OpenSolaris Service Management Facility Guide

Liane Praza

OpenSolaris Service Management Facility Guide

by Liane Praza

Published April 2007

Table of Contents

List of Tables

List of Examples

Preface

The Service Management Facility is a mechanism to define, deliver, and manage self-healing application services for the Solaris™ Operating System or other OpenSolaris™ works. The Service Management Facility (more compactly known as **smf(5)**) significantly augments the traditional **init.d(4)** and **inetd.conf(4)** models for service delivery. *The OpenSolaris Service Management Facility Guide* introduces important **smf(5)** concepts and provides details on delivering services which can be managed by the Service Management Facility.

Who Should Use This Book

If you have ever written or modified an **init.d(4)** script or **inetd.conf(4)** line, this book is for you. All administrators will also benefit from this book, as it expands significantly on the **smf(5)** administrative concepts introduced in *System Administration Guide: Basic Administration*.

How This Book Is Organized

XXX: need a definitive structure before writing this section. Remember to cross-link.

Related Information

- *System Administration Guide: Basic Administration*
- *Application Packaging Developer's Guide*
- *System Administration Guide: Solaris Containers -- Resource Management and Solaris Zones*
- OpenSolaris Service Management Facility Community: http://opensolaris.org/os/community/smf
- Predictive Self-Healing BigAdmin site: http://www.sun.com/bigadmin/content/selfheal

Chapter 1. Introduction

What it does

smf(5) promotes a service to a first-class operating system object. It allows administrators to:

- access information about misconfigured/misbehaving services,
- enable and disable services persistently across upgrades and patches,
- directly bind services to resource management configurations,
- delegate tasks to non-root users securely, and
- more reliably access the console in repair scenarios.

smf(5) gives developers:

- automated restart of services in dependency order due to administrative errors, software bugs, or uncorrectable hardware errors,
- a single API for service management, configuration, and observation, and
- simplified boot-process debugging.

Benefits of participation

Levels of Service Integration

No integration: compatibility. See the compatibility section.

Trivial integration: Create simple service manifest, convert init script to service methods, minimal testing

Full Restartability : Build on trivial conversion to split monolithic services -- each separately restartable component becomes its own service

Chapter 2. Service Concepts

Understanding a set of basic concepts will ease your interactions with **smf(5)** whether you're developing a complex suite of software or administering a Solaris system. These concepts apply to all services managed by **smf(5)**. There are some concepts which are specific to the type of service being implemented or managed, and subsequent chapters will cover the specifics of delivering individual service classes.

Service Model

The Service Management Facility defines a programming model for providing persistently running applications called *services*, and an extensive infrastructure for managing those services. An **smf(5)** service can represent a variety of software facilities, such as a single daemon, a set of running processes, a set of system configuration parameters, or even just a group of other services.

Adapting existing software to **smf(5)** brings a number of advantages. Services are automatically restarted if they fall victim to hardware failure, unexpected service failure, or administrative error. Participation in the service management facility also enhances observibility (with **svcs(1)** as well as future-planned GUI tools) and ease of management (with **svcadm(1M)** and other Solaris management tools). All that's required to adapt existing software to **smf(5)** is usually just the creation of a short XML file called a *service manifest* and a few simple modifications to the service `init.d(4)` script.

Service and Service Instance

Services are implemented as *service instances*. A service is the parent of one or more instances, and contains configuration information that is shared among all of the instances. The service should be considered only a configuration store, where the instance is the entity that executes on the system. The configuration of an instance is determined by composing configuration from the service and the instance. The instance's configuration will always be used for configuration defined both on the service and the instance.

All properties that would not be changed by a different copy of your service running (if your service supports that) should be defined at the service level. If your service may be implemented differently by a different instance (e.g. the `smtp` service may be implemented by `sendmail, postfix, qmail`, ...), properties that are specific to the current implementation should be specified at the instance, not the service.

Each service and service instance is named by a unique FMRI (Fault Managed Resource Identifier). The FMRI is prefixed with `svc:` and contains the service category, service and instance. For example, the FMRI for the default instance of the **fmd(1M)** service is `svc:/system/fmd:default`.

Property Groups and Properties

Service configuration is organized into a set of properties. These properties are typed to reduce incidence of administrative error. All properties are organized into property groups. The property group is where composition of service instances occurs. f a property group with a given name is defined both on the service and the instance, the instance's settings are used. As all configuration is stored as properties, a single interface can be used to view and modify any service configuration property.

While most properties groups are persistent, meaning that their settings are saved across system reboot, **smf(5)** also provides *non-persistent* property groups, which disappear when the system is rebooted. These non-persistent property groups are used for run-time information about services which needs to be re-generated when the system is rebooted.

A primary piece of configuration information is whether the service is enabled. This is also stored as a property. XXX?

Property Stability

Service Restarter and Delegation

In order to provide restart capabilities for services with different run-time characteristics, smf(5) allows for a variety of service models. The service model is implemented by a service restarter. The restarter sets up the environment for the service and starts, stops, restarts, and communicates configuration change to the service. Currently, these models are provided by the **svc.startd(1M)** and **inetd(1M)** restarters. Additional models may be provided in the future by either these restarters or by additional restarters. While this document describes the models for **svc.startd(1M)** and **inetd(1M)**, please also see the restarter documentation for more detail on the application model it provides.

XXX: specifics about delegation

Service Methods

A service's interface with its restarter is the service method. Each restarter defines the methods it requires for the services it manages. Most restarters minimally require a service provides a start method and a stop method.

Service Dependencies

Dependencies define the relationships between services. Dependency effects are tracked by the master restarter, **svc.startd(1M)**. When a service starts, fails, etc., **svc.startd(1M)** communicates the change to the service's restarter.

XXX -- more details about dependencies: restart_on and grouping

Service Repository

A service's configuration is kept in the service *repository*. The repository is a transactional database that contains the authoritative copy of service configuration. This configuration is used by the master restarter, **svc.startd(1M)**, the service's restarter, the service itself, and any management applications which wish to access information about the service. Once services have been delivered on a system, administrators or administrative applications may customize service settings in the repository.

The service repository has a number of characteristics which are important to the system.

- Transactional.

