ENOVIA DesignSync

CTS™ Programmer's Guide

3DEXPERIENCE 2022

Table Of Contents

Release Information

Documentation

Release-specific information is located on the Dassault Systèmes support website in the Program Directory (http://media.3ds.com/support/progdir/). The Program Directory contains release-specific information for all major DesignSync releases beginning with V6R2009x.

Selecting the appropriate release

1. Open the Program Directory (http://media.3ds.com/support/progdir/). You may be required to enter your username and password to access information on the 3ds support site.
2. Select the following options in the top bar:

   Select Line: **Version 6**
   Select Level: **V6R2014**
   Select Sub-Level: (use default)

   Note: By default, the sub-level is always the most current version of the Program Directory files for the selected Level. There should never be a reason that information you need for a release is not in the most current version.

Available Release-Specific Documentation

The documents listed in the following table are available.

Product Enhancement Overview	Contains the list of new features and enhancements for the release.
General and Open Issues	Contains any known release issues, platform support information, platform configuration information, and system configuration recommendations for the release.
Closed Issues	Contains a complete list of closed issues for the release.
Installation	Installation instructions for DesignSync clients on all supported platforms. For server configuration information, see the *ENOVIA Synchronicity DesignSync Administrator's Guide*.

Locating the Release Specific Documentation

Product Enhancement Overview

1. In the left frame, select **ENOVIA** in the **Product Enhancement Overview** Section. This opens the Product Enhancement Overview index in the right frame.
2. Navigate to the IP Work-in-Progress section and select **Synchronicity DesignSync Data Manager**, or use your browser search functionality to search for **Synchronicity DesignSync Data Manager**. Selecting **Synchronicity DesignSync Data Manager** opens the Product Enhancement Overview for DesignSync.

General and Open Issues

1. In the left frame, select **ENOVIA** in the **General and Open Issues** Section
2. Navigate to the IP Work-in-Progress: Semiconductor EDA section and select **Synchronicity DesignSync Data Manager (SYN)**, or use your browser search functionality to search for **Synchronicity DesignSync Data Manager**. Selecting **Synchronicity DesignSync Data Manager (SYN)** opens the General and Open Issues for DesignSync.

Closed Issues

1. In the left frame, select **List of Closed Issues** in the **Closed Issues** Section.
2. Use your browser search functionality to search for Synchronicity. This will bring you to the section of the closed issues list that includes the following products:

Synchronicity DesignSync (including DSclipse, and DSVS plug-ins)
Synchronicity DesignSync Add-On for DFII
Synchronicity DesignSync Add-On for DSMW
Synchronicity DesignSync Add-On for DSCC
Synchronicity DesignSync Add-On for CTS
Synchronicity ProjectSync

Note: Not all releases include closed issues for all DesignSync products.

Installation

1. In the left frame, select **ENOVIA Server** in the **Installation** Section
2. Select **ENOVIA Synchronicity DesignSync Data Manager** in the navigation links at the top of the page or use your browser search functionality to search for **ENOVIA Synchronicity DesignSync Data Manager**.
3. Select the **Installing Synchronicity DesignSync Data Manager** link to open the Installation document.

Introduction to the DesignSync Custom Type System

ENOVIA Synchronicity DesignSync® CTS(TM) is a programming interface used to customize DesignSync to manage your unique design data. DesignSync lets you define special object types and group files into abstract objects, such as a design view encompassing a number of files. You can check in, check out, and tag this abstract object, called a **collection**, as a single object. DesignSync safeguards your data by preventing users from checking in the constituent parts of a collection. Instead, users have to operate on the collection as a whole.

The Custom Type System can also be used to define special object types (files or folders). For example, particular object types might need to be checked in together or listed in a special way.

In effect, you use the Custom Type System to instruct DesignSync on the nature of your design data. In this way, DesignSync can efficiently traverse your data hierarchy, performing revision control operations on special objects or collections of your data.

To model your data, you create a Custom Type Package (CTP), a Tcl file containing procedures that recognize and traverse your custom data hierarchy, creating new object types and grouping the data into collections. You install the CTP within the DesignSync custom hierarchy. When next you invoke a DesignSync client, the DesignSync Custom Type System registers the CTP so that each revision control operation can now recognize and manage the special types and collections defined in your CTP.

In order to create CTPs, you must develop Tcl procedures. This document assumes in-depth knowledge of the Tcl programming language.

Important: When writing CTPs, you should not use any DesignSync commands that depends on object recognition such as the URL commands or primary DesignSync commands such as ci or populate. These commands rely on CTPs within DesignSync and can set up a recursive loop situation.

Note on the use of this guide: References from the *ENOVIA Synchronicity DesignSync Data Manager Custom Designer User's Guide* to the *ENOVIA Synchronicity Command Reference* guide always link to the ALL version of the guide, which contain information about all working methodologies for DesignSync. For more information about the available working methodologies, see ENOVIA Synchronicity Command Reference.

Installing Custom Type Packages

Installing Custom Type Packages

A Custom Type Package (CTP) is defined in a file with a `.ctp` extension, containing the procs that recognize and handle a collection of data. By installing a CTP in the site or project `ctp` area of your custom hierarchy, you register the CTP with the DesignSync Custom Type System (CTS). If you make changes to the CTP, you need to restart your client to force DesignSync to reread your `.ctp` file.

The following scenario shows how to install a sample CTP provided with this document. To create your own CTP, see Developing Custom Type Packages.

Where to install custom type packages

You can define a Custom Type Package for a particular project or for an entire site. You store the CTPs in the following locations in your DesignSync custom hierarchy:

- Project-level CTP:

  ```
  <SYNC_PROJECT_CFGDIR>/ctp/<ctp_file>.ctp
  ```

- Site-level CTP:

  ```
  <SYNC_SITE_CUSTOM>/share/client/ctp/<ctp_file>.ctp
  ```

`<SYNC_PROJECT_CFGDIR>` has no default; no project information is loaded if this environment variable is not set. <`SYNC_SITE_CUSTOM>` is equivalent to `<SYNC_CUSTOM_DIR>/site`; if <`SYNC_SITE_CUSTOM>` is not set, but <`SYNC_CUSTOM_DIR>` is set, DesignSync can still access the site-wide CTP.

Note:

Unlike registry files and access control files, there is no implied search order for CTPs. If you include a CTP of the same name in both the project and site `ctp` directories, CTP registration fails. Also, if a CTP of the same name exists already in the DesignSync ctp area (`<SYNC_DIR>/share/client/ctp`), CTP registration fails. Do not store custom CTPs in the DesignSync `<SYNC_DIR>/share/client/ctp` area because your CTPs will be overwritten if you reinstall DesignSync.

When are CTP files sourced?

Each `.ctp` file is sourced during initialization, or whenever a new client-sideTcl interpreter is created. The DesignSync Custom Type System invokes the custom CTP procedures (in the `.ctp` file) as appropriate during revision control operations and other operations that traverse the data hierarchy.

To register a CTP:

The following steps show how to install and thus register a sample CTP.

1. Create a `ctp` directory to store your CTPs if the `ctp` directory has not yet been created:

   ```
   $ cd $SYNC_SITE_CUSTOM/share/client

   $ mkdir ctp

   $ cd ctp
   ```

   This example shows how to store the CTP in the site `ctp` area, but you can store the CTP in your project `ctp` area, as well: `<SYNC_PROJECT_CFGDIR>/ctp`.

2. Copy the `collection.ctp` sample CTP from the examples directory to your custom ctp area:

   ```
   $ cp $SYNC_DIR/share/examples/doc/ctsguide/collection.ctp .
   ```

   Note: By default, `$SYNC_SITE_CUSTOM` resolves to `$SYNC_CUSTOM_DIR/site`.

   If you take a look at the contents of the `collection.ctp` file, you will notice the following `namespace eval` command:

   ```
   namespace eval collectionCTP {}
   ```

   For a CTP to be registered, the namespace name must correspond to the CTP filename. For example, if the CTP is named `mycollectionCTP`, it must be stored in a file named `mycollection.ctp`. Notice that the procedures defined in the `collection.ctp` file are in the `collectionCTP` namespace, for example:

   ```
   proc collectionCTP::getBase {filename} {
   ...
   }
   ```

3. Invoke a DesignSync client (DesSync, stclc, stcl, dssc, or dss).

   ```
   $ stclc
   ```

The DesignSync client is now configured to recognize the data represented by the `collection.ctp` CTP.

See DesignSync Data Manager User's Guide to learn how to set vaults and check in new design data. DesignSync will recognize the file structure of the data directory as being a collection. The Custom Type System then maps this system view into a custom view. The Custom Type System creates a collection object to represent the group of files.

The collection object is visible within DesignSync, but not by listing the contents of the file-system directory outside of DesignSync. Use the DesignSync graphical interface List View or the DesignSync ls -report OX command to view the collection object and its members. The collection is checked in as a whole; its constituent members are not checked in separately. This process is called view mapping and for custom collections, the CTP's custom mapView procedure performs this process. See DesignSync Recognition of Custom Type Packages: What Is View Mapping?

What if the CTP was not installed before check-in?

If you check in your design files before the CTP is installed, DesignSync does not recognize your design files as collections of related data. You can recover from this circumstance by installing the CTP and checking in the data again using the `ci -new` option (or the **Allow check in of new items** check box in the Check In dialog of the DesignSync graphical interface). DesignSync recognizes the data as a collection representing the CTP and checks in the collection as a single entity.

Developing Custom Type Packages

DesignSync Recognition of Custom Type Packages

DesignSync recognizes custom object types, collections, and collection members by applying the object recognition procedures the CTP developer includes in the `.ctp` file. The CTP developer creates procedures that traverse the data hierarchy and recognize data objects and collection members. Two important procedures the CTP developer creates are the `mapViews` and `updateObject` procedures which recognize specific file and directory attributes as those of a particular collection and perform **view mapping**.

What Is View Mapping?

There are two types of view mapping performed by the CTP:

- View mapping of a directory

  The `mapViews` procedure performs view mapping on the objects in a directory. The `mapViews` procedure detects data that matches particular criteria for special objects or collections. The `mapViews` procedure performs view mapping each time a DesignSync command browses a directory.

- View mapping of a single object

  The `updateObject` procedure performs view mapping on a single object. DesignSync commands invoke the `updateObject` procedure to request information on a specific object.

If the `mapViews` or `updateObject` procedures detect data that matches the special object or collection criteria, they map the system view of the data onto a custom view. The view mapping process generates this custom view by creating new objects, called collections, and assigning custom properties to existing objects, for example, by marking existing objects as special objects or collection members.

The Custom View of the Collection Data

The view mapping process generates a custom view of the data containing:

- Special objects

  CTPs can define special object types (files or folders) recognized by the view mapping algorithm. The view mapping algorithm assigns special properties to these objects.

- Collection objects

  Collection objects are new objects the `mapViews` procedure generates. Collection objects are named with the format `<object>.sgc.<collectiontype>`, where:

  `<object>` is the base name of the object, for example, a cell or view name.

  `sgc` indicates a custom collection defined in a Custom Type Package.

  `<collectiontype>` is the collection name defined in a Custom Type Package (CTP).

  An example of a custom generic collection object is `symbol.sgc.mytool`.

  This collection object is the handle by which DesignSync accesses all of the members of the collection. DesignSync performs revision control operations on the collection object, not its constituent members. The collection object does not exist as a regular file on the file system; you can only manage or view the object using DesignSync commands. Use the DesignSync List View or the `ls -report OX` text command to view collection objects.

- Collection members

  Collection members are the constituent objects of a collection. During view mapping, the `mapViews` procedure applies properties to the members, including their object types (`objtype`), such as `"mytool Member"`. The developer of the `mapViews` procedure decides which properties to set during the view mapping process. Specifically, the CTP developer must ensure that the object type for collection member files is non-versionable by either defining its `objtype` property such that it ends in `" Member"` (note the space before the `M`) or by using `sctp::setTypeProps` to set its `versionable` property to 0. See CTP Object Properties and CTP Object Type Properties for details.

  Collection members are regular files on the file system. DesignSync does not perform revision control operations explicitly upon collection members, but instead operates upon the collection object as a whole. The collection members do not display as managed objects in the DesignSync List View or the stclc/dssc `ls` listing. Instead DesignSync lists them with their object types, which typically includes the name of the member's owner collection.

- Non-member objects

Non-member files are regular files on the file system that are related to a collection, but not part of the collection. Non-member files are not operated on with the collection, but can be operated on by DesignSync commands as individual files. For example, non-member files might be derived files that do not need to be managed but are related to a collection.

- Non-versionable objects

  The CTP developer might want to mark some objects as non-versionable to prevent DesignSync from performing revision control operations on these objects. To do so, the CTP developer can set the `versionable` property to `false`. See CTP Object Type Properties for a description of the `versionable` property.

You view this custom view of the data using the DesignSync List View or the `ls -report O` command.

Exception Handling in View Mapping Procedures

If the `mapViews` or `updateObject` procedures throw an error, DesignSync assigns the `error` property to the affected objects. For a `mapViews` exception, DesignSync assigns the `error` property to all objects in the folder. If the CTP contains no `mapViews` procedure, DesignSync assigns the error property to all objects. You must add a `mapViews` procedure to the CTP before DesignSync can successfully check in any objects. For an `updateObject` exception, DesignSync assigns the `error` property to the object being updated.

If objects contain the `error` property, DesignSync prevents check-in operations on the affected objects to ensure the integrity of your collection data. In this case, DesignSync displays an error message indicating why the check-in operation failed.

For more information on the `error` property, see CTP Object Properties.

Developing Custom Type Packages

A Custom Type Package (CTP) defines special objects as well as generic collections -- groups of data files and folders that you want DesignSync to check in and out as a single entity. To create a CTP, you develop the procedures that recognize special objects and collection members and map them into abstract types and collections. For more information about this mapping process, see DesignSync Recognition of Custom Type Packages.

You store the CTP procedures in a Tcl file with a `.ctp` extension placed in the project or site `ctp` area of your custom hierarchy:

- Project-level CTP:

```
<SYNC_PROJECT_CFGDIR>/ctp/<ctp_file>.ctp
```

- Site-level CTP:

```
<SYNC_SITE_CUSTOM>/share/client/ctp/<ctp_file>.ctp
```

`<SYNC_PROJECT_CFGDIR>` has no default; no project information is loaded if this environment variable is not set. <`SYNC_SITE_CUSTOM>` is equivalent to `<SYNC_CUSTOM_DIR>/site`; if <`SYNC_SITE_CUSTOM>` is not set, but <`SYNC_CUSTOM_DIR>` is set, DesignSync can still access the site-wide CTP.

Types Versus Object Types

DesignSync has three distinct notions of types:

- The **basic** type, such as File, Folder or Collection, which is used internally and not reported to the user.
- The **objtype** type, which is a property set in the CTP to identify particular types of objects, such as "EDA Cell View". The `-report O` option to the DesignSync `ls` command shows the **object type.**
- The **label** value, which is a property set in the CTP, per object, to differentiate items of the same **objtype**. For instance, two "EDA Cell View" objects may be labeled "EDA Schematic" and "EDA Symbol". Both objects are cell views, but they are different kinds of cell views.

