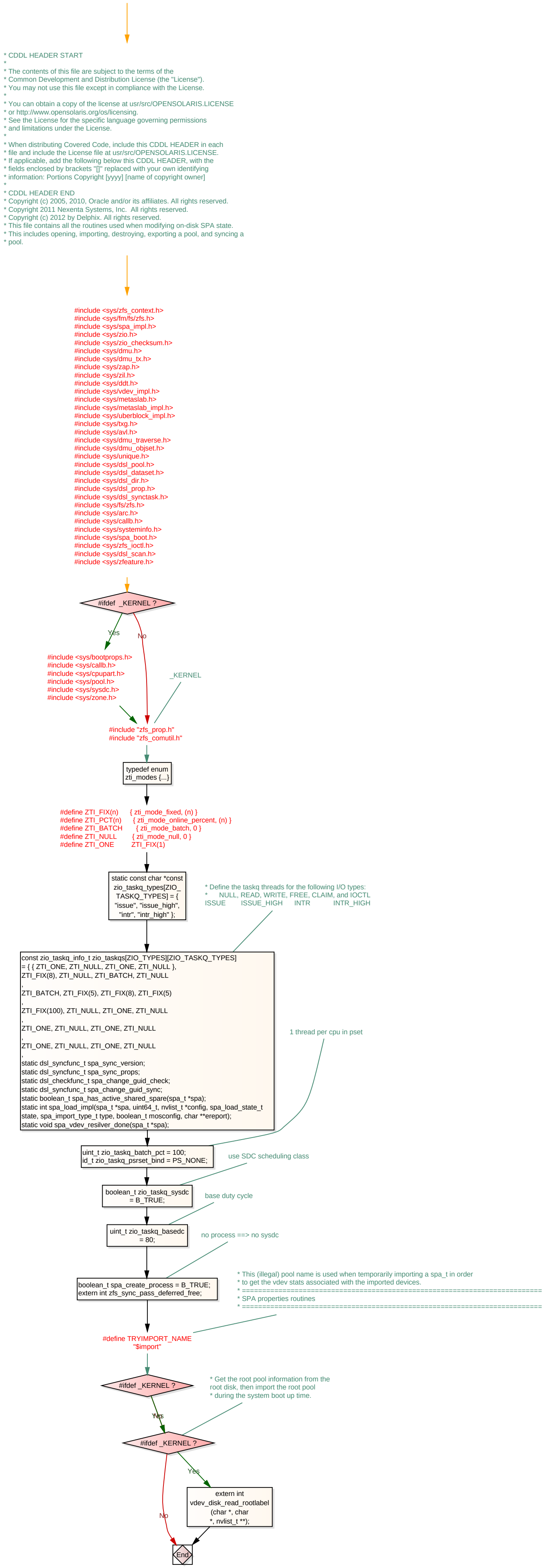


```
* CDDL HEADER START
*
* The contents of this file are subject to the terms of the
* Common Development and Distribution License (the "License").
* You may not use this file except in compliance with the License.
*
* You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
* or http://www.opensolaris.org/os/licensing.
* See the License for the specific language governing permissions
* and limitations under the License.
*
* When distributing Covered Code, include this CDDL HEADER in each
* file and include the License file at usr/src/OPENSOLARIS.LICENSE.
* If applicable, add the following below this CDDL HEADER, with the
* fields enclosed by brackets "[]" replaced with your own identifying
* information: Portions Copyright [yyyy] [name of copyright owner]
*
* CDDL HEADER END
* Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
* Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
* Copyright (c) 2012 by Delphix. All rights reserved.
* This file contains all the routines used when modifying on-disk SPA state.
* This includes opening, importing, destroying, exporting a pool, and syncing a
* pool.
```

```
#include <sys/zfs_context.h>
#include <sys/fm/fs/zfs.h>
#include <sys/spa_impl.h>
#include <sys/zio.h>
#include <sys/zio_checksum.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/ddt.h>
#include <sys/vdev_impl.h>
#include <sys/metaslab.h>
#include <sys/metaslab_impl.h>
#include <sys/uberblock_impl.h>
#include <sys/txg.h>
#include <sys/avl.h>
#include <sys/dmu_traverse.h>
#include <sys/dmu_objset.h>
#include <sys/unique.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/fs/zfs.h>
#include <sys/arc.h>
#include <sys/callb.h>
#include <sys/systeminfo.h>
#include <sys/spa_boot.h>
#include <sys/zfs_ioctl.h>
#include <sys/dsl_scan.h>
#include <sys/zfeature.h>
```

```
#ifdef _KERNEL ?
```

```
#include <sys/bootprops.h>
#include <sys/callb.h>
#include <sys/cpupart.h>
#include <sys/pool.h>
#include <sys/sysdc.h>
#include <sys/zone.h>
```

_KERNEL

```
#include "zfs_prop.h"
#include "zfs_comutil.h"
```

```
typedef enum
zti_modes {...}
```

```
#define ZTI_FIX(n)      { zti_mode_fixed, (n) }
#define ZTI_PCT(n)      { zti_mode_online_percent, (n) }
#define ZTI_BATCH       { zti_mode_batch, 0 }
#define ZTI_NULL        { zti_mode_null, 0 }
#define ZTI_ONE         ZTI_FIX(1)
```

```
static const char *const
zio_taskq_types[ZIO_
TASKQ_TYPES] = {
"issue", "issue_high",
"intr", "intr_high" };
```

```
* Define the taskq threads for the following I/O types:
*       NULL, READ, WRITE, FREE, CLAIM, and IOCTL
ISSUE        ISSUE_HIGH      INTR            INTR_HIGH
```

```
const zio_taskq_info_t zio_taskqs[ZIO_TYPES][ZIO_TASKQ_TYPES]
= { { ZTI_ONE, ZTI_NULL, ZTI_ONE, ZTI_NULL },
ZTI_FIX(8), ZTI_NULL, ZTI_BATCH, ZTI_NULL
,
ZTI_BATCH, ZTI_FIX(5), ZTI_FIX(8), ZTI_FIX(5)
,
ZTI_FIX(100), ZTI_NULL, ZTI_ONE, ZTI_NULL
,
ZTI_ONE, ZTI_NULL, ZTI_ONE, ZTI_NULL
,
ZTI_ONE, ZTI_NULL, ZTI_ONE, ZTI_NULL
,
static dsl_syncfunc_t spa_sync_version;
static dsl_syncfunc_t spa_sync_props;
static dsl_checkfunc_t spa_change_guid_check;
static dsl_syncfunc_t spa_change_guid_sync;
static boolean_t spa_has_active_shared_spare(spa_t *spa);
static int spa_load_impl(spa_t *spa, uint64_t, nvlist_t *config, spa_load_state_t
state, spa_import_type_t type, boolean_t mosconfig, char **ereport);
static void spa_vdev_resilver_done(spa_t *spa);
```

1 thread per cpu in pset

```
uint_t zio_taskq_batch_pct = 100;
id_t zio_taskq_psrset_bind = PS_NONE;
```

use SDC scheduling class

```
boolean_t zio_taskq_sysdc
= B_TRUE;
```

base duty cycle

```
uint_t zio_taskq_basedc
= 80;
```

no process ==> no sysdc

```
boolean_t spa_create_process = B_TRUE;
extern int zfs_sync_pass_deferred_free;
```

```
* This (illegal) pool name is used when temporarily importing a spa_t in order
* to get the vdev stats associated with the imported devices.
* ========================================================================
* SPA properties routines
* ========================================================================
```

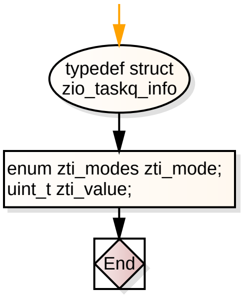
```
#define TRYIMPORT_NAME
"$import"
```

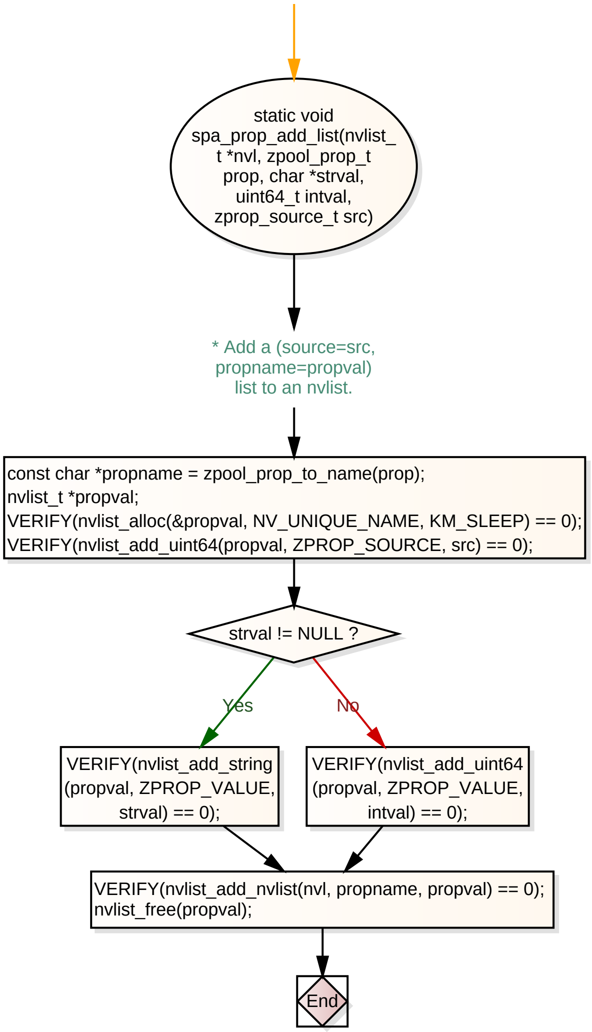
```
#ifdef _KERNEL ?
```

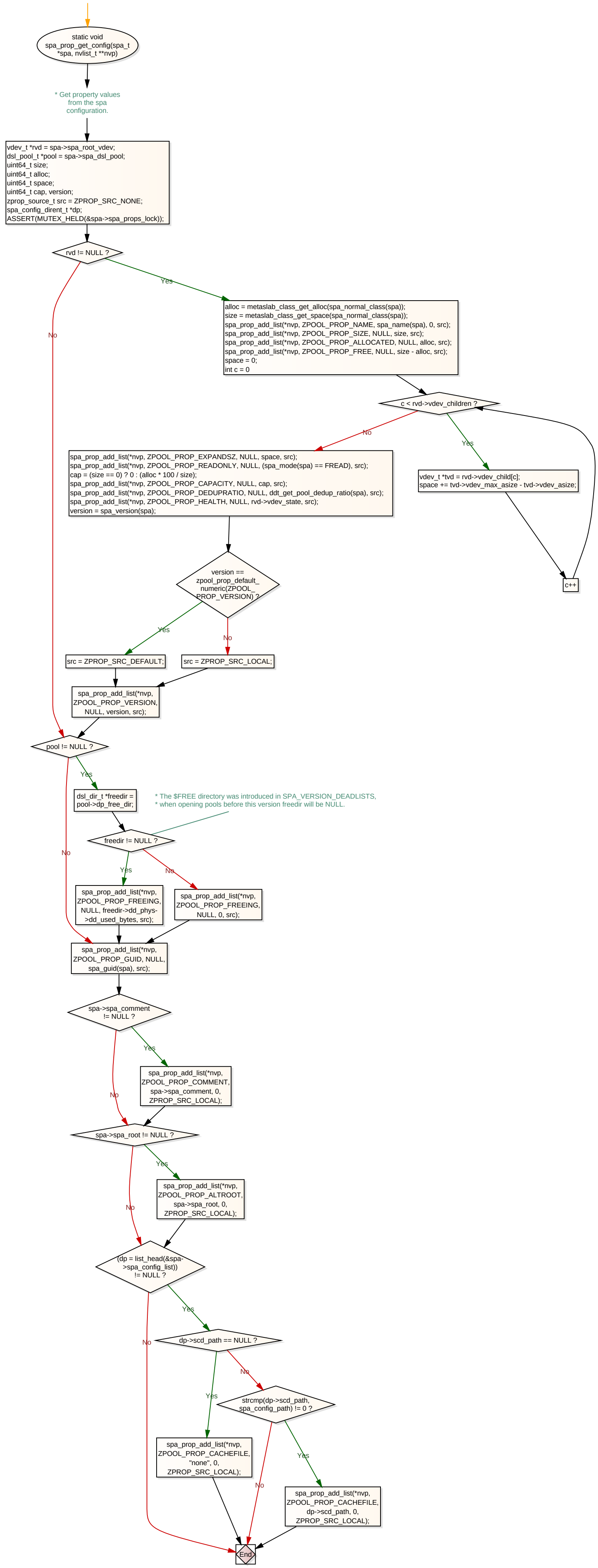
```
* Get the root pool information from the
root disk, then import the root pool
* during the system boot up time.
```