  All of the core **smf(5)** daemons are designed to be completely restartable if they fail due to hardware, software, or administrative failure. This requires a transactional backing store for all of our service information, including things like service state. If any core **smf(5)**daemons die halfway through an operation, they pick up where they left off when they return. Thus, the repository must be transactional to allow uto implement recoverability.

- Typed.

A strong typing system allows **smf(5)** to validate that configuration information is at least of the appropriate form. This reduces the chance of accidental misconfiguration by the administrator.

- Single point of access.

  All configuration and runtime data is accessed through a single API: **libscf(3LIB)**. This reduces the time required to write a management application for a service, and elimiates the need to write service-specific configuration parsers.

- Access control.

  A subset of configuration changes and administrative actions can be safely delegated to non-root users, without requiring that all changes and actions be allowed by those users.

  There are, however, no provisions for configuration data to be hidden from unprivileged users or applications. While modification is protected, reading is not. The repository should not be used for secret data.

- Snapshots.

  Allow administrators to easily revert to previous configuration versions.

- Checkable consistency.

  It must be simple to confirm on startup that the format of the system's configuration data is correct. Obvious filesystem corruption is flagged explicitly rather than parsed by higher-level applications as lack of or incorrect configuration.

- Fast

  Recovery algorithms require storing state in the repository, so updates must be fast for systems with many services.

The repository is split in implementation between the persistent properties which are not lost when the system restarts and non-persistent properties which are reset when the system reboots. The persistent database is critical for backup and is stored in `/etc/svc/repository.db`. The non-persistent database has no value in backups as its contents are regenerated every system boot. It is located in `/etc/svc/volatile/svc_nonpersist.db`.

Configuration Snapshots

Service Manifest

To deliver a service on a Solaris system, you create a service *manifest*, an XML file which describes a service and any instances associated with that service. The service manifest is imported into the repository either at boot time, or by using the **svccfg import** subcommand. The XML format of the service manifest is specified by the Service description DTD, located at `/usr/share/lib/xml/dtd/service_bundle.dtd.1`. The comments in the DTD will answer many questions about authoring service manifests.

Profiles

Milestones

A *milestone* service aggregates a set of service dependencies. Usually, a milestone service does nothing useful itself, but declares a specific state of system-readiness which other services can depend upon. One example is the `name-services` milestone service. The `name-services` milestone service is considered *online* as long as any name services which are enabled are running. Thus, **smf(5)** milestones are useful points for setting dependencies, as they reflect a specific state of system readiness. Milestone services are treated as normal services by **smf(5)**.

The standard Unix system run-levels are represented in **smf(5)** as milestone services. The `single-user`, `multi-user`, and `multi-user-server` milestone services correspond to run-levels S, 2, and 3, respectively.

smf(5) also allows you to reduce or increase the number of services running on the system by specifying. You can choose which milestone represents the set of services you want running, and ask to set the system milestone to the one you've selected. The **svcadm milestone**accepts the run-level milestone services, as well as the special `all` and `none` keywords. These aren't actual services, but shorthand for either no services, or all enabled services, respectively. This set of five special milestones can either be booted directly to (**boot -m milestone=**) or reached by running **svcadm milestone**.

A limited milestone (any special milestone but `all`) is reached by temporarily disabling all services which aren't required by the services defined in the milestone. If a service isn't a dependency of the milestone or one of the milestone's dependencies, it will be temporarily disabled when in that limited milestone. The `console-login` service, for example, is not a dependency of any of the reduced milestones and will always appear as disabled when in those milestones.

Reduced run-levels should still be reached by using **boot -s** or the **init(1M)** command directly. These commands set both the milestone and the run-level. Running **svcadm milestone** directly only sets the milestone, but not the system run-level.

Note

For system maintenance procedures, use the traditional **init(1M)** or **boot -s** invocations, *not* the **svcadm milestone** or **boot -m milestone** invocations. As the existing commands provide the same functionality they have in previous releases of the Solaris system, they offer the least incidence of surprise to the user when performing critical system maintenance.

Service States

Each service instance is always in a well-defined state. Define states XXX

Service states are stored in a non-persistent group, as the state must always be reset when the system reboots.

A Solaris service is only started if it is marked as enabled (by the administrator), and once all of its dependencies are satisfied.

Legacy Services

smf(5) provides start and stop of scripts placed in the `/etc/rc?.d/` directories. As these services aren't fully described to the system with a set of dependencies, they are called *legacy services*. Legacy services created for releases of the Solaris system older than 10 will continue to work without modification in almost all cases, starting when the system starts, and being shutdown during system shutdown or a change in the run-level. However, these services aren't monitored by **smf(5)** after they're started, and will never be automatically restarted by the system.

Chapter 3. Service Lifecycle

The service lifecycle describes

Determine service suitability

Not all software is appropriate for execution by **smf(5)**.

Write service methods

Write service manifest

Import manifest

Test/fix

Always look for errors or unusual messages in the service's log file, if it exists.

Package manifest and methods

Install service

Service startup/runtime

Start when...

Logfiles...

Configure/modify service

Upgrade service with no loss of configuration

Remove service

Chapter 4. smf(5)Commands

There is a rich **smf(5)** command set available too developers and administrators alike. The *Solaris System Administration Guide: Basic Administration* book focuses on administrators, while here we focus on commands a service developer might find valuable. First, an overview of general **smf(5)** commands.

Command overview

Table 4.1. smf(5) Commands

Command	Purpose
svcs(1)	List services, service information, discover and diagnose service problems.
svcadm(1M)	Perform general service administration.
svcprop(1)	Service information queries, suitable for scripting.
svccfg(1M)	Repository manipulation tool.

Service listings

Example 4.1. What services are enabled and running?

svcs(1) with no options answers that easily:

```
$ svcs
...
online         Feb_04   svc:/network/ntp:default
online         Feb_04   svc:/network/service:default
online         Feb_04   svc:/application/x11/xfs:default
online         Feb_04   svc:/application/font/stfsloader:default
```

Example 4.2. What services are available on the system?