The DesignSync object type is defined by the `objtype` property that CTP developers set in the `mapViews` and `updateObject` procedures. As a CTP developer, you can set the `objtype` property to a specific string for specific objects that your traversal algorithms can then use to more efficiently traverse the data hierarchy. For example, if a view is the most primitive object level in your data hierarchy, then an `objtype` property string that ends in 'View' might indicate that the algorithm need not traverse any deeper into the data hierarchy. Your traversal algorithms can take advantage of this knowledge and perform a more efficient traversal. Or if your search algorithm seeks a library file named '*.lib' but you find an object with an `objtype` property of `"EDAtool Cell"`, your algorithm need not search for a library file in the directory above the object. This type of algorithm is illustrated in the Library-View-Cell Example.

To learn more about setting object properties such as the `objtype` property, see CTP Object Properties.

CTP Namespaces

For your CTP procedures, you must create a Tcl namespace. For a CTP in a file named `<collection>.ctp` you would create procedures in the `<collection>CTP` Tcl namespace.

For example, for the `eda.ctp` CTP, you create the edaCTP namespace as follows:

```
  namespace eval edaCTP {}
```

DesignSync interprets `eda.ctp` as a CTP in the `edaCTP` namespace. To invoke the `mapViews` proc defined in `eda.ctp`, you indicate the `edaCTP` namespace as follows:

```
  edaCTP::mapViews
```

Required and Optional Procedures in CTPs

In order for DesignSync to recognize files and folders as the special objects and generic collections defined in a CTP, you define a number of required Tcl procedures in the `.ctp` file. These procedures instruct DesignSync to recognize particular files or folders as special objects or collection members. The procedures also indicate how to map the file-system view into a custom view by setting properties and grouping objects into collections. DesignSync invokes these custom CTP procedures to create and manage these special objects and collections. In this way, DesignSync is able to perform revision control operations on a collection object as a single entity rather than individual files.

The topic Custom Procedures in CTPs lists the key custom procedures needed in your CTPs. Some of these custom procedures are required procedures. If a required procedure is missing from the `.ctp` file, DesignSync fails to register the CTP during initialization. In this case, DesignSync issues an error message. If you continue to use DesignSync with an unregistered CTP, DesignSync fails to recognize your data as collection data. In this case, your data files might be interpreted as separate entities to be checked in rather than a collection of data. See Debugging Custom Type Packages for assistance in working through issues such as these with your CTP.

You can also develop optional procedures to be included in the CTP. DesignSync uses these procedures if they exist. In some cases, you include the optional procedures because they provide more efficient methods for DesignSync to handle your collection data. See Optional Custom Procedures in CTPs for a list and description of these procedures.

sctp Procedures

Within your custom required and optional procedures, you use the Custom Type System sctp procedures to manage CTP data. The sctp procedures let you manipulate collection objects. You can apply properties, query for properties, add objects to collections, and traverse design data hierarchies. See sctp Procedures Used in CTPs for descriptions of each of the sctp Procedures.

Utility Procedures for Recognizing Collection Data

As a CTP developer, you might want to create utility, or helper, procedures to be shared by your CTP procedures. For example, both the `mapViews` and `updateObject` procedures need utilities for recognizing collection data to determine which objects DesignSync is to handle as collection members. You can also develop shared utilities for updating the properties of objects or for traversing the data hierarchy.

Local Version Methodology

Some design tools implement their own basic version management by making local copies of design objects. A local copy of a design object is referred to as a 'local version', to distinguish it from the DesignSync version, which is created in the DesignSync vault upon checkin (a vault version).

Where DesignSync manages such data either through a predefined recognition package or through a developer's Custom Type Package (CTP), DesignSync incorporates the local version number into a tag name it applies upon checkin of the object.

If you intend to support your own local version methodology, the following topics will help you manage your CTPs:

Topic	**Overview**	**For More Information**
DesignSync `localversion` commands	These commands let you save and restore local versions, list the saved local versions, and delete local versions.	localversion delete localversion list localversion restore

		localversion save
Custom CTP local version procedures	You create your own local version management routines the CTS then invokes upon collection objects that use local versions.	getCurrentLocalVersion Procedure getLocalVersionFromTags Procedure localVersionChanged Procedure obsmembers Procedure
Local version properties	You can set the `obsmembers` property and create your own custom properties if necessary to help you manage local versions.	CTP Object Properties
Local Version Case Study Example	This case study implements a local version methodology.	Local Version Example

Ensuring Windows Compatibility

Because Windows and UNIX path formats differ, in some cases you must take steps to ensure that the paths are built correctly for each operating system:

Build paths that are appropriate for both UNIX an Windows platforms.

Collection objects have properties whose values are set in custom Tcl procedures. For the `owner` and `members` property, you must build the paths in such a way that they will be compatible for both UNIX and Windows platforms. In the case where members are in a folder below the collection object itself, your procedures must build the `members` path in a platform-dependant fashion. This means that you must use "/" on UNIX and "\" on Windows. Your procedures can use the DesignSync `url path` command to convert a path to the correct form and the Tcl `file separator` command to find the appropriate separator for the current platform. However, do not use the Tcl `file join` command to build these values, as that function always uses the UNIX "/" character to join the path elements. See CTP Object Properties for more information about properties.

Use the provided `sctp::glob` procedure rather than the Tcl `glob` command.

For pattern matching, the CTS provides the `sctp::glob` procedure which returns appropriate results for both the UNIX and Windows operating systems. For Windows operating systems, the `sctp::glob` procedure returns a list of filenames containing backslash '\' characters rather than forward slash '/' characters. See the Case Study Examples to see how you can use the `sctp::glob` procedure in CTPs.

CTP Object Type Properties

Your CTP objects obtain properties through:

- The Object Type Catalog: Your CTP objects inherit the properties of their object types as defined in the Object Type Catalog.
- Individual Object Properties: You can also set custom properties on individual objects. To set properties on individual objects, see CTP Object Properties.

Object Type Catalog

Each object type handled by a CTP has an associated **type catalog**. A type catalog defines the characteristics of the object type as properties.

The table below lists the properties you can assign to object types. The Custom Type System accesses these properties during revision control operations. CTP developers can also create custom properties to be stored in the object type catalog. These custom properties are not used by the Custom Type System. Instead, the CTP developer accesses them in custom CTP procedures. To learn how to set properties and values for the object type catalog, see the `sctp::setTypeProps` Procedure description.

Object Type Catalog Properties

Each object type can have the following properties:

Name	**Applies to**	**Purpose**
icon	All objects	Specifies the icon that DesignSync displays in its List View for this object type. See Predefined CTP Icons for a list of default icons you can specify. You can specify the icon as the name of a `.gif` in a `.jar` file or you can specify the path to a `.gif` or `.jpg` file. Create icons as 16x20 pixel `.gif` or `.jpg` files. The DesignSync graphical interface also supports the use of large (32x32 pixel) icons. Indicate a larger-sized icon by specifying the filename as: `<Icon>.L.gif`. If you specify this `<Icon>.L.gif` filename and the file does not exist, DesignSync programmatically enlarges the existing `<Icon>.gif` file.

openIcon	Folders only	Specifies the icon indicating an open folder that DesignSync displays in its List View for this object type. See Predefined CTP Icons for a list of default icons you can specify. You can specify the icon as the name of a `.gif` in a `.jar` file or you can specify the path to a `.gif` or `.jpg` file. Create icons as 16x20 pixel `.gif` or `.jpg` files. The DesignSync graphical interface also supports the use of large (32x32 pixel) icons. Indicate a larger-sized icon by specifying the filename as: `<Icon>.L.gif`. If you specify this `<Icon>.L.gif` filename and the file does not exist, DesignSync programmatically enlarges the existing `<Icon>.gif` file.
versionable	Files only	Specifies whether files of this object type can be revision-controlled. Set to 0 if the object is not versionable. Set to any other value if the object is versionable. **Note**: Collection members should be revision-controlled only as part of their collection and not separately. The CTP developer must ensure that the object type for collection member files is non-versionable by either setting its `versionable` property to 0 or by defining its `objtype` property such that it ends in `" Member"` (note the space before the `M`).
recurse	Folders only	Specifies whether revision-control operations are to recurse into folders of this object type. Set to 0 if revision-control operations should not recurse into folders. Set to any other value if revision-control operations should recurse into folders.

		Notes: • The `recurse` property overrides the `recurse` procedure; DesignSync does not invoke the `recurse` procedure on folders whose `recurse` property is set to 0. • If enabled, the `recurse` property prevents DesignSync commands from recursing into a folder with one exception: If you invoke the `ls` command within a folder that has the `recurse` property enabled, the `ls` command does recurse, although it does not recurse into subfolders in which the `recurse` property is enabled.
opNotify	Folders only	Specifies whether the `contentsChanged` notification procedure is called during revision control operations. Set to 1 for operation notification.
cntxMenu	All objects	Specifies the context menu for objects to this type. See Custom Context Menus for details.

Custom Context Menus

Each object type can have its own context menu. Within the DesignSync List View, a user positions the cursor over an object and clicks the right-mouse button to bring up the object type's context menu.

The following table lists the attributes you can define for context menus.

Name	Purpose
name	Specifies the item name as it will appear in the context menu
icon	Specifies the name of the .gif file that should appear in the menu. See Predefined CTP Icons for a list of default icons you can specify.
description	Specifies the description of the entry, which displays in the status bar of the DesignSync graphical interface when a user

	highlights the entry in the context menu.
tclAction	Specifies the Tcl command to be executed if a user selects the context menu entry.
shellAction	Specifies the shell command to be executed if a user selects the context menu entry.

You set up the context menu by defining a Tcl list of menu entries using the `sctp::setTypeProps` procedure.

You define the context menu in a Tcl list with a sublist for each context menu entry.

The following example creates a context menu for objects of `objtype` "Test Member". The context menu has three entries: "Open", "Unix listing", and "DS listing". The "Open" item opens the member in the emacs editor. The "Unix listing" item executes a shell command, the Unix `ls` command. The "DS listing" item runs a Tcl command, in this case the DesignSync `ls` command.

```
sctp::setTypeProps "Test Member" {icon "CTP_member.gif"
  cntxMenu {
    {name "Open" icon "open.gif" description \
        "Open in an editor" shellAction \
        "/usr/local/bin/emacs"}
    {name "Unix listing" description "Unix ls command" \
        shellAction "/bin/ls"}
    {name "DS listing" description "DS ls command" \
        tclAction "ls"}
  }
}
```

Notes:

- The attributes (listed in the table above) are name-value pairs you include in your `cntxMenu` Tcl list.
- The `name` attribute is required.
- The `icon` and `description` attributes are optional.
- The `tclAction` and `shellAction` attributes are mutually exclusive; define exactly one of these attributes for each context menu entry.
- The first item in the context menu has a special behavior such that the action listed by the `shellAction` or `tclAction` attribute becomes the default action for that particular object type. DesignSync adopts this default action under two circumstances: if the user double-clicks on the object or if the user selects **File=>Open**. Typically, the first item in a context menu is an open action, as in the example above.
- If no context menu is specified, the default action is for DesignSync to open the default editor if the user double-clicks on the object or select **File=>Open**.

- To use the "File Open" icon for an entry, specify "`open.gif`" using the `icon` attribute.

Predefined CTP Icons

The following icons are provided in the jar file supplied with DesignSync. You can specify these icons using the `icon` property:

Icons
CTP_collection
CTP_file
CTP_folder
CTP_folder_open
CTP_library
CTP_member
CTP_unmanaged
GEN_collection

CTP Object Properties

Your CTP objects obtain properties through:

- The Object Type Catalog: Your CTP objects inherit the properties of their object types as defined in the Object Type Catalog. See CTP Object Type Properties to learn more about the Object Type Catalog.
- Individual Object Properties: You can also set custom properties on individual objects. Your CTP procedures can access these properties in triggers for custom behavior.

The table below lists the properties you can assign to objects. The Custom Type System accesses these properties during revision control operations. CTP developers can also create custom properties for objects. These other properties are not used by the Custom Type System. Instead, the CTP developer accesses them in custom CTP procedures. To learn how to set properties and values on objects, see the `sctp::obj::setProp` and `sctp::obj::setProps` procedure descriptions. See also the Object Info Procedures topic for a description of other procedures you can use to manage properties.

Name	**Purpose**
objtype	Specifies the custom object type defined for an object. See Developing Custom Type Packages: Types Versus Object Types to understand the difference between types and objects types. **Note**: Collection members should be revision-controlled only as part of their collection and not separately. The CTP developer must ensure that the object type for collection member files is non-versionable by either defining its `objtype` property such that it ends in `" Member"` (note the space before the `M`) or by setting its `versionable` property to 0.

ciTag	Specifies a list of tags to be attached to a version of a collection when it is checked in. Following are examples of `ciTag` strings:
	
	`"tag1 tag2 tag3"`
	
	`[list tag1 tag2 tag3]`
	
	Note: The `ciTag` property is required if you are implementing a local version methodology. See Developing Custom Type Packages: Local Version Methodology for details.

owner	Specifies the collection object to which the object belongs. The `owner` property is required for collection member objects. Specify

	a full path or URL to the collection object. **Note**: In the case where members are in a folder below the collection object itself, your procedures must build the `owner` path in a platform-dependant fashion. This means that you must use "/" on UNIX and "\" on Windows. Your procedures can use the DesignSync `url path` command to convert a path to the correct form and the Tcl `file separator` command to find the appropriate separator for the current platform. However, do not use the Tcl `file join` command to build these values, as that function always uses the UNIX "/" character to join the path elements.
error	Indicates that a collection is invalid. If the `error` property exists, it prevents a collection or file from being checked in. The `error` property also marks files that the CTS recognizes as potential collection members, but an environment problem prevents the CTS from determining definitively that the object is a collection member. See DesignSync Recognition of Custom Type Packages: Exception Handling in View Mapping Procedures to learn more about how the CTS uses the `error` property.
members	Specifies a list of objects that are members of a collection. You can set the `members` property in the `mapViews` procedure if the members of the collection are a fixed set of objects your `mapViews` procedure can determine during view mapping. By doing so, you can omit the `members` procedure from your CTP. **Note**: In the case where members are in a folder below the collection object itself, your procedures must build the `members` path in a platform-dependant fashion. This means that you must use "/" on UNIX and "\" on Windows. Your procedures can use the DesignSync `url path` command to convert a path to the correct form and the Tcl `file separator` command to find the appropriate separator for the current platform. However, do not use the Tcl `file join` command to build these values, as that function always uses the UNIX "/" character to join the path elements.
obsmembers	Specifies a list of objects that are obsolete collection members. These objects are members of prior local versions of a collection; DesignSync's revision control operations handle only the current local version of a collection. The `obsmembers` list has the following format: `{1 {fileA:1 fileB:1}} {2 {fileA:2 fileB:2}}`

	where each sublist has two members: the local version number followed by a list of the files that comprise that local version. **Notes**: • The files must be provided as full paths or as paths relative to the directory containing the collection. • The current local version must not be included in the `obsmembers` list. Use this property if your CTP uses a local version methodology. Rather than assigning the `obsmembers` property, you can develop an `obsmembers` procedure. See also the `localversion` commands and the Local Version Example.
label	Specifies a subtype name used in reports about an object. For example, a collection of type Cadence View might have a label of type "Cadence Symbol View".
icon	Specifies an object-specific icon, which overrides the object type icon described in CTP Object Type Properties. See Predefined CTP Icons for a list of default icons you can specify. This `icon` property is typically used with the `label` property.
namespace	Specifies the CTP that customized the object. The `namespace` property is automatically attached to an object whenever a property is added or when a collection is created.

