The *localdev* Development Environment - A Tutorial

by Robert Prentice, SRG last modified: February 9, 2001

This tutorial covers both background concepts and practical usage information required to set up and use the **localdev** development environment. The **nwscm** accessory is covered in a separate tutorial.

Contents

Introduction

The *localdev* development environment consists of a set of scripts that are used to manage source files within an arbitrary number of related production areas, as shown in Figure 1. The environment not only provides baseline-oriented source code control within a particular production area, but also provides a simple means for developers to switch contexts between production areas. Facilities for combining and testing the products of multiple baselines and production areas exist, as do utilities supporting the import of files into production areas and the migration of files between production areas.

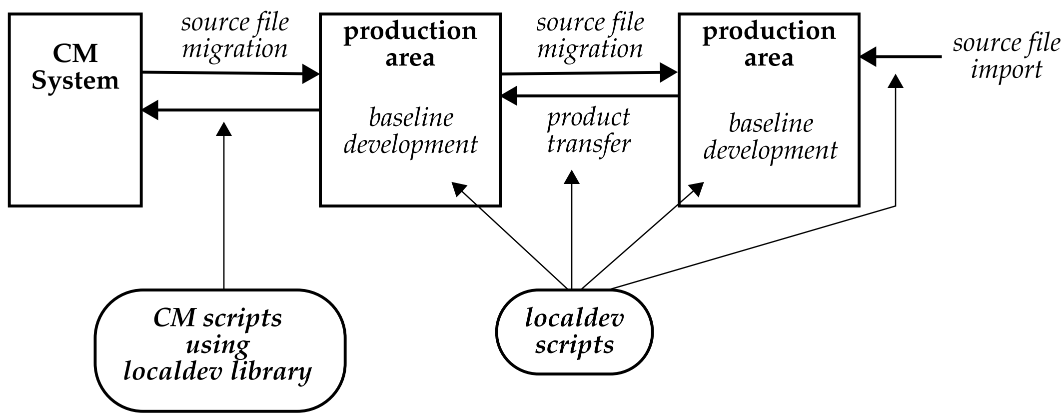


Figure 1. *localdev* **Features**

A capability that is of particular importance to FSL and the National Weather Service is the ability to easily migrate source files between the *localdev* environment and an external CM system. The actual scripts that implement this capability are dependent upon a particular CM system, and are hence not part of this generic package. The generic package includes a *CM interface library* that a set of CM scripts (implemented in Perl) can readily use for this purpose. This allows an implementation supporting a CM system to be installed as an accessory to this package. An *nwscm* accessory has been created to access the NWS CM system (refer to Appendix A).

The primary benefit to use of this package is that all of the diverse operations required to manipulate files in a complex development environment are performed safely and reliably by developer-oriented scripts, rather than through a collection of disparate, manual UNIX/RCS/BCS commands. This allows the following characteristics to be associated with all operations:

coordination	scripts ensure that developers coordinate activities within each production area
reliability	scripts have built-in safeguards to avoid inappropriate operations
efficiency	scripts expedite all operations, handling many details automatically
simplicity	scripts simplify development, making most development tasks routine
flexibility	scripts facilitate sharing of software and developers among multiple projects
compatibility	compatibility with an external CM system can be maintained

The *localdev* environment provides many useful capabilities to developers working within a particular production area that the underlying UNIX, BCS, and RCS utilities themselves cannot provide. These capabilities improve the efficiency of individual developers, and promote efficient interaction between the members of development groups. They also significantly reduce the amount of time required to maintain software baselines.

Background

The background information below summarizes concepts that constitute the foundation upon which the *localdev* environment is based. A general understanding of these concepts is a prerequisite to effective use of the environment.

Local Software Production

A **formal configuration management system** is designed to track all changes made to all files and other entities that affect released products, and to formalize the means whereby products are changed, built, and released. Reliable large-scale production of software requires such control. A CM system typically supports its own enterprise-wide access control, and uses a centralized database to hold and describe entities that are used to construct releasable products. Exercising such formal control over software development activities often requires significant amounts of administrative overhead, hardware resources, and developer time.

The formal control demanded by a configuration management system can increase the cost of changing software considerably. Its very formality makes it difficult to quickly adapt to change. This is naturally at odds with the process of software development, wherein change is anticipated as a frequent occurrence, and hence efficiency is achieved by keeping the cost of change small. There is thus a natural desire to apply configuration management control when and where it is needed, while using a less formal, more efficient system for all other activities. It is thus common to see software developed in small, independent packages that can be assembled into larger systems. These smaller packages can be developed by localized groups of developers who can take advantage of the high efficiency of less formal, localized development environments. Versions of software produced through these activities are typically fed into a formal CM system as they stabilize.

The *localdev* environment supports a **local software production environment**. It provides an integrated interface to a group of proven development tools, and adds to them a production orientation. It thus provides efficiency while maintaining production-oriented control over the development process. It's focus is on maximizing development efficiency and minimizing administrative overhead, while maintaining an ability to interface with formal CM systems. It supports development on multiple platforms within a single networked site.

Production Areas

Software development is more productive if many tasks can be performed in parallel, and if multiple versions of multiple products can be independently organized, managed, manipulated, tested, and released. Such independence is usually achieved through the establishment of **production areas**.

Production areas typically exist as directory hierarchies within shared file systems containing all files relevant to particular projects. Each production area is intentionally isolated, and measures are taken to control all interaction between them. These measures are intended to allow development activities to proceed independently within each production area. Developers can thus work on multiple projects, or on multiple versions of the same project without interfering with each other.

While it is possible for many developers to work in a single monolithic production area, it is often more efficient to organize activities into smaller, related projects that reuse shared software. It thus becomes important for developers to be able to easily interact with multiple production areas, and to transfer products or source files from one production area to another. Each of these topics is covered in detail herein.

The *localdev* environment maintains a context for developers. A selected **current production area** is a primary part of that context. It controls which production area a developer interacts with at any particular time. This single-context environment is quite adequate for most developers. A **multi-tasking** feature adds the ability to set multiple contexts within the environment simultaneously, so that any one of these contexts can be selected as active within each window.

Baselines and Staging Areas

A proven means of managing a production area is to maintain source files in baselines. A **baseline** consists of a hierarchical directory tree wherein all source files are maintained under revision control, and wherein the most recent revisions of each source file are readily accessible. Superseded revisions are still available for inspection, but are not as easily shared among developers. Baselines serve as shared repositories for source files. Development takes place within an arbitrary number of staging areas that are each linked to baseline files.

A **staging area** is a work area that mirrors a baseline. Each developer typically has their own staging area (for each baseline), within which their development activity takes place. Files within a staging area are usually symbolic links to baseline files. Each staging area has the same hierarchy of directories that exists within its baseline, except that the RCS subdirectories within its source directories are actually symbolic links to baseline RCS directories.

A staging area initially contains links to all “*important*” files in its baseline. Part of a baseline’s configuration determines what types of files are deemed important. If a developer updates a baseline file, then staging areas pick up the change immediately through symbolic links. It thus becomes convenient for developers to integrate their software frequently and regularly, rather than deferring it to a massive undertaking at the end of the development cycle.

A master staging area is typically created as an integration area for a project. Builds are regularly performed therein so as to immediately uncover build problems, and to provide a shared set of build-generated files for group use. Individual developers create their own staging areas to support development and testing in preparation for integration. File modification and build operations are performed within the context of a particular staging area.

A staging area that contains only directories and symbolic links is always **synchronized** with its baseline, with the exception of files created since the last synchronization operation. Any change to a baseline file is immediately propagated to such a staging area. Developers can exercise more control over when baseline changes impact a staging area by **staging** files. Staging a file replaces its link in a staging area with an actual copy of a file. This makes the staging area immune from changes to that file that might take place in the baseline, but also allows a file to fall out of synchronization with the baseline. A developer can later **unstage** the file to restore the link, thereby re-synchronizing the file with the baseline.

Baseline development utilizes file locking as a means of ensuring that developers do not make simultaneous changes to baseline files from their own staging areas. This precaution promotes communication between developers, and improves productivity associated with software integration. Files are **locked** and staged with a single command, making files available for modification. They can subsequently be **committed** to the baseline (that is, updated in the baseline, unlocked, and unstaged) when to make updated files available from the baseline.

Production areas can contain any number of baselines. Software development and resource utilization are often much more efficient if production areas are decomposed into multiple baselines of limited size. The limitation to such decomposition is that it must be possible to build baselines in sequence. All build steps must be completed in one baseline before starting the first build step in the next. Products from one baseline build can be used by other baseline builds if they are published into the production area.

Most of the time, developers interact with a single baseline at a time, through a particular staging area. A selected **current staging area** is included in the *localdev* context. It controls which staging area (and hence baseline) within the current production area is affected by *localdev* scripts.

Builds

Software can be built within any staging area. A **build** process performs compilation and other actions in an orderly and automated fashion, producing all files necessary for release of a product. A build process is usually modularized to make source directories independent of each other, but many of the build rules are placed in a single centralized file within each baseline.