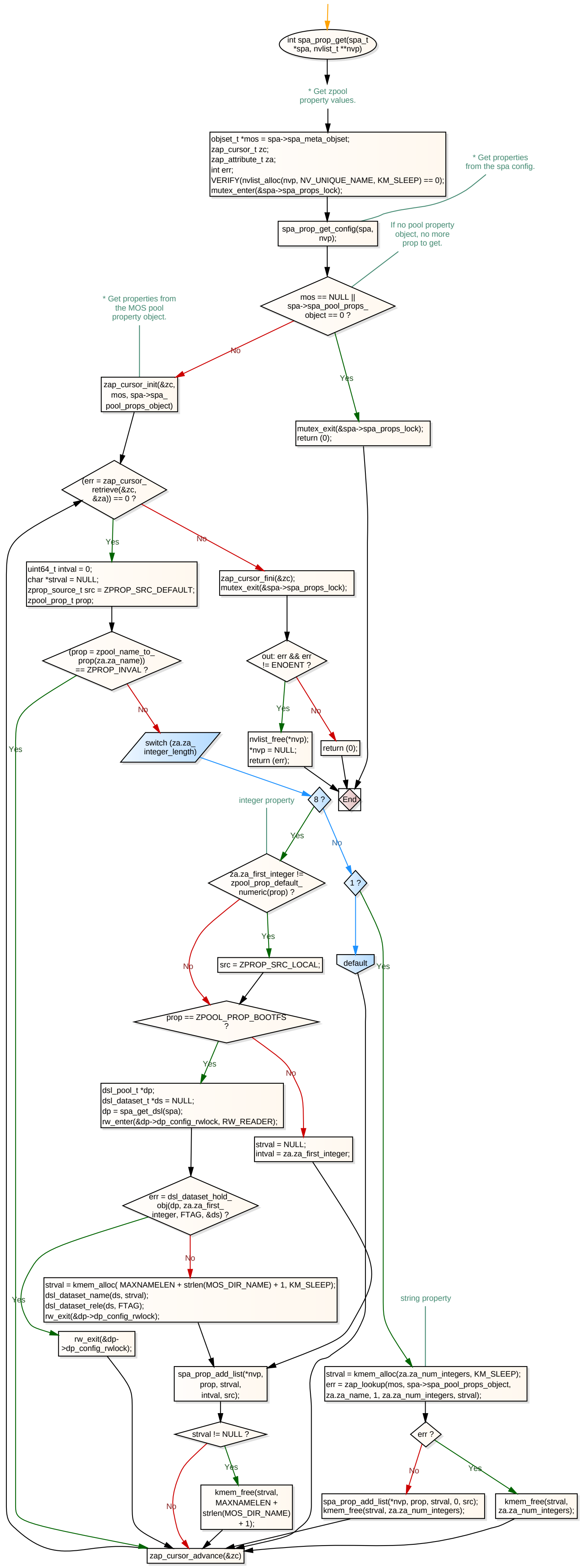
```
#ifdef _KERNEL ?
```

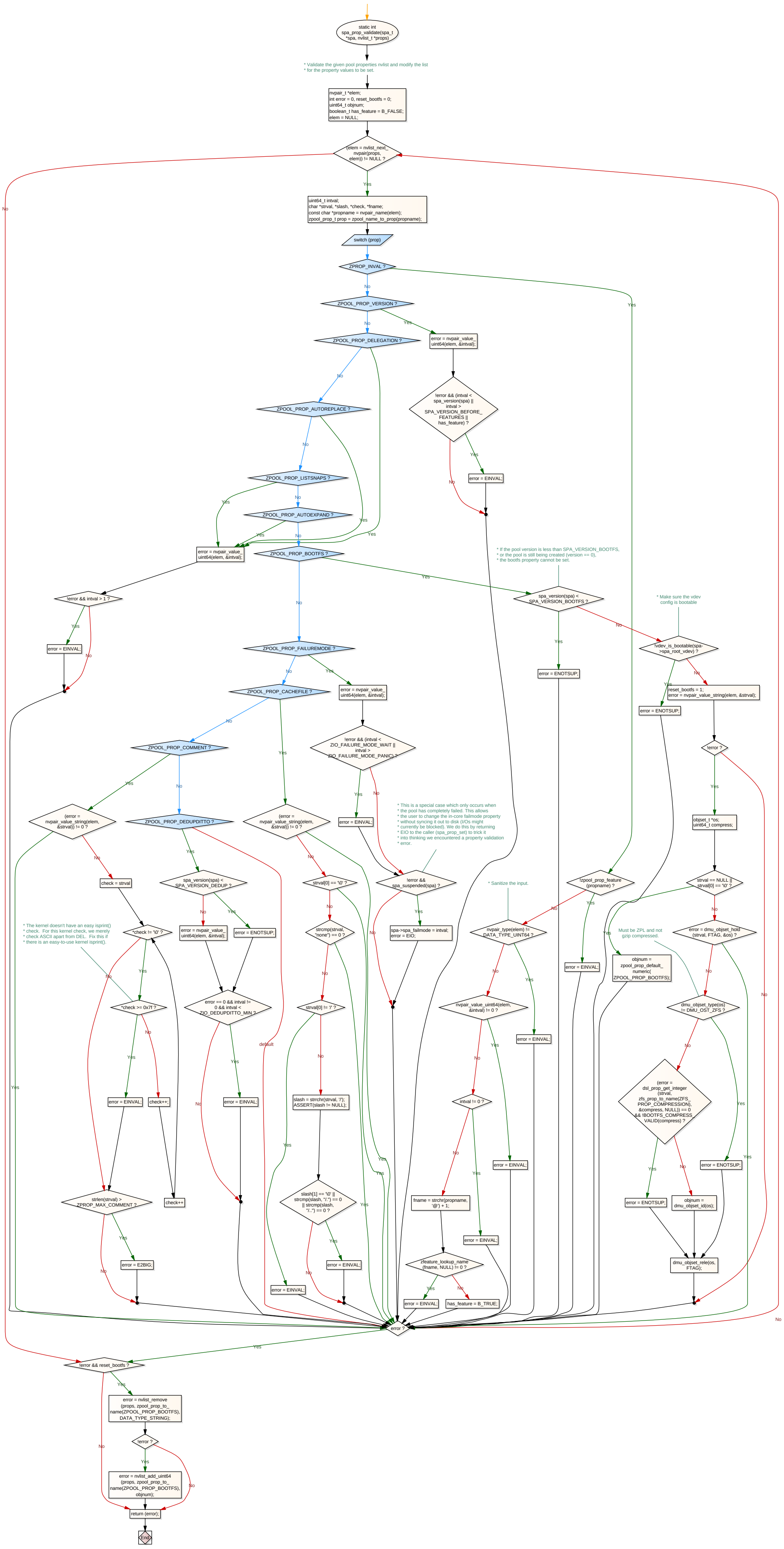
```
extern int
vdev_disk_read_rootlabel
(char *, char
*, nvlist_t **);
```