Debugging Custom Type Packages

Debugging Custom Type Packages

DesignSync helps you find the errors in your CTPs before you use these procedures on your production data. During development of your CTPs, you run the `ctp verify` command to test your CTP procedures. This command tests for inconsistencies among the CTP's procedures.

DesignSync also safeguards your collection design data by preventing check-in operations of objects in folders where the collection mapping by a CTP has failed. See DesignSync Recognition of Custom Type Packages: Exception Handling in View Mapping Procedures for details.

Using the 'ctp' Commands

DesignSync provides `ctp` commands to help you manage your CTPs. To check to see which CTPs are installed, use the `ctp list` command. To verify and debug your CTPs, use the `ctp verify` command. The `ctp verify` command validates all of the installed CPTs using either the data in the directory from which you invoke the command or against the data in a specified path. The command lists:

- The installed and active CTPs
- The folder and subfolders being validated
- A status of the collection members in the folder and its subfolder

  The command lists the objects that are not members of any of the installed CTPs. It also lists the collections that have no members. These occurrences might flag an error in a CTP. See Invalid CTPs for a list of other ways in which `ctp verify` checks for invalid CTPs.

Invalid CTPs

There are several ways in which a CTP can be defined such that it is not valid. For performance reasons, the system does not check for all of these errors during regular revision control operations. Instead, as the CTP developer, you validate your CTP before employing the CTP on production data by running the DesignSync `ctp verify` command against test data.

The `ctp verify` command checks that the CTP is internally consistent. For a CTP to be internally consistent, all type and property information is the same regardless of the path or command used to obtain that information. For example, if the `ls` command returns a CTP object's type as "Library File" but the `url getprop` command returns

"Special File", the CTP is not internally consistent. Among the ways a CTP can be internally inconsistent are:

- The `mapViews` and `updateObject` procedures perform view mapping of objects differently, thus generating inconsistent results for the same data.
- The `mapViews` and `determineFolderType` procedures return different values for a given folder.
- The `mapViews` procedure identifies an object as a member, but it is not returned by any collection's members procedure.
- The `mapViews` procedure fails to identify an object as a member when it is returned by a collection's members procedure.
- A collection member has an owner property identifying a collection, but that collection does not identify it as a member.
- More than one collection identifies a file as a member.
- The `members` custom property and the `members` procedure are inconsistent. In some cases, such as a broken symbolic, cache, or mirror link, an object might not be detected as a collection object and therefore the CTS does not set its custom `members` property. In this case, the `members` procedure would be called when the CTS encounters the object.

To learn more about the `ctp verify` command, see ENOVIA Synchronicity Command Reference: ctp verify Command.

Tips for Validating Your CTP

The following tips will help you use the `ctp verify` command and other methods to discover and resolve issues with your CTPs.

Verify one CTP at a time.

The `ctp verify` command validates all of the installed CTPs against the current data. You might want to move the other CTPs out of the custom ctp area of your installation hierarchy (`<SYNC_PROJECT_CFGDIR>/ctp` for project-level CPTs or `<SYNC_SITE_CUSTOM>/share/client/ctp` for site-level CTPs). If you move the other CTPs out of the custom ctp area, the `ctp verify` command operates only upon the single CTP, thus simplifying the diagnostic output of the `ctp verify` command.

Comment out the updateObject procedure until the CTP is verified.

One of the checks the `ctp verify` command makes is to verify that the `mapViews` and `updateObjects` procedures behave consistently. It might be easier to first verify that `mapViews` works correctly and then add the `updateObject` procedure back in and reverify the CTP using `ctp verify` again.

Execute individual CTP procedures from the command line.

A good method of isolating the behavior within a CTP is to execute a single CTP procedure from the command line. For a CTP named `myCTP`, you might enter the `myCTP::getCurrentLocalVersion` procedure on the command line. You can apply the procedure to particular objects to ensure that the CTP is handling local versions as expected for the workspace data.

Important: To use the CTP commands from the command line, you must be in an stcl/stclc shell and not in a dss/dssc shell.

The following example shows the invocation of a `members` custom procedure from the stcl command line:

```
stcl> lvcCTP::members ../schematic NAND.sgc.lvc "LVC Cell View"
    NAND/sch.db NAND/sch.prop NAND/lvc.celltype.schematic
```

Use DesignSync commands to verify output.

In addition to applying `ctp verify` to your CTPs, use the following DesignSync commands:

- `ls -report OX`: List objects and their collection owners
- `url members`: List a collection's members.
- `url exists`: Determine if collections and collection members exist
- `ci`, `co`, `populate`: Apply revision control operations and ensure that diagnostic messages contain no errors.

These commands will help ensure that your CTP is managing your data as intended.

Develop robust test data.

Your test data should exercise your CTP effectively to ensure that `ctp verify`, as well as your custom CTPs procedures, run cleanly on a complete set of test data. In addition to test data that exercises your unique CTP, your test data should handle the following types of general test cases:

- Valid collections
- CTP procedures that contain errors that should be flagged
- Collections with one or more members missing
- A collection that is a file on disk
- Collections or custom objects that are expected to be files but are folders, as well as objects that are expected to be folders but are files
- Empty folders, or workspaces that contain empty folders in the sub-hierarchy
- Collections with managed symbolic links
- Collections in cache and mirror mode
- Collections with missing cache, mirror, or symbolic links
- Collections in reference mode

- Data should include examples of all valid filenames, for example filenames with spaces, extra dots, or other special characters.

See DesignSync Data Manager User's Guide to learn how DesignSync manages symbolic links, cache links, mirror links, and references.

Use full paths for the `owner` property and to check whether files exist.

If a procedure is failing, you might have neglected to specify a full path where one is needed. For example, the owner property requires an absolute path. You must also use a full path when using the `file exists` Tcl command. For the `members` and `obsmembers` procedures, the directory must be a full or relative path.

Custom Procedures in CTPs

Custom Procedures in CTPs

As CTP developer, you create a number of Tcl procedures that DesignSync's Custom Type System (CTS) calls during revision control operations to recognize and manage collection data. You create these Tcl procedures in a Tcl namespace as described in Developing Custom Type Packages.

Note:

The procedures described in this section, as well as those described in the Optional Custom Procedures in CTPs section, are the procedures you, as the CTP developer, implement. These procedures are then called by the CTS (DesignSync) whenever appropriate.

Following are the key procedures you should include in your CTP:

Procedure	**Description**
mapViews	Maps a system view of a directory into a custom view with collections, applying appropriate custom properties to the objects.
updateObject	Performs view mapping like the `mapViews` procedure, but for individual objects. Applies any appropriate custom properties to the object.
members	Returns the members of a collection.

You can find examples of custom CTP procedures in the Case Study Examples.

Required Procedures for your CTP

In order for your CTP to be registered by the Custom Type System, you must include the `mapViews` and `members` procedures; however, you can omit the `members` procedure if either of the following are true:

- The members of a collection are fixed and known during view mapping so that your `mapViews` procedure assigns the `members` custom property for each collection.
- The CTP defines custom types for files but does not define any collections. For example a CTP might define a custom type, icon, and a context menu for `"*.cpp"` and `"*.html"` files.

You do not need to include the `updateObject` procedure for your CTP to be registered; however, you should include the `updateObject` procedure in your production CTP. If the `updateObject` procedure does not exist and the CTS needs to determine the nature of an object, it calls the `mapViews` procedure from the parent directory and extracts information about the object. Using the `mapViews` procedure in this way is less efficient because `mapViews` handles a complete directory, so most CTPs include an `updateObject` procedure. See the `updateObject` Procedure for more information.

mapViews Procedure

```
proc mapViews {dir contents}
  => return value unused
```

Description

The `mapViews` procedure is a required procedure that you must include in your CTP for the CTP collections to be recognized. Your `mapViews` procedure performs the view mapping of the CTP. View mapping is the process of mapping the system view of the collection data onto a custom view. To create this custom view, your `mapViews` procedure should

- Update the properties of objects as appropriate, using the `sctp::obj::setprop` or `sctp::obj::setprops` procedures.
- Add new objects, such as collections. To add collections, use the `sctp::objset::addobject` procedure.

For example, the `mapViews` procedure might scan a directory (folder), performing a pattern matching algorithm to detect potential members of a collection. If found, the `mapViews` procedure updates those objects with properties and creates the collection object for those objects. See DesignSync Recognition of Custom Type Packages for more information about view mapping.

The `mapViews` procedure is called by DesignSync whenever DesignSync traverses a directory and handles objects in the directory by:

- Browsing in the DesignSync graphical interface using the List View.
- Using the `ls` command in DesignSync.
- Executing any revision control operation, such as `ci`, `co`, `populate`, and `tag`.

Each time DesignSync traverses a directory, the CTS invokes the `mapViews` procedures for every installed CTP. If one of your CTPs has already assigned a custom folder type to a directory, DesignSync invokes only the `mapViews` procedure specified in that CTP. This is one way to ensure the efficiency of your custom code.

Within your `mapViews` procedure, you set properties on the objects deemed to be collection members, as well as the newly created collection objects, using the `sctp::obj::setprop` or `sctp::obj::setprops` procedures. You can also apply properties to directories. Examples of properties include:

- `owner`: The name of the collection containing the member object.
- `error`: The property applied to invalid collection objects.

- `members`: A list containing the members of the collection. Set this property if the members are a fixed set of objects your `mapViews` procedure can determine during view mapping.

See CTP Object Properties for the full set of properties you can apply to the objects.

The `mapViews` procedure performs the process of determining whether objects are collection members and updating their properties for all of the objects in a directory. To perform this same process on a single object, the CTS calls the `updateObject` procedure. If you do not develop an `updateObject` procedure, the CTS calls `mapViews` when it needs to recognize a single object -- a more inefficient method of obtaining the same information. An effective method of coding these two procedures is to develop a number of helper procedures that perform the mapping tasks. Then both the `mapViews` and `updateObject` procedures call these same routines. In this way, you are ensured consistent results from both procedures. Consistency between these two procedures is also one of the criteria used by the `ctp verify` command to validate CTPs.

You can find examples of custom `mapViews` procedures in the Case Study Examples.

Arguments

`dir`	Specifies the path (not the URL) of the directory to be mapped.
`contents`	Stores the contents of the directory. The `contents` argument is an Object Set object. See Object Set Procedures for descriptions of the procedures you use to manage Object Set objects. The `contents` argument is an in-out parameter. The procedure must modify it in place.

updateObject Procedure

```
proc updateObject {dir objInfo}
  => return value unused
```

Description

The `updateObject` procedure is an optional procedure you can develop to recognize individual objects. The CTS performs two types of view mapping:

- View mapping of a complete directory, performed by the `mapViews` procedure to recognize the objects in a directory.
- View mapping of a single object, performed by the `updateObject` procedure to recognize individual objects.

Note: Although the `updateObject` procedure is optional, you should include the `updateObject` procedure in order to optimize DesignSync's operations. If the `updateObject` procedure does not exist and the CTS needs to determine the nature of an object, it calls the `mapViews` procedure from the parent directory and extracts information about the object. Using the `mapViews` procedure in this way is less efficient because `mapViews` handles a complete directory, so most CTPs include an `updateObject` procedure. You might want to develop shared helper procedures called by both the `updateObject` and `mapViews` procedures because the functions of these procedures are similar. In this way, you are also ensured consistent results from both procedures. Consistency between these two procedures is also one of the criteria used by the `ctp verify` command to validate CTPs.

Your `updateObject` procedure should perform the following tasks:

- Determine the type of an object -- for example, whether the object is a collection member or a collection object.
- If the object is a collection object, determine whether the collection exists yet on disk and if so, use the `sctp::obj::collectionexists` procedure to mark that it exists. In this way, the CTS can distinguish between physical collections, references (which only exist as metadata), and objects that do not exist.
- Apply any appropriate custom properties to the object. See CTP Object Properties for the properties you can apply to the objects.

You can find examples of custom `updateObject` procedures in the Case Study Examples.

Arguments

`dir`	Specifies the path (not the URL) of the directory containing the object.
`objInfo`	The object to be recognized and updated. The `objInfo` object is an Object Info object. See Object Info Procedures for descriptions of the procedures you use to manage Object Set objects.

members Procedure

```
proc members {parentPath objName type}
  => list-of-relative-paths
```

Description

The `members` procedure determines and lists the members of a collection. The `members` procedure is a required procedure that you must include in your CTP unless the members of a collection are fixed and known during view mapping and thus your `mapViews` procedure assigns the `members` custom property for each collection. If you set the `members` custom property for each collection, you do not have to supply a `members` procedure. Another case where you can omit the `members` procedure is in that of a CTP that defines custom types for files but does not define any collections. For example a CTP might define a custom type, icon, and a context menu for "*.cpp" and "*.html" files.

Your `members` procedure should perform the following tasks:

- Recurse into sub-directories to obtain the collection members, if necessary.
- Return a list of paths to the collection's members. Your `members` procedure can return paths that are either full paths or relative to the specified path. For example, a path of `verilog/master.tag` indicates a member in the `verilog` subdirectory of the directory containing the collection.

See the Case Study Examples for examples of custom `members` procedures.

See the Collection Example for an example of a collection that sets the `members` property if the collection has fixed members and in these cases, does not have to call the `members` procedure.

Arguments

`parentPath`	Specifies the directory containing the collection object whose members are to be listed.
`objName`	Specifies the name of the collection object whose members are to be listed.
`type`	Specifies the object type of the specified object. The `type` is provided to the `members` procedure in case your algorithm for determining the collection's members needs this information.

Optional Custom Procedures in CTPs

Optional Custom Procedures in CTPs

As CTP developer, you create a number of Tcl procedures that DesignSync's Custom Type System (CTS) calls during revision control operations to recognize and manage collection data. You create these Tcl procedures in a Tcl namespace as described in Developing Custom Type Packages.

Note:

The procedures described in this section, as well as those described in the Custom Procedures in CTPs section, are the procedures you, as the CTP developer, implement. These procedures are then called by the CTS (DesignSync) whenever appropriate.

Following are optional procedures you might find useful to develop in your CTP:

Procedure	**Description**
contentsChanged	Lets you perform operations after revision control operations by detecting added, removed, or changed objects.
determineFolderType	Determines the custom folder type. Used for CTPs that define custom directory types.
getCurrentLocalVersion	Returns the local version currently checked out in a user's workspace.
getLocalVersionFromTags	Returns the local version by processing a list of tagnames.
localVersionChanged	Lets you perform operations each time a collection's local version changes.
obsmembers	Returns the members of a collection that belong to prior local versions of the collection.
operationBegin	Sends notice to the CTP whenever DesignSync operations begin. Used with the `contentsChanged` procedure to batch operations across directories.
operationEnd	Sends notice to the CTP whenever DesignSync operations end. Used with the `contentsChanged` procedure to batch operations across directories.
processKeyFiles	Uses a set of key files to recognize objects that have not been view mapped. Used in the case of an initial populate. You set the key files used to identify matching collection objects using the `sctp::setKeyFiles` procedure.
recurse	Determines whether the CTS should recurse into the specified directory (folder).

relations	Defines relationships for objects. Used by the DesignSync `url relations` command to return a list of related objects, such as dependencies, for a CTP's collection type.