Just ask **svcs(1)** to list all services, including the disabled ones:

```
$ svcs -a
disabled       Feb_04   svc:/system/metainit:default
disabled       Feb_04   svc:/network/rpc/nisplus:default
disabled       Feb_04   svc:/network/nis/server:default
```

Example 4.3. What services are provided on the system?

Again, just ask **svcs(1)**. This time, get the service description too:

```
$ svcs -a -o FMRI,DESC
svc:/milestone/name-services:default              name services milestone
svc:/platform/i86pc/kdmconfig:default             Display configuration
svc:/system/cron:default                          clock daemon (cron)
```

Example 4.4. How do I find out more about a specific service?

svcs(1M) gives more detailed information about a service with both the `-x` and `-l` options combined with the service name. The manpage references in **`svcs -x`** are particularly helpful.

```
$ svcs -x system-log
svc:/system/system-log:default (system log)
 State: online since Fri Feb 04 19:30:11 2005
   See: syslogd(1M)
   See: /var/svc/log/system-system-log:default.log
Impact: None.

$ svcs -l system-log
fmri         svc:/system/system-log:default
name         system log
enabled      true
state        online
next_state   none
state_time   Fri Feb 04 19:30:11 2005
logfile      /var/svc/log/system-system-log:default.log
restarter    svc:/system/svc/restarter:default
contract_id  51
dependency   require_all/none svc:/milestone/sysconfig (online)
dependency   require_all/none svc:/system/filesystem/local (online)
dependency   optional_all/none svc:/system/filesystem/autofs (online)
dependency   require_all/none svc:/milestone/name-services (online)
```

The **`svcs -p`** command allows you to determine which processes, if any, are in the service.

```
$ svcs -p system-log
STATE          STIME    FMRI
online         21:06:05 svc:/system/system-log:default
               21:06:05      272 syslogd
```

Additionally, **svcprop(1)** provides a dump of all service configuration. By default, it chooses to display the running configuration of the instance.

```
$ svcprop system-log
general/package astring SUNWcsr
general/enabled boolean true
general/single_instance boolean true
general/action_authorization string solaris.smf.manage.system-log
```

Chapter 5. Contracts and Service Restart

The Service Management Facility cooperates with the Solaris Fault Manager through *service contracts* to isolate and recover from hardware and software faults on the system. The Fault Manager essentially detects and predicts hardware faults, retiring bad hardware when it is appropriate and possible.

Hardware error handling before smf(5)

Earlier versions of the Solaris Operating System could often detect hardware faults, but couldn't usually recover from them without rebooting the system. For example, physical memory on the system can go bad. Depending on the type of memory and the type of error encountered, it can generate either a correctable error, or an uncorrectable one. The Solaris Operating System has always recovered gracefully from correctable errors. They're handled by the kernel and never seen by a user process.

But, uncorrectable ones mean that the system is unable find a good copy of the data. The error can occur either in the kernel's address space or in a user process's address space. An error in kernel address space means that the kernel is paniced immediately, restarting the system. An error in user space can be dealt with more gracefully. As we know which process the error affected, the kernel can kill it before it causes any more damage.

Prior to SMF, the relationships between user processes were unknown. As the system didn't know if the corrupted/absent memory in one process would cause corruption in another process which was cooperating very closely with the one that received the error, the entire system had to be gracefully shut down.

Hardware error handling within a service with smf(5)

Now **fmd(1M)** can take hardware that's about to have a failure offline in advance of that failure, or after that failure occurs. But, when a failure does slip through it is **smf(5)**'s job to know the relationships between processes/services on the system. There are two main types of relationships:

- processes part of the same service or fault boundary, and
- services which depend upon each other.

To track processes as part of the same service, the **smf(5)** restarters use the new kernel mechanism, **process(4)** *contracts* to group and monitor related processes. Certain types of events can be classified as important:

- `empty` - the last member of a process was killed
- `fork` - a new process was added to the contract
- `exit` - a member of the contract exited
- `core` - a process dumped core
- `signal` - a process received a fatal signal
- `hwerr` - a process was killed due to an uncorrectable hardware error

Each of these events is detected by the kernel, and passed to the contract owner. In the specific case of `hwerr`, if an uncorrectable hardware error does occur in a user process the kernel detects it and kills the process where the error occurs, just as in previous versions of the Solaris Operating System. With the introduction of SMF, we no longer need to restart the system - with `smf(5)` and **process(4)** contracts, we can just restart the "associated processes".

Contracts are written with three types of event sets: `informative`, `critical`, and `fatal`. Informative and critical only differ really in the guarantees about event delivery. `fatal` means all processes in the contract are killed if a fatal event is received. **smf(5)** puts the `hwerr` event into the `critical` event set. A few commands allow you to explore contract settings on the system. First, you can find out about contract and process relationships using:

```
$ ptree -c `pgrep sendmail`
  [process contract 1]
    1     /sbin/init
      [process contract 4]
        7     /lib/svc/bin/svc.startd
          [process contract 513]
            18676 /usr/lib/sendmail -Ac -q15m
            18678 /usr/lib/sendmail -bd -q15m
```

You can see that sendmail is in contract 513. Using that information, you can look at the terms of the contract:

```
ctstat -vi 513
     CTID    ZONEID  TYPE    STATE   HOLDER  EVENTS  QTIME   NTIME
     513     0       process owned   7       0       -       -
             cookie:                0x20
             informative event set: none
             critical event set:    hwerr empty
             fatal event set:       none
             parameter set:         inherit regent
             member processes:      18676 18678
             inherited contracts:   none
```

That output confirms what was described above: `hwerr` is in the `critical` event set. If there's a `hwerr` in either of the sendmail processes, the contract owner (7, **svc.startd(1M)** as you see above) will get a `critical` error. **svc.startd(1M)** then responds to the error by stopping the service, and restarting it if possible. Thus, when an uncorrectable memory error occurs in a process managed as an **smf(5)** service, **smf(5)** is able to detect an uncorrectable memory error in a process, and repair it by restarting the service.

Fault propagation between services with smf(5)

The previous section handles fault propagation within the first relationship type described above - processes related as part of the same service or fault boundary. Fault propagation between services are handled differently.