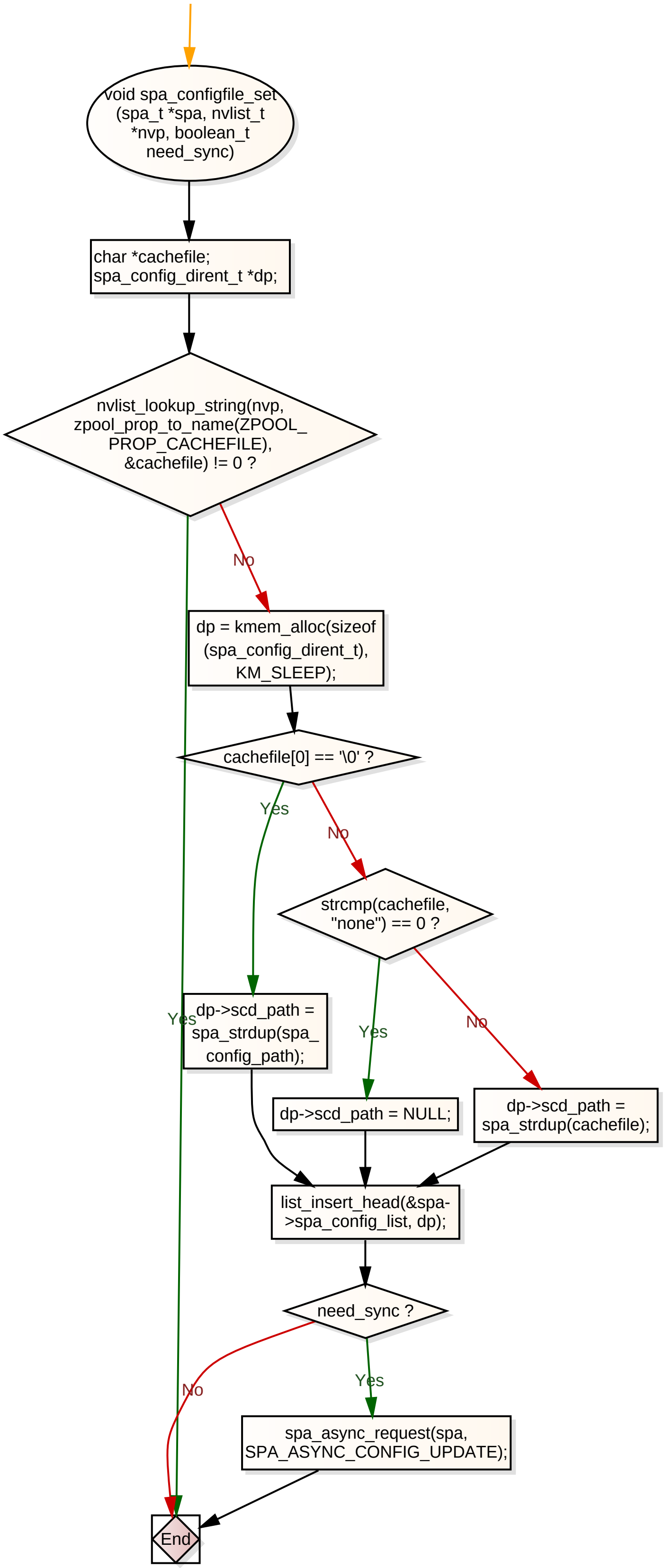

typedef struct
zio_taskq_info
enum zti_modes zti_mode;
uint_t zti_value;
End

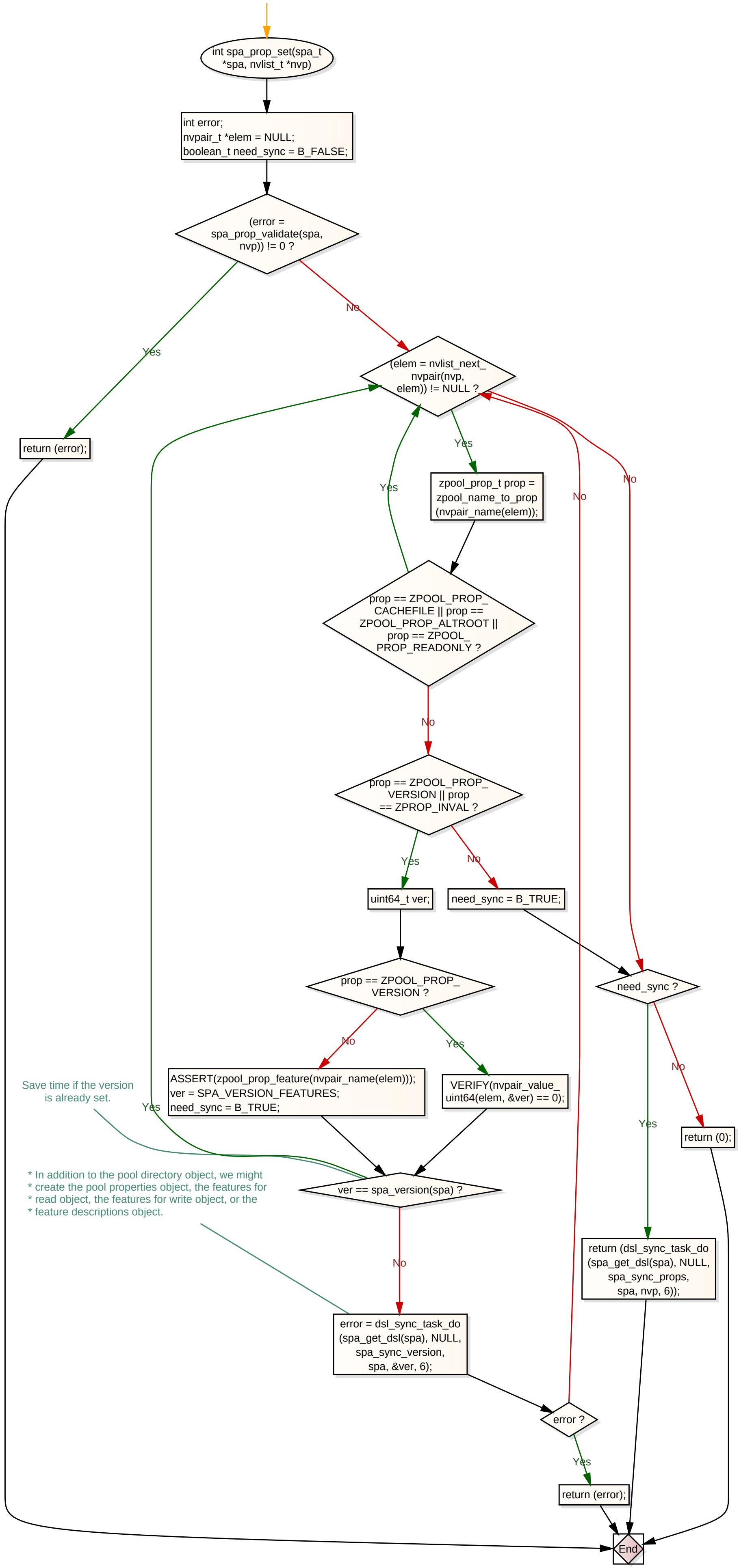

static void spa_prop_add_list(nvlist_t *nvl, zpool_prop_t prop, char *strval, uint64_t intval, zprop_source_t src)
* Add a (source=src, propname=propval) list to an nvlist.
const char *propname = zpool_prop_to_name(prop);
nvlist_t *propval;
VERIFY(nvlist_alloc(&propval, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_uint64(propval, ZPROP_SOURCE, src) == 0);
strval != NULL ?
Yes
No
VERIFY(nvlist_add_string(propval, ZPROP_VALUE, strval) == 0);
VERIFY(nvlist_add_uint64(propval, ZPROP_VALUE, intval) == 0);
VERIFY(nvlist_add_nvlist(nvl, propname, propval) == 0);
nvlist_free(propval);
End

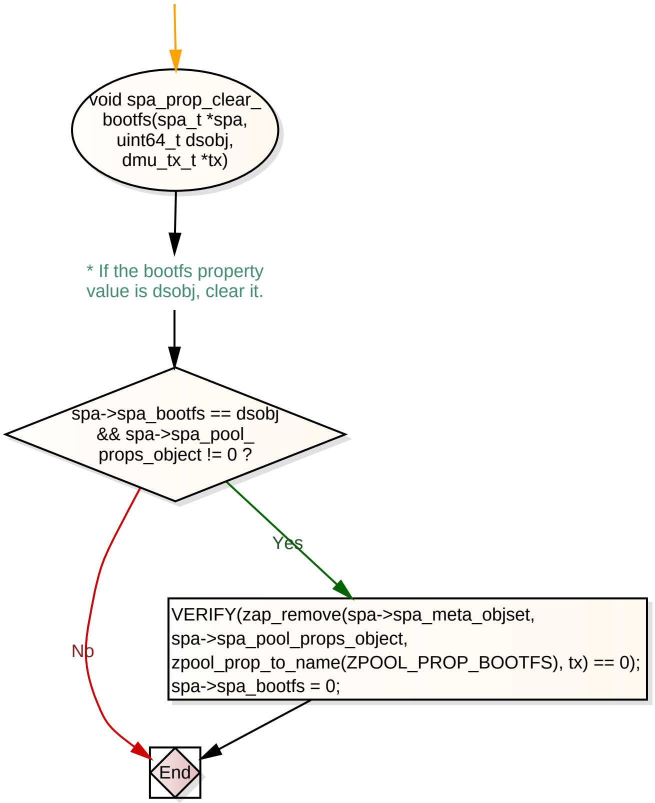

static void
spa_prop_get_config(spa_t
*spa, nvlist_t **nvp)
* Get property values
from the spa
configuration.
vdev_t *rvd = spa->spa_root_vdev;
dsl_pool_t *pool = spa->spa_dsl_pool;
uint64_t size;
uint64_t alloc;
uint64_t space;
uint64_t cap, version;
zprop_source_t src = ZPROP_SRC_NONE;
spa_config_dirent_t *dp;
ASSERT(MUTEX_HELD(&spa->spa_props_lock));
rvd != NULL ?
Yes
No
alloc = metaslab_class_get_alloc(spa_normal_class(spa));
size = metaslab_class_get_space(spa_normal_class(spa));
spa_prop_add_list(*nvp, ZPOOL_PROP_NAME, spa_name(spa), 0, src);
spa_prop_add_list(*nvp, ZPOOL_PROP_SIZE, NULL, size, src);
spa_prop_add_list(*nvp, ZPOOL_PROP_ALLOCATED, NULL, alloc, src);
spa_prop_add_list(*nvp, ZPOOL_PROP_FREE, NULL, size - alloc, src);
space = 0;
int c = 0
c < rvd->vdev_children ?
No
Yes
spa_prop_add_list(*nvp, ZPOOL_PROP_EXPANDSZ, NULL, space, src);
spa_prop_add_list(*nvp, ZPOOL_PROP_READONLY, NULL, (spa_mode(spa) == FREAD), src);
cap = (size == 0) ? 0 : (alloc * 100 / size);
spa_prop_add_list(*nvp, ZPOOL_PROP_CAPACITY, NULL, cap, src);
spa_prop_add_list(*nvp, ZPOOL_PROP_DEDUPRATIO, NULL, ddt_get_pool_dedup_ratio(spa), src);
spa_prop_add_list(*nvp, ZPOOL_PROP_HEALTH, NULL, rvd->vdev_state, src);
version = spa_version(spa);
vdev_t *tvd = rvd->vdev_child[c];
space += tvd->vdev_max_asize - tvd->vdev_asize;
c++
version ==
zpool_prop_default_
numeric(ZPOOL_
PROP_VERSION) ?
Yes
No
src = ZPROP_SRC_DEFAULT;
src = ZPROP_SRC_LOCAL;
spa_prop_add_list(*nvp,
ZPOOL_PROP_VERSION,
NULL, version, src);
pool != NULL ?
Yes
No
dsl_dir_t *freedir =
pool->dp_free_dir;
* The $FREE directory was introduced in SPA_VERSION_DEADLISTS,
* when opening pools before this version freedir will be NULL.
freedir != NULL ?
Yes
No
spa_prop_add_list(*nvp,
ZPOOL_PROP_FREEING,
NULL, freedir->dd_phys-
>dd_used_bytes, src);
spa_prop_add_list(*nvp,
ZPOOL_PROP_FREEING,
NULL, 0, src);
spa_prop_add_list(*nvp,
ZPOOL_PROP_GUID, NULL,
spa_guid(spa), src);
spa->spa_comment
!= NULL ?
Yes
No
spa_prop_add_list(*nvp,
ZPOOL_PROP_COMMENT,
spa->spa_comment, 0,
ZPROP_SRC_LOCAL);
spa->spa_root != NULL ?
Yes
No
spa_prop_add_list(*nvp,
ZPOOL_PROP_ALTROOT,
spa->spa_root, 0,
ZPROP_SRC_LOCAL);
(dp = list_head(&spa-
>spa_config_list))
!= NULL ?
Yes
No
dp->scd_path == NULL ?
Yes
No
strcmp(dp->scd_path,
spa_config_path) != 0 ?
Yes
No
spa_prop_add_list(*nvp,
ZPOOL_PROP_CACHEFILE,
"none", 0,
ZPROP_SRC_LOCAL);
spa_prop_add_list(*nvp,
ZPOOL_PROP_CACHEFILE,
dp->scd_path, 0,
ZPROP_SRC_LOCAL);
End

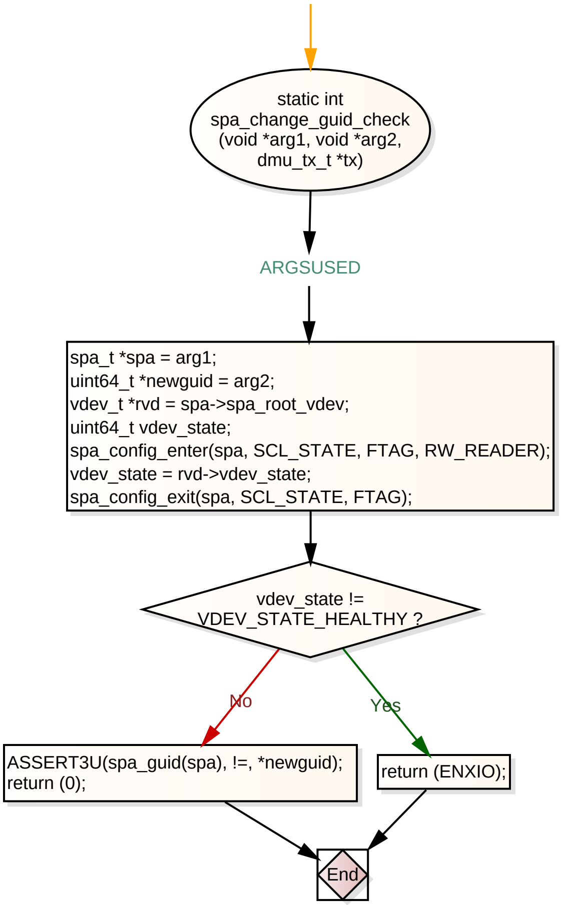

int spa_prop_get(spa_t *spa, nvlist_t **nvp)
* Get zpool property values.
objset_t *mos = spa->spa_meta_objset;
zap_cursor_t zc;
zap_attribute_t za;
int err;
VERIFY(nvlist_alloc(nvp, NV_UNIQUE_NAME, KM_SLEEP) == 0);
mutex_enter(&spa->spa_props_lock);
* Get properties from the spa config.
spa_prop_get_config(spa, nvp);
If no pool property object, no more prop to get.
mos == NULL || spa->spa_pool_props_object == 0 ?
No
Yes
* Get properties from the MOS pool property object.
zap_cursor_init(&zc, mos, spa->spa_pool_props_object)
mutex_exit(&spa->spa_props_lock);
return (0);
(err = zap_cursor_retrieve(&zc, &za)) == 0 ?
Yes
No
uint64_t intval = 0;
char *strval = NULL;
zprop_source_t src = ZPROP_SRC_DEFAULT;
zpool_prop_t prop;
zap_cursor_fini(&zc);
mutex_exit(&spa->spa_props_lock);
(prop = zpool_name_to_prop(za.za_name)) == ZPROP_INVAL ?
out: err && err != ENOENT ?
Yes
No
Yes
No
switch (za.za_integer_length)
nvlist_free(*nvp);
*nvp = NULL;
return (err);
return (0);
End
integer property
8 ?
Yes
No
za.za_first_integer != zpool_prop_default_numeric(prop) ?
1 ?
Yes
No
src = ZPROP_SRC_LOCAL;
default
Yes
prop == ZPOOL_PROP_BOOTFS ?
Yes
No
dsl_pool_t *dp;
dsl_dataset_t *ds = NULL;
dp = spa_get_dsl(spa);
rw_enter(&dp->dp_config_rwlock, RW_READER);
strval = NULL;
intval = za.za_first_integer;
err = dsl_dataset_hold_obj(dp, za.za_first_integer, FTAG, &ds) ?
No
Yes
strval = kmem_alloc(MAXNAMELEN + strlen(MOS_DIR_NAME) + 1, KM_SLEEP);
dsl_dataset_name(ds, strval);
dsl_dataset_rele(ds, FTAG);
rw_exit(&dp->dp_config_rwlock);
string property
rw_exit(&dp->dp_config_rwlock);
spa_prop_add_list(*nvp, prop, strval, intval, src);
strval = kmem_alloc(za.za_num_integers, KM_SLEEP);
err = zap_lookup(mos, spa->spa_pool_props_object, za.za_name, 1, za.za_num_integers, strval);
strval != NULL ?
err ?
Yes
No
No
Yes
kmem_free(strval, MAXNAMELEN + strlen(MOS_DIR_NAME) + 1);
spa_prop_add_list(*nvp, prop, strval, 0, src);
kmem_free(strval, za.za_num_integers);
kmem_free(strval, za.za_num_integers);
zap_cursor_advance(&zc)