You can find examples of custom CTP procedures in the Case Study Examples.

contentsChanged Procedure

```
proc contentsChanged  \
{path list-of-added-objects list-of-removed-objects}
  => return value unused
```

Description

The `contentsChanged` procedure is an optional procedure you can include in your CTP if you want to perform post-processing of objects affected by a revision control operation. The `contentsChanged` procedure supplies you with the list of objects added, changed, and removed from the affected directory. Within the `contentsChanged` procedure, you can perform any appropriate custom operations.

Use the `contentsChanged` procedure with the `operationBegin` and `operationEnd` procedures so that you can batch operations across objects. Those objects may be in different directories. You can check which operation has triggered the `operationBegin` procedure so that you can apply the `contentsChanged` procedure only to objects that have been handled by a particular command such as `co` or `ci`. See the Library-View-Cell Example for an example of these procedures working together in this way.

To ensure efficient performance, the CTS applies the `contentsChanged` procedure only for directories whose Object Type Catalog has the `opNotify` property set. See CTP Object Type Properties: Object Type Catalog for details about the `opNotify` property, which sets up notification so that the CTS calls the `contentsChanged` procedure if a revision control operation modifies the objects in a directory.

Arguments

`path`	The path to the directory whose contents were changed by the operation. For example, fetching a different version of the data may change the data files in the directory.
`list-of-added-objects`	The list of objects within the directory that have been added or changed by the operation.
`list-of-removed-objects`	The list of objects within the directory that have been removed by the operation.

determineFolderType Procedure

```
proc determineFolderType {dir}
  => dirType
```

Description

The `determineFolderType` procedure is an optional procedure you can develop to help the CTP procedures navigate within the design data hierarchy. Some collections can occur within any directory (folder) in the design data hierarchy. See the Collection Example for an example of this kind of collection. Other collections, however, can only occur in a particular hierarchy context. For example, an EDA tool might have a collection type that can only exist in a directory named `cellview`. In this case, the CTP's `determineFolderType` procedure might indicate whether the directory is a cellview directory. See the Library-View-Cell Example for an example of a collection that exists in a particular hierarchy context and thus implements the `determineFolderType` procedure. The types of CTPs that require a `determineFolderType` procedure also define custom property types on directories within the Object Type Catalog. See CTP Object Type Properties: Object Type Catalog for more information about these property types.

Unlike the `mapViews` procedure, which is passed a list of the contents of a directory, the `determineFolderType` procedure must access the specified directory on the file system to determine its object type.

Your `determineFolderType` procedure must:

- Process the directory and its contents to determine what folder type it should have.

  Utilize criteria such as the following to determine the folder type: Do all of your collection folders contain files with a particular extension? Do collection folders have particular naming conventions? Do collection folders contain library files with particular naming conventions?

- Return the directory type.

  Note: The `determineFolderType` procedure must return the same property type for the directory as that returned by the `mapViews` procedure. The `ctp verify` command performs this check.

Note: If the `determineFolderType` procedure needs to find the type of the parent folder or a sub-folder, it should call the sctp::getFolderType procedure rather than calling itself, to take advantage of the directory type caching built into the CTS system.

Arguments

`dir`	The path (not the URL) to the directory whose folder type is to be determined.

getCurrentLocalVersion Procedure

```
proc getCurrentLocalVersion {url type {usetag 1} {members ""}}
  => an integer identifier
```

Description

If your CTP implements local versions, it must include the `getCurrentLocalVersion` procedure. DesignSync uses this procedure, like the `getLocalVersionFromTags` procedure, to determine the local version during checkins and checkouts in order to manage the local versions in users' workspaces and in the DesignSync vaults. For example, if a user has applied the `co` command with the `-savelocal fail` option (the default value for this option), DesignSync must verify that the user does not have a higher local version for the collection than that in the vault, otherwise the checkout fails. See Developing Custom Type Packages: Local Version Methodology for an overview of local versions. See also the `localversion`, `co -savelocal`, and `populate -savelocal` command descriptions for details about local versions.

The CTS calls the `getCurrentLocalVersion` procedure to determine the local version currently checked out in the workspace. If a mirror is being used, the CTS calls the `getCurrentLocalVersion` procedure with the `members` argument. Before determining the local version, your `getCurrentLocalVersion` procedure must first determine which objects to operate on by doing the following:

- Check if the `usetag` argument is set and if so, use a tag to determine the current local version.

  The name of the tag corresponds to the tag specifier of the collection. Set the tag specifier using the `ciTag` property during view mapping (in the `mapViews` and `updateObject` procedures). If the `ciTag` property is set, DesignSync applies the specified tag to the collection automatically during check-in, thus identifying the collection's local version. Following is an example of a statement from a `mapViews` procedure that sets the `ciTag` property:

  ```
  sctp::obj::setprop $colobj ciTag [join [list \
  "CTP_LV" $vernum ] "" ]
  ```

  In this example, the `ciTag` property is generated by joining the `"CTP_LV"` string with the local version number (`$vernum`).

  Your `getCurrentLocalVersion` procedure must look up the tags on the specified object, using the `url tags` command, and match against the `"CTP_LV"` string to extract the local version number.

- If the `usetag` argument is not set, check if a `members` argument was passed into the `getCurrentLocalVersion` procedure and if so, determine the local version based on the contents of the `members` argument.
- If the `members` argument is empty, determine the local version based on the specified `url` argument (the URL of the collection object whose local version you must determine). You determine the local version typically by examining the files on disk that make up that object.
- Return the local version number as an integer identifier for the local version.

Once your `getCurrentLocalVersion` procedure has determined which objects to operate on, it determines the current local version. Typically the current local version is the local version with the highest number appended, but your `getCurrentLocalVersion` procedure can determine the current local version using whatever method is required by your local version methodology.

Arguments

`url`	Specifies the URL of the collection object whose current local version is being determined.
`type`	Specifies the object type of the specified object. The `type` is provided to the `getCurrentLocalVersion` procedure in case your algorithm for determining the current local version needs this information.
`usetag`	If set to 1, the `getCurrentLocalVersion` procedure uses a tag to determine the local version; otherwise, if the `usetag` argument is not supplied or is set to 0, the `getCurrentLocalVersion` procedure determines the local version by processing the `members` argument or the contents of the user's workspace. Your `getCurrentLocalVersion` procedure (or a helper procedure called by the `getCurrentLocalVersion` procedure) defines the tag name against which to match the objects. The tag name corresponds to the `ciTag` property set by the `mapViews` and `updateObject` procedures.
`members`	Specifies a list of members. **Note**: The `members` argument is passed into the `getCurrentLocalVersion` procedure only if the collection data is mirrored. (See DesignSync Data Manager User's Guide: Administering Mirrors to learn about mirroring data.) If a `members` argument is passed into the procedure, your `getCurrentLocalVersion` procedure must process the `members` argument instead of monitoring the contents of the workspace. However, if the members of a collection are fixed

and thus defined in the `members` property initially during view mapping (using the `mapViews` procedure), your `getCurrentLocalVersion` procedure does not have to process the `members` argument. After you develop your `getCurrentLocalVersion` procedure to process the `members` argument, test your procedure by invoking the `populate -mirror` command.

getLocalVersionFromTags Procedure

```
proc geLocalVersionFromTags {url type tags}
  => an integer identifier
```

Description

If your CTP implements local versions, it must include the `getLocalVersionFromTags` procedure. DesignSync uses this procedure, like the `getCurrentLocalVersion` procedure, to determine the local version during checkouts and populates in order to manage the local versions in users' workspaces and in the DesignSync vaults.

The CTS calls the `getLocalVersionFromTags` procedure if a user has applied the `co` or `populate` command with the `-savelocal fail` option (the default value for this option). The CTS passes to the `getLocalVersionFromTags` procedure the list of all tags attached to the version being checked out or populated. In this way, DesignSync can verify that the user does not have a higher local version for the collection than that in the vault, otherwise the checkout fails. Your `getLocalVersionFromTags` procedure must

- Determine which of the tags passed in with the `tags` argument identifies the local version.
- Find the local version number from that tag.

  The name of the local version's tag corresponds to the tag specifier for the collection. Note that you set the tag specifier in the `mapViews` and `updateObject` procedures using the `ciTag` property during view mapping . If the `ciTag` property is set, DesignSync applies the specified tag to the collection automatically during check-in, thus identifying the collection's local version. Following is an example of a statement from a `mapViews` procedure that sets the `ciTag` property:

  ```
  sctp::obj::setprop $colobj ciTag [join [list \
  "CTP_LV" $vernum ] "" ]
  ```

  In this example, the `ciTag` property is generated by joining the `"CTP_LV"` string with the local version number (`$vernum`). Your `getLocalVersionFromTags` procedure must look at the `tags` argument value and match against the `"CTP_LV"` string to extract the local version number.

- Return the local version number as an integer identifier for the local version.

See Developing Custom Type Packages: Local Version Methodology for an overview of local versions. See the Local Version Example for an example of the `getLocalVersionFromTags` procedure. See also the `localversion` command descriptions in the ENOVIA Synchronicity Command Reference.

Arguments

`url`	Specifies the URL of the collection object whose local version is being determined.
`type`	Specifies the object type of the specified object. The `type` is provided to the `getLocalVersionFromTags` procedure in case your algorithm for determining the local version needs this information.
`tags`	Specifies the list of tagnames from which the `getLocalVersionFromTags` procedure determines the local version.

localVersionChanged Procedure

```
proc localVersionChanged {url type oldLv newLv members obsInfo}
  => return value unused
```

Description

The `localVersionChanged` procedure is an optional procedure you can include in your CTP if you want to perform operations each time a collection's local version changes. Within the `localVersionChanged` procedure, you can perform any appropriate custom operations on objects such as the new collection members or the obsolete collection members, as these values are passed to the `localVersionChanged` procedure.

See Developing Custom Type Packages: Local Version Methodology for an overview of local versions. See the Local Version Example for an example of the `localVersionChanged` procedure. See also the `localversion` command descriptions in the ENOVIA Synchronicity Command Reference.

Arguments

`url`	Specifies the URL of the collection object whose local version has changed.
`type`	Specifies the object type of the specified object.
`oldLv`	Specifies the previous local version number.
`newLv`	Specifies the new local version number.
`members`	Lists the members that have been added to the collection.

`obsInfo`	Lists the obsolete local versions in the following format:
	
	`{1 {fileA:1 fileB:1}} {2 fileA:2 fileB:2}}`
	`...`
	
	where each sublist has two members: the local version number and the list of files making up that obsolete local version.

operationBegin Procedure

```
proc operationBegin {cmd}
  => return value unused
```

Description

The `operationBegin` procedure works with the `operationEnd` procedure to allow the CTP to receive notice whenever DesignSync operations occur. Use the `operationBegin` and `operationEnd` procedures with the `contentsChanged` procedure so that you can batch operations across objects. Those objects may be in different directories. You can check which operation has triggered the `operationBegin` procedure so that you can apply the `contentsChanged` procedure only to objects that have been handled by a particular command such as `co` or `ci`. Then, you can use the `operationEnd` procedure to operate on the results of the `contentsChanged` procedure upon completion of the operation. See the Library-View-Cell Example for an example of these procedures working together in this way.

Note: The command names passed to `operationBegin` are not necessarily formal DesignSync command names. In some cases, DesignSync passes internal command names; for example, DesignSync passes the `populate` command as `_populate` (note the underscore preceding the `populate` command name). If your code utilizes the `cmdName` input, you might initially want to write a test `operationBegin` procedure that prints the `cmdName` arguments, so that you know the exact command strings being passed into `operationBegin`.

Arguments

`cmd`	The command that triggered the `operationBegin` procedure. DesignSync passes the full command including its arguments to the `operationBegin` procedure so that you can test for particular DesignSync commands and their argument values.

operationEnd Procedure

```
proc operationEnd {}
  => return value unused
```

Description

The `operationEnd` procedure works with the `operationBegin` procedure to allow the CTP to receive notice whenever DesignSync operations occur. Use the `operationBegin` and `operationEnd` procedures with the `contentsChanged` procedure so that you can batch operations across objects. Those objects may be in different directories. You can check which operation has triggered the `operationBegin` procedure so that you can apply the `contentsChanged` procedure only to objects that have been handled by a particular command such as `co` or `ci`. Then, you can use the `operationEnd` procedure to operate on the results of the `contentsChanged` procedure upon completion of the operation. See the Library-View-Cell Example for an example of these procedures working together in this way.

Arguments

None

processKeyFiles Procedure

```
proc processKeyFiles {path list-of-key-files}
  => CTP name (Tcl namespace)
```

Description

The `processKeyFiles` procedure works with the `operationBegin`, `operationEnd`, and `contentsChanged` procedures to assist with the recognition of objects that have been recently fetched and therefore have not undergone view mapping. The `processKeyFiles` procedure thus performs a bootstrapping operation; the CTS calls the `processKeyFiles` procedure when a `populate` command operates on a new folder, thus flagging the collection objects. You set the key files used to identify matching collection objects using the `sctp::setKeyFiles` procedure. See the Library-View-Cell Example for an example of the `processKeyFiles` procedure.

If the `processKeyFiles` procedure returns a Tcl namespace, the CTS invokes the `contentsChanged` procedure for that CTP, with the containing folder and the key file as a member of the list of added files.

Arguments

`path`	The path to the directory whose contents are to be processed.
`list-of-key-files`	List of the key files found in the path.

recurse Procedure

```
proc recurse {path type}
  => 1 or 0
```

Description

The `recurse` procedure is an optional procedure you can develop to determine whether the CTS should recurse into the specified directory (folder). Upon encountering a directory (folder), a recursive operation invokes the `recurse` procedure only if the following are true:

- The directory (folder) has a custom object type defined by the CTP (using the `sctp::setTypeProps` procedure)
- The `recurse` property has **not** been set on the directory. If the `recurse` property has been set, the CTS uses that value to determine whether to recurse into the directory and thus does not invoke the `recurse` procedure. See CTP Object Type Properties for details of the `recurse` property.

The `recurse` procedure is optional; if the `recurse` procedure exists, the CTS passes it the path to the directory and the directory's object type. Your `recurse` procedure should return a 1 (or any non-zero value including an empty string) to indicate that the directory is to be recursed and a 0 to indicate that the directory must be skipped. If the `recurse` procedure is not included in the CTP, the directory is recursed by default, as if the `recurse` procedure had returned a 1.

For an example of a custom `recurse` procedure, see the Library-View-Cell Example.