Build requirements can play an important role in determining how source directories are split among baselines and production areas. In general, source directories within a baseline can be built via a multiple-step process, wherein each build step is performed within each directory before proceeding to the next build step. Releases and builds generally apply to entire baselines, though developers can opt to rebuild portions of baselines, as desired, to enhance efficiency.

Builds within different baselines are performed separately, in a sequence determined by production area configuration. If the products of one baseline build are referred to by a subsequent baseline build, then the first build must have published its products into the '`built`' directory within the production area, so that the subsequent build can find them. For testing purposes, build products that are part of the runtime system are installed into a runtime hierarchy within an '`installed`' directory in the production area. These products can be used directly for testing and product release, or can be **imported** by other production areas.

The *localdev* environment does not prescribe a particular build process. Doing so would be too constraining. The build process must be designed with consideration for any Configuration Management (CM) system that may be in use. A build process must function independently of a CM system if it is to be able to exist outside of the CM system at all. It is highly desirable from a reliability standpoint for a CM system to precisely match the build process that exists external to it. The *localdev* environment provides facilities that simplify the design and use of a build process, while placing few constraints upon it. Detailed consideration of build the process design is beyond the scope of this document.

Baseline and Production States

Each baseline within each production area typically cycles through a number of **baseline states** that determine what types of operations can be performed upon its files. Controlled procedures like builds and releases, for example, can usually only run when a baseline is protected in a way that prevents certain types of development activity. Use of a number of explicit production states is a simple means of implementing such protection. These states aid developers by making it easier to coordinate their actions with others.

The *localdev* environment supports baseline states, and also supports a master **production area state** that acts globally upon all baselines within a production area.

Tools within a production-oriented development environment, such as *localdev*, must be cognizant of baseline and production area states. Such tools must enforce appropriate use of production areas, thereby providing more reliable coordination of activities than can be achieved through adherence to conventions by individual developers. The *localdev* environment supports a fixed number of generic baseline and production area states. See *Baseline and Production States* on page 5 for a description of supported states.

Testing and Relocatability

Each production area typically has at least one staging area reserved for group use. Such **master staging areas** are where integration takes place. Once a product is built in this area, it can be **installed** into a test area. Although it is sometimes possible to test software as it resides in a staging area, it is usually desirable to be able to install products into a runtime configuration within a test area that is free of source code and software build artifacts. This allows testing to closely emulate an operational environment. It also facilitates integration of products from multiple baselines, for they can easily install into the same area. The *localdev* environment places an '`installed`' directory within each production area to serve as such a test area.

Software is usually developed so that its runtime location is configurable. This **relocatability** offers the advantage of making it possible to install multiple versions of a product on a single machine. Any particular version can then be accessed and/or executed at will. A tested version of relocatable software can be taken directly from a trial area, placed into an archive for distribution, and installed on a target system at a desired location. It can also be released into a Configuration Management system, if desired, which may have its own notion of location.

A similar relocatability is desirable in the location of baselines, staging areas, and the entire development environment. The *localdev* environment assumes that the locations of baselines, staging areas, and production areas are arbitrary. Software can thus be tested within *localdev* production areas only if it is relocatable. Non-relocatable software must be installed in a compatible target location before testing.

Production Areas

Production Area Structure

Figure 2 shows the file system structure of *localdev* production areas. An *owner account* is set up with ownership of a number of *production area directories*, each uniquely named within one or more shared *hub directories*. This shared portion of each production area is augmented with private extensions (*spoke directories*) for individual developers. Spoke directories contain product directories for each production area that a developer works in. Developers create these private production work areas, as needed, for their exclusive use.

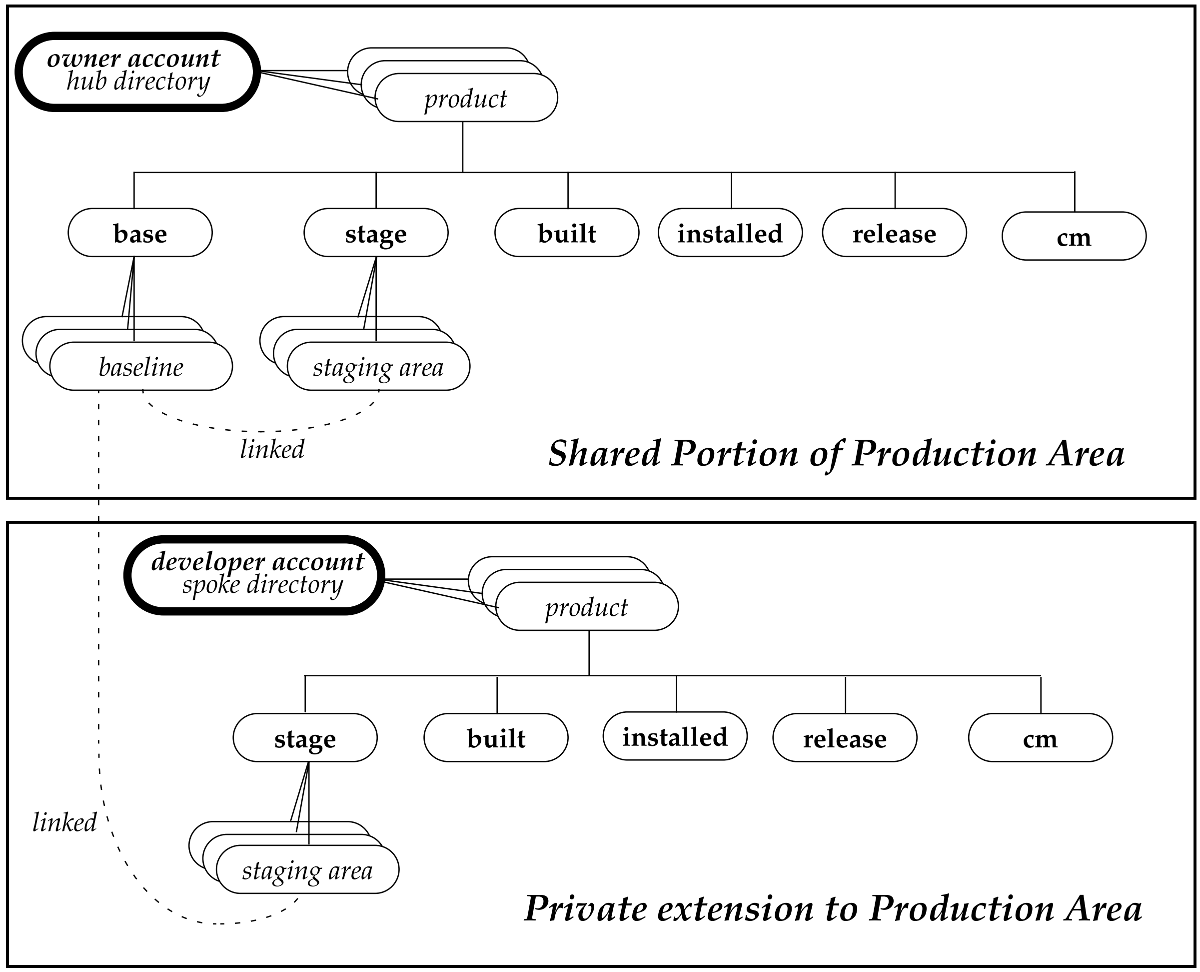


Figure 2. Production Area Structure

A number of configuration files and some standard directories are contained within each production area. Configuration files are maintained via *localdev* scripts, and are thus not designed for direct manipulation by developers. Standard directories serve to organize production area files. The shared portion of each production area contains the following directories:

- A **base** directory contains all baselines in the production area. Files within baselines are source files that are kept under revision control.
- A similar **stage** directory contains master staging areas for each baseline. Integrated builds are periodically performed in these staging areas.
- A **built** directory provides a location within which build products are placed for reference by other baseline builds within this production area.
- An **installed** directory provides a location within which build products are installed for testing and export to other production areas.
- A **release** directory provides a location to store distributable release archives and associated

release files.

- A **cm** directory provides a place for CM work areas to reside. These might be useful for, examining or building file revisions fetched from a CM system.

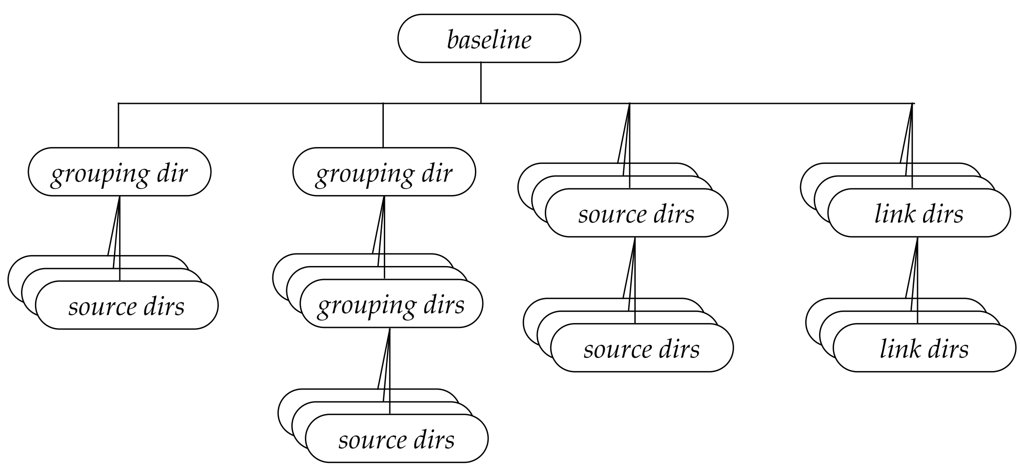
Private extensions to production areas replicate this production area layout, except that they contain no **base** directory. They rely on links from private staging areas to their baselines for access to baseline files. The same build and release processes performed within the shared portion of a production area can also be performed within its private extensions. This is, in fact, how such procedures are developed and tested.