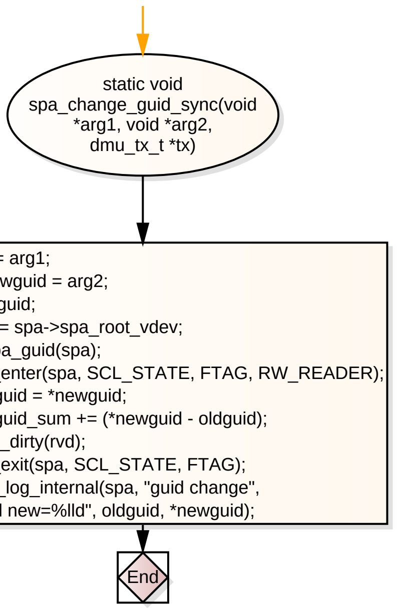

static int spa_prop_validate(spa_t *spa, nvlist_t *props)
* Validate the given pool properties nvlist and modify the list
* for the property values to be set.
nvpair_t *elem;
int error = 0, reset_bootfs = 0;
uint64_t objnum;
boolean_t has_feature = B_FALSE;
elem = NULL;
(elem = nvlist_next_nvpair(props, elem)) != NULL ?
Yes
No
uint64_t intval;
char *strval, *slash, *check, *fname;
const char *propname = nvpair_name(elem);
zpool_prop_t prop = zpool_name_to_prop(propname);
switch (prop)
ZPROP_INVAL ?
No
Yes
ZPOOL_PROP_VERSION ?
No
Yes
error = nvpair_value_uint64(elem, &intval);
!error && (intval < spa_version(spa) || intval > SPA_VERSION_BEFORE_FEATURES || has_feature) ?
Yes
No
error = EINVAL;
ZPOOL_PROP_DELEGATION ?
No
Yes
ZPOOL_PROP_AUTOREPLACE ?
No
Yes
ZPOOL_PROP_LISTSNAPS ?
No
Yes
ZPOOL_PROP_AUTOEXPAND ?
No
Yes
error = nvpair_value_uint64(elem, &intval);
ZPOOL_PROP_BOOTFS ?
Yes
No
* If the pool version is less than SPA_VERSION_BOOTFS,
* or the pool is still being created (version == 0),
* the bootfs property cannot be set.
spa_version(spa) < SPA_VERSION_BOOTFS ?
Yes
No
error = ENOTSUP;
* Make sure the vdev config is bootable
!vdev_is_bootable(spa->spa_root_vdev) ?
Yes
No
error = ENOTSUP;
reset_bootfs = 1;
error = nvpair_value_string(elem, &strval);
!error ?
Yes
No
objset_t *os;
uint64_t compress;
strval == NULL || strval[0] == '\0' ?
Yes
No
Must be ZPL and not gzip compressed.
objnum = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
error = dmu_objset_hold(strval, FTAG, &os) ?
No
Yes
dmu_objset_type(os) != DMU_OST_ZFS ?
No
Yes
(error = dsl_prop_get_integer(strval, zfs_prop_to_name(ZFS_PROP_COMPRESSION), &compress, NULL)) == 0 && !BOOTFS_COMPRESS_VALID(compress) ?
Yes
No
error = ENOTSUP;
objnum = dmu_objset_id(os);
error = ENOTSUP;
dmu_objset_rele(os, FTAG);
!error && intval > 1 ?
Yes
No
error = EINVAL;
ZPOOL_PROP_FAILUREMODE ?
No
Yes
error = nvpair_value_uint64(elem, &intval);
!error && (intval < ZIO_FAILURE_MODE_WAIT || intval > ZIO_FAILURE_MODE_PANIC) ?
Yes
No
error = EINVAL;
* This is a special case which only occurs when
* the pool has completely failed. This allows
* the user to change the in-core failmode property
* without syncing it out to disk (I/Os might
* currently be blocked). We do this by returning
* EIO to the caller (spa_prop_set) to trick it
* into thinking we encountered a property validation
* error.
!error && spa_suspended(spa) ?
Yes
No
spa->spa_failmode = intval;
error = EIO;
ZPOOL_PROP_CACHEFILE ?
No
Yes
(error = nvpair_value_string(elem, &strval)) != 0 ?
No
Yes
strval[0] == '\0' ?
No
Yes
strcmp(strval, "none") == 0 ?
No
Yes
strval[0] != '/' ?
No
Yes
slash = strrchr(strval, '/');
ASSERT(slash != NULL);
slash[1] == '\0' || strcmp(slash, "/.") == 0 || strcmp(slash, "/..") == 0 ?
Yes
No
error = EINVAL;
error = EINVAL;
ZPOOL_PROP_COMMENT ?
Yes
No
(error = nvpair_value_string(elem, &strval)) != 0 ?
No
Yes
check = strval;
* The kernel doesn't have an easy isprint()
* check. For this kernel check, we merely
* check ASCII apart from DEL. Fix this if
* there is an easy-to-use kernel isprint().
*check != '\0' ?
Yes
No
*check >= 0x7f ?
Yes
No
error = EINVAL;
check++;
check++
strlen(strval) > ZPROP_MAX_COMMENT ?
Yes
No
error = E2BIG;
ZPOOL_PROP_DEDUPDITTO ?
Yes
default
spa_version(spa) < SPA_VERSION_DEDUP ?
No
Yes
error = nvpair_value_uint64(elem, &intval);
error = ENOTSUP;
error == 0 && intval != 0 && intval < ZIO_DEDUPDITTO_MIN ?
Yes
No
error = EINVAL;
* Sanitize the input.
!zpool_prop_feature(propname) ?
Yes
No
error = EINVAL;
nvpair_type(elem) != DATA_TYPE_UINT64 ?
No
Yes
error = EINVAL;
nvpair_value_uint64(elem, &intval) != 0 ?
No
Yes
error = EINVAL;
intval != 0 ?
No
Yes
error = EINVAL;
fname = strchr(propname, '@') + 1;
zfeature_lookup_name(fname, NULL) != 0 ?
Yes
No
error = EINVAL;
has_feature = B_TRUE;
error ?
Yes
No
!error && reset_bootfs ?
Yes
No
error = nvlist_remove(props, zpool_prop_to_name(ZPOOL_PROP_BOOTFS), DATA_TYPE_STRING);
!error ?
Yes
No
error = nvlist_add_uint64(props, zpool_prop_to_name(ZPOOL_PROP_BOOTFS), objnum);
return (error);
End


void spa_configfile_set (spa_t *spa, nvlist_t *nvp, boolean_t need_sync)
char *cachefile;
spa_config_dirent_t *dp;
nvlist_lookup_string(nvp, zpool_prop_to_name(ZPOOL_PROP_CACHEFILE), &cachefile) != 0 ?
No
dp = kmem_alloc(sizeof (spa_config_dirent_t), KM_SLEEP);
cachefile[0] == '\0' ?
Yes
No
strcmp(cachefile, "none") == 0 ?
Yes
dp->scd_path = spa_strdup(spa_config_path);
Yes
No
dp->scd_path = NULL;
dp->scd_path = spa_strdup(cachefile);
list_insert_head(&spa->spa_config_list, dp);
need_sync ?
No
Yes
spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
End


int spa_prop_set(spa_t *spa, nvlist_t *nvp)
int error;
nvpair_t *elem = NULL;
boolean_t need_sync = B_FALSE;
(error = spa_prop_validate(spa, nvp)) != 0 ?
Yes
No
return (error);
(elem = nvlist_next_ nvpair(nvp, elem)) != NULL ?
Yes
zpool_prop_t prop = zpool_name_to_prop (nvpair_name(elem));
Yes
No
No
prop == ZPOOL_PROP_ CACHEFILE || prop == ZPOOL_PROP_ALTROOT || prop == ZPOOL_ PROP_READONLY ?
No
prop == ZPOOL_PROP_ VERSION || prop == ZPROP_INVAL ?
Yes
No
uint64_t ver;
need_sync = B_TRUE;
need_sync ?
prop == ZPOOL_PROP_ VERSION ?
No
Yes
No
ASSERT(zpool_prop_feature(nvpair_name(elem)));
ver = SPA_VERSION_FEATURES;
need_sync = B_TRUE;
VERIFY(nvpair_value_ uint64(elem, &ver) == 0);
Save time if the version is already set.
Yes
Yes
return (0);
ver == spa_version(spa) ?
* In addition to the pool directory object, we might
* create the pool properties object, the features for
* read object, the features for write object, or the
* feature descriptions object.
No
return (dsl_sync_task_do (spa_get_dsl(spa), NULL, spa_sync_props, spa, nvp, 6));
error = dsl_sync_task_do (spa_get_dsl(spa), NULL, spa_sync_version, spa, &ver, 6);
error ?
Yes
return (error);
End

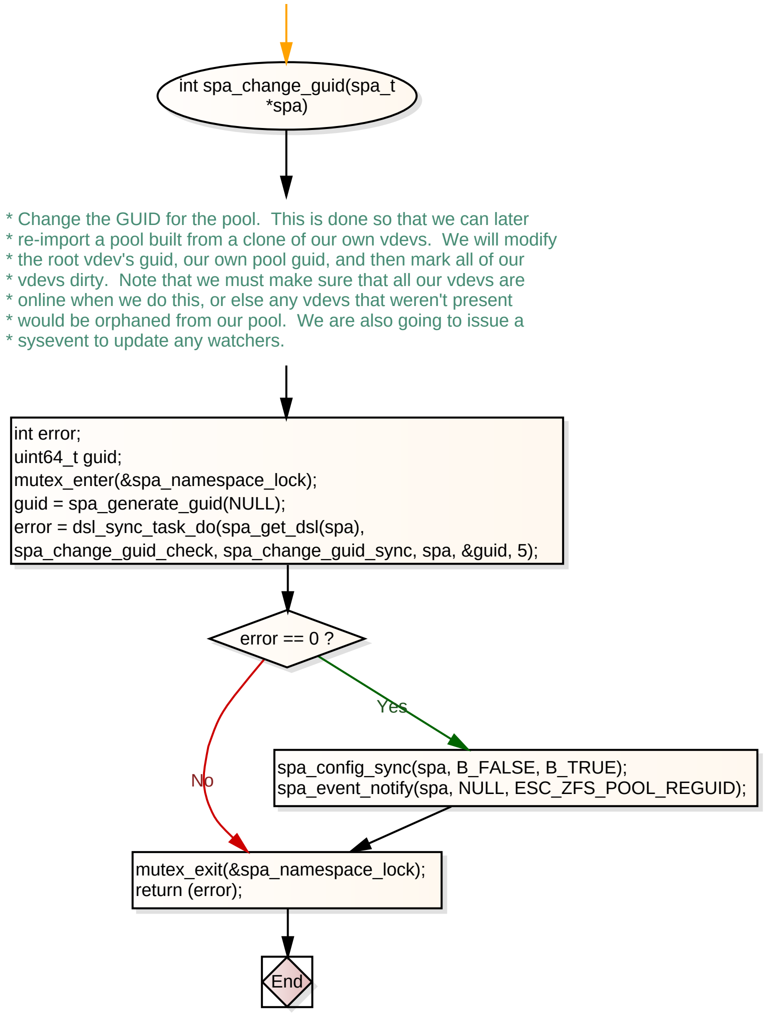

void spa_prop_clear_ bootfs(spa_t *spa, uint64_t dsobj, dmu_tx_t *tx)
* If the bootfs property value is dsobj, clear it.
spa->spa_bootfs == dsobj && spa->spa_pool_ props_object != 0 ?
Yes
No
VERIFY(zap_remove(spa->spa_meta_objset,
spa->spa_pool_props_object,
zpool_prop_to_name(ZPOOL_PROP_BOOTFS), tx) == 0);
spa->spa_bootfs = 0;
End


static int
spa_change_guid_check
(void *arg1, void *arg2,
dmu_tx_t *tx)
ARGSUSED
spa_t *spa = arg1;
uint64_t *newguid = arg2;
vdev_t *rvd = spa->spa_root_vdev;
uint64_t vdev_state;
spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
vdev_state = rvd->vdev_state;
spa_config_exit(spa, SCL_STATE, FTAG);
vdev_state !=
VDEV_STATE_HEALTHY ?
No
Yes
ASSERT3U(spa_guid(spa), !=, *newguid);
return (0);
return (ENXIO);
End



```
static void
spa_change_guid_sync(void
*arg1, void *arg2,
dmu_tx_t *tx)
```

```
spa_t *spa = arg1;
uint64_t *newguid = arg2;
uint64_t oldguid;
vdev_t *rvd = spa->spa_root_vdev;
oldguid = spa_guid(spa);
spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
rvd->vdev_guid = *newguid;
rvd->vdev_guid_sum += (*newguid - oldguid);
vdev_config_dirty(rvd);
spa_config_exit(spa, SCL_STATE, FTAG);
spa_history_log_internal(spa, "guid change",
tx, "old=%lld new=%lld", oldguid, *newguid);
```