Arguments

`path`	Specifies the directory to be checked for recursion.
`type`	Specifies the object type (`objtype`) of the directory.

relations Procedure

```
proc relations {objName type relationName}
  => list-of-urls
```

Description

You can develop a `relations` procedure if you want the DesignSync `url relations` command to return a list of related objects, such as dependencies, for a CTP's collection type. You create the relation by creating an optional `relations` procedure in the CTP. Then CTP users can apply the `url relations` command to the collection to view the results of the relation. If the `relations` procedure is not defined, the `url relations` command returns an empty list.

If the relations procedure throws an exception such as "Relation type not supported", the exception is propagated through the `url relations` command.

As the CTP developer, you can customize the format of the `list-of-urls` return value.

Arguments

`objName`	Specifies the object whose relatives are to be identified. The object can be a collection, a collection member, or a directory.
`type`	Specifies the object type of the specified object. The `type` is provided to the `relations` procedure in case your algorithm for determining the relatives needs this information.
`relationName`	Specifies the name of the relation query; this value can be any string.

Examples

The following example shows a custom relations procedure included in a CTP in the namespace `mycollectionCTP`:

```
proc mycollectionCTP::relations {objName type relName} {
    return "../doc.css"
}
```

The `relations` procedure is called below:

```
stcl> mycollectionCTP::relations SPEC.sgc.myc \
"Doc Files collection" dependencies
../doc.css
```

The DesignSync `url relations` command invokes the relations procedure:

```
stcl> url relations SPEC.sgc.myc  dependencies
../doc.css
```

sctp Procedures Used in CTPs

sctp Procedures Used in CTPs

The sctp procedures are the procedures you call from your custom procedures:

- Object Info Procedures: Use these procedures to determine and set properties of individual objects.
- Object Set Procedures: Use these procedures to determine and set properties on groups of objects.

Example of an sctp Procedure

You include the sctp procedures in your custom procedures. During CTP development, you can also apply these commands directly from the stcl shell. The following shows an example of the `sctp::obj::getprops` command invoked in the stcl shell.

In this example, `$NANDobj` is a variable containing the Object Info object for the `NAND.sgc.lvc` collection. The `getprops` procedure accepts the `$NANDobj` Object Info object as an input parameter and loads the `newvar` array with the property/value pairs for the collection:

```
stcl> set NANDobj [sctp::getObjectInFolder [pwd] NAND.sgc.lvc]
NAND.sgc.lvc Collection {branch 1 members {NAND/sch.db
NAND/sch.prop NAND/lvc.celltype.schematic} conflict 0
_SyncColStampDataVers 1 vault
{sync://puccini:30048/Projects/sep9lvc/schematic/NAND.sgc.lvc;}
objtype {LVC Cell View} _SyncColStampData {NAND/sch.db
1091092806 NAND/sch.prop 1091092806 NAND/lvc.celltype.schematic
1091092797} vctime 1094751543 mergedeps {} state Copy label {LVC
schematic Cell View} namespace lvcCTP selectors {} log {}
version 1.1 size 1635019107 mtime 15 datefetched 1091092806}

stcl> sctp::obj::getprops $NANDobj newvar

stcl> stcl> parray newvar
newvar(_SyncColStampData)     = NAND/sch.db 1091092806
NAND/sch.prop 1091092806 NAND/lvc.celltype.schematic 1091092797
newvar(_SyncColStampDataVers) = 1
newvar(branch)                = 1
newvar(conflict)              = 0
newvar(datefetched)           = 1091092806
newvar(label)                 = LVC schematic Cell View
newvar(log)                   =
newvar(members)               = NAND/sch.db NAND/sch.prop
```

```
                                NAND/lvc.celltype.schematic
newvar(mergedeps)              =
newvar(mtime)                  = 15
newvar(namespace)              = lvcCTP
newvar(objtype)                = LVC Cell View
newvar(selectors)              =
newvar(size)                   = 6172092
newvar(state)                  = Copy
newvar(type)                   = Collection
newvar(vault)                  =
sync://myvault:2647/Projects/sep9lvc/schematic/NAND.sgc.lvc;
newvar(vctime)                 = 1094751543
newvar(version)                = 1.1
```

You can find other examples of the sctp commands in the Case Study Examples.

Object Info Procedures

Object Info Procedures

An Object Info object represents information about a single CTP object. The Object Info object is a C object that is exposed via Tcl. You use the Object Info Tcl procedures to set or retrieve properties of objects, including their names and types. The Object Info procedures are in the `sctp::obj` namespace. To invoke, for example, the `getprop` proc, you indicate the namespace as follows:

```
  sctp::obj::getprop
```

You can use these procedures in any of the procedures you define in your `.ctp` file.

For examples of object info procedures, see sctp Procedures Used in CTPs and Case Study Examples.

sctp::obj::collectionexists Procedure

```
sctp::obj::collectionexists $obj
  => return value undefined
```

Description

The `collectionexists` procedure marks that a collection exists on disk. It is important that the `updateObject` procedure call this procedure to identify whether the collection exists yet on disk. In this way, the CTS can distinguish between physical collections, references (which only exist as metadata), and objects that do not exist.

Arguments

`$obj`	Specifies the collection object which exists on disk. **Note**: The `$obj` name must contain `.sync.` or the `collectionexists` procedure throws an error.

sctp::obj::getprop Procedure

```
sctp::obj::getprop $obj name
  => property
```

Description

The `getprop` procedure retrieves the value of the specified property.

Arguments

`$obj`	Specifies the object whose property is to be returned.
`name`	Specifies the name of the property whose value is to be returned.

sctp::obj::getprops Procedure

```
sctp::obj::getprops $obj var
  => return value undefined
```

Description

The `getprops` procedure loads an array (a var) with the object's properties (the object's `Object Info` data).

For an example that invokes `getprops` procedure, see sctp Procedures Used in CTPs.

Arguments

`$obj`	Specifies the object whose properties are to be returned
`var`	Specifies the array (var) to store the properties of the specified object.

sctp::obj::name Procedure

```
sctp::obj::name $obj
  => objname
```

Description

The `name` procedure returns the name of the object.

Arguments

`$obj`	Specifies the object whose name is to be returned.

sctp::obj::setprop Procedure

```
sctp::obj::setprop $obj name value
  => return value undefined
```

Description

The `setprop` procedure sets the specified property of the object.

Arguments

`$obj`	Specifies the object whose property is to be set.
`name`	Specifies the property to be set. See CTP Object Properties for the properties you can set.
`value`	Specifies the value to which the property is to be set.

sctp::obj::setprops Procedure

```
sctp::obj::setprops $obj {name value name value …}
  => return value undefined
```

Description

The `setprops` procedure sets the specified properties of the object. Specify the properties as a list of property name/value pairs.

Arguments

`$obj`	Specifies the object whose properties are to be set.
`name`	Specifies the property to be set. See CTP Object Properties for the properties you can set.
`value`	Specifies the value to which the property is to be set.

sctp::obj::type Procedure

```
sctp::obj::type $obj
  => type
```

Description

The `type` procedure returns the type of the object. The type returned is the web data type, such as File or Folder. The type procedure does not return the more granular object type which indicates, for example, a specific type of collection object. See Developing Custom Type Packages: Types Versus Object Types for more information.

Arguments

`$obj`	Specifies the object whose type is to be returned.

Object Set Procedures

Object Set Procedures

An object set is the entity that represents the contents of a directory. The object set is a C object that is exposed via Tcl. You use the object set procedures to retrieve the objects in a set or to create new objects. The object set procedures are in the `sctp::objset` namespace. To invoke, for example, the `addobject` procedure, you indicate the namespace as follows:

```
  sctp::objset::addobject
```

You can use these procedures in any of the procedures you define in your `.ctp` file.

For examples of object set procedures, see Case Study Examples.

sctp::objset::addobject Procedure

```
sctp::objset::addobject $set objname [{name value name value…}]
  => CTP object
```

Description

The `addobject` procedure adds an object to the specified set. Optionally, you can use the `addobject` procedure to set properties on the object as you add it to the set. The procedure returns the new CTP object (an Object Info object).

Arguments

`$set`	Specifies the set of objects to which the new CTP object is added. Typically custom procedures pass the contents of a directory into the `addobject` procedure using the `$set` argument.
`objname`	Specifies the name of the object to be added. **Note**: The object's name must contain `.sgc` **.** Otherwise the Custom Type System throws an exception.
`name`	Specifies the name of a property to be set. (Optional)
`value`	Specifies the value to be set for the corresponding property (the `name` argument). (Optional)

sctp::objset::contains Procedure

```
sctp::objset::contains $set objname [var]
=> 0/1
```

Description

The `contains` procedure detects whether a set contains the specified object. If you supply the `var` argument, the `contains` procedure sets the `var` variable to the CTP object (an Object Info object).

The `contains` procedure returns a 0 if the specified object is not in the set. It returns a 1 if the object is in the set.

Arguments

`$set`	Specifies the set of objects against which you want to check for a particular object.
`objname`	Specifies the name of the object to be added.
`var`	Specifies the variable that is set to the CTP object if the set contains the object. (Optional)

sctp::objset::foreachfile Procedure

```
sctp::objset::foreachfile $set var pattern {code}
=> return value undefined
```

Description

The `foreachfile` procedure iterates over all objects in the specified set matching the given pattern. For each iteration, if there's a match, the `foreachfile` procedure sets the `var` loop variable to the CTP object (an Object Info object) and executes the specified Tcl code.

Arguments

`$set`	Specifies the set of objects against which you want to match a specified pattern.
`var`	Specifies the loop variable. The `var` argument is set, in turn, to each CTP object that matches the specified pattern.
`pattern`	Specifies a glob-style pattern to be matched against objects in the set.
`code`	Specifies the Tcl code to be executed for each match.

sctp::objset::foreachfolder Procedure

```
sctp::objset::foreachfolder $set var {code}
=> return value undefined
```

Description

The `foreachfolder` procedure iterates over each folder in the specified set. For each folder object found in the set, the `foreachfolder` procedure sets the `var` loop variable to the CTP object (an Object Info object) and executes the specified Tcl code.

Arguments

`$set`	Specifies the set of objects containing the folders being processed.
`var`	Specifies the loop variable. The var argument is set, in turn, to each CTP object in the set that is a folder.
`code`	Specifies the Tcl code to be executed for each iteration.

sctp::objset::getinfo Procedure

```
sctp::objset::getinfo $set objname
=> Object Info object
```

Description

The `getinfo` procedure returns the CTP object (an Object Info object) that corresponds to the specified `objname` argument.

Arguments

`$set`	Specifies the set containing the object whose properties are to be returned.
`objname`	Specifies the name of the object whose CTP object (Object Info object) is to be returned. **Note**: The object's name must contain `.sgc` . Otherwise the Custom Type System throws an exception.

sctp::objset::names Procedure

```
sctp::objset::names $set
=> list of names
```

Description

The `names` procedure returns the list of object names in the specified set.

Arguments

`$set`	Specifies the set whose object names are to be returned.

General sctp Procedures

sctp::fileExists Procedure

```
sctp::fileExists filename
=> 0,1,2,3,4,-1
```

Description

The `fileExists` procedure detects whether the specified file or folder exists, returning the following values:

`0`	The object does not exist.
`1`	The object exists and is a file.
`2`	The object exists and is a directory.
`3`	The object exists and is a link to an existing file.
`4`	The object exists and is a link to an existing directory.
-1	The object exists but is a broken link (links to nowhere).

It is recommended that you use the `sctp::fileExists` procedure in your procedure rather than the Tcl `file exists` command because this procedure can better identify the different cases shown above.

Arguments

`filename`	Specifies the path to the file or folder.

sctp::getFolderType Procedure

```
sctp::getFolderType $dir
=> folder_type
```

Description

The `getFolderType` procedure returns the type of folder corresponding to the specified directory. The procedure determines whether

- The folder is already included in the folder type cache, which stores the folder types of the folders that have already been mapped.

  In this case, `getFolderType` returns the stored folder type.

- The folder is the user's HOME directory (on UNIX) or a root directory (/ or a drive).

  In this case, `getFolderType` returns `"Folder"`.

- The folder type has not yet been determined.

  In this case, if the `determineFolderType` procedure exists in the namespace of the function that called `getFolderType`, then `determineFolderType` is called. If the call to `determineFolderType` returns an empty string, `getFolderType` returns `"Folder"`. If the call to `determineFolderType` returns a string, the `getFolderType` procedure adds the returned folder type value to the folder type cache and returns the value.

Arguments

`$dir`	Specifies the folder whose folder type is being retrieved.

sctp::glob Procedure

```
sctp::glob [flags] pattern
=> list of filenames
```

Description

The `glob` procedure accepts a pattern and returns the filenames that match the pattern. The filenames are returned with any path that was specified in the pattern. For example:

- "./*" will return paths containing "./"
- "/users/fred/mydir/*" will return paths containing "/user/fred/mydir/"
- "*" will return the leaf names

If the CTS is running on UNIX, the `glob` procedure returns the identical list of filenames that the standard Tcl `glob` command returns, given the same arguments.

If the CTS is running on Windows, the `glob` procedure returns the list of filenames containing the backslash '\' characters appropriate for the Windows operating system.

Arguments

`flags`	The flags you can pass to the `glob` procedure are the same flags you can pass to the standard Tcl `glob` command. The `glob` procedure runs in 'nocomplain' mode (as if the `-nocomplain` option is passed to the Tcl `glob` command). In this case, the `glob` procedure returns an empty list if no files match the pattern. See a Tcl reference for more details about the flags accepted by the `glob` command.
`pattern`	The patterns you can pass to the `glob` procedure are the same patterns you can pass to the standard Tcl `glob` command. See a Tcl reference for more details about the patterns accepted by the `glob` command.

sctp::setKeyFiles Procedure

```
sctp::setKeyFiles namespace {list-of-wildcard-patterns}
=> return value undefined
```

Description

The `setKeyFiles` procedure supplies the patterns to identify the special files that represent collection objects. The patterns set by calling `setKeyFiles` are used to determine when the `processKeyFiles` procedure is called.

Note: The `setKeyFiles` procedure takes glob-style expressions, not `regexp` expressions.

Arguments

`namespace`	The CTP namespace for the CTP objects to be matched against the key files, for example, `lvcCTP`.
`list-of-wildcard-patterns`	A list of glob expressions to represent the patterns the CTP objects must match. **Note**: The `setKeyFiles` procedure takes glob expressions, not `regexp` expressions.

sctp::setTypeProps Procedure

```
sctp::setTypeProps typeName {name value name value …}
=> return value undefined
```

Description

The `setTypeProps` procedure lets you define custom types that have one or more properties associated with them. Specify the properties as a list of property name/value pairs. See CTP Object Type Properties for information about the Object Type Catalog, which stores these type definitions.

Arguments

`typeName`	Specifies the name to be assigned to the new object type. To create an object of this object type, use the `addobject` command and set the `objtype` property to this `typeName` string.
`name`	Specifies the property to be set. See CTP Object Type Properties: Object Type Catalog for the predefined properties you can set.
`value`	Specifies the value to which the property is to be set.