Baseline Structure

Baseline structure is arbitrarily defined by production area owners. There are three types of directories within baselines, as shown in Figure 3.

- **Grouping directories** serve only to help organize a baseline, and contain only subdirectories.
- **Source directories** contain source files that are maintained under revision control. Source directories always contain '`RCS`' subdirectories.
- **Link directories** serve as locations within staging areas where files can be published during a build process, so that they become accessible to other source directories. Link directories within baselines contain no files, but may contain subdirectories. Link directories are identified by '`.link`' files that appear within them.

As indicated in Figure 3, the internal organization of a baseline is left undefined by the *localdev* environment. It is up to those maintaining the production area to define its structure.



Baseline organization is unrestricted, and usually determined by breakdown of software into design components.

Figure 3. Baseline Structure

Some programming languages place constraints on the arrangement of source directories. Java, for example, requires that its class package hierarchy correspond to the portion of the source directory hierarchy within which Java source files reside. Otherwise, the organization of source directories is arbitrary, and usually designed to represent a logical breakdown of source files into design components.

Supported States

A key aspect of a production area is the fact that it has a state, as does each baseline within it. These states assist developers to act as part of a coordinated group. State information can signify that it is inappropriate for some types of actions to occur. *Localdev* scripts check state information for developers before taking such actions, to ensure that the production area does not become corrupted.

The production area state can be considered a master switch that affects all baselines within it. Production area state overrides the states of individual baselines. Baseline states are set independently. A particular baseline state is relevant only when the current *localdev* context includes an associated staging area. Supported states are as follows:

Production Area States

normal	No restrictions - individual baseline states apply
locked	Baselines cannot be modified (unless forced by owner) - does not override **frozen** or **offline** baseline states
offline	Only baseline owners can access baselines - does not override **frozen** baseline states
frozen	Baselines cannot be modified by anyone

Baseline States

open	No restrictions
locked	Restricted state for build or other operation - no baseline changes (unless forced by owner)
stable	Stable state after a completed build - no baseline changes except creation of new source files (unless forced) - consistent set of build-generated files
recovery	Recovery after a failed build - no restrictions - files produced by a build are in an unknown state
offline	Only the baseline owner can access baseline files - special types of maintenance operations can occur
frozen	Baseline cannot be modified by anyone

Development within any particular baseline normally cycles through **open**, **locked**, and **stable** states, as shown in Figure 4. Periodically, a baseline is placed into **locked** state in order to perform a build within the integration staging area. This is often performed automatically by `cron` after-hours. The **locked** state prevents developers from making changes to the baseline that would corrupt the build.

If the build is successful, the baseline is placed into **stable** state. This state indicates that source files and build-generated files are all in a consistent state. This is the best time to synchronize staging areas and stage or import build-generated files.

At some point, often at a pre-arranged time of day via `cron`, baseline state changes from **stable** to **open**. Developers are now free to commit new file revisions, and they will not expect source files to be consistent with build-generated files. The cycle then repeats as the time for the next build approaches.

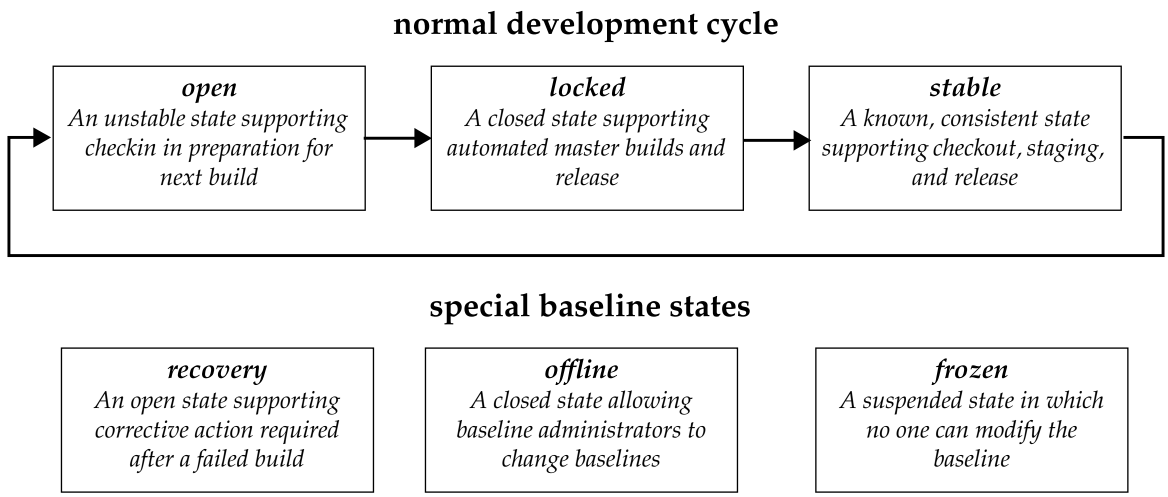


Figure 4. Baseline States

Special baseline states are provided to handle abnormal conditions that are known to frequently occur during the course of development. If a baseline's build fails, then it is placed into **recovery** state. This signals to developers that the build failed, and that they should not rely on build-generated files in the integration staging area. It also indicates that a post-build recovery procedure is needed or in progress. Such a procedure involves determining why a build failed and altering source files accordingly so that a subsequent build will succeed. Depending on the circumstances, a baseline owner may either run a new build in **locked** state after such a procedure, or may simply change the baseline state directly to **open**.

There are certain types of maintenance operations that can only reasonably be applied to a baseline when no one is accessing its files. Such operations include changes in baseline configuration and manipulation of baseline directories. An **offline** state is provided for this purpose. It allows only the baseline owner to access baseline files.

There are times when it is desirable to ensure that a baseline does not change. A baseline placed in **frozen** state is known by all developers to be unchanging. Even the baseline owner must first change the baseline's state before modifying it.

Installation and Setup

Accessing Scripts

The *localdev* package is installed within a single directory that will be referred to herein as `DV_HOME`. The executable files within the package (all Perl scripts) are located in '`DV_HOME/bin`'. Setting up your account to access *localdev* scripts is very simple.

Proceed through the following steps:

1. The *localdev* scripts work in conjunction with any UNIX shell. Use of the `tcsh` shell (a publicly available *csh-compatible* shell featuring command-line editing) has been found useful for achieving highest levels of productivity.
2. Ensure that version 5.003 or higher of `Perl` is accessible to your shell through the path `/usr/local/bin/perl`. You can check this by executing '`/usr/local/bin/perl -V`'. The scripts always access Perl via `/usr/local/bin/perl`. If necessary, have a symbolic link created there to support such access.You should get a response something like the following:

   % /usr/local/bin/perl -V

   ```
   Summary of my perl5 (5.0 patchlevel 3 subversion 0) configuration:
     Platform:
       osname=hpux, osver=10, archname=PA-RISC1.1
   ...
   ```

3. Place the '`DV_HOME/bin`' directory in your command-path. For a csh-compatible shell, you would place the following in your `.cshrc` file, where `DV_HOME` is replaced by the actual path of the `DV_HOME` directory, provided that the `.cshrc` file completely defines the path variable::

   ```
   # define other path components first, then...
   if (-d DV_HOME/bin) set path = ( DV_HOME/bin $path )
   ```

4. Source your initialization file in each existing shell you wish to use. New shells will automatically do this as they are invoked. You can then run '`dv-help -s`' to get a summary of *localdev* commands, and run any *localdev* command with a '`-h`' option to get usage information.

Now that you have access to *localdev* scripts, you can set up your environment, as explained in the following section.

Setting up your Environment

A few primary settings govern the scope of your *localdev* environment. These are set via the `dv-scope` script, and stored within a `.localdev` file within your home directory. Run '`dv-scope -m`' to modify these settings. You'll be asked to specify the following information:

Spoke Directory

The path to your private spoke directory controls where all of your private work areas for each production area that you use will reside. This might be set to something like

'`/home/`*machine*`/`*account*`/spoke`' for a developer account. Owner accounts do not use this setting. (See *Setting up Owner Accounts* on page 17 for details.)