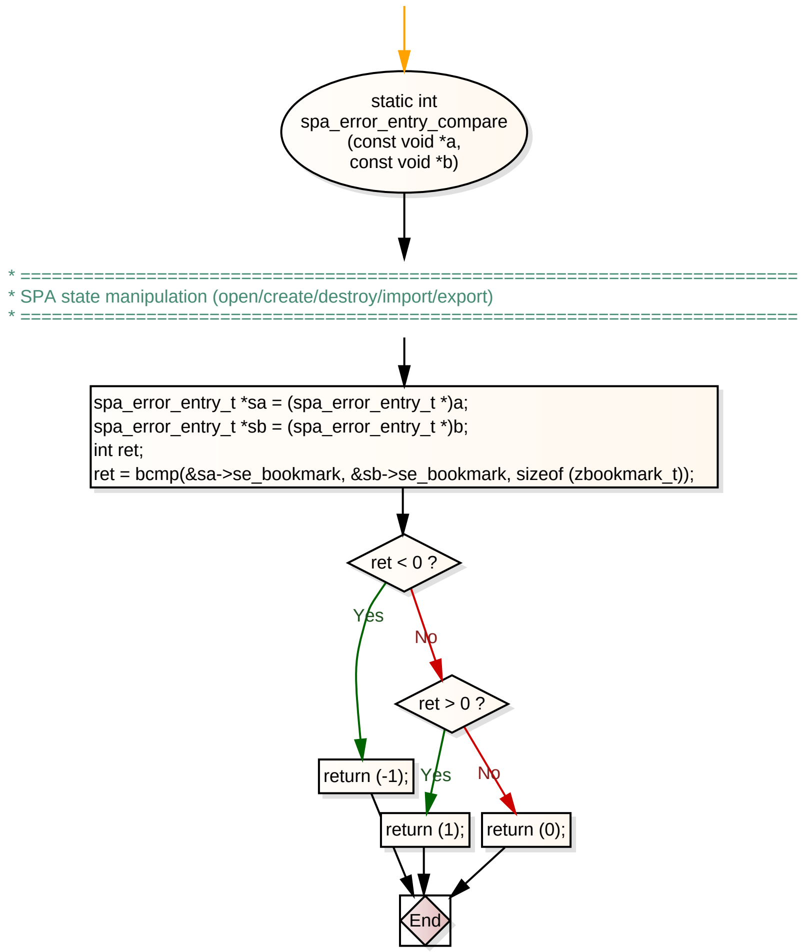


```
* Change the GUID for the pool.  This is done so that we can later
* re-import a pool built from a clone of our own vdevs.  We will modify
* the root vdev's guid, our own pool guid, and then mark all of our
* vdevs dirty.  Note that we must make sure that all our vdevs are
* online when we do this, or else any vdevs that weren't present
* would be orphaned from our pool.  We are also going to issue a
* sysevent to update any watchers.
```

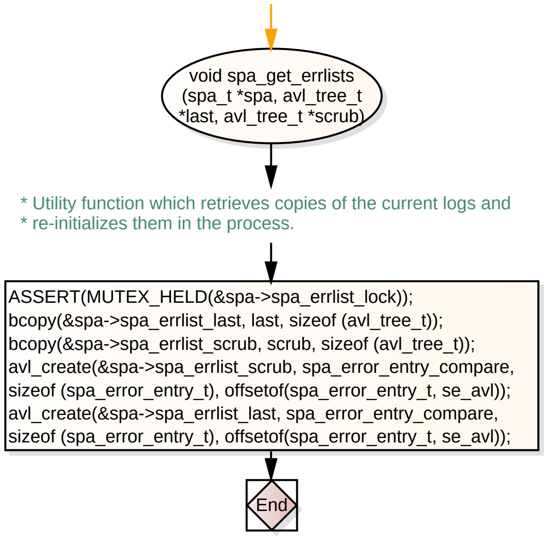
```
int error;
uint64_t guid;
mutex_enter(&spa_namespace_lock);
guid = spa_generate_guid(NULL);
error = dsl_sync_task_do(spa_get_dsl(spa),
spa_change_guid_check, spa_change_guid_sync, spa, &guid, 5);
```

```
spa_config_sync(spa, B_FALSE, B_TRUE);
spa_event_notify(spa, NULL, ESC_ZFS_POOL_REGUID);
```

```
mutex_exit(&spa_namespace_lock);
return (error);
```



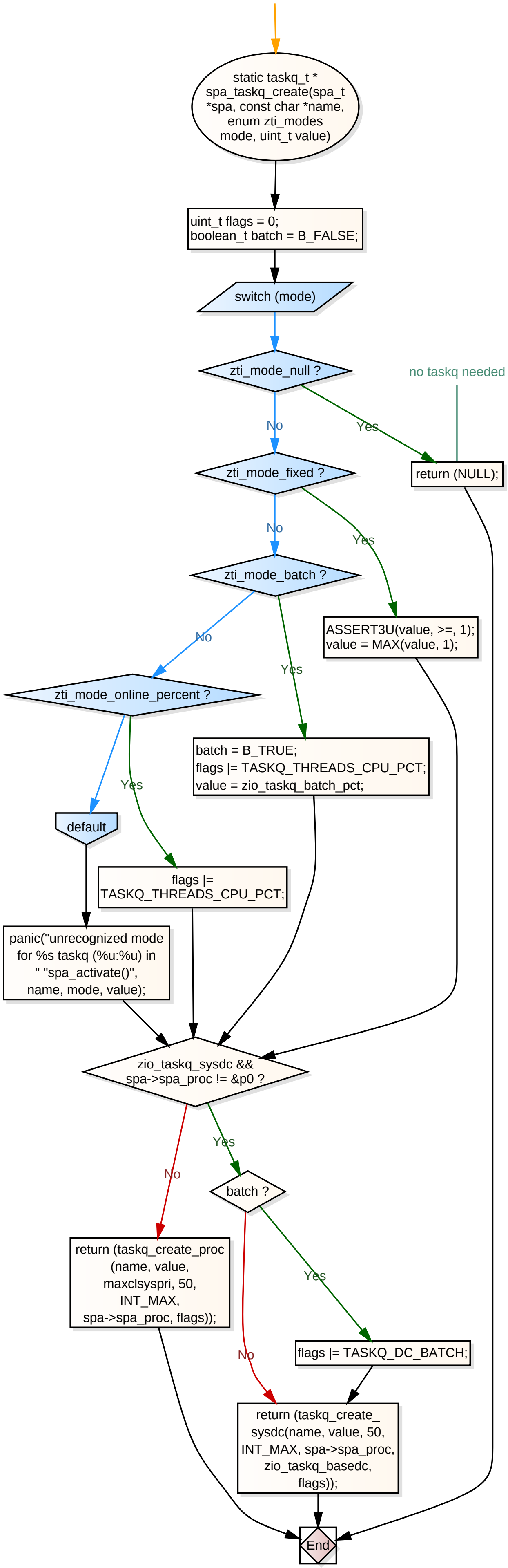
```
spa_error_entry_t *sa = (spa_error_entry_t *)a;
spa_error_entry_t *sb = (spa_error_entry_t *)b;
int ret;
ret = bcmp(&sa->se_bookmark, &sb->se_bookmark, sizeof (zbookmark_t));
```