Service relationships are expressed by **smf(5)** dependencies. Most dependencies are used to specify startup order, by using `grouping=require_all` and `restart_on=none`. However, you can also specify that a service is restarted if its dependency experiences any type of error (hardware error, core dump, etc.). You do this by using `restart_on=error` as opposed to `none`. Then when the dependency is stopped due to that error, your dependent service will be too.

Fault handling for legacy services

Uncorrectable errors are handled differently for processes that aren't explicitly part of an `smf(5)` service. How does SMF know what to do if you didn't write a service manifest to describe how faults should be handled? Since **init(1M)** is in a **process(4)** contract, all processes are part of a **process(4)** contract. If no software creates a new contract, the process is in the same contract as its parent. The default terms for a contract are not the same as what **svc.startd(1M)** uses. Instead, the default **process(4)** contract is written such that hardware errors are fatal. Remember, that means all processes in the contract are killed if any process sees an uncorrectable memory error. **svc.startd(1M)** also helpfully puts each legacy-run service in its own contract. Thus, if any processes launched out of a legacy-run service (e.g. **vold** or **dtlogin**) fall victim to an uncorrectable memory error, all processes in the contract will be killed.

```
$ ptree -c `pgrep vold`
   [process contract 81]
     481   /usr/sbin/vold
   $ ctstat -vi 81
   CTID    ZONEID  TYPE    STATE   HOLDER  EVENTS  QTIME   NTIME
   81      0       process orphan  -       0       -       -
           cookie:                0
           informative event set: core signal
           critical event set:    hwerr empty
           fatal event set:       hwerr
           parameter set:         none
           member processes:      481
           inherited contracts:   none
```

Note that for vold's process, `hwerr` is in the fatal event set. But, since there's no service manifest to tell SMF how to deal with the legacy-run service, the system can't restart it. This is a primary reason why adapting your service to **smf(5)** is valuable, even though compatibility is provided for legacy services.

Handling faults within Zones

As a zone doesn't have a kernel of its own, an uncorrectable memory error in the kernel still means that the entire system goes down. However, each zone has its own copy of **smf(5)** inside which is completely separate from the other zones on the system. As **smf(5)** runs inside the zone as well, faults are handled inside the local zone the same was as they are in the global zone. There's no need to isolate the fault to the zone because we isolate the fault to a finer granularity -- the service. **smf(5)**and zones are highly complementary technologies.

Ignoring errors in your service

If you've specified the following with your service manifest, you've told **smf(5)** that you don't care about what happens to the processes that your start method starts up.

```
<property_group name='startd' type='framework'>
    <propval name='duration' type='astring' value='transient' />
</property_group>
```

We provided this functionality for configuration services which need to tell **smf(5)** that they don't have processes that need to be restarted if they fail. Basically, no processes in the contract isn't an error. But, this has also (understandably) been abused to shoehorn legacy services which may or may not have processes running when their start method exits into **smf(5)**. Some of these examples of incomplete conversions even exist within SMF. `svc:/network/initial` may start up a number of daemons on your system, but you don't see them under **`svcs -p`**. That's because the duration property is set to `transient`. You can see this with:

```
$ svcprop -p startd/duration network/initial transient
```

svc.startd(1M) believes there are no important processes to worry about restarting, so it doesn't track them under `svcs -p`, and won't restart the service if one of the processes is killed due to an uncorrectable memory error. These services will be converted more thoroughly in a future release of SMF. But, if you want the processes in your service to be restarted on failure, never set `startd/duration` to `transient`.

Chapter 6. `svc.startd(1M)` Service Development

XXX: first confirm the service's model (start, stop, etc.) is compatible with svc.startd's.

Manifest Creation

Name your service

Services have names, which may have slashes included in the name. Unlike in the filesystem, the `/` is not special in the service name. This allows *categories* to be included in the service name, which allows administrators to easily group service types and refer to them more easily. These categories aren't used by the system, but help the administrator in identifying the general use of the service. These categories are shown in `/var/svc/manifest`, and include:

`application`	higher level applications, such as `apache`
`milestone`	collections of other services, such as `name-services`
`platform`	platform-specific services, such as Dynamic Reconfiguration daemons
`system`	OpenSolaris system services, such as `coreadm`
`device`	device-specific services
`network`	network/internet services, such as protocol implementations
`site`	site specific descriptions

Categories may also have subcategories to further classify similar services. For example, `network/rpc` is used for RPC services. Additional subcategories may be added if a Java™ - style reversed domain prefix or your companies stock symbol are used in the category name to avoid conflicts with other add-on products.

The service name should describe what is being provided, and includes both any category and subcategory identifier and the actual service name, separated by '/'. Service names should usefully identify to the administrator the service being provided.

The instance name describes any specific features about the instance. Most services deliver a 'default' instance. Some (e.g. ORACLE™ software or other services with complex configuration) may want to only create instances based on administrative configuration choices.

Services that are shipped as part of a product or generally extend beyond a site-specific definition should include either the stock symbol or Java™-style reversed domain prefix followed by a comma as part of the category or service name for uniqueness. As an example of the naming conventions above, the `cron` service specifies as its prelude:

```
<service
```

```
    name='system/cron'
    type='service'
    version='1'>
```

Identify whether your service may have multiple concurrent instances

If multiple binaries of your service running simultaneously on the system would cause an error, you must define it as a `single_instance` service. This tag tells the restarter to not start multiple service instances simultaneously, regardless of administrative configuration.

Most configuration and system services require `single_instance` tags. Services such as web servers or databases which could run multiple configurations simultaneously (such as use a different database source or run on a different port) should not be specified as `single_instance`.