Hub List

This setting determines which production areas that *localdev* scripts allow you to select. In this sense, it defines the scope of your *localdev* environment. Actual access to production areas, which will be described shortly, is determined by your membership in the group owning each production area. The `dv-scope` script does not check the group ownership of any hubs you specify, but does require that they exist. Access control is handled by `dv-select`. (See *Selecting a Production Area*, below, for details.) The `dv-scope` script prompts you individually for each directory path that you want to place in your hub list. Each prompt displays the current entry in the list, if any, and allows you to do any of the following:

If no entries exist in the list:

- *specify* the current entry by entering an absolute pathname
- *end* the list by entering '`.`' (or by hitting `<Enter>` if '`.`' is the default)

If an entry already exists in the list, then you have additional options:

- *replace* the current entry by entering an absolute pathname
- *accept* this setting as is, by hitting `<Enter>`
  (*If '.' is shown as the default value, then this terminates the list.*)
- *delete* this setting by entering '`-`'
- *insert* a path before this setting by entering '`+`'

A hub might have a path something like '`/home/`*machine*`/`*owneracct*`/`*project*Hub'. You can run `dv-scope` without options at any time to display your current settings. A sample session is shown below.

Note that the interface by which `dv-scope` settings are entered is **generic to all** *localdev* **scripts**. Any time configuration information is needed, it will be acquired through this interface. The interface is designed to be easy to use, always giving you the ability to hit `<Enter>` to accept defaults of current settings.

```
% dv-scope -m
Define Scope for a Personal Account...
------------------------------------------------------------

Property: Spoke Path
----------------------------------------------------------------------
Your spoke directory is where all of your private staging areas will reside.

Specify its path by specifying an absolute path that is accessible network-wide
from all relevant development machines.
The specified value must be of the form: /dir[/dir]...
>> Hit <Enter> to accept the shown value.
Selection? []: /home/quartz/scotty/spoke
```

```
Property: Accessible Hub List
----------------------------------------------------------------------
The scope of your localdev environment is defined by a list of
hub directories containing production areas that you want to work in.

Specify each as an absolute path that is accessible network-wide
from all development machines. Each must be of the form:  /dir[/dir]...
>> Hit <Enter> to accept a shown value, if shown,
   or to terminate (if '.' is shown).
>> Enter a value to replace a shown value, if any, or to append to the list.
>> Enter '+' to insert a value before a shown value.
>> Enter '-' to delete a shown value.
>> Enter '.' to terminate the list.
Selection? [.]: /home/maya/pcscm/linuxdev
Selection? [.]: /home/nymph/rdscm/newdev
Selection? [.]:

SCOPE-RELATED PROPERTIES
------------------------------
Personal Account
Spoke Path = '/home/quartz/scotty/spoke'
Accessible Hub List =
   '/home/maya/pcscm/linuxdev'
   '/home/nymph/rdscm/newdev'
```

Selecting a Production Area

Assuming that you've completed the setup described above and that production areas of interest have already been created, then you are ready to select a current production area and baseline within which to work. If you are using n owner account and want to create a production area, then skip to the next section (*Setting up Owner Accounts* on page 17). Run `dv-scope` without arguments to see what is available, as shown below.

```
% dv-scope
AVAILABLE PRODUCTION AREAS
------------------------------
   PRODUCTION AREA      OWNER      GROUP      STATE
/home/maya/pcscm/linuxdev:
   prism1.0            pcscm    pcstaff   normal
   prism1.1            pcscm    pcstaff   normal
/home/nymph/rdscm/newdev:
   jewel               rdscm    rdstaff   normal
   stardust            rdscm    rdstaff   normal

SCOPE-RELATED PROPERTIES
------------------------------
Personal Account
Spoke Path = '/home/quartz/scotty/spoke'
Accessible Hub List =
   '/home/maya/pcscm/linuxdev'
   '/home/nymph/rdscm/newdev'
```

At this point, you must change the group ownership of your shell session to match the group ownership of your selected production area. Note that the `dv-select` script, if run without options, lists all available production areas within *localdev* scope (as defined by `dv-scope`), including their owners and group owners. The following UNIX commands may be helpful:

- `id -gn` indicates your shell's current group name
- `groups` indicates the groups of which you are a member
- `newgrp` *name* sets your shell's current group to *name*

Once you've set the proper group, `dv-select` will allow you to select any production area owned by that group. Select a production area by specifying the product name, as in '`dv-select -P` *product*'. This will create a private extension to the production area in your spoke directory, if one does not already exist.

```
% dv-select -P stardust
Creating a product area extension...

MODIFIED SELECTIONS
----------------------------------------
Current Production Area = 'stardust'
Current Staging Area = ''
RCS Access Switch = 'off'
```

You can now run `dv-select` without arguments to see which staging areas are available to you. The example below shows three baselines, none of which have staging areas created for them.

```
% dv-select
CURRENT SELECTIONS
----------------------------------------
Current Production Area = 'stardust'
Current Staging Area = ''
RCS Access Switch = 'off'

AVAILABLE STAGING AREAS (listed by baseline)
----------------------------------------
   STAGING AREA         BASELINE STATE
commlib:                stable
   none available
receptor:               open
   none available
interface:              offline
   none available
```

If you have already created a staging area for the baseline you want to work in, then you can select it at the same time, as in '`dv-select -P` *product stagearea*'. Otherwise, you need to create a staging area using `dv-mkstaging`. If you do not specify a staging area name (as in '`dv-mkstaging` *baseline*'), then your staging area is named after its baseline. An invocation like '`dv-mkstaging -N` *stagename baseline*' creates a staging area with a custom name. This is useful when you want multiple staging areas for a single baseline. Each staging area must have a name that is unique within its production area. When you create a staging area, it is automatically *selected* for you (placing it in the current context).

```
% dv-mkstaging receptor
Staging Area Property Overrides...
------------------------------------------------------------

Do you wish to override the 'Staging Area Location' property
for this staging area? (y/n) [n]:
...
Staging area 'receptor' was created (for baseline receptor)
New staging area 'receptor' is now selected as current.
Run 'dv-sync' to synchronize it with its baseline.
```

Once you have selected a production area and associated staging area, you are free to use any of the *localdev* scripts. They will apply their actions to the current production area and associated staging area only, and you will not have to worry about inadvertently affecting files in other areas. Note that you will need to change the *current directory* to somewhere within your current staging area in order for many of the scripts to work. If you are out of position, then the scripts will fail gracefully with an error message.

When production areas and associated baselines are created, they are configured using `dv-config` to have a set of default properties that will usually be appropriate for use by all developers. You are free, however, to override these defaults within your own staging areas as desired, simply by invoking `dv-config`. (See *Configuring Production Areas* on page 19 for details.)

Setting Up Multiple Contexts

The setup procedure described in the previous two sections defines a *single context*. With a single defined context, all development is focused on a single staging area at any time. While you can change this context using `dv-select`, doing so will change your context within all of your windows and shells. In some cases it is convenient to predefine a number of different contexts, so that you can actually assign each window or shell to a desired context and work in all of them simultaneously. Within *localdev*, this is referred to as **multi-tasking**.

The procedure for setting up a multi-tasking environment is simple. A `LOCALDEV_TASK` environment variable controls which task a shell (or window) is associated with. Your default task has an undefined value of `LOCALDEV_TASK`, and saves context information into a `.localdev` file in your home directory. If `LOCALDEV_TASK` is defined, then contect information is saved into a `.localdev_`*task* file, where task is the value of `LOCALDEV_TASK`.

The setup procedure within each task is identical to that described above for defining a single context. You simply define `LOCALDEV_TASK`, and then run through the setup. You will notice that the output from `dv-scope` and `dv-select` each indicate the name of the current task.

Once each task is set up, you can simply set `LOCALDEV_TASK` to access that task from a particular shell. If you want to get fancy, you can create aliases that set this variable for you. For example:

```
alias stardust `setenv LOCALDEV_TASK stardust'
```

Advanced Shell Initialization

The *localdev* environment cannot set environment variables for you.[1] It is often desirable, especially for builds, to be able to set variables to context-specific values. At present, without *localdev* build support, *build variables* must be set within the shell environment outside of *localdev*.

It is typically beneficial to set up a more elaborate means of changing shell context that affects your shell environment as well as your *localdev* environment. One way to do this is described below. It involves setting up the localdev multi-tasking environment, as described above, and then creating both an initialization script and a shell alias for each task.

The example below assumes that two tasks (prism and commlib) have been set up. You could do the same thing without defining tasks, but then dv-select would generate output each time you actually switched contexts. Tasks allow you to switch context quietly.

