cadence

Packaging Reusable Components EZ-Start Guide

September 2006

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Packaging Reusable Components EZ-Start Guide

About EZ-Start Guides

EZ-Start guides are provided by Cadence Design Systems as entry point tutorials for various technologies. EZ-Start guides are meant to quickly review high-level concepts of a specific technology, and allow the user to independently experience and explore it. For a deeper understanding of how to architect a quality verification environment, consult a Cadence methodology specialist or refer to the *Plan-to-Closure Methodology User Guide*.

Introduction

The complexity of today's designs and protocols demand *verification reuse*, a fast-growing trend common in ASIC design houses and commercial component vendors. Verification reuse starts with breaking a desired environment into generic sub-components that can laterbe used collaboratively to verify the current DUT, as well as other DUTs. By creating and developing these types of components, you can distribute the development effort, minimize the required protocol knowledge, and reduce the overall verification effort.

Because typical system-on-chip devices (SoCs) have multiple interfaces, they can benefit greatly from multiple verification component integration. However, it can be challenging to understand and use components that originated from multiple sources and are constructed differently. For example, how do you adjust a generic component to your local requirements, or how difficult will it be to learn the test user interface of each component? You might also face challenges with the packaging and integration of verification components. For example, do you have all of the files, documentation, and usage examples for this component, and can the component code coexist with your own files?

Although common sense should govern the right directory structure for your environment, it is still most beneficial to adhere to some type of standard. In this case, adhering to a standard will enable you to smoothly adopt a component developed by another team that you might not be familiar with, or that might be working for a different company.

This EZ-Start guide outlines simple rules that can help component developers and integrators successfully share and reuse verification components. The standards recommended in this EZ-Start guide have been proven to work with different requirements and in different application domains.

Universal Verification Components (uVCs)

While this guide focuses on SystemVerilog, the past and present indicate that the future is a multi-language one. Factors like language inherent differences, existing legacy code, and personal preferences create a colorful world with multiple languages.

Universal verification components (uVCs) are verification components that are implemented in one or more computer languages. Typically, these components contain all the logic required for a verification task (such as coverage, checking, and stimuli generation). The Universal Reuse Methodology (uRM) is a combination of a methodology and technology that enables rapid reusable component development. Among others, the uRM defines standard packaging guidelines for delivering verification components, regardless of their implementation language. A uVC *package* is a standard directory structure, and it is the basic unit for developing and sharing any verification logic.

The following lists the different types of packages that can be shared:

- Interface verification components—This package captures the logic of an interface protocol and provides all of the logic required to verify a device which supports that interface. Examples include PCI-E uVCs and Ethernet uVCs.
- Module verification components—This package is a container for design-specific verification logic. Examples include reference-models for checking, high-level models for faster simulation, whitebox checking, and assertions. Module verification components travel with the design IP, which enables verification of this IP within various contexts.
- Common utilities—Wrapping utilities as packages is beneficial, because it can facilitate easier usage of the utilities. Examples of utilities include math tasks, visualization, and debugging aids.
- A verification environment—A verification environment is a collection of collaborating verification components with some logic. Organizing all of your packages in a consistent way allows easy integration and reduces the support required for sharing code.

Note: For more guidelines on code sharing, such as environment structure, configuration, and test writer interfaces, refer to the *Plan-to-Closure Methodology Guide.*.