Case Studies

Case Study Examples

The case study topics describe CTP examples included with this document. The examples are stored in the following directory within the DesignSync installation hierarchy:

```
  <SYNC_DIR>/share/examples/doc/ctsguide
```

The following sample CTPs are included in the examples directory:

- `collection.ctp`: Illustrates a simple CTP.
- `lvc.ctp`: Illustrates a collection hierarchy that contains a library-view-cell directory structure.
- `local.ctp`: Illustrates how a CTP can implement a local versioning scheme reflective of some EDA tool suites.

Collection Example

To view the local version sample code, see collection.ctp.

To try the `collection.ctp` collection, install the following example in your `<SYNC_SITE_CUSTOM>/share/client/ctp` directory:

```
  <SYNC_DIR>/share/examples/doc/ctsguide/collection.ctp
```

The `collection.ctp` example creates collections of pairs of `.txt` and `.html` files.

Notice the following features of the Collection Example:

The members in the `mapViews` and `members` procedures match.

The `mapViews` procedure marks objects as collection members if both a `.txt` and an `.html` file exist. Notice that the `members` procedure returns the same members as the members added by the `mapViews` procedure.

The .txt/.html collections can occur anywhere in the data hierarchy.

Some collections depend on a particular organization of the data hierarchy in order for a collection to be recognized. This example assumes that a collection (a pair of `.txt` and `.html` files in this case) can occur anywhere in the data hierarchy. In other terminology, unlike the Library-View-Cell Example, this example requires no **hierarchy context**. Collections that do not rely on a hierarchy context for collection recognition do not need the `determineFolderType` procedure.

Example: collection.ctp

```
# $Revision$  $Date$
#
#
# Copyright (c) 1997-2010 Dassault Systemes. All rights reserved.
# Use of this source code is restricted to the terms of your license
# agreement with Dassault Systèmes Any use, reproduction, distribution,
# copying or re-distribution of this code outside the scope of that
# agreement is a violation of U.S. and International Copyright laws.
#
# DISCLAIMER: The following sample code is intended as a learning
# tool and not for use as production CTP code.
#
# Collection CTP : simple collection of a couple of files.
# If both exist, they are a collection. If one or the other exists,
# then they are not.


namespace eval collectionCTP {}
#
#
# NOTE: This example expects collection member files with
# names of the form "name.txt" and "name.html". The
# body of the name cannot contain other dot "." characters.
# This example does not handle the condition where a folder
# unexpectedly has one of the collection member file names.
#
#

sctp::setTypeProps "Test Collection"  {icon "CTP_collection.gif"}

#
# common subroutines
#
#
proc collectionCTP::fileExists { filename } {
# File must be a plain file on disk or a link to a plain file
# on disk. Nothing else is acceptable.
    if { [sctp::fileExists $filename] == 1 || \
         [sctp::fileExists $filename] == 3 } {
        return 1
    }
    return 0
}

proc collectionCTP::existsOnDisk {basename} {
# Both members must be real files on disk.
    return [expr [collectionCTP::fileExists $basename.html] \
                 && [collectionCTP::fileExists $basename.txt] ]
}

# Get the base name of a file.
proc collectionCTP::getBase {filename} {
    set dot [string first . $filename]
    if {$dot == -1} {
        return $filename
    }
    return [string range $filename 0 [expr $dot - 1]]
}

# Get the extension of a file.
proc collectionCTP::getExt {filename} {
    set dot [string first . $filename]
    if {$dot == -1} {
        return $filename
    }
    return [string range $filename [expr $dot] end]
}

# The following proc is here to illustrate that the member
# list can be set via a proc, or by the members property.
# If the members property is set, the members proc is ignored.
# To call the members proc instead, set the value of the
# FixedMembers variable to 1.
#

set collectionCTP::FixedMembers 0
# Set the member list. If FixedMembers is 0, do nothing
# and ::members will be called instead.
proc collectionCTP::setMemberList {obj base path} {
    if {$collectionCTP::FixedMembers} {
        sctp::obj::setprop $obj members [list $base.txt $base.html]
    }
}



# mapViews
# This is the main proc (required).
#
proc collectionCTP::mapViews {path set} {

    sctp::objset::foreachfile $set f *.txt {

        # For each txt, see if an html exists
        set base [collectionCTP::getBase [sctp::obj::name $f]]
        if [sctp::objset::contains $set $base.html] {

           # Found a matching html. Now make sure it's a file and
           # not a folder.
           if { [sctp::obj::type [sctp::objset::getinfo $set $base.html]] \
              == "File" } {
              # We have both, so create a collection.
              # Catch the addobject. This keeps the ctp from throwing
              # an exception if the collection object exists
              # on disk as a real file.
              if [catch {
                 set col [sctp::objset::addobject $set $base.sgc.tst ]
                 sctp::obj::setprop $col objtype "Test Collection"
              } addobjerr] {
                 set col [sctp::objset::getinfo $set $base.sgc.tst]
                 sctp::obj::setprop $col objtype "Invalid collection"
                 sctp::obj::setprop $col error \
                      "Collection object is already a disk file - \
                      found in mapViews."
              } else {

               # Now mark this with a special label for the GUI.
               # This is not required, but makes the GUI label more
               # distinctive for each separate collection.
               sctp::obj::setprop $col label [join [list $base \
                   "Test Collection"] ]

               #
               # Now declare membership and ownership.
               # This is required for collections.
               #
               setMemberList $col $base $path
               set htmlobj  [sctp::objset::getinfo $set $base.html]
               set txtobj  [sctp::objset::getinfo $set $base.txt]
               sctp::obj::setprop $htmlobj objtype "Test Member"
               sctp::obj::setprop $htmlobj owner [file join \
                   $path $base.sgc.tst ]
               sctp::obj::setprop $htmlobj label [join [list \
                   $base "Test Member"] ]
               sctp::obj::setprop $txtobj  objtype "Test Member"
               sctp::obj::setprop $txtobj  owner [file join $path \
                   $base.sgc.tst]
               sctp::obj::setprop $txtobj  label [join [list $base \
                   "Test Member"] ]
        } }
    } }



    # Make sure referenced collections are recognized properly.
    sctp::objset::foreachfile $set f *.sgc.tst {
        if { [collectionCTP::fileExists [file join $path \
                [sctp::obj::name $f]] ] } {
            sctp::obj::setprop $f objtype "Invalid collection"
            sctp::obj::setprop $f error "Collection object is \
                 already a disk file."

        } else {
            sctp::obj::setprop $f objtype "Test Collection"
        }
            set base [collectionCTP::getBase [sctp::obj::name $f]]
            collectionCTP::setMemberList $f $base $path
   }
}


proc collectionCTP::updateObject {path object} {

    set name [sctp::obj::name $object]
    set base [collectionCTP::getBase $name]

    set extlen 8
```

```
     set length [string length $name]
    if {$length > $extlen} {
        set ext [string range $name [expr $length - $extlen] $length]
        if {$ext == ".sgc.tst"} {
            # mark as a collection
            sctp::obj::setprop $object objtype "Test Collection"
#
# collectionexists - Why do this?
# This correctly identifies the object for the sake of 'url exists'.
# If an object physically exists, or it has metadata (thus is a
# reference) then 'url exists' always returns 1. In the case of a
# collection that has not yet been checked in, the object is neither
# an actual object on disk, nor does it have metadata. So I have to
# tell 'url exists' that yes, this thing really exists. To see how
# this makes a difference, comment out the next section, create a new
# collection pair: n.txt and n.html, then type 'url exists n.sgc.tst'.
# Without this section, you get back a no answer. That's a problem
# because some operations might then actually skip processing this
# object because it thinks it doesn't exist.
#
            if { [collectionCTP::existsOnDisk [file join $path $base]]  } {
                 if [catch {
                      sctp::obj::collectionexists $object
                 } existserr] {
                      sctp::obj::setprop $object objtype \
                         "Invalid collection"
                      sctp::obj::setprop $object error \
                         "Collection object is already a disk file - found \
                         in updateObject."
                      setMemberList $object $base $path
                 } else {
                      # In this case, we have a new object not yet checked
in,
                      # or an existing managed collection.
                      setMemberList $object $base $path

                      # Now mark this with a special label for the GUI
                      sctp::obj::setprop $object label [join [list $base \
                         "Test Collection"] ]

                 }
            } else {
             if { [collectionCTP::fileExists [file join $path \
                     $base.sgc.tst] ] } {
                # Really odd case. There is a file on disk with the
collection
                # extension, but the members don't exist.
                       sctp::obj::setprop $object objtype "Invalid
collection"
                       sctp::obj::setprop $object error "Collection object \
                           is already a disk file."
                       setMemberList $object $base $path

               } else {
                   # I have metadata, but the object and members are not on
disk.
                   # Must be a reference.
```

```
                    setMemberList $object $base $path

                    # Now mark this with a special label for the GUI.
                    sctp::obj::setprop $object label [join [list $base \
                        "Test Collection"] ]
                }
            }


        }
    }

    set ext [collectionCTP::getExt $name]
#
# Identify if the file is a member
#

    if {$ext == ".txt" && [collectionCTP::fileExists [file join $path \
            $base.html ] ] } {
        sctp::obj::setprop $object objtype "Test Member"
        sctp::obj::setprop $object owner [file join $path $base.sgc.tst ]
    }
    if {$ext == ".html" && [collectionCTP::fileExists [file join $path \
            $base.txt ] ] } {
        sctp::obj::setprop $object objtype "Test Member"
        sctp::obj::setprop $object owner [file join $path $base.sgc.tst  ]
    }
}


proc collectionCTP::members {path object type} {
    if {$collectionCTP::FixedMembers} {

    # This proc should not be called if "FixedMembers" variable was
    # set to 1.  If we get here, then some error was encountered -
    # probably a broken symlink issue.  In that case, throw an error and
    # mark the collection as invalid.
    sctp::obj::setprop $object error "Error in member set for this
collection."
    }
    set base [collectionCTP::getBase $object]
    return [list $base.txt $base.html]
}
```

Library-View-Cell Example

To view the Library-View-Cell sample code, see lvc.ctp.

To try the `lvc.ctp` collection, install the following example in your `<SYNC_SITE_CUSTOM>/share/client/ctp` directory:

```
  <SYNC_DIR>/share/examples/doc/ctsguide/lvc.ctp
```

You can use this CTP on the following data by untarring the tar file and running DesignSync from the generated `lib` directory:

```
  <SYNC_DIR>/share/examples/doc/ctsguide/lvc.tar
```

Notice the following features of the Library-View-Cell Example:

Hierarchy Context

Some collections can occur within any directory (folder) in the design data hierarchy. Other collections, however, can only occur in a particular hierarchy context for the collections to be recognized as valid. For the data managed by the `lvc.ctp` to be recognized as a valid collection, it must be in a directory that contains an `lvc.lib` file. Thus, the CTP needs a `determineFolderType` procedure which will detect whether a particular directory contains an `lvc.lib` file and therefore might contain a collection.

Use of contentsChanged Procedure

The `lvc.ctp` CTP calls the custom `contentsChanged` procedure to monitor for the situation when a new schematic cell view is fetched into the workspace. In this case, the CTP provides notification that objects in the affected libraries have changed. Notice that the `operationBegin` and `operationEnd` procedures work in tandem with the `contentsChanged` procedure to implement this notification. The `operationBegin` procedure clears the `libsToNotify` variable which stores the libraries that have changed. The `operationBegin` procedure also captures the command name that caused the `operationBegin` procedure to be triggered. Only if the operation is a checkout or a populate does the `contentsChanged` procedure check for changed objects. After the operation completes, the `operationEnd` procedure provides notification of the changed libraries. As an enhancement to this CTP, notification of changed libraries could be replaced with a system call to notify the EDA tools of potential new schematic data.