Changing Shell Environment Context

1. Set up each task using `dv-scope` and `dv-select`.
2. Place something like the following in your `.cshrc` file. These lines define an alias for each task that invokes your .cshrc file with the appropriate task set. It also calls one of a number of task-specific scripts within '~/lib' to actually perform task initialization. This specific example has a few extra bells and whistles, but you can trim it down to the essentials if you'd like. Note that, again, `DV_HOME` should be replaced by the directory where *localdev* is installed.

```
###-----------------------------------------------------------------------
### Configure build environment (including localdev)
### Use a LOCALDEV_TASK environment variable and a collection of
### aliases to control build variable settings by sourcing external
### ~/lib/*.env files. Note that this should be done ONLY for
### interactive shells.
###-----------------------------------------------------------------------
if (-d DV_HOME) then
    ### Set default LOCALDEV_TASK value here
    if ($?LOCALDEV_TASK == 0) setenv LOCALDEV_TASK ""
    if ("$LOCALDEV_TASK" == "") setenv LOCALDEV_TASK "prism"

    ### Setup access to localdev environment.
    if (-d DV_HOME/bin) set path = ( DV_HOME/bin $path )
    alias tostage 'cd `dv-path -s`'
    alias tobase 'cd `dv-path -B`'

    ### Define aliases for build configurations.
    alias prism 'setenv LOCALDEV_TASK prism; source ~/.cshrc'
    alias commlib 'setenv LOCALDEV_TASK commlib; source ~/.cshrc'

    ### Select localdev task and initialize build variables.
    if (-r $HOME/lib/$LOCALDEV_TASK.env) source $HOME/lib/$LOCALDEV_TASK.env

    ### Override prompt here, if desired.
    set prompt = "${LOCALHOST}{${LOCALDEV_TASK}}: "
endif
```

3. Create each of the task-specific initialization files, which are named in this example as '`~/lib/$LOCALDEV_TASK.env`'. Note that having the value of `LOCALDEV_TASK` in the name is essential to invoking the right script without using an explicit switch in the `.cshrc` file.

---

1. The *parent* shell that accepts your *localdev* command invokes a *child* shell to actually run the Perl script. Your parent shell passes its environment to this child shell, but the child shell has no way of affecting the environment of its parent shell. Other commands you invoke from the parent shell, like `make`, would thus not benefit from *localdev* settings. Other *localdev* scripts acquire the settings from `.localdev` (or `.localdev_`*task*) files as they start up, thus getting around this limitation.

```
#---------------------------------------------------------------------------
# prism.env   Sample environment file for prism project
# Each LOCALDEV_TASK.env file must set/unset the exact same collection of
# variables.
#---------------------------------------------------------------------------
# Select localdev task (redundant, but useful for standalone invocation)
setenv LOCALDEV_TASK prism

if (-r $HOME/.localdev_prism) then
    # Build variable setup
    setenv WHATEVER_PATH `/usr/local/whatever`
endif
```

Creating and Maintaining Production Areas

Setting up Owner Accounts

Owner accounts are given ownership of shared production areas so that anyone given the responsibility of administering them can login to the account to do so. Administrators responsible for software configuration management for particular projects are given access to these accounts. These accounts are NOT group accounts. The *localdev* package tacitly assumes that only one owner is working on a particular production area at any given time. Localdev multi-tasking can be effectively used to deal with situations where more than one person must work with such an account.

Group ownership of each production area controls who has the ability to select that production area, and thereby controls who can perform development within it. By default, read access to production areas is granted to all accounts, though they can be configured (during creation or via `dv-config`) to be readable only by its group. Files within all baselines in a production area are always readable and writable to group members.

An owner account environment is set up to access *localdev* scripts in the same way as are individual accounts. Use of the `dv-scope` script differs in that the '`-g`' form is used. This causes only the hub list to be prompted for, but requires that this list contain all hub directories owned by the owner account. An owner account does not have a *spoke* directory. It instead places staging areas within the '`stage`' directories in its shared production areas. The `dv-scope` script otherwise behaves for an owner account just as it does for developer accounts.

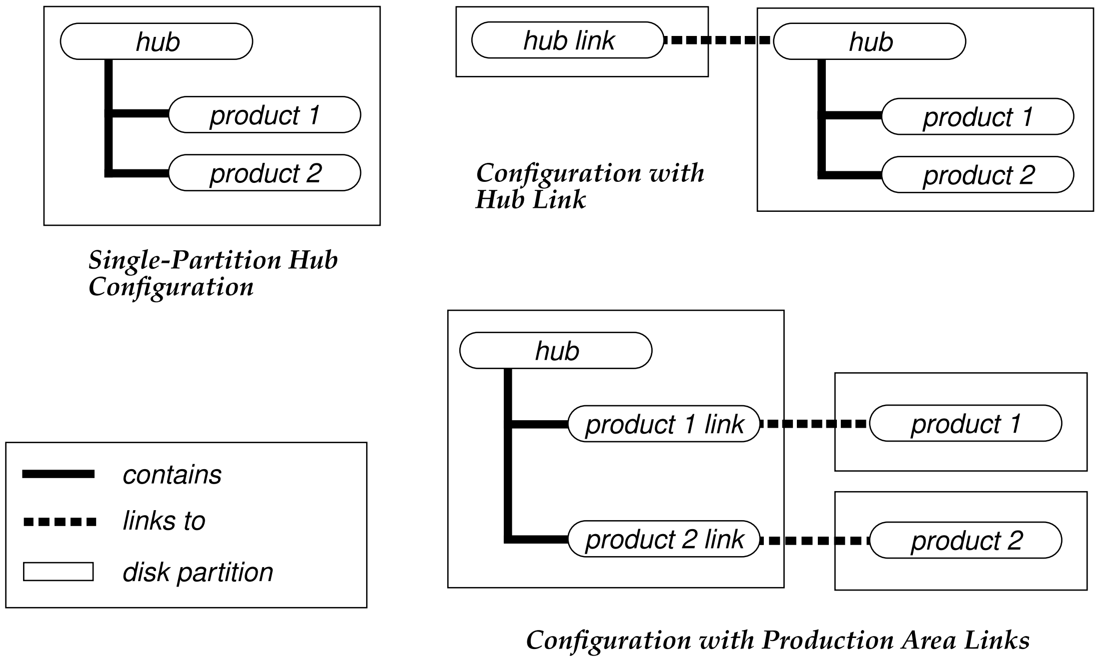


Figure 5. Use of Symbolic Links

Hub directories can exist in arbitrary locations, and can be accessed through symbolic links, if desired, as shown in Figure 5. Such links preserve the integrity of the environment, while allowing more flexible utilization of disk space. The `dv-scope` script, when used with the '`-g`' option, will attempt to create each hub directory if it does not already exist. Such a hub directory can then be relocated if the original location is replaced with a symbolic link that points to the new location. The path through which baselines are referred to thereby remains unchanged.

Individual production areas can similarly be located in arbitrary locations, as shown in Figure 5. In this case, `dv-mkarea` prompts for a non-default location. If one is specified, then the symbolic link is created automatically. Staging areas can also be created in non-default locations (via `dv-mkstaging`). The *localdev* environment has been designed to allow symbolic links to exist in these places without impacting everyone's *localdev* configurations. Hubs, production areas, and staging areas can be manually moved after creation, though one must be sure to preserve file permissions and ownership. *Dot* files must be included in the copy (use of `tar` is recommended).

Creating Production Areas

Once a hub has been created, the `dv-mkarea` script can be used to create a production area within it. An invocation such as '`dv-mkarea` *hub product*' will use a default configuration for the production area.

It is important that the path used to specify the hub be the same network-wide path that all developers will use to access the hub.

If the path contains elements that are only accessible locally, even if equivalent to a network-wide path, then developers may not be able to create staging areas for production area baselines within the hub. In the example below, a production area is created with default configurable properties.

```
% dv-mkarea /home/maya/rdscm/newdev stardust
Define Production Area Properties...
------------------------------------------------------------
Do you want to override any default properties? (y/n) [n]

Creating production areas...
Do you want a non-default location for the 'stardust' production area? (y/n) [n]
\/home/maya/rdscm/newdev...
   stardust:    production area created
```

Before creating many production areas, you may want to experiment with production area configuration, as described below. Creation of a custom configuration template can save time by automatically customizing each production area as it is created. Once such a template is created, an invocation such as '`dv-mkarea -t` *configname product hub*' will create a production area using your custom configuration.

Note that production areas must be uniquely named across all hubs that a developer will access. It is desirable that product names be meaningful, so that their purpose can be easily deduced from their names. If you plan to eventually have production areas for multiple versions of a product, then it is desirable to place distinguishing version information in the product name.

Configuring Production Areas

The `dv-config` script is used both to create production area configuration templates and to alter the configurations of existing production areas. Once you have created a template, it is simple to update existing production areas and to create new ones that share the template's configuration. You can get an up-to-date description of all configurable properties by invoking '`dv-config -D`'. Those pertaining to production areas are described below.

Property: Public Accessibility Boolean

The file access permissions associated with production areas can be set so as to either grant or disallow read access to accounts outside the owning group. A true setting grants such access.

Property: Source File Disclaimer

Headings generated within source files carry a customizable disclaimer. This is usually a short statement designed to protect the legal interests of the organization that developed the files. A disclaimer for use within the current production area is selected by name. Disclaimer files reside in the '`DV_HOME/config/disclaimers`' directory. See the `README` file therein for more information about disclaimers.

