the entity
<code>error[file-read-form]</code>	<code>READ</code> with <code>LENGTH</code> from a file of a scalar or constrained type, or without it from a <code>file of string</code>
<code>error[index-out-of-range]</code>	An index or slice outside its prefix's range, known at analysis
<code>error[call-before-body]</code> , <code>error[protected-before-body]</code>	A subprogram called, or a protected object created, before its body is elaborated

Fix the code, or run with `--compat`, which makes each of these a warning. Warnings can then be hidden with `--allow RULE`.

A procedure may not assign a signal

```
error[assignment-target]: `s` is not a signal parameter of this procedure, which is not declared in a process: it can assign only its own signal parameters (and those of a procedure it is in)
```

A procedure declared outside a process (in an architecture or a package) may assign only its own signal parameters (LRM 10.5.2.2); `--compat` does not change that. Pass the signal as a parameter of class `signal` and mode `out` or `inout`, or declare the procedure in the process that calls it: there it may assign the signals the process can see.

UVVM stops with errors about shared variables

```
error[unprotected-shared-variable]: a shared variable must be of a protected type, not T_LOG_DESTINATION
```

UVVM needs compatibility mode: add `--compat` (or `compat on` in a project, or tick **Compatibility mode** in the Run dialog). Without it, the packages with such variables fail to analyze and every unit that uses them reports errors as well, so the output is long; the first error is the one that matters ([UVVM](#)). Or compile UVVM

once (`jhdl lib compile uvvm FOLDER`, [Compiled libraries](#)) and leave its files out of your list: a compiled UVVM is analyzed in compatibility mode whatever your design's mode, so your own files need no `--compat`.

A file uses VHDL-2019 directives

```
error[syntax]: tool directives are VHDL-2019; this file is analyzed as VHDL-2008
```

Run with `--std 19`, or give that file or its library VHDL-2019 in a project ([VHDL versions](#)).

A word is reserved

A name such as `context`, `force`, `release`, `parameter`, `view` or `private` is reserved in later VHDL versions. Code that uses one as a name analyzes as an older version: give that library its own version in a project (`library legacy std=1993`).

Running

The simulation never ends

A free-running clock always has something to do, so a testbench with one runs until it stops itself or reaches a stop time. End the testbench with `std.env.finish` (or `stop`), or give `--stop-time`. In the window, **Pause** (F6) or **Stop** (Shift+F5) it.

The exit code is 1, though the testbench says it passed

The exit code is 1 when any report or assertion of severity `error` or `failure` came up, also one the testbench made on purpose to check that an error is caught. Raise the level for such a testbench: `--exit-level failure` ([Severity levels](#)).

A generic is refused

```
error: generic period: `20` is not a value of type TIME: give a number and a unit, such as 20 ns, 20ns or 1.5 us (the units: fs, ps, ns, us, ms, sec, min, hr)
```

The message says which forms the generic's type takes; [Generics](#) lists them all. The usual causes: a time without its unit, a lower-case `x` for `'X'` (character literals keep their case), a vector of the wrong length, or shell quoting that ate the quotes of `x"A5"` (write `'-gINIT=x"A5"'`).

The run stops with a memory error

```
error: the simulation uses 4100 MiB, over the --max-memory limit of 4096 MiB; stopped (--max-memory 0 turns the limit off)
```

Each run stops at 4 GiB by default. If the design needs more, raise the limit (`--max-memory 16G`) or turn it off (`--max-memory 0`). Objects that nothing reaches any more are reclaimed between simulation cycles, in every VHDL version, so a run that grows without end is usually a testbench that keeps what it allocates: a queue or a scoreboard that is never emptied, a list that only grows. A process that allocates in a loop without waiting keeps that memory until it waits.

The verdict changes with the process order

A testbench that passes under one `--order-seed` and fails under another has a race between processes: see [Finding races](#).

Native code

Native code gives the same results as the interpreter, messages included. If a run with native code behaves differently from the same run with `--jit off`, that is a defect: please report it with the design.

A typed file cannot be opened

```
cannot open file "data.bin": it holds values of type INTEGER, not REAL (the element type of
REAL_FILE). ...
```

A file of a type other than `text` and `character` holds a header that says what it holds ([Files the design reads and writes](#)). Read it with a file type of the element type it was written with. A typed file written by an earlier version of JanusHDL, or by another program, has no such header and is not opened: write it again with this version, or, for another program's binary data, read it as a file of `character`.

Waveforms

The insight panels say the recording has no structure

The panels need a waveform database written by this version: run in the window, or record with `jhd1 --wave`. A VCD file, or a recording made by an earlier version, holds values only ([What the panels need](#)).