Example: lvc.ctp

```
# $Revision$    $Date$
#
# Copyright (c) 1997-2010 Dassault Systemes. All rights reserved.
# Use of this source code is restricted to the terms of your license
# agreement with Dassault Systèmes Any use, reproduction, distribution,
# copying or re-distribution of this code outside the scope of that
# agreement is a violation of U.S. and International Copyright laws.
#
# DISCLAIMER: The following sample code is intended as a learning
# tool and not for use as production CTP code.
#
# Lib-View-Cell Custom Type Package
#
# DESCRIPTION
# -----------
# This sample Custom Type Package implements an imaginary directory
# structure that contains design data in a library-view-cell hierarchy.
#
# A "LVC Library" directory is identified as any directory containing a
# a file named lvc.lib.
#
# Under a "LVC Library", each sub-directory is considered a "LVC View".
#
# Under a "LVC View", each subdirectory is potentially a "LVC Cell" if
# it contains the contents of a "LVC Cell View" object.
#
# A cell-level directory is an "LVC Cell" if it contains a file matching:
#   lvc.celltype.*
#
# Where there is a "LVC Cell", then a collection object is created to contain
# the set of files that make up the cell. The set of files is dependent on
# the cell type, and the cell type is identified by the tail part of the
# lvc.celltype.* name. A mapping identifying the set of member files for
# any particular type is hard-coded into this example. See below.
#
# For example, Consider the following directory structure:
#
# mylib/
#     lvc.lib
#     drawing/
#         NAND/
#             lvc.celltype.schematic
#             sch.db
#             sch.props
#             tmp.bck
#         NOR/
#             lvc.celltype.symbol
#             symbol.db
#             symbol.pinmap
#         NonCell/
#             tmp.txt
#
```

```
# In this example, "mylib" is a "LVC Library", because it contains a
"lvc.lib"
# file. The library has a single "LVC View" type folder called "drawing".
# The "drawing" view contains two "LVC Cell" type folders, identified by the
# lvc.celltype.* files, and one other folder.
# The NAND "LVC Cell" contains a "schematic" type object. The member file
# mapping for "schematic" indicates that anything matching "sch.*" is a
# members, and so the sch.db and sch.props are members, but not the tmp.bck.
#
# This CTP defines two collection objects from this data. The collections are
# created within the "LVC View" directory, and they are "schematic.sgc.lvc"
and
# "symbol.sgc.lvc"
#
# By default, the LVC Cell directories are marked as "non-recursable", so
# any files other than the members within them are not processed by recursive
# operations, but this can be changed by setting the variable:
#    set lvcCTP::recurse_into_cells 1
#
# When a "schematic" view is modified/fetched as a result of a co/pop
# operation, the design tools need to be notified of that. This is achieved
# through the operationBegin/operationEnd and contentsChanged system.
# If any schematic is modified (by co/pop) in an operation, then a system
# call (commented out in this example) is made to notify the design tools
# that there may be new data present.
# The sctp::setKeyFiles call and lvcCTP::processKeyFiles proc are also
# related to this requirement to perform an update on an initial populate.
#
############################################################################
##

namespace eval lvcCTP {

    #-------------------------------------------------------------------
    # Initialize the CTS Type Catalog with our types and the properties
    # for these types.
    #
    # NOTES:
    #   1. Recurse on folders/non-view folders set according
    #      to variable
    #   2. Collection members are not versionable, the collection as
    #      a whole is.
    #   3. opNotify is set for LVC View.
    #-------------------------------------------------------------------
    sctp::setTypeProps {LVC Library} \
                    [list recurse 1]
    sctp::setTypeProps {LVC View} \
                    [list recurse 1 opNotify 1]
    sctp::setTypeProps {LVC Cell} \
                    [list opNotify 1]

    sctp::setTypeProps {LVC Cell View} \
                    [list versionable 1]

    sctp::setTypeProps {LVC Member} \
                    [list versionable 0]
    sctp::setTypeProps {LVC Lib File} \
                        [list versionable 1]


    #------------------
    # Key files identify special files that might belong to us. This helps
    # on an initial populate.
    # The files we want to key off of are the LVC collection objects.
    # NOTE: the setKeyFiles proc takes glob expressions, not
    # regexp.
    #------------------
    sctp::setKeyFiles lvcCTP [list \
        {*\.sgc\.lvc} \
    ]

    #--------------------------------------------------------------------
    # This variable controls whether we recurse into cell directories.
    #--------------------------------------------------------------------
    variable recurse_into_cells 0

    #--------------------------------------------------------------------
    # This mapping defines how types are mapped to a set of "glob"
    # expressions that identify the members of that view. Multiple
    # expressions are allowed for each type.
    # At present only two type are defined, but this would in practice
    # be a larger list.
    #--------------------------------------------------------------------
    variable cellMap
    set cellMap(schematic) [list sch.*]
    set cellMap(symbol) [list symbol.*]

}

#------------------------------------------------------------------------
# CTP Procs
#
# The next set of procedures are those required by the CTS system.
# This is followed by "helper functions" that are specific to this
# CTP.
#------------------------------------------------------------------------



#---------------------------------------------------------------------
# ::lvcCTP::mapViews {dir contents}
#
# dir:      The path to the directory containing contents
# contents: An object set containing the contents of $dir
#
# This proc is the core of object recognition for the LVC CTP.
# This is called by the CTS system whenever a whole directory is
# being expanded to identify its contents.
#
# We will determine the type of the dir and then if it is a LVC type we
# will process the contents.
#---------------------------------------------------------------------
proc ::lvcCTP::mapViews {dir contents} {
    #-------------------------
    # Figure out the type of this directory
    #-------------------------
    set type [sctp::getFolderType $dir]

    set skip 0
    #-------------------------
    # If it matches one of these then process the files in these
    # directories.
    #-------------------------
    switch -exact -- $type {
        {LVC Library} {
            #------------------------------------------------------------
            # Map the files in a Library
            #------------------------------------------------------------
            lvcMapLibraryContents $dir $contents

        }
        {LVC View} {
            #------------------------------------------------------------
            # Map the files in a View
            #------------------------------------------------------------
            lvcMapViewContents $dir $contents
            set skip 1
        }

        {LVC Cell} {
            #------------------------------------------------------------
            # Map the files in a Cell
            #------------------------------------------------------------
            lvcMapCellContents $dir $contents
        }
    }


    #-----------------
    # Now determine the types of all the folders in the dir,
    # except for LVC View where this has already been done.
    #-----------------
    if {!$skip} {
        sctp::objset::foreachfolder $contents obj {
            if {[sctp::obj::type $obj ] == {Folder} } {
                set ftype [sctp::getFolderType [file join $dir \
                                                [sctp::obj::name $obj]]]
                if {$ftype != {Folder}} {
                    sctp::obj::setprop $obj objtype $ftype
                }
            }
        }
    }

    return

}

#---------------------------------------------------------------------
```

```
# ::lvcCTP::updateObject {dir objInfo}
#
# dir:      Directory path
# objInfo:  A CTS object item
#
# This proc is called whenever the CTS system needs to identify
# a specific object.
#
# We see whether the name matches one of our objects, and of so then
# set any appropriate properties.
#
#-----------------------------------------------------------------
proc ::lvcCTP::updateObject {dir objInfo} {

    #-----------------------------------------------------------------
    # Figure out the type of this directory
    #-----------------------------------------------------------------
    set type [sctp::getFolderType $dir]

    #-----------------------------------------------------------------
    # If it matches one of these then set the appropriate information
    # for the given objInfo. This handles everything but directory types.
    #-----------------------------------------------------------------
    switch -exact -- $type {
        {LVC Library} {
            lvcUpdateLibObject $dir $objInfo

        }
        {LVC View} {
            lvcUpdateViewObject $dir $objInfo
        }

        {LVC Cell} {
            lvcUpdateCellObject $dir $objInfo
        }

    }
    #-----------------------------------------------------------------
    # If the object is itself a folder of one of our types, then set the
    # type of it.
    #-----------------------------------------------------------------
    if {[sctp::obj::type $objInfo ] == {Folder} } {
        set ftype [sctp::getFolderType [file join $dir \
                                        [sctp::obj::name $objInfo]]]
        if {$ftype != {Folder}} {
            sctp::obj::setprop $objInfo objtype $ftype
        }
    }

}


#-----------------------------------------------------------------
# ::lvcCTP::determineFolderType {dir}
# => type
#
# dir:  Directory path.
```

```
#
# Figure out if the given directory is one of our special directory
# types.
# Returns the folder type if we know, otherwise return nothing
#
#-------------------------------------------------------------------
proc ::lvcCTP::determineFolderType {dir} {

    if {$dir == {/}} {return {}}

    #-------------------------------------------------------------------
    # If there is a lvc.lib, this is a library.
    #-------------------------------------------------------------------
    if {[file exists [file join $dir lvc.lib]]} {
        return "LVC Library"
    }

    #-------------------------------------------------------------------
    # If there is a lvc.lib above, this is a view.
    #-------------------------------------------------------------------
    if {[file exists [file join [file dirname $dir] lvc.lib]]} {
        return "LVC View"
    }

    #-------------------------------------------------------------------
    # If there is a file matching lvc.celltype.*, then this is a cell.
    # NOTE: In theory, we should check that the parent is a view as
    # well.
    #-------------------------------------------------------------------
    if {[llength [glob -nocomplain -dir $dir "lvc.celltype.*"]] > 0} {
        return "LVC Cell"
    }

    return {}

}


#-------------------------------------------------------------------
# ::lvcCTP::members {parentPath objName type}
#
# Figure out the members of the LVC Cell View.
# This is called when CTS needs to know the set of members of a
# collection object crated by this CTP.
#
# Return: a list of members or nothing
#-------------------------------------------------------------------
proc ::lvcCTP::members {parentPath objName type} {

    if {$type == {LVC Cell View}} {
        #---------------------------------------------------------------
        # Extract the cell directory name and look up the members.
        #---------------------------------------------------------------
        return [lvcMembers [lvcGetDir [file join $parentPath $objName]]]

    }
```

```
}

#-----------------------------------------------------------------------
# ::lvcCTP::recurse { path type }
#
# path: Directory path.
# type: Type of the directory.
#
# This is called by the CTS system to decide whether to recurse into
# a folder of a type that does not have a specific recurse setting
# defined in the type properties.
#
# In our case, this wwill be called on a "LVC Cell", and we decide
# whether to recurse depending on the package variable, so that it
# can be overridden by the user.
#-----------------------------------------------------------------------
proc ::lvcCTP::recurse { path type } {

    if {$type == "LVC Cell"} {
        return $::lvcCTP::recurse_into_cells
    }
    return 1

}


#-----------------------------------------------------------------------
# ::lvcCTP::operationBegin
#
# Clear the libsToNotify list.
# Set the operation variables, so that when contentsChanged is
# called we know whether we are interested in the objects.
#
#-----------------------------------------------------------------------
proc ::lvcCTP::operationBegin { cmd } {

    #-------------------------
    # clear the libsToNotify data
    #-------------------------
    if {[ info exists ::lvcCTP::libsToNotify]} {
        array unset ::lvcCTP::libsToNotify
    }

    set ::lvcCTP::operation [lindex $cmd 0]

}

#-------------------------------------------------------------------------
# ::lvcCTP::contentsChanged {path added removed}
#
# path:     Directory path
# added:    Objects added/changed in that dir
# removed:  Objects removed from that dir
#
# This is called whenever anything changes in a dir.
# If we are in a pop/co operation, then record the library path IF
# the path given is a schematic view folder. (Note that this means
# we record the name whether items are added or removed, and also
# whether those items are cell objects or plain files. This could
# be improved in practice by examining the added/removed lists.
# NOTE: The actual command value for populate operations is _populate,
# the actual values for commands can be found by examining the operation
# values in the operationBegin proc.
#-----------------------------------------------------------------------
proc ::lvcCTP::contentsChanged {path added removed} {

    #-------------------------------------------------------------------
    # We are only interested if the command was co or populate.
    #-------------------------------------------------------------------
    if {$::lvcCTP::operation == "co" || $::lvcCTP::operation == "_populate"}
{

        #---------------------------------------------------------------
        # Add if the tail of the path is "schematic"
        #---------------------------------------------------------------
        if {[file tail $path] == "schematic"} {

            #-----------------------------------------------------------
            # Then add the library path to our array.
            #-----------------------------------------------------------
            set ::lvcCTP::libsToNotify([file dirname $path]) 1
        }
    }

}


#--------------------------------------------------------------------
# ::lvcCTP::operationEnd
#
#
# When the operation ends, see whether there are libraries to
# be notified.
#--------------------------------------------------------------------
proc ::lvcCTP::operationEnd {} {

    #---------------------------------------------------------------------
    # Process each library that we found updates for in this operation.
    # In this example, we just generate a message.
    #---------------------------------------------------------------------
    if {[info exists ::lvcCTP::libsToNotify]} {
        foreach lib [array names ::lvcCTP::libsToNotify] {
            puts "Updated data in library: $lib. Need to notify."
        }
    }

}

#-----------------------------------------------------------------------
# ::lvcCTP::processKeyFiles {path files}
#
# path:     Directory path
# files:    Key files found in that path.
#
```

```
# This proc will be called when a populate command operates on a new
# folder and adds files to the the folder that match the file names
# specified above in sctp::setKeyFiles.
# This handles "bootstrapping", where we need to be notified about the
# new objects, but because they are only just being fetched, the real object
# recognition has not been able to come into play.
#
# In this case, all we need to do is act as though
# ::lvcCTP::contentsChanged() was called, so we pass this call through to
that.
#
# See the sctp::setKeyFiles call above.
#-----------------------------------------------------------------------
proc ::lvcCTP::processKeyFiles {path files} {

    ::lvcCTP::contentsChanged $path $files {}

}



#-----------------------------------------------------------------------
# HELPER FUNCTIONS
#
# These are the procs used by the above code.
#-----------------------------------------------------------------------


#-----------------------------------------------------------------------
# lvcGetDir {objName}
# => cell_dir
#
# Given an object name of the form /dir/cell.sgc.lvc, return /dir/cell
# Used to convert a cell view object name into the cell directory name.
#-----------------------------------------------------------------------
proc lvcGetDir {objName} {

    set dir [file dirname $objName]
    set leaf [file tail $objName]
    set cell [string range $leaf 0 [expr [string length $leaf] - 9]]

    return [file join $dir $cell]
}

#-----------------------------------------------------------------------
# lvcGetType {dir}
# => type
#
# Finds the lvc.celltype.* file and extracts the type name from it.
#-----------------------------------------------------------------------
proc lvcGetType {dir} {

    #-------------------------------------------------------------------
    # Extract the type value from the lvc.celltype.* file.
    # NOTE: We assume there is only one such file!
    #-------------------------------------------------------------------
    set type_file [lindex [glob -nocomplain [file join $dir lvc.celltype.*]]
0]
    set type [string range [file tail $type_file] 13 end]

    return $type

}


#------------------------------------------------------------------------
# lvcMembers {dir}
# => members
#
# dir:  Cell directory path
#
# Given a LVC Cell directory, identify the member files.
#------------------------------------------------------------------------
proc lvcMembers {dir} {

    set type [lvcGetType $dir]

    #--------------------------------------------------------------------
    # If there was no type found, simply return no members.
    #--------------------------------------------------------------------
    if {$type == {}} {
        return
    }

    #--------------------------------------------------------------------
    # Check we have a type mapping for this type.
    #--------------------------------------------------------------------
    if {![info exists ::lvcCTP::cellMap($type)]} {
        error "No mapping found for type: $type"
    }

    #--------------------------------------------------------------------
    # Build a list of glob expressions, and then match them to find the
    # members.
    # NOTE: We assume no file is matched by more than one expression.
    # NOTE: members are specified as relative paths to the collection
    # object, and must be separated by the system file separator.
    #--------------------------------------------------------------------
    set lst {}
    foreach glob_str $::lvcCTP::cellMap($type) {
        lappend lst [file join $dir $glob_str]
    }
    set len [string length [file dirname $dir]]
    incr len
    if {$lst != {}} {
        set res {}
        foreach glob_str $lst {
            set match [sctp::glob $glob_str]
            foreach m $match {
                lappend res [string range $m $len end]
            }
        }
        lappend res "[file tail $dir][file separator]lvc.celltype.$type"
        return $res
    }
    return {}
}


#--------------------------------------------------------------------
# lvcMapLibraryContents {dir contents}
#
# dir:      Directory to be processed
# contents: A CTS contents list for the dir.
#
# The contents are from a known LVC Library directory.  Determine the
# types of the files and set them.
# The only thing we are intersted in is the lvc.lib file.
# Note that types of sub-dirs are handled in the calling function.
#
#--------------------------------------------------------------------
proc lvcMapLibraryContents {dir contents} {

    #------------------
    # Look for the library identification file
    #------------------
    if {[sctp::objset::contains $contents {lvc.lib} lfb]} {
        sctp::obj::setprop $lfb objtype {LVC Lib File}
    }

}


#--------------------------------------------------------------------
# lvcMapViewContents {dir contents}
#
# dir:      Directory to be processed
# contents: A CTS contents list for the dir.
#
# The contents are from a known LVC View directory.
# There are no specific file types in this directory, but
# add any LVC Cell View objects that are needed.
#
#--------------------------------------------------------------------
proc lvcMapViewContents {dir contents} {

    #----------------------------------------------------------------
    # Find all dirs, and if have a lvc.celltype.* set up the collection
    # object.
    #----------------------------------------------------------------
    sctp::objset::foreachfolder $contents obj {

        set dname [sctp::obj::name $obj]
        set vdir [file join $dir $dname]

        if {[glob -nocomplain -dir $vdir "lvc.celltype.*"] != {}} {
            #--------------------------------------------------------
            # Add the collection object.
            #--------------------------------------------------------
            set collObj [sctp::objset::addobject $contents $dname.sgc.lvc]
            sctp::obj::setprop $collObj objtype {LVC Cell View}

            #--------------------------------------------------------------
            # Get the members, and set them for the collection object.
            #--------------------------------------------------------------
            set mems [lvcMembers $vdir]
            sctp::obj::setprop $collObj members $mems

            #--------------------------------------------------------------
            # Set the view type label. This is displayed as the extended
            # type name by the ls command.
            #--------------------------------------------------------------
            set vtype [lvcGetType $vdir]
            if {$vtype != ""} {
                sctp::obj::setprop $collObj label "LVC $vtype Cell View"
            }

            #--------------------------------------------------------------
            # Set the type of the cell folder itself.
            #--------------------------------------------------------------
            sctp::obj::setprop $obj objtype "LVC Cell"

        }
    }

}

#---------------------------------------------------------------------
# lvcMapCellContents {dir contents}
#
# dir:      Directory to be processed
# contents: A CTS contents list for the dir.
#
# The contents are from a known LVC Cell directory.
# Find any LVC Cell View members and mark their type and owner.
#
#---------------------------------------------------------------------
proc lvcMapCellContents {dir contents} {

    #------------------------------------------------------------------
    # Get the members.
    #------------------------------------------------------------------
    set mems [lvcMembers $dir]
    set dname [file tail $dir]

    #------------------------------------------------------------------
    # Look up the members and set their type, and set the
    # owner to the collection object.
    # NOTE: The owner is a full path, in Unix path format, which is
    # how the directory is passed into here.
    #------------------------------------------------------------------
    sctp::objset::foreachfile $contents obj {*} {
        set objname [sctp::obj::name $obj]
        if {[lsearch -exact $mems "$dname[file separator]$objname"] != -1} {
            sctp::obj::setprop $obj objtype "LVC Member"
            sctp::obj::setprop $obj owner $dir.sgc.lvc
        }
    }
}

#--------------------------------------------------------------------
# lvcUpdateLibObject {dir objInfo}
#
# dir:      Directory containing the object
# objInfo:  The TS object
#
# Called by ::lvcCTP::updateObject
#
# If the object is our lvs.lib, set its type.
#
#--------------------------------------------------------------------
proc lvcUpdateLibObject {dir objInfo} {

    set objname [sctp::obj::name $objInfo]
    if {$objname == "lvc.lib"} {
        sctp::obj::setprop $objInfo objtype "LVC Lib File"
    }

}


#--------------------------------------------------------------------
# lvcUpdateViewObject {dir objInfo}
#
# dir:      Directory containing the object
# objInfo:  The TS object
#
# Called by ::lvcCTP::updateObject
#
# If we are given a LVC Cell View, then we need to set the type and
# the members.
#
#--------------------------------------------------------------------
proc lvcUpdateViewObject {dir objInfo} {

    set objname [sctp::obj::name $objInfo]
    if {![string match *.sgc.lvc $objname]} {
        return
    }

    sctp::obj::setprop $objInfo objtype "LVC Cell View"

    set dname [lvcGetDir [file join $dir $objname]]

    #--------------------------------------------------------------------
    # Set the view type label. This is displayed as the extended type
    # name by the ls command.
    #--------------------------------------------------------------------
    set vtype [lvcGetType $dname]
    if {$vtype != ""} {
        sctp::obj::setprop $objInfo label "LVC $vtype Cell View"
    }

    #--------------------------------------------------------------------
    # Set the view members. Allow for them being empty, which might
    # happen for a view in reference mode.
    #--------------------------------------------------------------------
    set mems {}
    catch {set mems [lvcMembers $dname]} res

    #--------------------------------------------------------------------
    # See if any of the files exist on disk. As soon as one exists
    # assume they all exists. Mark the object.
    #--------------------------------------------------------------------
    foreach i $mems {
        if {[file exists [file join $dir $i]]} {
            sctp::obj::collectionexists $objInfo
            break
        }
    }

    #--------------------------------------------------------------------
    # Update the collections members.
    #--------------------------------------------------------------------
    sctp::obj::setprop $objInfo members $mems

}


#--------------------------------------------------------------------
# lvcUpdateCellObject {dir objInfo}
#
# dir:      Directory containing the object
# objInfo:  The TS object
#
# Called by ::lvcCTP::updateObject
#
# See if the given item is a LVC Member, and if so set its type and owner.
#
#--------------------------------------------------------------------
proc lvcUpdateCellObject {dir objInfo} {

    set objname [sctp::obj::name $objInfo]
    set mems [lvcMembers $dir]
    set dname [file tail $dir]

    if {[lsearch -exact $mems "$dname[file separator]$objname"] != -1} {
        sctp::obj::setprop $objInfo objtype "LVC Member"
        sctp::obj::setprop $objInfo owner $dir.sgc.lvc
    }

}
```