Property: Checks On Commit

When text files are committed (or newly introduced) into baselines, a number of tests can be performed to verify that they meet certain minimum legibility criteria. The following checks are supported:

tabs	denies commit of files that have unexpanded tabs (except as required in special files, makefiles, and shell script quotes)
returns	denies commit of files having carriage returns at the ends of lines
terminated	denies commit of files having an unterminated line
content	denies commit of empty files

lines	denies commit of files having line lengths greater than 80 characters
headings	denies commit of files not having standardized headings

Property: Unique Directory Boolean

A production area can be configured to ensure that all directory names within a baseline are unique. This is generally advantageous. A true setting activates the check when new baseline directories are created.

Property: Request Descriptions Boolean

The underlying RCS system is capable of maintaining a file description for each created baseline file. This option controls whether or not such information is requested when a new file is committed to a baseline. A true setting activates file description prompts for each new baseline file that it creates.

Property: File Stamping On Commit

The *localdev* environment supports the notion of '*stamping*' file revisions as they are committed. In some environments, this is a desirable means of tracking file modifications. Each stamp keyword represents a trackable task or change order, which is to be associated with all files bearing that particular stamp. Internally, *localdev* does not actually make such associations, nor does it maintain a list of available stamp keywords. It does, however, have the ability to work in conjunction with an external CM system to perform such tasks.

This setting controls whether `dv-commit` requires or allows a stamp option to be used when committing files. This is controlled by the first word of the setting, which can be set to '`required`', '`allowed`', or '`disabled`'. The second word of the setting (an accessory name) controls whether or not an accessory is used to validate stamps, which would ensure that only valid stamps are created, and only by authorized developers. Such validation can occur only on machines that have access to an accessory's CM system.

Note that if you do validate stamping it is recommended that you organize authorization for stamping such that all developers likely to modify a particular set of files are authorized to use all stamps that must be associated with those files. This makes handling of multiply-stamped files much easier.

Property: Stamp Pattern

Production areas may be configured so as to allow or require that committed revisions be associated with some sort of label, called a *stamp*, for configuration management purposes (see *Property: File Stamping On Commit*, above). This property defines an optional pattern that restricts the syntax of commit stamps. This is useful as an efficient means of performing rudimentary validation without access to a CM system. The pattern is defined using Perl regular expression syntax. This consists of a sequence of atoms, each of which may be single characters or meta-characters having special meanings. Some of these are listed below:

\(	an escaped punctuation character always represents itself
A	an alphanumeric character represents itself
.	represents any character
\w	represents any alphanuumeric
\d	represents any digit
\s	represents any whitespace character
*****	represents zero or more occurrences of the preceding atom
+	represents one or more occurrences of the preceding atom
?	represents zero or one occurrences of the preceding atom

[ab]	represents any of the characters within the braces
[^ab]	represents any characters except those within the braces
(...)	groups contents into a single atom
(a|b)	represents a or b, where a and b can be any atom
^	beginnning of string
$	end of string

A pattern defining `DR`### or `DCS`### stamps, for example, would be '`^D(R|(CS))†+$`'. If no pattern is specified, then there are only basic restrictions on stamp syntax (a series of alphanumerics, possibly including '-', and '_').

Property: Stamp Choices

Production areas may be configured so as to allow or require that committed revisions be associated with some sort of label, called a *stamp*, for configuration management purposes (see *Property: File Stamping On Commit*, above). This property defines an optional array of choices that can be used to limit stamps to a predefined list of strings.

Property: Stamp Prompt

Production areas may be configured so as to allow or require that committed revisions be associated with some sort of label, called a *stamp*, for configuration management purposes (see *Property: File Stamping On Commit*, above). This property defines a custom prompt that is used by `dv-commit` to request a stamp. If left unspecified, then a default prompt is generated. The default prompt is generic, however, and cannot convey the context or purpose associated with stamping in a particular production area.

Property: Ignored Patterns

Some scripts examine filenames to determine which files within a source directory can be committed to a baseline. This property provides a list of patterns that are used to ignore all files that ARE NOT considered capable of being committed. These are typically files that are created as part of the build process within a staging area. Binary executable files and all files that are not 'plain' files are AUTOMATICALLY ignored, so you need not include patterns to explicitly ignore them. Pattern syntax is identical to that accepted by Perl (see '`dv-config -P`' for details).

Property: Source File Types

The syntax of generated headings is customized based upon a source file type. Each such '*source type*' is associated with a '*source template*' that contains the information defining that type's characteristics. This property allows you to select the file types that you want to be able to generate headings for within the current production area. Each file type is specified as follows:

```
keyword    templatePath    description
```

The first word is a '*source type*' keyword, the name by which the type is known to developers. The second word is the path of the source type template (relative to '`DV_HOME/config/sourceTypes`') that you want to associate with it. The remainder of the line, consisting of one or more words, is a description of the file type.

In order to facilitate file migration and update, you may want to define outdated or foreign file types as '*aliases*' for current file types. This is done by defining them with description='`alias for` *currenttype*'. This allows scripts to update files that have been created with outdated file types, giving them each a currently configured file type.

Creating Baselines

A baseline represents a set of source files that is built as an independent entity. Baselines builds never depend upon each other directly, but can depend upon products from other baselines that have been installed into the current production area's '`built`' directory. Each baseline must be able to build independently using only its own source files and the products available in the '`built`' directory.

Splitting a product into multiple baselines can significantly reduce the amount of time it takes to synchronize and build any single baseline. One can also save disk space by requiring developers to build only the portion of the project that they are working on. Products that exist in the shared '`built`' and/or '`installed`' directories can easily be copied to a private work area using `dv-stage`. It is likely that most developers would be able to use `dv-stage` to acquire at least a few products from a master build, rather than rebuilding them all from scratch.

Use the `dv-mkbase` command to create a baseline. Realize that baseline names need to be unique only within the context of their production area. It is thus usually unnecessary (and undesirable) to burden them with redundant version or other information that is already contained in production area names.

```
% dv-mkbase receptor
Baseline Property Overrides...
------------------------------------------------------------

Do you wish to override the 'Checks On Commit' property
for these baselines? (y/n) [n]:

Do you wish to override the 'Unique Directory Boolean' property
for these baselines? (y/n) [n]:

Do you wish to override the 'Request Descriptions Boolean' property
for these baselines? (y/n) [n]:
/home/maya/rdscm/newdev/stardust...
   receptor:    baseline created
```

There are a variety of production area properties that affect baselines. It is usually easiest to configure these before creating a baseline, though you can do so at any time. Production area configuration applies globally to all contained baselines, but some properties can be overridden by configuration at baseline and staging area levels. Use the `dv-config` script with an appropriate option ('`-P`' for production area, '`-b`' for baseline, and '`-s`' for staging area) to configure production areas, baselines, or staging areas. You will be prompted for all relevant property settings.

When a baseline is first created, it is placed in **offline** state, so that only you (as the production area owner) can access it. This gives you the opportunity to alter its configuration, and to create directories and files within it before anyone else accesses it. Most privileged actions that significantly change baselines must be performed with affected baselines in offline state. This includes any subsequent use of `dv-config` to change production area or baseline configuration.You can use `dv-state` to easily change baseline state at any time, or simply to examine the current state. Before doing so, however, you'll need to create and select a staging area for the baseline (see *Selecting a Production Area* regarding use of `dv-select` and `dv-mkstaging`).

Note that production area state overrides baseline state unless production area state is set to **normal**.

```
% dv-state
The state of baseline 'receptor' is 'offline'.
```

```
% dv-state open
The state of baseline 'receptor' has been set to 'open'.
Production area state (offline) overrides.
```

```
% dv-state -P open
The state of the current production area has been set to 'normal'.
```

```
% dv-state
The state of baseline 'receptor' is 'open'.
```

Populating Baselines

A comprehensive set of scripts is provided for creating and manipulating both directories and files within *localdev* baselines. There should be little or no need to perform such operations manually. The scripts perform many bookkeeping operations for you that would be manual operations if BCS/RCS were used directly. Take advantage of these capabilities. It is very easy to make a mistake or omission when manually manipulating baseline files, causing developers needless grief. The following scripts are provided:

dv-repair	validate baseline state
dv-download	download revisions to a child baseline
dv-import	import source directories into *localdev* baselines
dv-merge	merge revisions in parent and child baselines
dv-mkdir	create directories within the baseline that is linked to the current staging area
dv-mvdir	rename a directory within the baseline that is linked to the current staging area
dv-rmdir	remove directories within the baseline that is linked to the current staging area
dv-tag	tag revisions of baseline files
dv-upload	upload revisions to a parent baseline

Use of these scripts is detailed in their individual help pages. The `dv-import` script, in particular, is useful for importing source directories containing plain and/or RCS files. As indicated below, use of this script to import source directories involves several steps:

Importing source directories with dv-import:

1. Files are imported into the baseline that is linked to the current staging area. You can only *import* directories that do not already exist in the baseline. Dot files are not imported. Any`.unimportant` files you may have in source directories must be recreated within localdev using `dv-shelve`.Your shell's account and group must be set so that you are able to select the current (importing) staging area. Use `dv-state` to place the baseline offline, if it is not already offline.
2. Run `dv-import` to import source directories. Note that if importing a directory hierarchy, you can only specify the name of the top-most directory, as in:

   `dv-import -R` *externalPath baseDir*

   where *externalPath* is the path to the top-most external source directory and *baseDir* is the name of the top-most baseline directory where they are to be copied to.