Specify after the service block:

```
<single_instance />
```

Identify your service model

svc.startd(1M) is a process-based restarter. It provides three distinct models for service processes:

contract Most services are contract services. That is, they are implemented by long-running processes. Standard system daemons are almost always contract services. They require processes which run forever once started to provide service. Death of all processes in a contract service is considered a service error, which will cause **svc.startd(1M)** to restart the service. The default service model is contract. No additions to the manifest are required to use this service model.

transient A transient service is expected not to start a long-running process. Transient services are often configuration services, which require no long-running processes to provide service. Common transient services perform boot-time cleanup or load configuration properties into the kernel.

svc.startd(1M) does not monitor the child processes of a transient service beyond the execution of the method - processes started by a transient service aren't considered part of the service once the method exits. Failures in the child processes are not detected as an error. Transient services are therefore sometimes used to overcome difficulties in conforming to the method requirements for contract or wait services. This is not recommended and should be considered a stopgap measure.

svc.startd(1M) treats a service as transient if its `startd/duration` property is set to `transient`. If your service should be defined as transient, insert the following into your service manifest:

```
<property_group
```

```
    name='startd' type='framework'>
 <propval
     name='duration'
     type='astring'
     value='transient' />
</property_group>
```

wait

Wait services are implemented by a single child process, and are restarted when that process exits. Wait services are very rare; consider use of a different service model first.

svc.startd(1M) treats a service as transient if its `startd/duration` property is set to `child`. If your service should be defined as wait, insert the following into your service manifest:

```
<property_group
    name='startd' type='framework'>
 <propval
     name='duration'
     type='astring'
     value='child' />
</property_group>
```

Identify how your service is started/stopped.

smf(5) manipulates a service with methods. Methods are procedures specified by a services' properties. **svc.startd(1M)** requires services to provide *stop* and *start* methods. **svc.startd(1M)** methods can name a program, such as a shell script or a binary, or an action to be taken by **svc.startd(1M)**The *refresh* method is optional for **svc.startd(1M)** managed services. Different restarters may require different methods.

Existing `init.d` scripts can usually serve as the basis for service methods. The following rules and guidance are appropriate for the methods supported by **svc.startd(1M)**:

all methods

- Shell scripts should include `/lib/svc/share/smf_include.sh` to gain access to convenience functions and return value definitions.
- Failures must cause explicit error returns. All non-0 values are considered errors. Additional information (for example, to avoid restart due to configuration errors) may be provided to the restarter with the `SMF_EXIT_*` shell variable definitions. See the individual method descriptions for further details on exit code behavior.
- Method should print messages to `stdout` or `stderr` on error or failure. They'll be redirected by **svc.startd(1M)** to the service log file, which the administrator will be directed to in case of error so they can determine potential courses for repair.
- The keywords `:kill` and `:true` are available for all method definitions. `:true` instructs **svc.startd(1M)** to do nothing. `:kill` kills all processes started by your start method. The list of all processes is determined by the service's contract.

`:kill` is primarily effective for *contract* services. **svc.startd(1M)** doesn't track processes for *transient* and *wait* services in the service's primary contract. Therefore, `:kill` will not effectively kill all processes in *transient* and *wait* services.

`start` methods

- A `start` method is required for all **svc.startd(1M)**-managed services.
- `start` methods are only run when the service is enabled and dependencies are already met.
- `start` methods should exit with `$SMF_EXIT_ERR_CONFIG` if the service cannot come online due to any configuration error. XXX - other exit codes and their meanings
- For *contract* services, the `start` method must leave your daemon running if returning success, as exit of all processes will cause the service to be restarted. That is, the start method should only return with exit code 0 once the processes have started and will likely stay running until an explicit error occurs.
- *Contract* and *transient* service start methods should not return success until the service is completely ready to talk to its clients. Note that this is true for daemons as well; daemons shouldn't `fork` then `exit` from their initial process, they should wait to return until startup errors have been accumulated and can be reported. Many `init.d` scripts traditionally execute a daemon and return immediately without waiting for the service to start, counting on the fact that the serial boot took 'a while' to start dependent services. Now that dependent services are started precisely (often immediately) after your service returns successfully from its start method, precise semantics are required.

  If code changes to the daemon/service can't be made, the method should wait for service to be provided before returning success. If no other options are available, insert an appropriately long `sleep` before successful return.

`stop` methods

- A `stop` method is required for all **svc.startd(1M)**-managed services.
- `Stop` methods are run in a number of different scenarios, including when a dependency (with `restart_on` set to something more than none) has gone offline or failed, when the service fails, and when an administrator requests disable or restart.
- Thus, `stop` methods should return success if the service is no longer running after execution is complete, even if the service wasn't running when the execution started. This is because `stop` methods may be called in error scenarios.

`refresh` methods

- `Refresh` methods are optional for all **svc.startd(1M)**-managed services.
- Any defined refresh method must have very precise semantics; it must reload appropriate configuration parameters from the repository or other configuration source without interrupting service. It must not cause exit of the existing processes for *contract* or *wait* services.

Timeouts must be specified for all methods. **svc.startd(1M)** will consider the method to have failed if the timeout expires during the method's execution. The timeout should be defined to be the maximal amount of time in seconds that your method might take to run on a slow system or under heavy load. A method which exceeds its timeout will be killed. If the method could potentially take an unbounded amount of time, such as a large filesystem fsck, an infinite timeout may be specified as '0'.

We strongly discourage expecting user interaction (i.e. via console input) as part of the service methods. Scripts which do so will not work without modification, as the `stdin/stdout/stderr` are not `/dev/console` for service methods. XXX - example needed

We provide a set of method tokens available for use in method specification for commonly used property values. A comprehensive list is available in **smf_method(5)**. XXX - example needed

The default method environment is inherited from **init(1M)**, with the **PATH** set to `/usr/sbin:/usr/bin`. Variables beginning with `SMF_` are reserved for framework use. The `SMF_` variables defined in **smf_method(5)** are provided to all methods; these include `SMF_FMRI`, `SMF_METHOD,` and `SMF_RESTARTER.`

Finally, each method may specify a `method_context`, to define system, resource management, and security attributes used during method execution. We recommend long-running services are started with reduced privileges and safe uids and gids, when possible. XXX - examples

An example of a start method specification is below.

```
<exec_method
    type='method'
    name='start'
    exec='/lib/svc/method/svc-cron'
    timeout_seconds='60'
    <method_context>
 <method_credential
      user='root'
            group='root' />
    </method_context>
</exec_method>
```

Determine faults to be ignored

If either your service is poorly behaved itself, or it might spawn poorly behaved subprocesses, you will want to inform the restarter that certain errors are expected and don't constitute service faults.