Local Version Example

The local version example illustrates a methodology whereby a design tool implements its own basic version management by making local copies of design objects. See Developing Custom Type Packages: Local Version Methodology for a discussion of local versions.

To view the local version sample code, see local.ctp.

To try the local.ctp collection, install the following example in your `<SYNC_SITE_CUSTOM>/share/client/ctp` directory:

```
  <SYNC_DIR>/share/examples/doc/ctsguide/local.ctp
```

The Local Version Methodology of the local.ctp Example

Typically a CTP with a local version methodology might have a methodology such that the local version with the highest local version number is the current local version. In `local.ctp`, the collections are appended with `_v1`, `_v2`, and `_v3`. The current local version is the local version with the `_v3` extension. If `_v3` doesn't exist, the current local version is the local version with the `_v2` extension. If `_v2` doesn't exist, the current local version is the local version with the `_v1` extension.

Example: local.ctp

```
# $Revision$   $Date$
#
# Local CTP : collection for illustrating local versions
# This example sets versionable and ownership properties.
# Procedures used in this example include: obsmembers,
# collectionexists,getCurrentLocalVersion, localVersionChanged,
# getLocalVersionFromTags.
#
# Copyright (c) 1997-2010 Dassault Systemes. All rights reserved.
# Use of this source code is restricted to the terms of your
# license agreement with Dassault Systèmes Any use, reproduction,
# distribution, copying or re-distribution of this code outside
# the scope of that agreement is a violation of U.S. and
# International Copyright laws.
#
# DISCLAIMER: The following sample code is intended as a learning
# tool and not for use as production CTP code.
#
#

namespace eval localCTP {}

# In this simple CTP, the collection consists of all of the members
# with the name *_v3, or if none exist, then the collection is all
# files with the name *_v2, or if none exist,  then the collection
# is all files with names matching the pattern *_v1. The local
# version is thus 1, 2, or 3. Note that this is not the same thing
# as saying "the highest local revision of each file is part of the
# "latest" collection" - not so.
#
# (4's are beyond the intelligence of this ctp example.)
# Note that "_" characters anywhere else in the name string can
# confuse this ctp. Note also that this ctp does not account for
# the fact that you might encounter a folder that follows the
# collection member naming convention.
#
# Current local versions are recognized as Loc Members, while the
# "older" local versions are marked as Obsolete.
#
# To see how this works, create some test data where the files end
# in "_v1". Check those in. Then create more files that have the
# same prefix, but end in "_v2" this time. Check those in.
#
# Anything recognizable as a current or obsolete version is in
# itself non-versionable. Only the collection object itself is
# versionable.
#
sctp::setTypeProps "Loc Member" { versionable 0 \
     icon CTP_member.gif }
sctp::setTypeProps "Obsolete File" { versionable 0 \
     owner "" icon CTP_unmanaged.gif }
```

```
# Get the base name of a file.
# The collection has the name local.sgc.loc.

proc localCTP::getBase {filename} {
    set dot [string first "_v" $filename]
    if {$dot == -1} {
        return $filename
    }
    return [string range $filename 0 [expr $dot - 1]]
}

# Get the extension of a file.
proc localCTP::getExt {filename} {
    set dot [string first "_v" $filename]
    if {$dot == -1} {

# Not a version, so check to see if we have a dot file.
        set dot [string first "." $filename]
        if {$dot == -1} {
           return $filename
        }
    set ext [string range $filename [expr $dot] end]
    return $ext

    }
    set ext [string range $filename [expr $dot] end]

# Check for proper format, otherwise ignore the file
    if {[string length $ext] != 3} {
        return $filename
    }

    return $ext
}


proc localCTP::mapViews {path set} {
    set found3 0
    set found2 0
    set found1 0
    if [sctp::objset::contains $set local.sgc.loc object] {
      # make sure we don't have a real file - that would be an error.
      if { [sctp::fileExists [file join $path local.sgc.loc]] != 0 } {
        sctp::obj::setprop $object error "Invalid collection: \
            local.sgc.loc is a file."
        return
      } else {
        sctp::obj::setprop $object objtype "Local Collection"
      }
    }
    sctp::objset::foreachfile $set object *_v3 {
        sctp::obj::setprop $object objtype "Loc Member"
        set found3 1
        set vernum 3
    }
    set twoType "Loc Member"
    set oneType "Loc Member"
    if {$found3} {
        set oneType "Obsolete File"
        set twoType "Obsolete File"
    }
    sctp::objset::foreachfile $set object *_v2 {
        sctp::obj::setprop $object objtype $twoType
        set found2 1
        if { $found3 == 0 } {set vernum 2}
    }
    if {$found2} {
        set oneType "Obsolete File"
    }

    sctp::objset::foreachfile $set object *_v1 {
        sctp::obj::setprop $object objtype $oneType
        set found1 1
        if { $found2 == 0  && $found3 == 0 } {set vernum 1}
    }
    if {$found3 || $found2 || $found1} {
        set colobj [sctp::objset::addobject $set local.sgc.loc \
           {objtype "Local Collection"} ]

        # Must also set the ciTag value for this local version
        # on the collection object; otherwise, getCurrentLocalVersion
        # won't find anything when passed the usetag option.
        sctp::obj::setprop $colobj ciTag [join \
             [list "CTP_LV" $vernum ] "" ]
        sctp::obj::setprop $colobj label [join \
             [list "Local Version" $vernum ] ]

        # This is not very efficient programmatically, but we need
        # to set the owner property now on anything that is a
        # collection member.
        sctp::objset::foreachfile $set object {*_v[1-3]} {
        if { [sctp::obj::getprop $object objtype] == \
                "Loc Member" } {
            sctp::obj::setprop $object owner \
                [file join $path "local.sgc.loc"]
            sctp::obj::setprop $object label [join \
                [list "Local Member" $vernum ] ]
        }
      }

    }
}

proc localCTP::updateObject {path object} {

# Ignore folders; just look at files and collection objects.
  if { [sctp::obj::type $object] != "Folder" } {
    set name [sctp::obj::name $object]
    set ext [localCTP::getExt $name]
    if {$ext == "_v3" } {
        sctp::obj::setprop $object objtype "Loc Member"
        sctp::obj::setprop $object label "Local Member 3"
        sctp::obj::setprop $object owner \
            [file join $path "local.sgc.loc"]
    }
    if {  $ext == "_v2"  } {
        set twoType "Loc Member"
        if {[llength [sctp::glob [file join $path *_v3] ]] > 0} {
            set twoType "Obsolete File"
        } {
            sctp::obj::setprop $object owner [file join $path \
               "local.sgc.loc"]
            sctp::obj::setprop $object label "Local Member 2"
        }

        sctp::obj::setprop $object objtype $twoType
    }
    if {  $ext == "_v1"  } {
        set oneType "Loc Member"
        if {[llength [sctp::glob [file join $path "*_v3"] ]] > 0 || \
               [llength [sctp::glob [file join $path "*_v2"] ]] > 0} {
            set oneType "Obsolete File"
        } {
            sctp::obj::setprop $object owner [file join $path \
                "local.sgc.loc"]
            sctp::obj::setprop $object label "Local Member 1"
        }

        sctp::obj::setprop $object objtype $oneType
    }
    if { $ext == ".sgc.loc" } {
       sctp::obj::setprop $object objtype "Local Collection"

       # Figure out the highest local version number and mark that
       # as the active one.
       set vernum 0
       if { [llength [sctp::glob [file join $path "*_v1"] ]] > 0 } \
          { set vernum 1 }
       if { [llength [sctp::glob [file join $path "*_v2"] ]] > 0 } \
          { set vernum 2 }
       if { [llength [sctp::glob [file join $path "*_v3"] ]] > 0 } \
          { set vernum 3 }
       if { $vernum > 0 } {
          sctp::obj::setprop $object ciTag [join \
              [list "CTP_LV" $vernum ] "" ]
          sctp::obj::setprop $object label [join \
              [list "Local Version" $vernum ] ]

          # Announce that this is a real object, just in case there's
          # no metadata yet. But throw error if there's an actual file
          # with this name.
          if { [sctp::fileExists [file join $path local.sgc.loc]] != 0 } {
              sctp::obj::setprop $object error "Invalid collection: \
                 local.sgc.loc is a file."
          } else {
              sctp::obj::collectionexists $object
          }
      }
    }
 }
}
```

```
proc localCTP::members {path object type} {
    # Return either the list of *_v2 or *_v1 or *_v3
    set mem [sctp::glob [file join $path "*_v3" ] ]
    if {[llength $mem] == 0} {
        set mem [sctp::glob [file join $path "*_v2" ] ]
        if {[llength $mem] == 0} {
                set mem [sctp::glob [file join $path "*_v1" ]]
        }
    }
    return $mem
}

#
# If there's a higher local version, the lower versions will be
# marked obsolete.
#
proc localCTP::obsmembers {path object type} {
    set one [sctp::glob [file join $path "*_v1" ] ]
    set two [sctp::glob [file join $path "*_v2" ]]
    set three [sctp::glob [file join $path "*_v3" ] ]
    set obs {}
    set lone [llength $one]
    set ltwo [llength $two]
    set lthree [llength $three]
    if { $lone > 0 } {
      if { $ltwo > 0  || $lthree > 0  } {
        lappend obs [list 1 $one]
    }}
    if { $ltwo > 0  && $lthree > 0 } {
        lappend obs [list 2 $two]
    }
    return $obs
}


proc localCTP::getCurrentLocalVersion {url type {usetag 1} \
     {members ""}} {
if {$usetag} {
     set res [::localCTP::getTagVal $url]
     return $res
    }

if { [llength $members] > 0 } {

# The member list is passed in. Ignore what's in the workspace
# and go by what was passed in. In this case, get the version
# number off the member file names.  Of course we are assuming
# that the members list is correct for this ctp.

   if { [lsearch -glob $members *_v1] > -1 } {
        return 1
   } else {
        if {[lsearch -glob $members *_v2] > -1 } {
        return 2
   } else {
        return 3
   }}

} else {


    set parentPath [url path [url container $url]]
    if {[llength [sctp::glob [file join $parentPath "*_v3"] \
         ]] >  0} {
        return 3
    }
    if {[llength [sctp::glob [file join $parentPath "*_v2"] \
         ]] > 0} {
        return 2
    }
    return 1
} }


#
# localVersionChanged proc
# This proc is used like contentsChanged or a post trigger,
# when a post-processing step is desired if the localversion
# info changes on a  collection object. That post-processing step
# would be programmed here. In this example, we just output a
# message that the LV number has changed.
#
proc localCTP::localVersionChanged {url type lv1 lv2 newMembers \
    obsoleteInfo} {
    puts "A local version changed."
    return 1
}


#
#
proc localCTP::getLocalVersionFromTags {url type tags} {
# NOTE: This function is only called directly by the system when
# performing a checkout with the savelocal -fail option.
   return [ ::localCTP::getTagVal $url ]
}



#
# This is a support proc that looks at all the tags and finds the
# one related to local versioning.
#
proc ::localCTP::getTagVal {url} {
    set int 0
    set tags [url tags $url]

    set index [lsearch -glob $tags {CTP_LV*}]
    if {$index >= 0} {
        #--------------
        # The tags is of the form {CTP_LVinteger}
        #--------------
        if {[scan [lindex $tags $index] {CTP_LV%d} int] != 1} {
            puts "Tag not in proper form: [lindex $tags $index]"
        }
    }

    return $int


}
```

Index