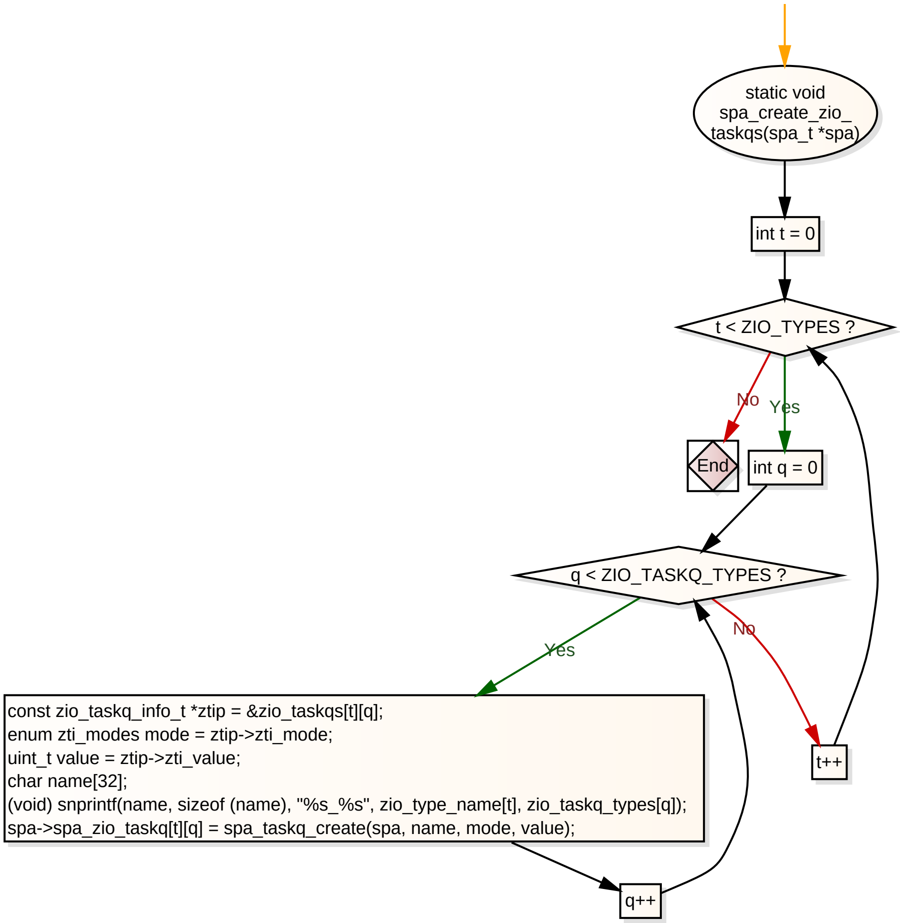


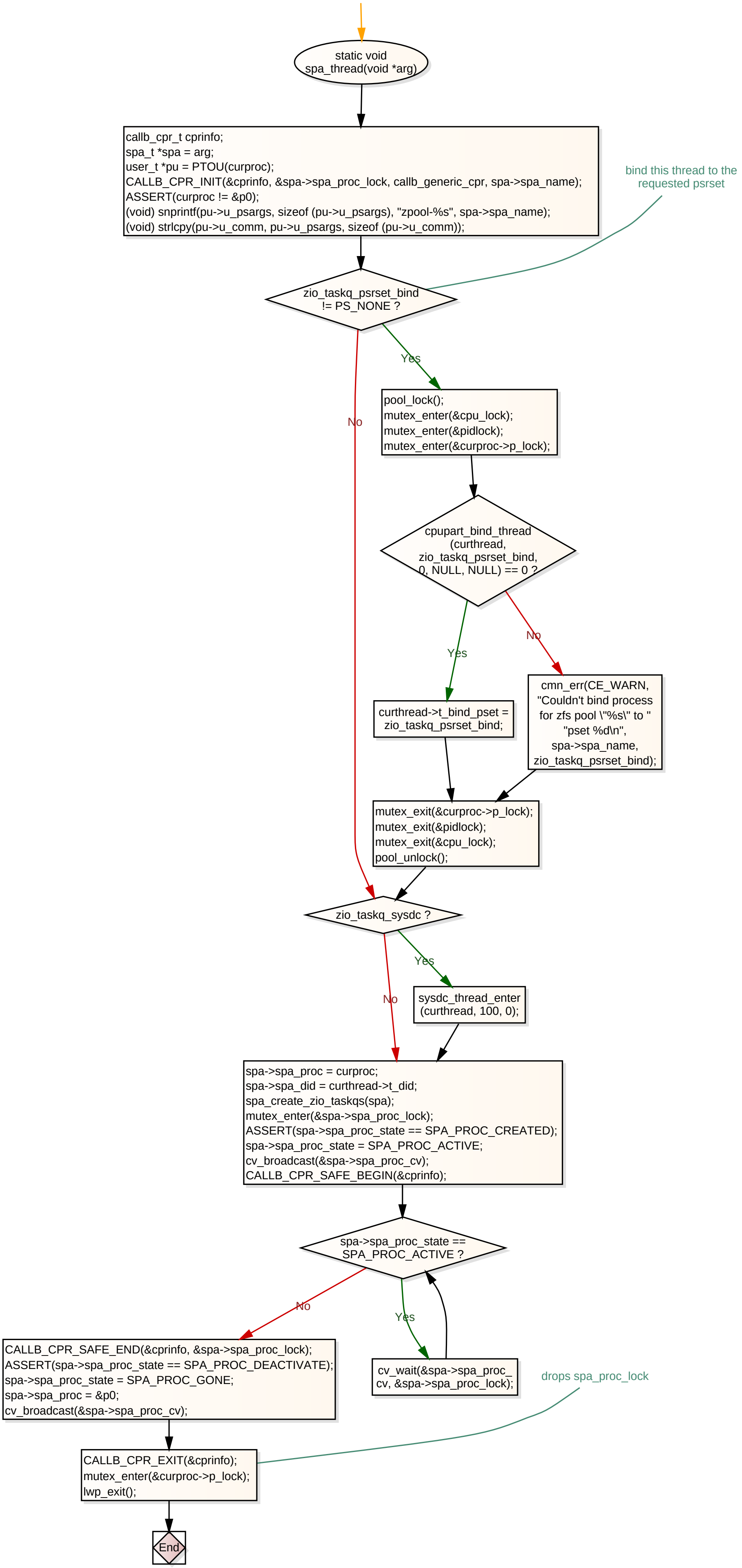
```
void spa_get_errlists
(spa_t *spa, avl_tree_t
*last, avl_tree_t *scrub)
```

* Utility function which retrieves copies of the current logs and
* re-initializes them in the process.

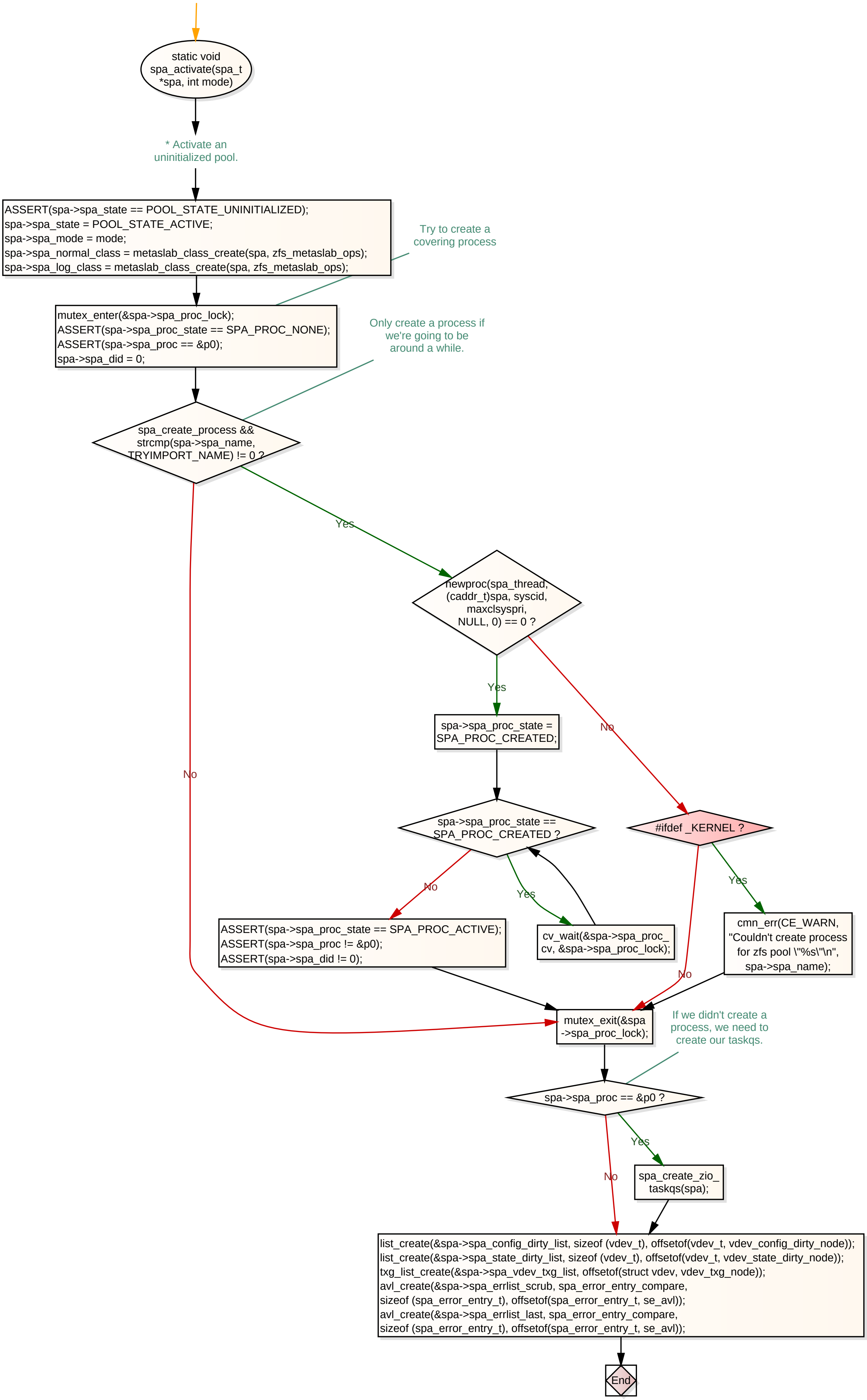
```
ASSERT(MUTEX_HELD(&spa->spa_errlist_lock));
bcopy(&spa->spa_errlist_last, last, sizeof (avl_tree_t));
bcopy(&spa->spa_errlist_scrub, scrub, sizeof (avl_tree_t));
avl_create(&spa->spa_errlist_scrub, spa_error_entry_compare,
sizeof (spa_error_entry_t), offsetof(spa_error_entry_t, se_avl));
avl_create(&spa->spa_errlist_last, spa_error_entry_compare,
sizeof (spa_error_entry_t), offsetof(spa_error_entry_t, se_avl));
```

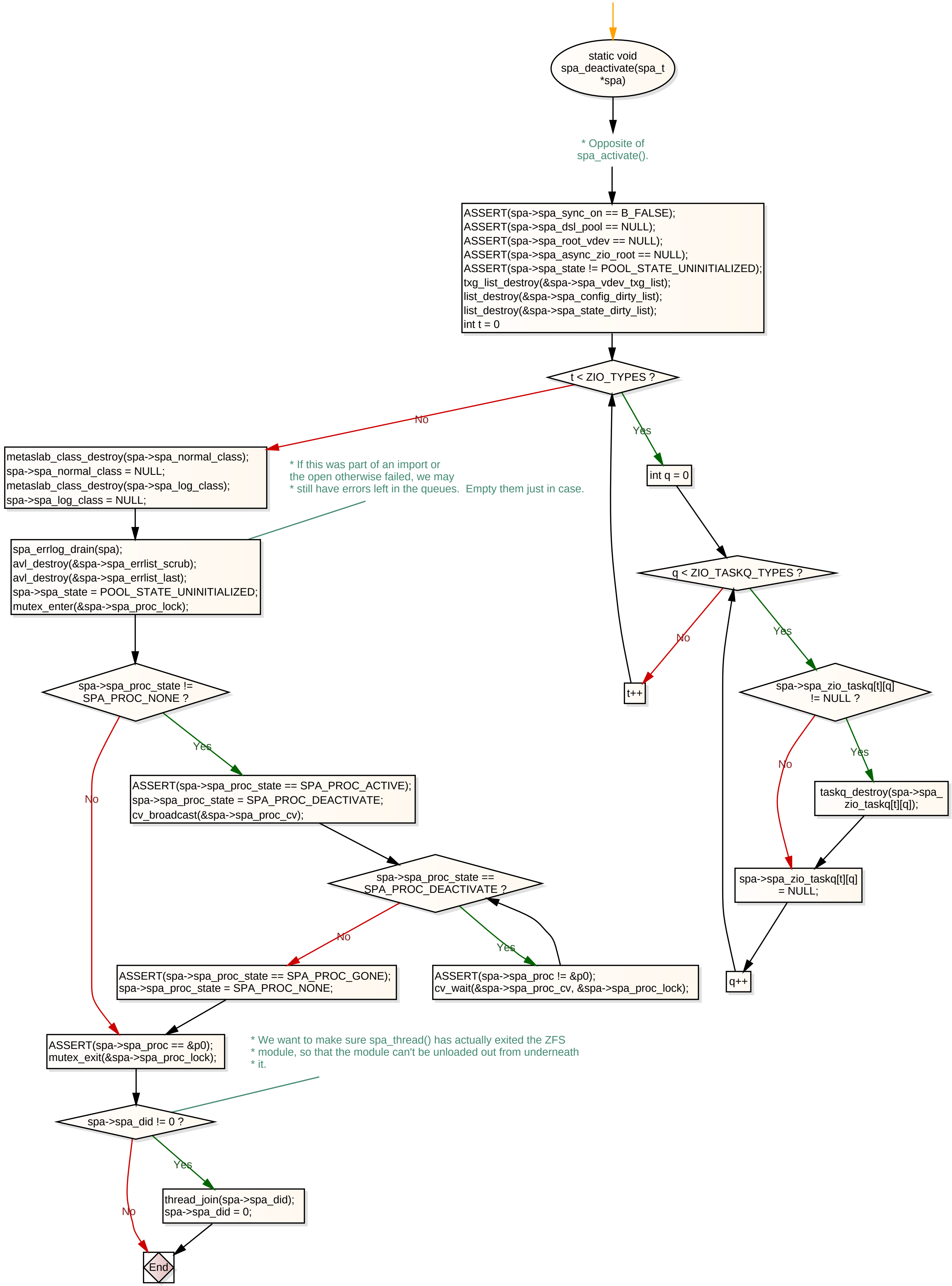

static taskq_t *
spa_taskq_create(spa_t
*spa, const char *name,
enum zti_modes
mode, uint_t value)
uint_t flags = 0;
boolean_t batch = B_FALSE;
switch (mode)
zti_mode_null ?
no taskq needed
Yes
return (NULL);
No
zti_mode_fixed ?
Yes
ASSERT3U(value, >=, 1);
value = MAX(value, 1);
No
zti_mode_batch ?
No
Yes
zti_mode_online_percent ?
batch = B_TRUE;
flags |= TASKQ_THREADS_CPU_PCT;
value = zio_taskq_batch_pct;
Yes
default
flags |=
TASKQ_THREADS_CPU_PCT;
panic("unrecognized mode
for %s taskq (%u:%u) in
" "spa_activate()",
name, mode, value);
zio_taskq_sysdc &&
spa->spa_proc != &p0 ?
Yes
No
batch ?
Yes
return (taskq_create_proc
(name, value,
maxclsyspri, 50,
INT_MAX,
spa->spa_proc, flags));
flags |= TASKQ_DC_BATCH;
No
return (taskq_create_
sysdc(name, value, 50,
INT_MAX, spa->spa_proc,
zio_taskq_basedc,
flags));
End