A signal in a wave setup shows as missing

The waveform has no signal with that path: it was renamed, or the setup is for another design. The row stays, so that the setup still serves other designs; fix the path, or remove the row.

The front ends and VUnit

- **"not supported yet"**. The command, option or script command is not handled yet; the message names it. Verilog commands answer the same way.
- **A script stops at `if`, `foreach`, `proc` or `[...]`**. The front ends understand a small part of Tcl ([Limits of the front ends](#)). Move the control logic into the shell script that calls the simulator, or call `jhd1` with a file list or project.

- **The wrong command runs.** Another tool with the same command name comes first on the `PATH`. Put the JanusHDL `compat` folder first, and only one family's folder.
- **VUnit does not find JanusHDL.** With `VUNIT_SIMULATOR=janushdl`, VUnit must be 5.0.0.dev16 or later, `run.py` must add the `vunit-janushdl` package before any call that depends on the simulator, and JanusHDL's `bin` folder must be on the `PATH` (VUnit).

The editor and the language server

- **The editor marks names from other files as undeclared.** The editor analyzes a file as the current run does: the open project, else the last simulation run ([Diagnostics](#)). With no run, or for a file that is not among the run's files, names from other files are not known. Open the project, or run the file list, that the file belongs to.
- **The language server does not see the design.** It looks for the project or file list `projectFile` names, then a `.jhd1proj` in the workspace's top folder, then `jhd1.f` there ([How the server finds your design](#)). A project with an error is reported in the server's log. Check what it sees with `jhd1-lsp --check FOLDER`.

Known limits

What JanusHDL does not do yet. Where one of these is met, the message says so, by name.

Language

- In PSL, a built-in function with a clock argument inside an expression, such as `rose(b, clk)` and `c ;` the error says how to write it with `{...} @ clk`. As a whole Boolean of a sequence or property, `rose(b, clk)` works: it is `{rose(b)} @ clk`.
- Protected (IEEE 1735 encrypted) code is not decrypted.
- Verilog and SystemVerilog are not supported yet.
- Foreign code (VHPI) is not supported yet. A foreign subprogram (attribute `'FOREIGN`) runs its VHDL body if it has one; a call of one without stops the run with an error that says so.

Command line

- `--top` names an entity or a configuration; to choose an architecture, use a configuration.

Files

- Typed files written by earlier versions of JanusHDL are not read; write them again. Text files and files of `character` are not affected.

The window

- A file list or VHDL file run from the window takes a VHDL version, generics and compatibility mode from the Run dialog; per-library versions, rule levels and the other options need a project.
- The window records each run into a temporary file; keep recordings with `jhd1 --wave`.
- Extended VCD files (`$dumpports`) cannot be opened.

Language server and front ends

- Without a project, the language server analyzes every file as one VHDL version (its `std` option); a compiled UVVM or OSVVM keeps its own version and mode.
- The front ends' scripts understand a small part of Tcl; a design they open in the window does not run its scripts there.
- Through the front ends (VUnit 4), VUnit's `--gui` option is not supported; through VUnit's package `vunit-janushdl` it is ([VUnit](#)).

Reporting a problem

A good report lets the problem be reproduced and fixed quickly. Please include:

1. The version: `jhd1 --version`.
2. Your system: Windows or Linux, and which processor.
3. The exact command line, or the project file.
4. What `jhd1` printed, in full.
5. The smallest design that shows the problem. The determinism of JanusHDL means that a design that fails once fails every time, on any machine.

For a difference between native code and the interpreter, say whether it goes away with `--jit off`. For a race, give the seed (`--order-seed N`) that shows it.

Glossary

Analysis. Reading VHDL source files, checking them against the standard and adding their design units to libraries. The first phase of a run.

Analog row. A row of the wave view that draws a real, or a vector read as a number, as a line instead of a segment per value. See [Analog rows](#).

Block (editor). A rectangle of text selected in columns, with `Alt + drag` or `Alt+Shift` and the arrow keys; typing and deleting act on every line of it. See [Block editing](#).

Built-in library. A library that comes with JanusHDL and needs no analysis: `std`, `ieee` and the legacy arithmetic packages in `ieee`. Its sources are in the download's `vhd1` folder. See [Built-in libraries](#).

Cache. A file in which `--cache` keeps the analysis of a list of files, loaded on the next run while no file changed. See [Caching the analysis](#).

Compatibility mode. `--compat`: code the standard forbids but whose meaning is clear is reported as a warning and simulated as written, instead of stopping analysis. See [Compatibility mode](#).

Compiled library. UVVM or OSVVM, analyzed once from your copy with `jhd1 lib compile`: a design that names one of its libraries finds it in every run, the window and the language server, without listing its files. See [Compiled libraries](#).