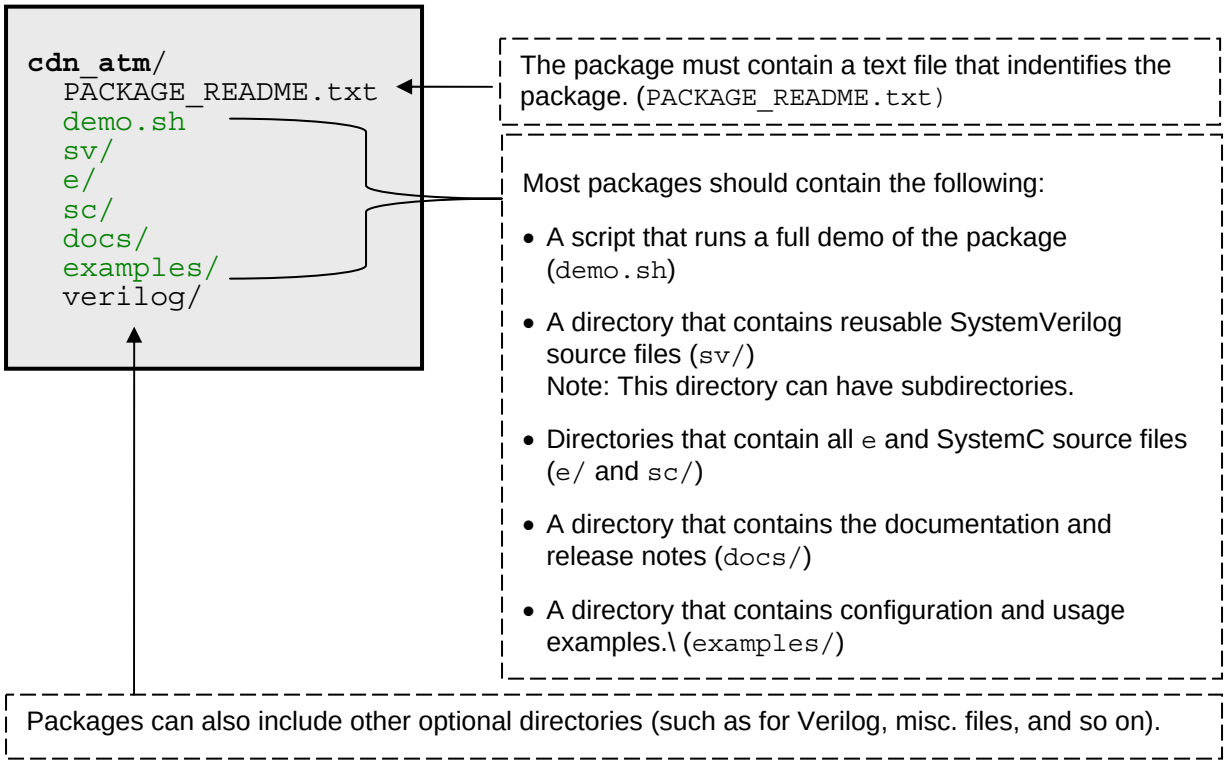
Reusable Packages

A *package* is the basic unit for sharing code, which can be for uVCs or other utilities. This section describes how you can use the uRM guidelines to build your uVC packages so that they can be easily explored and used by other in their own local environments.

Directory Structure for Packages

The following illustrates a typical uVC package directory structure.

Figure 1 uVC Package Structure



Note the following when developing the directory structure of a uVC package:

- The `PACKAGE_README.txt` file is a structured text file that describes the content of the uVC, its directory structure, and any accompanying scripts.

  For example, the following is the content of the `PACKAGE_README.txt` file included with this EZ-Start.

```
1 * Title: EZ start packaging tutorial
2 * Name: my_first_package
3 * Version: 0.1
4 * Requires:
5 * Modified: Aug-2006
6 * Category: Tutorial
7 * Support: sharonr@cadence.com
8 * Description:
9
10 It contains the following files and directories:
11
12  demo.sh             : The demo script
13  docs/               : Contains a short document to describe the lab
```

```
14  sv/
15  examples/
```

For more information on the required fields in the `PACKAGE_README.txt`, refer to the *Plan-to-Closure Methodology Guide*.

Tip A quick way to create a `PACKAGE_README.txt` file is to copy an existing one and modify it to suit your environment. Sample `PACKAGE_README.txt` files are included as a part of the golden examples located in the `urm_lib` directory.

- Most often, users will want to check a complete installation of a reusable component. The `demo.sh` script runs an independent check of the environment.
- For SystemVerilog uVC packages, you should have a `run.f` script in the `examples/` directory that will load all of the reusable code.
- Packaged code must be generic and reusable, so that it can apply to the verification of multiple devices. Thus, the configuration and settings of the reusable code for a target environment must be done in a `config` file.

  Tip Usually, examples of package usage are in the `examples/` directory.
- Use standard naming conventions when naming your uVC packages. This is described in the following section.

Standard Naming Conventions for Packages

When you use standard naming conventions, it is easy to navigate through the contents of a uVC package. You find things easily when you know how files and directories should be named. This also makes it easier to predict the contents of a directory.