You may specify that coredumps from service subprocesses or fatal signals from processes outside the service aren't fatal to the service. An example of specifying that neither are errors is below.

```
<property_group
    name='startd'
    type='framework'>
 <propval
     name='ignore_error'
     type='astring'
     value='core,signal' />
</property_group>
```

XXX - separate examples?

Identify dependencies

This is the most difficult part of service conversion, as most dependencies are not explicitly stated. There are two different types of dependencies; `file` and `service` dependencies. XXX - warn about file: dependencies

First, identify what other services are required for yours to be started. For example, does your service require the network to be plumbed, local devices to be configured, name services to be available? Are there services that yours would be useless without?

Once you've decided what your service is dependent on, you'll need to determine and specify the fault propagation model. For each dependency, decide whether your service should restart if:

`none`	service can withstand faults and restarts in the dependency
`fault`	restart if the dependency has a fault (core dump, system fault, etc.)
`restart`	if the dependency is restarted for any reason, including fault, your service should be
`refresh`	if the dependency is refreshed (its configuration is changed), restarted, or has a fault, your service should be restarted

These values correspond to the ability to handle restart of the specified dependency, via the `restart_on` property.

Dependencies may be grouped. The potential groupings are:

`require_all`	all in the group must be running (*online* or *degraded*) before the dependency can be started
`require_any`	any one of the services in the group must be *online* or *degraded* before the dependency can be started
`optional_all`	all services must be running (*online* or *degraded*), *disabled*, in the *maintenance* state, or not present before the dependency can be started. In other words, if the dependency will get to *online* or *degraded*, wait for it, including if it gets stuck in *offline* due to unsatisfied dependencies.
`exclude_all`	if the dependency is enabled and *online* or *degraded,* the service should not be started

Dependencies may specify service FMRIs or instance FMRIs. A dependency on the *instance* is evaluated precisely. Dependencies specified on a *service* rather than a specific instance are evaluated as `require_any` for all configured instances. If your service does not require a specific instance, always use the *service* as the dependency for maximum flexibility.

If your service is dependent on a legacy script having run, we strongly recommend you either convert or encourage your vendor to convert the legacy script to an **smf(5)** service. Barring that, you can specify a dependency on the *milestone* that script is part of. Since **svc.startd(1M)** doesn't track legacy services, this will never propagate errors from the legacy service, so only makes sense as a `restart_on=none` dependency.

Finally, since you did the hard work to determine why a certain dependency was required, write a comment to help future maintainers!

```
<!-- Must be able to resolve hostnames. -->
<dependency
    name='nameservice'
    type='service'
    grouping='require_all'
    restart_on='none'>
 <service_fmri
     value='svc:/milestone/name-services' />
</dependency>
```

Identify dependents

If you wish to deliver a service on which another service should depend, you can specify this in your manifest without modifying the manifest you don't own. That is, *dependent* specifications are an easy way to have your service run before a service delivered by Sun.

If not all of your dependent services have been converted, you'll either need to convert those too, or specify the *milestone* the legacy service runs in as a dependent. See the next section for instructions.

To avoid conflicts, we require prefacing your dependent name with the name of your service.

For example, if you're delivering a service (`mysvc` in the example below) that must start before `syslog`, use the following:

```
<dependent
    name='mysvc_syslog'
    grouping='optional_all'
    restart_on='none'>
 <service_fmri
     value='svc:/system/system-log' />
</dependent>
```

If appropriate, insert your service into a *milestone*

If your service was previously delivered into an `rc?.d` directory, you should make the *milestone* corresponding to your previous delivery location a dependent. A milestone should almost never be restarted due to failure of your service. Therefore, `restart_on` should be specified as `none`.

For example, if your service was previously started at runlevel 2, this clause will make sure that runlevel 2 is not considered complete until your service has started.

```
<dependent
    name='mysvc_multi-user'
    grouping='require_all'
    restart_on='none'>
    <service_fmri value='svc:/milestone/multi-user' />
</dependent>
```

Note that the dependent name is created by connecting your service name and the dependent's name by an underscore (_).

Create, if appropriate, a default instance

If your service does not require extensive configuration before it can be started the first time, you should configure a default instance for your service.

If the instance has no configuration differences from the service, this can easily be done with:

```
<create_default_instance enabled='false' />
```

Alternatively, you can explicitly define the instance.

```
<instance name='default' enabled='false'>
    <!-- instance-specific properties, methods, etc. go here. -->
</instance>
```

We recommend that all instances are delivered as disabled unless if they are critical to system boot. Customization can then be done by either the administrator or a *profile* (described elsewhere).

Create *template* information to describe your service

Document at least a common name in the C locale and a manpage reference. The common name should

- be short (40 characters or less),
- avoid capital letters aside from trademarks like Solaris or OpenSolaris,
- avoid punctuation, and
- avoid the word *service* (but do distinguish between client and server).

This information is presented by various forms of **svcs(1)** to provide the administrator with concise detail about your service and where to get more technical information. Common names may be localized.

```
<template>
    <common_name>
        <loctext xml:lang='C'>
        fault manager
        </loctext>
    </common_name>
    <documentation>
        <manpage
     title='fmd'
     section='1M'
     manpath='/usr/share/man' />
    </docmentation>
```

```
</template>
```

Write/update an administrative command

If your service already has an administrative command which stops, starts, or restarts your service, update it to use **svcadm(1M)**, or **libscf(3LIB)** calls. If an administrative command explicitly starts a daemon outside of **smf(5)**, the system won't know there are other daemons running. Conflicts between daemons, incorrect contracts, and lack of visibility using **svcs(1)** are among the problems that will occur.

Remove your script from `/etc/rc?.d` locations and `/etc/init.d`

If you don't remove your `init` script, it will still be run in legacy mode. If your `/etc/init.d` script is well-documented, you may wish to ease the transition for administrators by providing a compatibility script. XXX - example.

Method Context

XXX: how to set, etc.

Method Creation

Logging recommendations - no need to stop using syslog, all exits from methods with non-zero exit values should have an accompanying helpful log message to stderr to guide the administrator to resolution.

svcprop example

Moving configuration to repository

XXX: which properties to move, new properties to create

Importing and Testing

XXX: use of mark/clear

Manifest to repository mapping

Examples

Sun delivers many manifests in `/var/svc/manifest`. These may be used as templates and examples. A few to start with on your Solaris system:

- `/var/svc/manoifest/system/utmp.xml` is a simple standalone daemon, and
- `/var/svc/manifest/system/coreadm.xml` is a simple transient service

In addition, we'll cover some examples from start to finish here.

