Use Modulefiles with Linux*

There are two methods for configuring your environment in Linux*:

Use modulefiles, as described on this page

Use a setvars.sh configuration file

Most of the component tool folders contain one or more modulefile scripts that configure the environment variables needed by that component to support development work. Modulefiles are an alternative to using the `setvars.sh` script to set up the development environment. Because modulefiles do not support arguments, multiple modulefiles are available for oneAPI tools and libraries that support multiple configurations (such as a 32-bit configuration and a 64-bit configuration).

NOTE:

The modulefiles provided with the Intel oneAPI toolkits are compatible with the Tcl Environment Modules (Tmod) and Lua Environment Modules (Lmod). The following minimum versions are supported:

Tmod 3.2.10 (compiler modulefile requires 4.1, see below)

Tcl version 8.4

Lmod version 8.2.10

Test which version is installed on your system using the following command:

module --version

Each modulefile automatically verifies the Tcl version on your system when it runs.

If your modulefile version is not supported, a workaround may be possible. See Using Environment Modules with Intel Development Tools for more details.

As of the oneAPI 2021.4 release you can use the `icc` modulefile to setup the `icc` and `ifort` compilers if you are using version 3.2.10 of the Tcl Environment Modules. A future oneAPI release will resolve the support for the `compiler` modulefile.

The oneAPI modulefile scripts are located in a modulefiles directory inside each component folder (similar to how the individual vars scripts are located). For example, in a default installation, the `ipp` modulefiles script(s) are in the `/opt/intel/ipp/latest/modulefiles/` directory.

Due to how oneAPI component folders are organized on the disk, it can be difficult to use the oneAPI modulefiles directly where they are installed. Therefore, a special `modulefiles-setup.sh` script is provided in the oneAPI installation folder to make it easier to work with the oneAPI

modulefiles. In a default installation, that setup script is located here: `/opt/intel/oneapi/modulefiles-setup.sh`

The `modulefiles-setup.sh` script locates all modulefile scripts that are part of your oneAPI installation and organizes them into a single directory of versioned modulefiles scripts.

Each of these versioned modulefiles scripts is a symlink that points to the modulefiles located by the `modulefiles-setup.sh` script. Each component folder includes (at minimum) a "latest" version modulefile that will be selected, by default, when loading a modulefile without specifying a version label. If you use the `--ignore-latest` option when running the `modulefiles-setup.sh` script, the modulefile with the higest semver version will be loaded if no version is specified by the `module_load` command.

Creating the `modulefiles` Directory

Run the `modulefiles-setup.sh` script.

NOTE:

By default, the `modulefiles-setup.sh` script creates a folder named modulefiles in the oneAPI toolkit installation folder. If your oneAPI installation folder is not writeable, use the `--output-dir=<path-to-folder>` option to create the modulefiles folder in a writeable location. Run `modulefiles-setup.sh --help` for more information about this and other modulefiles-setup.sh script options.

Running the `modulefiles-setup.sh` script creates the `modulefiles` output folder, which is organized like the following example (the precise list of modulefiles depends on your installation). In this example there is one modulefile for configuring the Intel® Advisor environment and two modulefiles for configuring the compiler environment (the compiler modulefile configures the environment for all Intel compilers). If you follow the latest symlinks, they point to the highest version modulefile, per semver rules.

```
|-- advisor
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/advisor/2021.2.0/modulefiles/advisor
|   `-- latest -> /home/ubuntu/intel/oneapi/advisor/latest/modulefiles/advisor
|-- ccl
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/ccl/2021.1.1/modulefiles/ccl
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/ccl/2021.2.0/modulefiles/ccl
|   `-- latest -> /home/ubuntu/intel/oneapi/ccl/latest/modulefiles/ccl
|-- clck
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/clck/2021.1.1/modulefiles/clck
|   `-- latest -> /home/ubuntu/intel/oneapi/clck/latest/modulefiles/clck
|-- compiler
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler
|-- compiler-rt
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt
|-- compiler-rt32
```

```
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt32
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt32
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt32
|-- compiler32
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler32
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler32
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler32
```

Update your MODULEFILESPATH to include to the `modulefiles` output folder that was created by the `modulefiles-setup.sh` script or run the `moduleuse <folder_name>` command.

Installing the Tcl Modulefiles Environment onto Your System

The instructions below will help you quickly get started with the Environment Modules utility on Ubuntu*. For full details regarding installation and configuration of the `module` utility, see http://modules.sourceforge.net/.

$ sudo apt update
$ sudo apt install tcl
$ sudo apt install environment-modules

Confirm that the local copy of `tclsh` is new enough (see the beginning of this page for a list of supported versions):

$ echo 'puts [info patchlevel] ; exit 0' | tclsh
8.6.8

To test the `module` installation, initialize the `module` alias.