In order to prevent naming collisions, uVC packages should have unique names and use the following form:

```
<prefix>_<descriptive_name>
```

where:

- `<prefix>_` is two to four characters and usually denotes the company name (for example, `cdn_` for Cadence, or `arm_` for ARM).

  Note: Cadence monitors company prefixes to prevent duplication. If you do not know your company's prefix or need to register a new one, ask your local customer engineer or send an e-mail to urm_admin@cadence.com.

  The following are other types of prefixes:

  - `shr_`—Denotes shareware. Use this if you do not want to attach the package to a company name.

  - ex_ —Use this for small example packages.

- <descriptive_name> provides a unique identifier for the package. For example, cdn_**atm** or vr_**xbus**.

The uVC package name precedes all global names within the package. For example:

- vr_xbus_types.sv and vr_xbus_env.sv (for filenames)
- vr_xbus_master_t and vr_xbus_transfer_t (for type names)

For more information about packaging, refer to the *Plan-to-Closure Methodology Guide*.

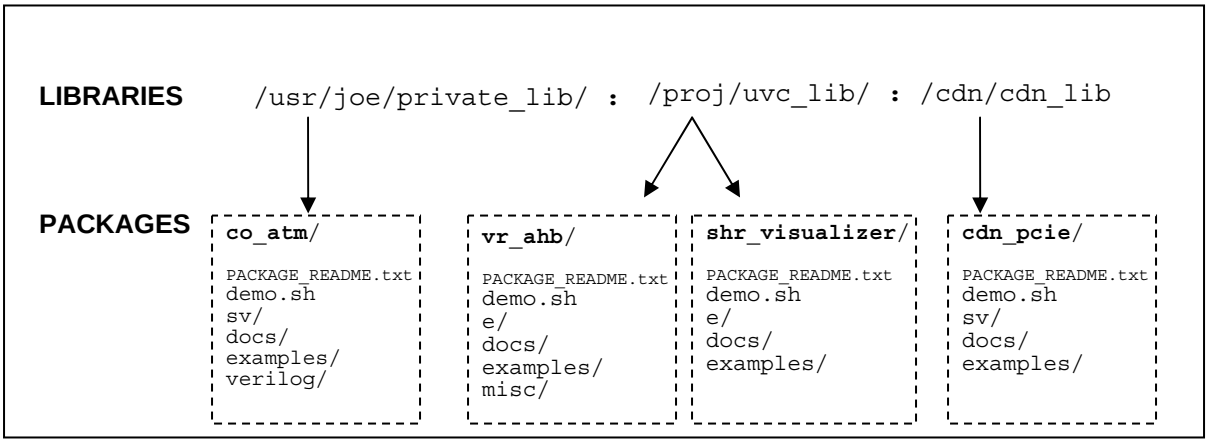
Reusable Libraries

Now that you know the standard directory structure and naming conventions for a uVC package, you need to know how to organize uVC packages on the disk. This section will help answer the following questions:

- Can I arbitrarily place packages in various directories?
- How can I find a specific uVC?
- What versions of a specific uVC exist on my disk?

Libraries offer a solution to these questions. A *library* is a UNIX directory that contains uVC packages (as illustrated below). Libraries organize your packages on the disk so that they are easier to use and locate.

The following illustrates how packages can be grouped inside of libraries.



Note the following when defining your libraries:

- Choose the order of your directories carefully.

  Libraries should be listed in the `$IPCM_PATH` in the order in which they should be searched. For example (as illustrated in the previous figure): `/usr/joe/private_lib/:/proj/uvc_lib/:/cdn/cdn_lib...`

- To control things like visibility and compilation order, organize uVC packages into a combination of private, project, company, and commercial libraries (as illustrated in the previous figure).

- Each library directory should contain a text file called `LIBRARY_README.txt`, which describes the contents of the library, such as the various uVC packages and sub-directories.

  For example, the following is the content of the `LIBRARY_README.txt` included with this EZ-Start.

```
1 * Title: my_first_library library
2 * Version:  0.1
3 * Modified: Aug-2006
4 * Description:
5
6   The my_first_library is part of the EZ start tutorials
```

  As for packages, you can copy an existing file and modify it for your local environment.