Example 6.1. Creating a simple standalone daemon manifest

We'll begin with a small toy daemon in this example. The `/opt/SUNWtoyd/sbin/toyd` command returns success only after the daemon is up and providing service. It returns failure if it cannot start successfully. There is always a process associated with this service if it is running correctly. The service manifest for `toyd` specified below would be delivered into `/var/svc/manifest/application/SUNW-toyd.xml`.

The manifest is created by answering the questions in the section above.

- Our toy daemon is an application service, not critical to system operation. Thus, we name it `application/toy`.
- Multiple copies of the toy daemon running simultaneously would cause problems, as the daemon isn't written to handle that. It should be specified as `single_instance`
- As this is a standard system daemon which always has at least one process associated with it while it is running, we use a `contract` service model.
- Our daemon doesn't require any specific setup, so we can just execute the daemon directly. There's no requirement for an additional method script for start and stop. The start method is therefore specified as `/opt/SUNWtoyd/sbin/toyd`. The toy daemon is quick to start up - we know a longer than 60 second startup time probably means there's something wrong. To stop the service, it just needs to be killed, so we use the `:kill` keyword for the stop method. This is also quick, so we also use a default 60 second timeout. Our daemon doesn't support reloading its configuration without a restart, so we don't specify a refresh method.
- There are no faults that need to be ignored by the toy daemon service; core dumps and external fatal signals are errors that should cause the service to be restarted, so we add no ignored faults to the manifest.
- Our toy daemon doesn't do much, but it does require that `/opt` is mounted so that it has access to its binaries. We specify a dependency on `svc:/system/filesystem/local` to reflect that, and also consider the case where `/opt` is an NFS automount by specifying an optional dependency on `svc:/system/filesystem/autofs`. Our daemon also uses **syslog(3C)** to log problems, but can still run even if **syslogd(1M)** isn't running. An optional dependency on `svc:/system/system-log` is in order. None of these services restarting should cause our daemon to restart, so all dependencies have `restart_on` set to `none`.
- There are no individual services which depend on the toy daemon.
- But, we do want the toy daemon to always start as part of `multi-user-server`, the `rc3` milestone. Thus, we create a dependent for that milestone.
- The toy daemon does have a default instance, and should be, like all delivered services, disabled by default.
- The toy daemon has a descriptive common name which fits the naming rules specified above: smf(5) Guide example daemon. Its manpage lives in `/opt/SUNWtoyd/man/man1m/toyd.1m`.

```
<?xml version="1.0"?>
<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">

<service_bundle type='manifest' name='SUNWtoyd:toyd'>

<service
 name='application/toy'
```

```
type='service'
version='1'>

<create_default_instance enabled='false' />

<single_instance/>

<!-- This daemon requires files located on /opt.  -->
<dependency
 name='filesystem'
 grouping='require_all'
 restart_on='none'
 type='service'>
 <service_fmri
     value='svc:/system/filesystem/local' />
</dependency>

<!-- /opt may be automounted  -->
<dependency
 name='autofs'
 grouping='optional_all'
 restart_on='none'
 type='service'>
 <service_fmri
     value='svc:/system/filesystem/autofs' />
</dependency>

<!-- We use syslog(3C) to log messages.  -->
<dependency
 name='system-log'
 grouping='optional_all'
 restart_on='none'
 type='service'>
 <service_fmri
     value='svc:/system/system-log' />
</dependency>

<dependent
 name='toyd_multi-user-server'
 grouping='optional_all'
 restart_on='none'>
 <service_fmri
     value='svc:/milestone/multi-user-server' />
</dependent>

<exec_method
 type='method'
 name='start'
 exec='/opt/SUNWtoyd/sbin/toyd'
 timeout_seconds='60' />

<exec_method
 type='method'
 name='stop'
```

```
      exec=':kill'
      timeout_seconds='60' />

     <template>
      <common_name>
      <loctext xml:lang='C'>
          smf(5) Guide example daemon
      </loctext>
      </common_name>
      <documentation>
       <manpage title='toyd' section='1M'
           manpath='/opt/SUNWtoyd/man' />
      </documentation>
     </template>
    </service>
    </service_bundle>
```

Example 6.2. Creating a simple configuration service manifest

This example uses as its basis a simple configuration command which uploads configuration to a fictional kernel component. The `/opt/SUNWkconfig/bin/kconfig -u` command returns success after successfully performing a structured set of `ioctl(2)` calls with arguments determined by a configuration file. It returns failure if the configuration file is invalid or the `ioctl(2)` fails. After the service completes its startup, no processes are left associated with the service. The service manifest for `kconfig` specified below would be delivered into `/var/svc/manifest/application/SUNW-kconfig.xml`.

This example is appropriate if your software has an existing configuration file. Software which lacks an existing configuration file with a well-known format, or a configuration file that must be portable amongst multiple operating systems should use **smf(5)** properties for configuration. Using those properties will be handled in another example.

The manifest is created by answering the questions in the section above.