3. Run `dv-repair` to ensure that all files imported by `dv-import` conform to *localdev* requirements. If not, perform manual adjustments until they do, as guided by `dv-repair` output.
4. Synchronize the current staging area with the baseline (via `dv-sync`). Check that the staging area looks as expected. If not, make adjustments as necessary until it does.
5. Place the baseline back on-line by changing its state appropriately (via `dv-state`).

The following section provides general information about how `dv-download`, `dv-merge`, `dv-tag`, and `dv-upload` are used to *migrate* (and merge) files between production areas.

Migrating Files between Production Areas

☛ ***NOTE: This section will soon be undergoing revision.***

In many cases, the need to migrate files between baselines can be avoided by placing shared source directories into an independent baseline, whose products can be imported into '`built`' or '`installed`' directories of other production areas (via `dv-import`). This avoids the necessity of maintaining multiple sets of source directories and the file-merging tasks associated with migration. In some cases, however, migration is unavoidable.

Migration of files from one baseline to another is a periodic process. The need for it arises when two projects share a common base of source code, but each customize or maintain it separately for their own purposes. Files are thus generally being simultaneously changed in each baseline. Migration thus requires an ability to merge updates from one baseline into files of the other.

A number of scripts are provided with *localdev* to support migration. A `dv-download` script is used to initially establish a relationship between an existing parent baseline and a child baseline that is to be populated with a shared set of files.

Unlike `dv-import`, which could also be used to import the source directories into the child baseline, `dv-download` tags each downloaded file with a transfer tag. On subsequent downloads, `dv-download` uses these tags to determine which revisions in the child baseline correspond to revisions in the parent baseline. From this information, `dv-download` can determine which files need to be merged, which files can simply be updated, and which files require no update at all.

When changes to a file in both baselines do need to be merged, a `dv-merge` script can often perform the merge automatically, and tag it in such a way that `dv-download` recognizes the merged revision. If `dv-merge` cannot reliably merge the revisions, then it does not attempt to do so. Instead, it simply indicates that a manual merge is required.

Access scripts are provided with *localdev* allowing direct access to underlying utilities that are useful for merging files (`dv--diff`, `dv--diff3`, `dv--rcsdiff`, `dv--cmp`, `dv--patch`). These may aid in performing a manual merge. Once a merged revision is created, a `dv-tag` script can be used to associate a special merge tag with the revision that `dv-download` will recognize.

Downloading is only half of the migration story. It is often also useful to be able to introduce changes that have been made within a child baseline back into its parent. A `dv-upload` script is provided for this purpose. It operates in much the same way as does `dv-download`. Uploads recognize the transfer tags generated by downloads, and vice versa, so that merges and updates need not be performed redundantly.

Transfer tags are an essential ingredient in making uploads and downloads efficient. Their presence makes it possible to avoid performing the same merge operations repeatedly, each time an upload or download is performed. These tags are generated in such a way that a child baseline can become the parent of yet another child baseline (a grandchild to the original parent). The grandchild baseline can upload either to its parent or directly to its grandparent. This flexibility is supported by propagating relevant transfer tags from generation to generation. While this is a useful feature, it is advisable to limit the number of generations involved to only a few, so that an excessive number of transfer tags do not need to be generated.

Other Administrative Tasks

A few additional scripts are provided to ease the task of maintaining production areas. The scripts perform many bookkeeping operations for you that would have been manual operations if UNIX, BCS, and RCS were used directly. Take advantage of these scripts, which are listed below.

dv-mvbase	rename a baseline in the current production area
dv-rmbase	remove baselines from the current production area
dv-sever	sever locks that others have placed on files
dv-shelve	make files or directories unimportant (a reversible operation)
dv-repair	scan baselines for inconsistencies and repair them

Use of these scripts is detailed in their individual help pages.

Developing Software

General Script Characteristics

All *localdev* scripts maintain an appropriate level of awareness of production area context and state. Scripts have names of the form **dv-xxxx** to make their association as a script set apparent. Scripts are presented below in the order they might typically be used. First, however, it is useful to note a few common characteristics that all scripts share:

Baseline States

frozen	When a baseline is in a **frozen** state, scripts that modify baseline files will notify you of the baseline state and abort. There is no way to override this behavior.
offline	When a baseline is in an **offline** state, scripts that access baseline files will work only for the owner of the baseline. Others are notified of the baseline state before the script aborts. Baseline owners place baselines in this state in order to perform certain types of privileged operations. There is no way to override this behavior.
locked	When a baseline is in a **locked** state, scripts that access baseline files will notify you of the baseline state and abort. A baseline is typically locked when integration builds are in progress. The baseline owner can override this behavior with the **force** option (see below), causing the script to operate normally.

stable When a baseline is in a **stable** state, scripts that modify baseline files will notify you of the baseline state and abort. The intent of this state is to provide access to a consistent set of files, preventing additional changes from being committed. You can override this behavior with the **force** option (see below), causing the script to operate normally.

recovery When a baseline is in a **recovery** state, this is an indication that a build has failed, and that efforts are underway to repair it. Although you have complete access to baseline files, you may want to limit changes only to those required to recover from the failure.

open When a baseline is in an **open** state, you are free to perform all normal operations on baseline files.

Common Options and Behavior

force (-f) When a baseline is in a **stable** state, scripts that modify baseline files will tell you so and abort. Developers can override this behavior with the **force** option, causing the script to operate normally.

noop (-n) Scripts that impact baseline files support a *noop* option. This option causes a script to inform you what would happen if executed normally, without actually doing so. It exits without taking any action.

confirm (-c) Scripts that impact baseline files support a *confirm* option. This option causes a script to first inform you what would happen if executed normally, and then prompts for confirmation before going ahead with execution.

interrupt (^C) Once a script is executing, it can be stopped immediately by issuing an interrupt request with a `control-C`. Recovery is performed on interrupt, so your files won't be left in abnormal states. No attempt is made, however, to roll back changes that have been completed before an interrupt occurs.

eligible files Most scripts that operate on files know what state the file needs to be in, in order for the operation to succeed. You can't lock a file that you have staged, for example. You must unstage it first, before locking it. Scripts filter specified file lists, determining which are eligible for the specified operation. Scripts, operate only upon eligible files, generating skip messages for the rest.

files... Most scripts that take lists of files as arguments expect files to be specified relative to the current directory. There is no restriction with regard to the number of files or directories involved, provided that both the current directory and all directories containing specified files exist in the current staging area. Files are specified relative to the current directory.

dirs... Most scripts that take lists of files as arguments allow you to specify a directory to represent all eligible files in the specified directory. As with files, directories must exist in the current staging area. Directories are specified relative to the current directory, and '.' is used to specify the current directory.

recursive (-R) Some scripts allow you not only to specify directory names to represent eligible files within it, but also provide a **recursive** option that causes a directory to represent eligible files within it and (recursively) within all of its subdirectories.

Creating Source Files

A number of customizable templates are provided to ease the process of creating new files. A `dv-create` script uses these templates to create new files with standardized file headings. An invocation such as '`dv-create cinc whatever.H`', for example, might create the file `whatever.H` with the following content:

```
//F>===================================<type=cinc>========================
// whatever.H
// RCS: $Revision$ $Date$ $Author$
//========================================DISCLAIMER========================
// This file has been developed by and/or for the U.S. government.
// As required by government regulations, this file is in the public
// domain, and is furnished "as is", without technical support, and with
// no warranty, express or implied, as to its usefulness for any purpose.
//F>>===================================description (optional)=============
//F>>>=====================================================================
#ifndef whatever_H
#define whatever_H

#ifdef IDENT_H
static const char* const whatever_H_Id =
  "$Id$";
#endif

class whatever
    {
    public:

    private:
    };

#endif // whatever_H
```

Notice that the standardized file heading consists of three parts:

- file identity
- disclaimer
- description

The *file identity* and *disclaimer* portions of the file heading are controlled by the script. The disclaimer is selected from among available disclaimers, based upon your current *localdev* configuration. The *description* portion of the heading is designed to contain whatever custom information you want to place in the heading to describe the file.

The first argument passed to `dv-create` is a file type, which is used to access an appropriate source file template. Information associated with each template controls characteristics such as comment delimiters and *shebang lines*[2] that should be associated with the heading. It also controls the entire content of new files following the heading, which usually provides just enough code to simplify the process of adhering to some simple coding conventions. The `dv-create` script skips files that already exist, generating an appropriate message.

A `dv-update` script can be used to update existing files with heading information. It examines the file to determine if it has a *localdev* heading already. If so, it updates it to match the current

2. A *shebang line* is an initial comment-line in a file that is interpreted in order to determine what kind of shell or interpreter should be used to execute it. On UNIX systems, these are of the form '#!path options'. The required exclamation point, which is often referred to as a 'bang', gave rise to the term shebang.

configuration, being sure to preserve the *description* portion of the heading. Otherwise, it simply prepends a new heading to the file. If the file had an unrecognizable (non-*localdev*) heading, then it is up to you (the developer) to place relevant portions of it into the *description* portion of the new heading and delete the rest. In either case, if the file exists, nothing past the file heading is modified. The `dv-update` script normally skips files that do not exist, generating an appropriate message. Both `dv-create` and `dv-update` operate only on writable, ASCII files.