static void spa_create_zio_ taskqs(spa_t *spa)
int t = 0
t < ZIO_TYPES ?
No
Yes
End
int q = 0
q < ZIO_TASKQ_TYPES ?
Yes
No
const zio_taskq_info_t *ztip = &zio_taskqs[t][q];
enum zti_modes mode = ztip->zti_mode;
uint_t value = ztip->zti_value;
char name[32];
(void) snprintf(name, sizeof (name), "%s_%s", zio_type_name[t], zio_taskq_types[q]);
spa->spa_zio_taskq[t][q] = spa_taskq_create(spa, name, mode, value);
t++
q++



```
static void
spa_thread(void *arg)

callb_cpr_t cprinfo;
spa_t *spa = arg;
user_t *pu = PTOU(curproc);
CALLB_CPR_INIT(&cprinfo, &spa->spa_proc_lock, callb_generic_cpr, spa->spa_name);
ASSERT(curproc != &p0);
(void) snprintf(pu->u_psargs, sizeof (pu->u_psargs), "zpool-%s", spa->spa_name);
(void) strlcpy(pu->u_comm, pu->u_psargs, sizeof (pu->u_comm));

zio_taskq_psrset_bind
!= PS_NONE ?

pool_lock();
mutex_enter(&cpu_lock);
mutex_enter(&pidlock);
mutex_enter(&curproc->p_lock);

cpupart_bind_thread
(curthread,
zio_taskq_psrset_bind,
0, NULL, NULL) == 0 ?

curthread->t_bind_pset =
zio_taskq_psrset_bind;

cmn_err(CE_WARN,
"Couldn't bind process
for zfs pool \"%s\" to "
"pset %d\n",
spa->spa_name,
zio_taskq_psrset_bind);

mutex_exit(&curproc->p_lock);
mutex_exit(&pidlock);
mutex_exit(&cpu_lock);
pool_unlock();

zio_taskq_sysdc ?

sysdc_thread_enter
(curthread, 100, 0);

spa->spa_proc = curproc;
spa->spa_did = curthread->t_did;
spa_create_zio_taskqs(spa);
mutex_enter(&spa->spa_proc_lock);
ASSERT(spa->spa_proc_state == SPA_PROC_CREATED);
spa->spa_proc_state = SPA_PROC_ACTIVE;
cv_broadcast(&spa->spa_proc_cv);
CALLB_CPR_SAFE_BEGIN(&cprinfo);

spa->spa_proc_state ==
SPA_PROC_ACTIVE ?

cv_wait(&spa->spa_proc_
cv, &spa->spa_proc_lock);

CALLB_CPR_SAFE_END(&cprinfo, &spa->spa_proc_lock);
ASSERT(spa->spa_proc_state == SPA_PROC_DEACTIVATE);
spa->spa_proc_state = SPA_PROC_GONE;
spa->spa_proc = &p0;
cv_broadcast(&spa->spa_proc_cv);

CALLB_CPR_EXIT(&cprinfo);
mutex_enter(&curproc->p_lock);
lwp_exit();
```


static void spa_activate(spa_t *spa, int mode)
* Activate an uninitialized pool.
ASSERT(spa->spa_state == POOL_STATE_UNINITIALIZED);
spa->spa_state = POOL_STATE_ACTIVE;
spa->spa_mode = mode;
spa->spa_normal_class = metaslab_class_create(spa, zfs_metaslab_ops);
spa->spa_log_class = metaslab_class_create(spa, zfs_metaslab_ops);
Try to create a covering process
mutex_enter(&spa->spa_proc_lock);
ASSERT(spa->spa_proc_state == SPA_PROC_NONE);
ASSERT(spa->spa_proc == &p0);
spa->spa_did = 0;
Only create a process if we're going to be around a while.
spa_create_process && strcmp(spa->spa_name, TRYIMPORT_NAME) != 0 ?
Yes
No
newproc(spa_thread, (caddr_t)spa, syscid, maxclsyspri, NULL, 0) == 0 ?
Yes
No
spa->spa_proc_state = SPA_PROC_CREATED;
spa->spa_proc_state == SPA_PROC_CREATED ?
No
Yes
ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
ASSERT(spa->spa_proc != &p0);
ASSERT(spa->spa_did != 0);
cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
#ifdef _KERNEL ?
Yes
No
cmn_err(CE_WARN, "Couldn't create process for zfs pool \"%s\"\n", spa->spa_name);
mutex_exit(&spa->spa_proc_lock);
If we didn't create a process, we need to create our taskqs.
spa->spa_proc == &p0 ?
Yes
No
spa_create_zio_taskqs(spa);
list_create(&spa->spa_config_dirty_list, sizeof (vdev_t), offsetof(vdev_t, vdev_config_dirty_node));
list_create(&spa->spa_state_dirty_list, sizeof (vdev_t), offsetof(vdev_t, vdev_state_dirty_node));
txg_list_create(&spa->spa_vdev_txg_list, offsetof(struct vdev, vdev_txg_node));
avl_create(&spa->spa_errlist_scrub, spa_error_entry_compare,
sizeof (spa_error_entry_t), offsetof(spa_error_entry_t, se_avl));
avl_create(&spa->spa_errlist_last, spa_error_entry_compare,
sizeof (spa_error_entry_t), offsetof(spa_error_entry_t, se_avl));
End



```
static void
spa_deactivate(spa_t
*spa)
```

* Opposite of
spa_activate().

```
ASSERT(spa->spa_sync_on == B_FALSE);
ASSERT(spa->spa_dsl_pool == NULL);
ASSERT(spa->spa_root_vdev == NULL);
ASSERT(spa->spa_async_zio_root == NULL);
ASSERT(spa->spa_state != POOL_STATE_UNINITIALIZED);
txg_list_destroy(&spa->spa_vdev_txg_list);
list_destroy(&spa->spa_config_dirty_list);
list_destroy(&spa->spa_state_dirty_list);
int t = 0
```

```
t < ZIO_TYPES ?
```

```
int q = 0
```

```
q < ZIO_TASKQ_TYPES ?
```

```
spa->spa_zio_taskq[t][q]
!= NULL ?
```

```
taskq_destroy(spa->spa_
zio_taskq[t][q]);
```

```
spa->spa_zio_taskq[t][q]
= NULL;
```

```
q++
```

```
t++
```

```
metaslab_class_destroy(spa->spa_normal_class);
spa->spa_normal_class = NULL;
metaslab_class_destroy(spa->spa_log_class);
spa->spa_log_class = NULL;
```

* If this was part of an import or
the open otherwise failed, we may
* still have errors left in the queues. Empty them just in case.

```
spa_errlog_drain(spa);
avl_destroy(&spa->spa_errlist_scrub);
avl_destroy(&spa->spa_errlist_last);
spa->spa_state = POOL_STATE_UNINITIALIZED;
mutex_enter(&spa->spa_proc_lock);
```

```
spa->spa_proc_state !=
SPA_PROC_NONE ?
```

```
ASSERT(spa->spa_proc_state == SPA_PROC_ACTIVE);
spa->spa_proc_state = SPA_PROC_DEACTIVATE;
cv_broadcast(&spa->spa_proc_cv);
```

```
spa->spa_proc_state ==
SPA_PROC_DEACTIVATE ?
```

```
ASSERT(spa->spa_proc != &p0);
cv_wait(&spa->spa_proc_cv, &spa->spa_proc_lock);
```

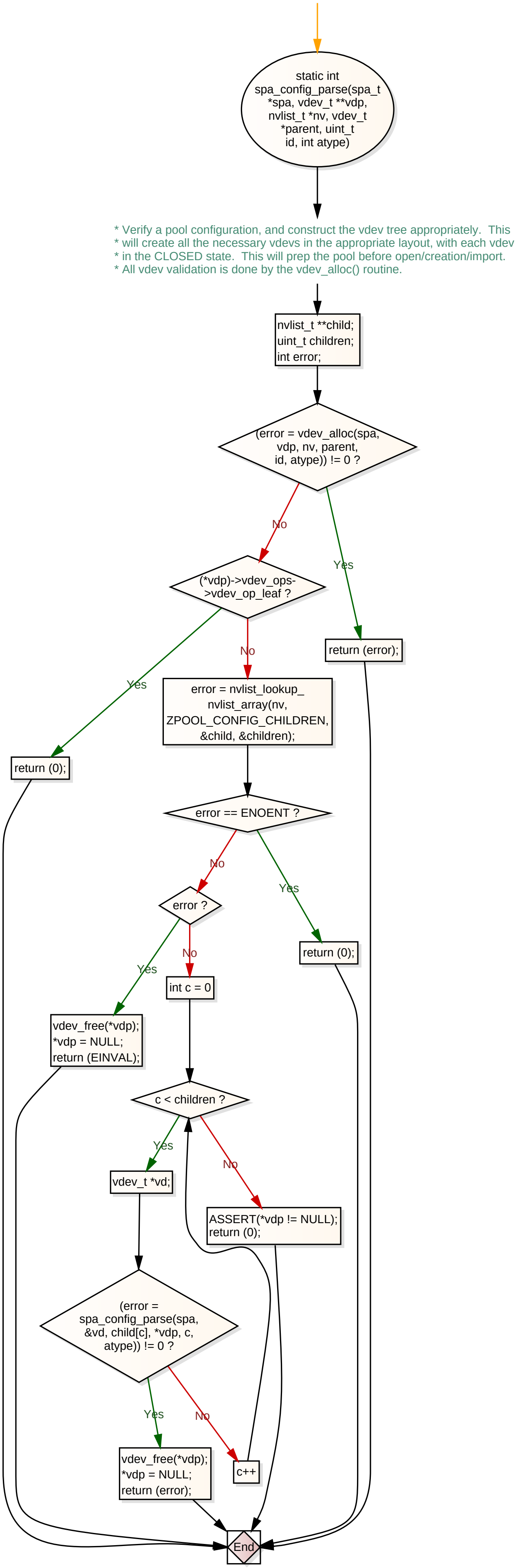
```
ASSERT(spa->spa_proc_state == SPA_PROC_GONE);
spa->spa_proc_state = SPA_PROC_NONE;
```

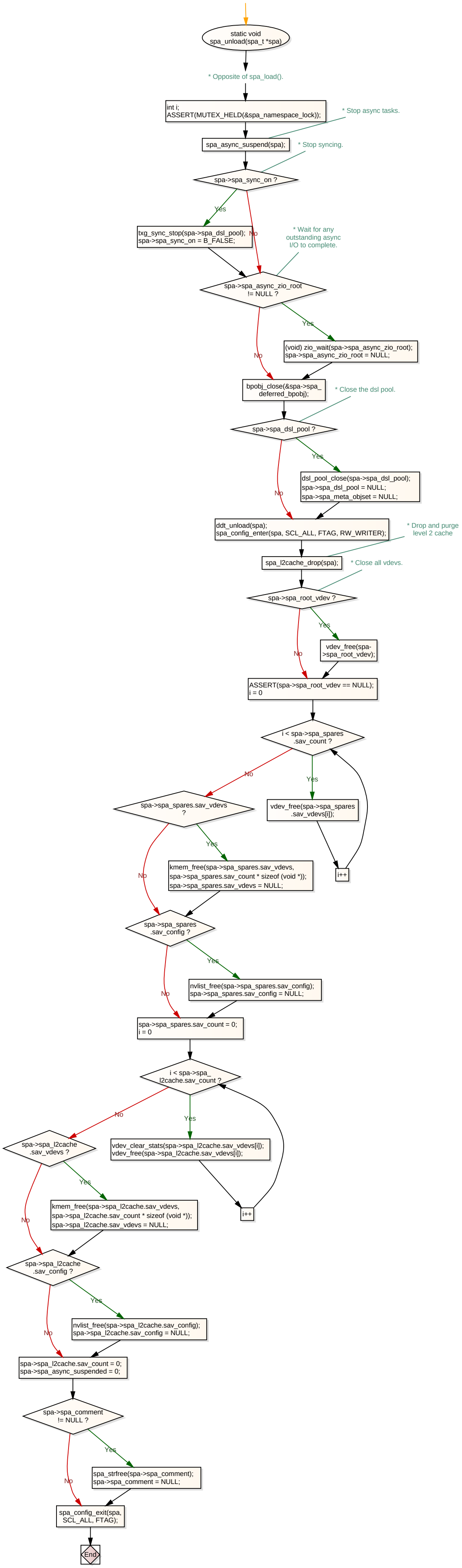
```
ASSERT(spa->spa_proc == &p0);
mutex_exit(&spa->spa_proc_lock);
```