$ source /usr/share/modules/init/sh
$ module

NOTE:

Initialization of the Modulefiles environment in POSIX-compatible shells should work with the `source` command shown above. Shell-specific init scripts are provided in the `/usr/share/modules/init/` folder. See that folder and the initialization section in `man module` for more details.

Source the `module` alias init script (`.../modules/init/sh`) in a global or local startup script to ensure the module command is always available. At this point, the system should be ready to use the module command as shown in the following section.

Getting Started with the `modulefiles-setup.sh` Script

The following example assumes you have:

installed `tclsh` on to the Linux development system

installed the Environment Modules utility (i.e., `module`) onto the system

sourced the `.../modules/init/sh` (or equivalent) module init command

installed the oneAPI toolkits required for your oneAPI development

```
$ cd <oneapi-root-folder>       # cd to the oneapi_root install directory
$ ./modulefiles-setup.sh        # run the modulefiles setup script
$ module use modulefiles        # use the modulefiles folder created above
$ module avail                  # will show tbb/X.Y, etc.
$ module load tbb               # loads tbb/X.Y module
$ module list                   # should list the tbb/X.Y module you just loaded
$ module unload tbb             # removes tbb/X.Y changes from the environment
$ module list                   # should no longer list the tbb/X.Y env var module
```

Before the unload step, use the `env` command to inspect the environment and look for the changes that were made by the modulefile you loaded. For example, if you loaded the `tbb modulefile`, the command will show you some of the env changes made by that `modulefile` (inspect the `modulefile` to see all of the changes it will make):

```
$ env | grep -i "intel"
```

NOTE:

A `modulefile` is a script, but it does not need to have the 'x' (executable) permission set, because it is loaded and interpreted by the "module" interpreter that is installed and maintained by the end-user. Installation of the oneAPI toolkits do not include the `modulefile` interpreter. It must be installed separately. Likewise, `modulefiles` do not require that the 'w' permission be set, but they must be readable (ideally, the 'r' permission is set for all users).

Versioning

The oneAPI toolkit installer uses version folders to allow oneAPI tools and libraries to exist in a side-by-side layout. These versioned component folders are used by the `modulefiles-setup.sh` script to create the versioned `modulefiles`. The script organizes the symbolic links it creates in the `modulefiles` output folder as `<modulefile-name>/version`, so that each respective `modulefile` can be referenced by version when using the module command.

```
$ module avail
---------------- modulefiles -----------------
ipp/1.1  ipp/1.2  compiler/1.0  compiler32/1.0
```

Multiple modulefiles

A tool or library may provide multiple `modulefiles` within its `modulefiles` folder. Each becomes a loadable module. They will be assigned a version per the component folder from which they were extracted.

Understanding How the `modulefiles` are Written when using oneAPI

Symbolic links are used by the `modulefiles-setup.sh` script to gather all the available `modulefiles` into a single `modulefiles` folder. This means that the actual `modulefile` scripts are not moved or modified. As a consequence, the `${ModulesCurrentModulefile}` variable points to the symlink to each `modulefile`, not to the actual `modulefile` located in the respective installation folders. To determine the full path to the actual modulefiles, each modulefile starts with a statement like this:

[file readlink ${ModulesCurrentModulefile}]

to get a direct reference to the original `modulefile` in the product install directory. This is done because the actual install location can be customized and is, therefore, unknown at runtime and must be deduced. For that reason, the actual modulefile cannot be moved outside of the installed location, otherwise it will not be able to locate the absolute path to the library or application that it must configure.

For a better understanding, review the `modulefiles` included with the installation. Most include comments explaining how they resolve `symlink` references to a real file, as well as parsing the version number (and version directory). They also include checks to insure that the installed TCL is an appropriate version level.

Use of the `module load` Command by `modulefiles`

Several of the `modulefiles` use the `module load` command to ensure that any required dependent modules are also loaded. There is no attempt to specify the version of those dependent modulefiles. This means you have the option to load a specific version of a dependent module prior to loading the module that requires that dependent module. If you do not preload a dependent module, the latest available version is loaded.

This is by design because it gives you the flexibility to control the environment. For example, you may have installed an updated version of a library that you want to test against a previous version of the compiler. Perhaps the updated library has a bug fix and you are not interested in changing the version of any other libraries in your build. If the dependent modulefiles were hard-coded to require a specific dependent version of this library, you could not perform such a test.

NOTE:

If a dependent `module load` cannot be satisfied, the currently loading module file will be terminated and no changes will be made to your environment.