- We name our configuration service `application/kconfig`, as for this example the kernel component is assumed to be application-specific. We'd use a `system` category if it was a core system component instead.
- There's no reason to allow multiple instances, so this service is specified as `single_instance`
- The run-once nature of this configuration service along with its lack of long-term processes clearly require a `transient` service model.
- `kconfig` is a simple command, so we can execute it directly as the start method. The start method is therefore specified as `/opt/SUNWtoyd/bin/kconfig`. This service is quick to start up - we know a longer than 60 second startup time probably means there's something wrong. No action is necessary to stop the service, so we use the `:true` keyword for the stop method. This is also quick, so we also use a default 60 second timeout.
- As there are no long-running processes for this service, there are no faults that need to be explicitly ignored.
- The configuration service does require that `/opt` is mounted so that it has access to its binaries. We specify a dependency on `svc:/system/filesystem/local` to reflect that, and also consider the case where `/opt` is an NFS automount by specifying an optional dependency on `svc:/system/filesystem/autofs`. None of these services restarting should cause our daemon to restart, so both dependencies have `restart_on` set to `none`.
- There are no individual services which depend on our configuration service.
- But, we do want the service to always start as part of `multi-user`, the `rc2` milestone. Thus, we create a dependent for that milestone.
- The `kconfig` service does have a default instance, and should be, like all delivered services, disabled by default.
- `kconfig` has a descriptive common name which fits the naming rules specified above: smf(5) Guide example configuration service. Its manpage lives in `/opt/SUNWkconfig/man/man1m/kconfig.1m`.

```
<?xml version="1.0"?>
<!DOCTYPE service_bundle
```

```
    SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">

<service_bundle type='manifest' name='SUNWkconfig:kconfig'>

<service
 name='application/kconfig'
 type='service'
 version='1'>

 <create_default_instance enabled='false' />

 <single_instance/>

 <!- This service requires files located on /opt.  ->
 <dependency
  name='filesystem'
  grouping='require_all'
  restart_on='none'
  type='service'>
  <service_fmri value='svc:/system/filesystem/local' />
 </dependency>

 <!- /opt may be automounted  ->
 <dependency
  name='autofs'
  grouping='optional_all'
  restart_on='none'
  type='service'>
  <service_fmri value='svc:/system/filesystem/autofs' />
 </dependency>

 <dependent
  name='kconfig_multi-user'
  grouping='optional_all'
  restart_on='none'>
  <service_fmri value='svc:/milestone/multi-user' />
 </dependent>

 <exec_method
  type='method'
  name='start'
  exec='/opt/SUNWkconfig/bin/kconfig'
  timeout_seconds='60' />

 <exec_method
  type='method'
  name='stop'
  exec=':true'
  timeout_seconds='60' />

 <property_group name='startd' type='framework'>
  <propval name='duration' type='astring'
   value='transient' />
 </property_group>
 <template>
  <common_name>
   <loctext xml:lang='C'>
   smf(5) Guide example configuration service
   </loctext>
  </common_name>
  <documentation>
   <manpage title='kconfig' section='1M'
    manpath='/opt/SUNWkconfig/man' />
  </documentation>
 </template>
</service>
</service_bundle>
```

Example 6.3. Converting an existing init.d(4) service

Example 6.4. Creating a new smf(5) service

Chapter 7. inetd(1M) Service Development

inetconv(1M)

Start with **inetconv(1M)**, and include other modifications such as adding templates and refining the name. More to come on this and: * packaging * removing pre-converted inetd.conf services.

1. Create an **inetd.conf(4)**-style file which contains only your service's entries
2. Run **inetconv -i "your inetd.conf file"**.

/etc/services

Describing the service

Examples

Sun delivers many manifests in `/var/svc/manifest`. `/var/svc/manifest/network/telnet.xml` is an **inetd(1M)**-based daemon.

Chapter 8. Delegated Administration

RBAC integration, how to specify, how to manage, etc.

Chapter 9. Service Packaging

Delivery Locations

methods are delivered with your service. If your service is delivered in /opt/SUNWfoo, your method should be delivered as /opt/SUNWfoo/lib/svc/svc-foo. If your method would otherwise share a name with your service's executable, prefixing the method with `svc-` helps to easily differentiate the two.

Using the manifest class-action scripts

Chapter 10. Service Testing and XXX

Service debugging modes

XXX: pre and post temporary methods

Temporary disable

Changing the restarter of a service

XXX where to put this?

Chapter 11. Compatibility

init

smf(5) maintains compatibility for most applications started by **init(1M)** by placement in the `/etc/rc?.d` directories, and for applications delivered into `inetd.conf.`

Some init services, however, must be converted to smf to preserve their boot-time ordering. An init service needs to convert if it affects other infrastructure services, like the early setup of devices, filesystems, or network configuration. A service also needs to convert if it requires input from the console during the boot process. (Such services are strongly discouraged.)

Services that are started from the rc directories are referred to as legacy services.

inetd

XXX -- autoconvert on upgrade, running inetconv with no manifest modification, when we print warnings, see inetd development chapter

inittab

Chapter 12. Troubleshooting

A number of standard procedures and tricks can ease troubleshooting during service development. This chapter covers some of those techniques.

XXX -- likely different organization.

Problem: Service not running

(include uninitialized explanation)

Problem: Manifest won't import

Problem: Service restarting too rapidly

Problem: Repository corrupt or missing

Problem: System hangs during boot

If the system hangs during boot (e.g. you never receive a console login prompt or a graphical login screen), you can use **smf(5)** to essentially watch boot happen. **svc.startd(1M)** makes a concerted effort to bring up a console login prompt early in boot, and to bring up an **sulogin(1M)** prompt if something goes wrong, but there are some cases where the system appears hung and there's no login prompt to be seen. They're rare, but not completely impossible.

At the boot prompt (`ok` on sparc, `Select (b)oot or (i)nterpreter:` on x86), type **`b -m milestone=none`**. That'll get you to here:

```
Select (b)oot or (i)nterpreter: b -m milestone=none
SunOS Release 5.10 Version gate:2005-01-10 32-bit
Copyright 1983-2005 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
Booting to milestone "none".
Requesting System Maintenance Mode
(See /lib/svc/share/README for more information.)
Console login service(s) cannot run

Root password for system maintenance (control-d to bypass):
```

Log in. If you run **`svcs`** now, you'll note that all services are `disabled` or `uninitialized`. The disabled services are temporarily disabled by **svc.startd(1M)** because that's how a limited milestone is implemented: we temporarily disable all services that aren't part of that milestone's subgraph. The uninitialized services are managed by a different restarter than svc.startd. Their restarters haven't shown up yet to manage their state, so they remain uninitialized. Now, to start up the rest of the system. Run **`svcadm milestone all`**, then use **`svcs`** to watch your system start up. If you're looking to debug a specific problem, wait until the **svcs(1M)** output stabilizes, then run **`svcs -x`** to see what services are causing trouble. Look at the services' logfiles for more details on what's going wrong. Finally, when you're done poking around, just exit the login shell to resume normal console login.