Cursor, second cursor. The vertical lines in the wave view at which values are read (A) and measured from (B). See [Cursors, markers and measuring](#).

Delta cycle. A simulation cycle that does not advance time. Signal assignments made in one delta cycle take effect in the next. Messages show it as `(delta n)`. See [Determinism and races](#).

Design structure. What a waveform database records besides values: every process with its kind and place in the source, the signals it drives (and the lines that assign them) and the signals it reads. The insight panels use it.

Diagnostic. A message from analysis: an error or a warning, with its rule, place, help and LRM section. See [Reading a message](#).

Driver. A process (or concurrent statement, or port connection) that assigns a signal. A resolved signal can have several.

Elaboration. Building the design hierarchy from the top level down: instances, generics, ports, signals and processes. The second phase of a run.

Exit level. The least severity of a report that makes `jhd1` exit with code 1 (`--exit-level`, default `error`).

Expression row. A row of the wave view whose value is computed from other signals, such as `valid` and `ready`. See [Expression rows](#).

File list. A text file (`.f`) naming source files in analysis order, with `library NAME` lines choosing their libraries. See [File lists](#).

Front end. One of the small programs in the `compat` folder that answer to the command names existing scripts call, and run them on JanusHDL. See [Running existing scripts](#).

Group. A named row of the wave view that holds other rows, and can be closed and opened. See [Groups](#).

Insight panels. The **Why this value?**, **Dataflow** and **Delta cycles** tabs, which explain a signal's value from the recording. See [Insight panels](#).

Interpreter. The part of JanusHDL that runs VHDL directly. It is the reference: native code gives the same results.

Language server. `jhd1-lsp`: JanusHDL's analyzer for other editors, through the Language Server Protocol. See [External editors and the language server](#).

Legacy arithmetic packages. `std_logic_arith`, `std_logic_unsigned`, `std_logic_signed`, `std_logic_misc` and the VHDL-93 `std_logic_textio`: packages that are not IEEE standards but that many designs use from library `ieee`. Built in.

LRM. The Language Reference Manual: the VHDL standard, IEEE Std 1076. Messages cite its sections as in IEEE 1076-2019.

Marker. A named point in time in the wave view, such as `M1`. See [Cursors, markers and measuring](#).

Native code. Machine code that JanusHDL compiles from the parts of a design that qualify, as the simulation runs. On by default; `--jit off` turns it off. See [Native code](#).

Process order. The order in which the processes of one simulation cycle run, which the standard leaves open. JanusHDL's is fixed; `--order-seed` chooses another. See [Determinism and races](#).

Project, project file. A `.jhd1proj` file holding a design's sources with their libraries and VHDL versions, its top level, generics, stop time and options. See [Projects](#).

Race. A testbench whose result depends on the process order: legal VHDL, whose result the standard leaves open. See [Finding races](#).

Radix. How a vector's value is written in the wave view: binary, hexadecimal, decimal, ASCII, fixed or floating point, and so on. See [Radixes](#).

Rule. The name of a kind of diagnostic, such as `undeclared` or `unprotected-shared-variable`, whose level can be set with `--allow`, `--warn` and `--deny`. See [The rules](#).

Scope. A level of the design hierarchy: the top level, an instance, a block, a generate, a process or a package. The **Hierarchy** tab lists them.

Seed. The number `N` of `--order-seed N`, from which a process order is drawn.

Severity. The level of a report or assertion: `note`, `warning`, `error` or `failure`.

Split view. Two or more panes of the code editor showing files side by side, or the same file twice. See [Split view](#).

State file. `.jhd1/NAME.state` beside a project: your layout, open files and last wave view of that project. Not for version control.

Stop level. The least severity of a report that stops the simulation (`--stop-level` , default `failure`).

Stop time. The time at which a run ends (`--stop-time`), whatever is still scheduled.

Tab window. A tab of the window moved into a window of its own, for another monitor. See [Tab windows](#).

Time step. All the delta cycles at one simulation time.

Top level. The design unit that elaboration starts from: usually the testbench entity, or a configuration.

Typed file. A file of a file type other than `text` , such as `file of integer` . JanusHDL writes such files with a header that says what they hold; files of `character` are plain bytes. See [Files the design reads and writes](#).

VCD. Value Change Dump, the text waveform format of IEEE 1364, read and written by many tools. See [VCD files](#).

Waveform database. JanusHDL's own recording format (`.jhw`): compressed, written as the simulation runs, readable while it is written, with the design's structure and delta cycles. See [Recording waveforms](#).

Wave setup. What the wave view shows (signals, radices, colors, groups, cursors, markers, visible span), saved as a `.wavesetup` file. See [Wave setup files](#).

Work library. The library that sources are analyzed into when no other is named: `work` . Inside a library's own files, `work` means that library.