A compatible, but much more sophisticated, script exists for use with Java files. The `dv-jdoc` script not only creates file headings, but provides capabilities that make it simple to maintain and extract internal documentation within/from Java source files. It can expand markers that you place in Java source files into extractable method headings. Both these headings and method signatures can be extracted by `dv-jdoc`. The standardized format for headings is simple, clean, and mostly free-format.

A `dv-commit` script is available for placing new files into a baseline. Files to be placed into the baseline must already exist (as writable files) in the current staging area. Invoking `dv-commit` places files into the baseline, and can either unstage them for you or leave them locked and ready for further modification. A '`-b`' option can be specified to add *binary* (non-ASCII) files to the baseline. Note that production area owners will have already created appropriate baseline directories, and may populate a baseline with source files via other means.

Maintaining Source Files

Tools have been provided with the *localdev* environment to simplify the process of maintaining source files. The `dv-update` script, for example, supports a *check* option ('`-c`') that looks for anomalies within source files. These anomalies, listed below, can make files illegible or different in appearance to different developers. While compilers do not often care about these types of anomalies, humans who need to view, print, or edit source files often care very much. Utilities used to merge and compare files, for example, can be much more effective if operating on anomaly-free files. You can improve the efficiency of your development team by removing them promptly when detected.

tabs Unexpanded tabs generally need not appear in source files, except within Makefiles, or occasionally within scripts. Those viewing or printing files may have tab settings that are inconsistent with those of a file's author, making the presence of tabs difficult to deal with. Provided the tab width is known, tabs can almost always be readily expanded without harming a file. The practice of placing dependencies on tabs within files (as `make` does), such that expanding them causes the file to fail, is by today's standards very poor programming practice. A `dv-expand` script is available for ridding files of this anomaly.

line wraps Line lengths greater than 80 characters are not necessary in modern programming languages. It is always possible to break long lines using some form of continuation mechanism. This anomaly must be eliminated manually with an editor.

last end-of-line Text files are generally assumed to have a *newline* or *carriage return* + *newline* combination at the end of each line, including the last one. Some applications that generate text files have bugs that cause them to fail to properly terminate the last line. This makes the files illegible to some utilities. A `dv-expand` script is available for ridding files of this anomaly.

control characters There is no need to place control characters in source files. Modern programming languages always provide some form of alternative mechanism, such as hexadecimal escape sequences, for expressing such characters. This anomaly must be eliminated manually with an editor.

Accessing Source Files

Source files are initially created in staging areas as symbolic links to baseline files. They can be viewed in this form, but cannot be modified. Source files must be *locked* for modification (via `dv-lock`) if modifications to them are to reach the baseline.

Developers can obtain local copies of source files without locking them via a `dv-stage` command. This *staging* of a file replaces a symbolic link with a local copy of a file, thereby shielding a staging area from outside change, even when a staging area is synchronized with its baseline (see *Synchronizing Staging Areas* on page 31). A `dv-unstage` script performs the inverse operation, causing a staged file to revert to a link. Such control over a staging area can be useful when developers are working on file changes not yet ready for integration.

Staged files are by default not writable, though it can be desirable to manually make them so. Files must be locked, however, before they can be modified in a way that will propagate changes back to the baseline.

The usual mechanism for modifying files is to:

- lock them with `dv-lock`
- test them locally in your staging area
- commit the revised files back into the baseline with `dv-commit`
  OR discard changes by reverting to the last-committed revision with `dv-revert`

```
% cd /home/quartz/scotty/spoke/stardust/stage/receptor/receiver
% dv-lock *.C
receiver...
   alpha.C:  SKIPPED (file is already locked by scotty)
   beta.C:   SKIPPED (file is staged)
   gamma.C:  locked latest revision
   omega.C:    locked latest revision
```

The `dv-commit` script prompts for a revision message, if a file has been modified. *It does not create a new revision unless the file has actually been modified.* Revision messages can be viewed by developers or production area administrators (via `dv--rlog` or `dv-list`) to aid in identifying and characterizing file revisions. Note that the existence of (RCS) revision messages makes it redundant to encumber file headings with revision history.

If `dv-commit` creates a new revision of a file (which happens if it has been modified), then it automatically updates the baseline. Any staging area associated with a production area that has not shielded itself by staging files is immediately impacted by committed files.

```
% cd /home/quartz/scotty/spoke/stardust/stage/receptor/receiver
% dv-commit *.C
Revision message for all files (terminate with a '.' line):
This would be a relevant message that briefly identifies the group of changes
being committed.
.
receiver...
   alpha.C:  lock discarded (file has not changed)
   beta.C:   SKIPPED (file is not writable in the current staging area)
   gamma.C:  committed new revision
   omega.C:  committed new revision
```

Note that one of the files in the above example was not commited ('`lock discarded`'). This happens when the locked file has not been changed, thus avoiding the creation of a redundant revision. The `dv-revert` command unlocks and *unstages* files, causing each to essentially revert to its last committed revision.

```
% cd /home/quartz/scotty/spoke/stardust/stage/receptor/receiver
% dv-revert *.C
receiver...
   alpha.C:  reverted to last-committed revision
   beta.C:   SKIPPED (file is not writable in the current staging area)
   gamma.C:  reverted to last-committed revision
   omega.C:  reverted to last-committed revision
```

Only locked files or non-baseline files are usually writable in a staging area. As shown in the above examples, scripts may skip processing of baseline files in a staging area that are detected as being in a state that is unsuitable to the requested operation.[3]

Examining Source Files

A `dv-list` script provides a simple mechanism for examining the status of staging area files. It can be instructed to:

- list files that you have locked
- list all files
- list all files that anyone has locked
- list all staged files
- list all files locked by a particular owner
- list revision history

By default, it operates on the current directory, but it can be made to operate on the entire staging area or any part thereof. Output looks similar to that shown below:

---

3. In some cases, such as when developing code for multiple platforms, it is desirable for a developer to create multiple staging areas for the same baseline. The *localdev* environment allows such activity, but offers no special support for it. If a file is locked (and hence staged) within one such staging area, it will be modifiable only in that staging area. Other staging areas will treat the file as if it were locked by someone else. There are ways to get around this by manually manipulating files. Doing so, however, can interfere with *localdev* script operation, leading to unpredictable results.

```
% cd /home/quartz/scotty/spoke/stardust/stage/receptor/receiver
% dv-list -a
Source file listing: All Files
receiver...
                 STATUS     VER OWNER         MODTIME
   alpha.C       locked     1.2 scotty   08/27/99 11:29:48
   beta.C        current    1.1 spock    06/02/99 13:09:40
   gamma.C       SHIELDED   1.5 ---      08/27/99 11:32:22
   omega.C       locked     1.1 ---      08/27/99 11:35:45
   receiver.H    private    --- ---      08/27/99 12:12:06
```

```
% cd /home/quartz/scotty/spoke/stardust/stage/receptor/receiver
% dv-list
Source file listing: Personally Locked Files
receiver...
                 STATUS     VER OWNER         MODTIME
   alpha.C       locked     1.2 scotty   08/27/99 11:29:48
   omega.C       locked     1.1 ---      08/27/99 11:35:45
```

```
% cd /home/quartz/scotty/spoke/stardust/stage/receptor/receiver
% dv-list -s
Source file listing: Staged Files
receiver...
                 STATUS     VER OWNER         MODTIME
   alpha.C       locked     1.2 scotty   08/27/99 11:29:48
   gamma.C       SHIELDED   1.5 ---      08/27/99 11:32:22
   omega.C       locked     1.1 ---      08/27/99 11:35:45
```

Synchronizing Staging Areas

A `dv-sync` script is provided to synchronize an existing staging area with its baseline. This action causes a staging area to be updated so as to reflect the current state of its baseline. Symbolic links are added, removed, or modified accordingly. New directories are created if necessary. Any files that are locked or staged are not affected by `dv-sync`.

Note that in cases where directories or staged files have been removed or moved in the baseline, dv-sync will not remove outdated files or directories. It will simply create directories and links for the new locations, as necessary, and inform you of the presence of outdated directories. It is incumbent on developers to remove or move such directories or files themselves. Moving them before `dv-sync` performs synchronization is usually most convenient.

Building Source Files

A `dv-build` script will eventually be available for building source files within production areas. This script will likely generate one or more shell scripts that can be called to perform builds either within or outside the localdev environment. The idea will be to use dv-build to infuse these files with build variables and source directory selections so as to control a build, while calling a standard `make` utility to actually perform the build.

Details regarding the way builds operate are deferred to a separate document. Suffice it to say that a single, unified build process involving the UNIX `make` utility will support both Java and traditional (C, C++. Perl, and others) builds.