- Each library should contain various packages as subdirectories

Locating a Package

Cadence provides a script called `ipcm_which.sh`, which will go through the `IPCM_PATH` and identify all of the packages installed on the disc.

For example, you set the `IPCM_PATH` to:

```
%setenv IPCM_PATH /user/sharon/my_lib:/projects/sbt_router_lib:/company/company_lib
```

Now, you can use the `ipcm_which.sh` script to search for a package along this path. For example, if you type the following from the command prompt:

```
% ipcm_which.sh ex_my_first_package
```

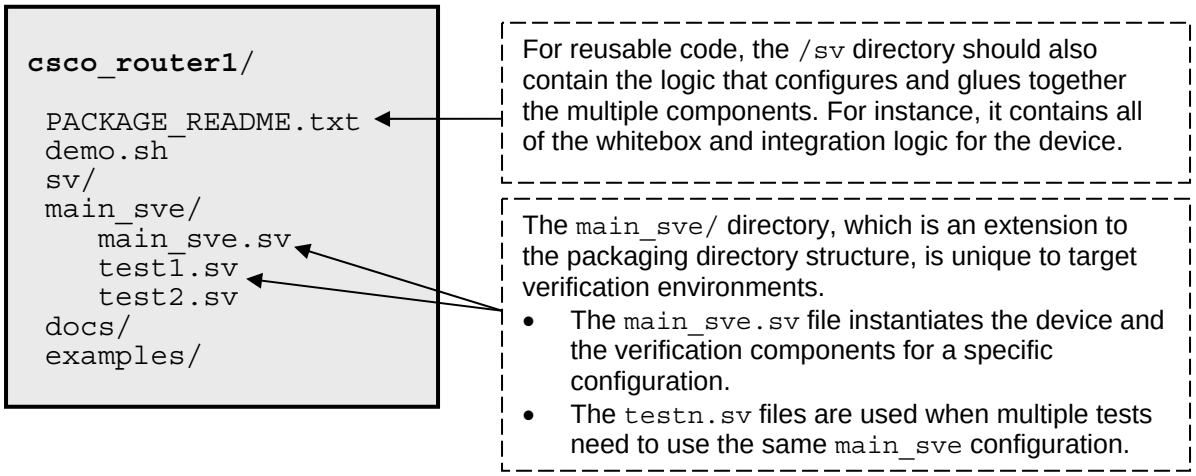
you will get:

```
/home/sharonr/EZStartPackaging/solutions/my_first_library/ex_my_first_package
```

Simulation Verification Environments (SVEs)

Now that you have learned about organizing and packaging reusable code, this section will describe the directory structure for using your code on a target device under verification (DUV).

uVC packages and reusable code use the same directory structure. The following figure illustrates the directory for reusable code and notes any extensions to the structure for uVC packages.



Note: Using a central `sve` file for multiple tests allows tests that are shorter, easier to read, and easier to maintain (in case of configuration changes).

For more information about the `sve/` directory structure, refer to the *Plan-to-Closure Methodology Guide.*

Packaging Environments Lab

You will use the directory structures described in the previous sections to create your own library and reusable package directory.

In this lab, you will create a library and a reusable package directory. You will ensure that the package directory has the correct structure and that it contains the right files. You will then set your `$IPCM_PATH` path so that it points to the right location. At the end of this lab, you will check your work using a script called `demo.sh`.

Lab instructions:

1. Untar and unzip the `EZStartPackaging*.tar.gz` file included with this document:

   ```
   % gtar –zxvf EZStartPackaging.tar.gz
   ```

   This creates an `EZStartPackaging` directory.

2. Create a directory called `my_first_library`.

3. Within `my_first_library`, create a subdirectory called `my_first_package`.

4. Using the directory structures discussed in the previous sections, create three directories: `sv`, `examples`, and `docs`. Place them in the correct location.

5. Copy all of the files from the `EZStartPackaging/files` directory into the correct sub-directories

6. Set the `$IPCM_PATH` path so that it points to the correct location.

7. Use the `ipcm_which.sh` script to determine whether the package is in the correct location. For example:

   ```
   % ipcm_which.sh my_first_package
   ```

8. Run the `demo.sh` script, which is located in the `EZStartPackaging/files` directory.

   If you receive the "*well done*" message, then you have successfully completed the lab. Otherwise, review your directory structure, make adjustments, and retry step #8.