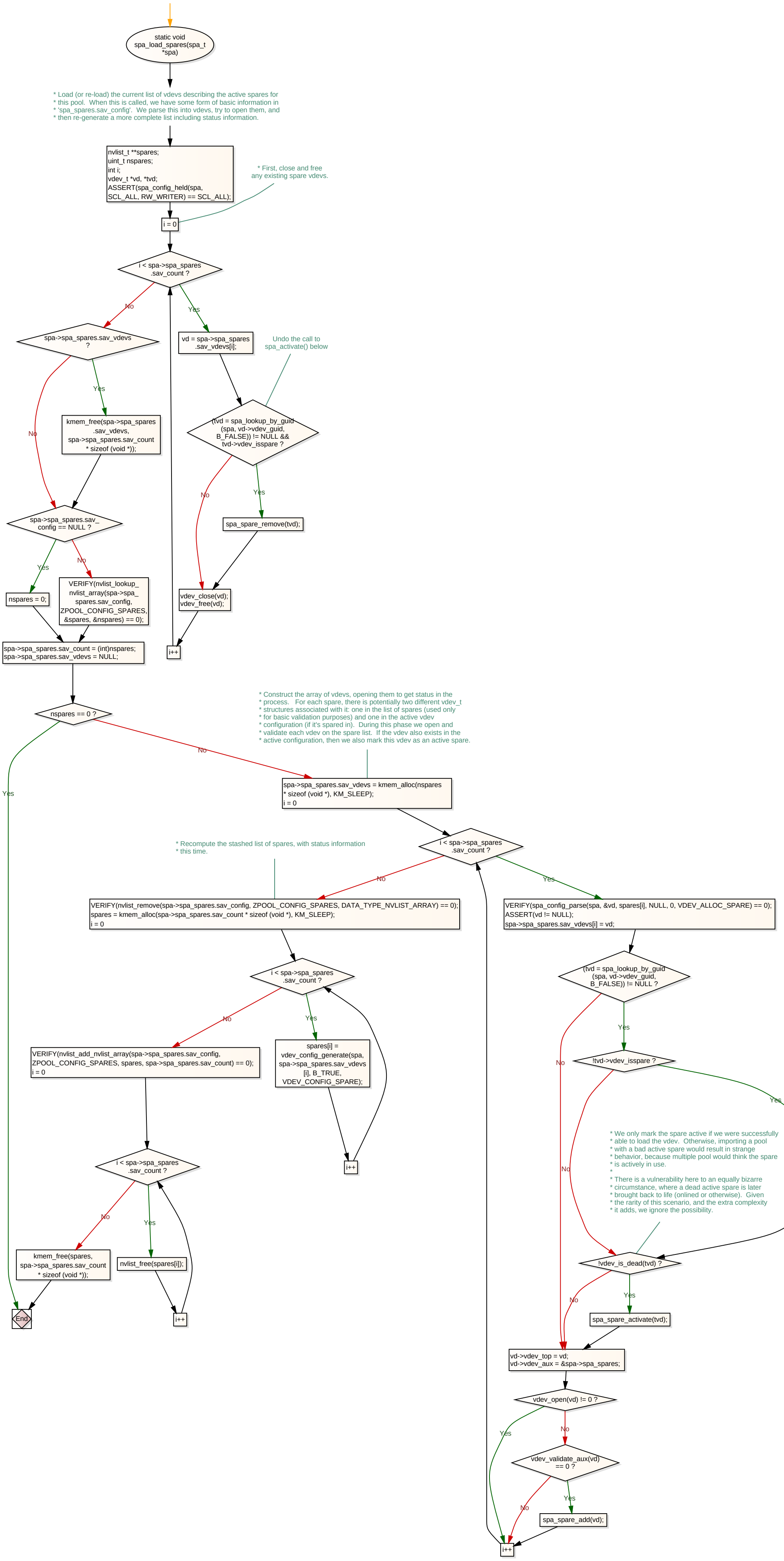
* We want to make sure spa_thread() has actually exited the ZFS
* module, so that the module can't be unloaded out from underneath
* it.

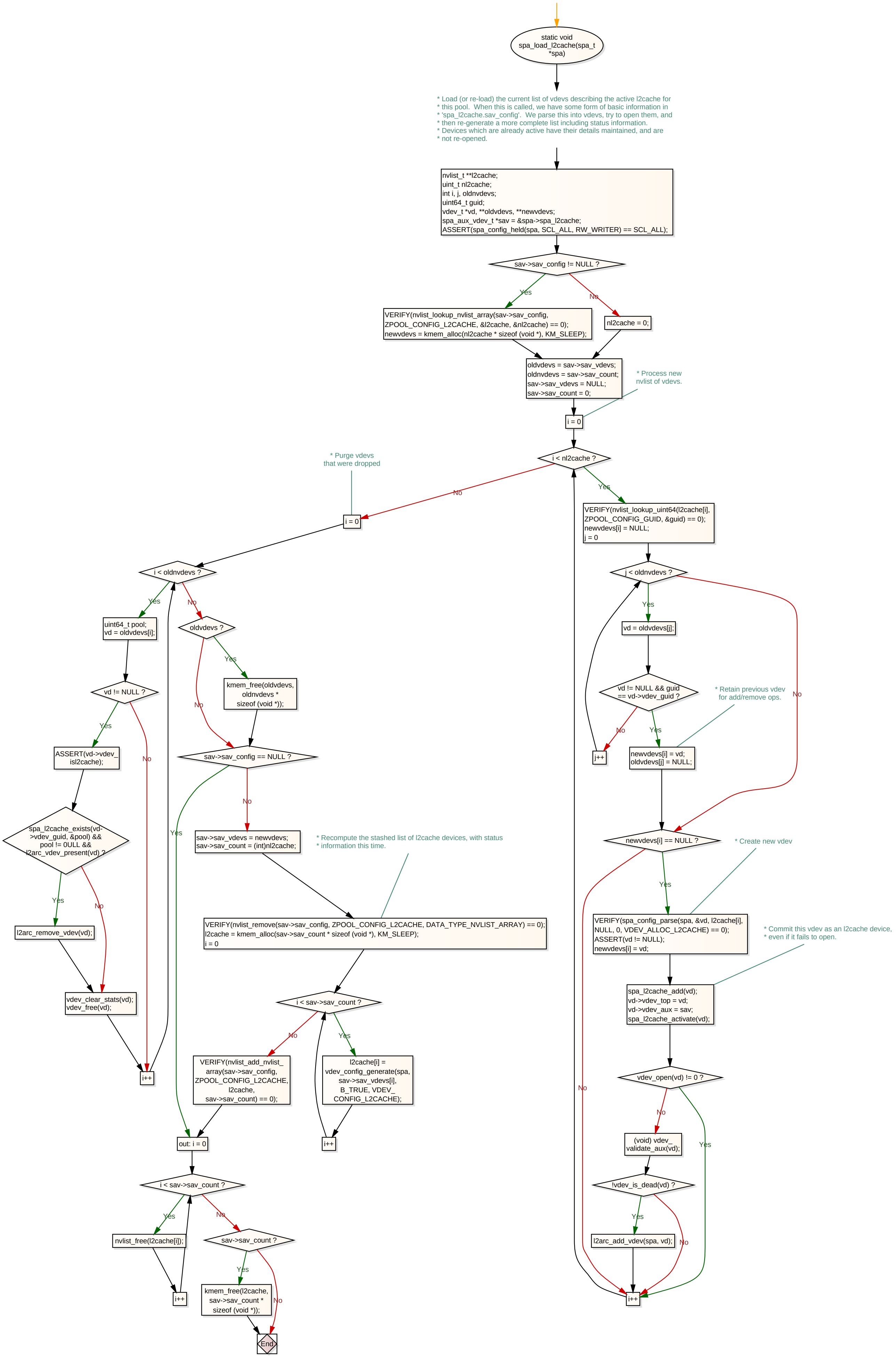
```
spa->spa_did != 0 ?
```

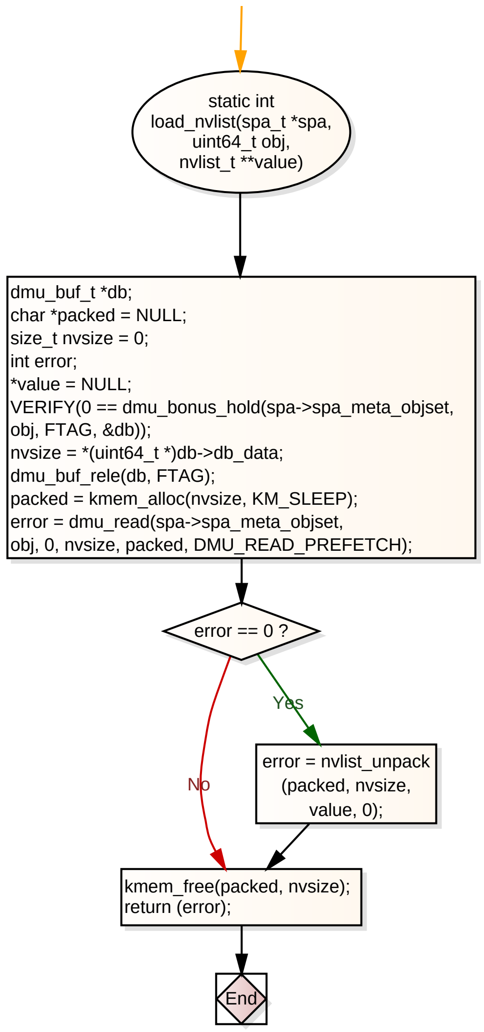
```
thread_join(spa->spa_did);
spa->spa_did = 0;
```

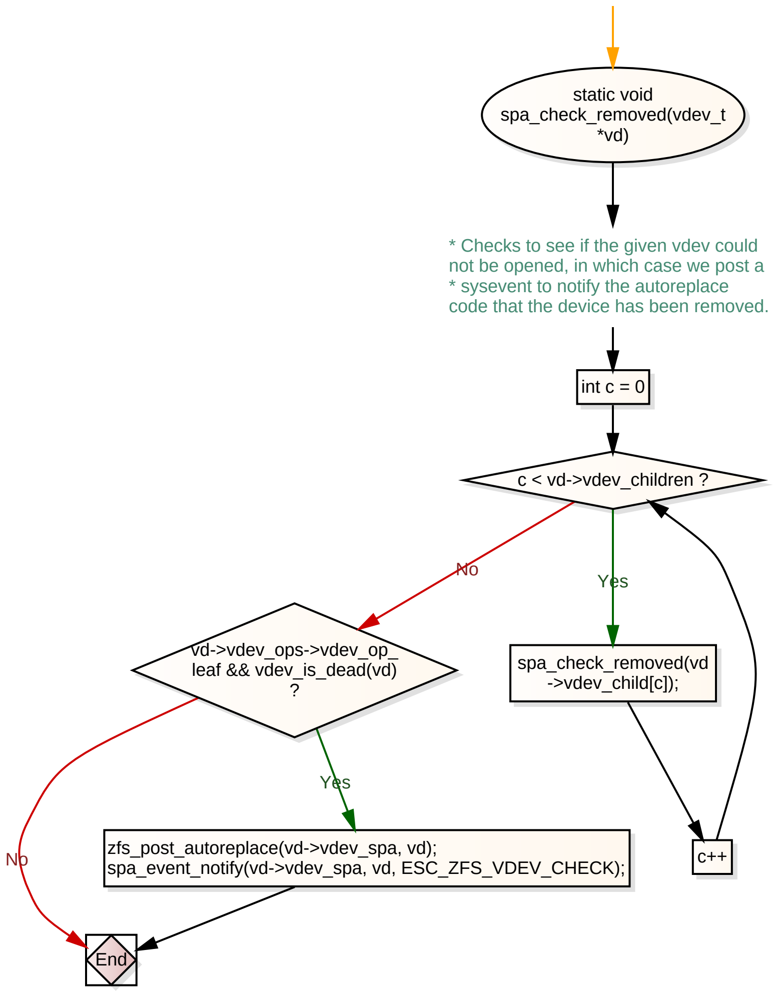

static int spa_config_parse(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent, uint_t id, int atype)
* Verify a pool configuration, and construct the vdev tree appropriately. This
* will create all the necessary vdevs in the appropriate layout, with each vdev
* in the CLOSED state. This will prep the pool before open/creation/import.
* All vdev validation is done by the vdev_alloc() routine.
nvlist_t **child;
uint_t children;
int error;
(error = vdev_alloc(spa, vdp, nv, parent, id, atype)) != 0 ?
No
Yes
(*vdp)->vdev_ops->vdev_op_leaf ?
return (error);
Yes
No
error = nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN, &child, &children);
return (0);
error == ENOENT ?
No
Yes
error ?
return (0);
Yes
No
int c = 0
vdev_free(*vdp);
*vdp = NULL;
return (EINVAL);
c < children ?
Yes
No
vdev_t *vd;
ASSERT(*vdp != NULL);
return (0);
(error = spa_config_parse(spa, &vd, child[c], *vdp, c, atype)) != 0 ?
Yes
No
vdev_free(*vdp);
*vdp = NULL;
return (error);
c++
End

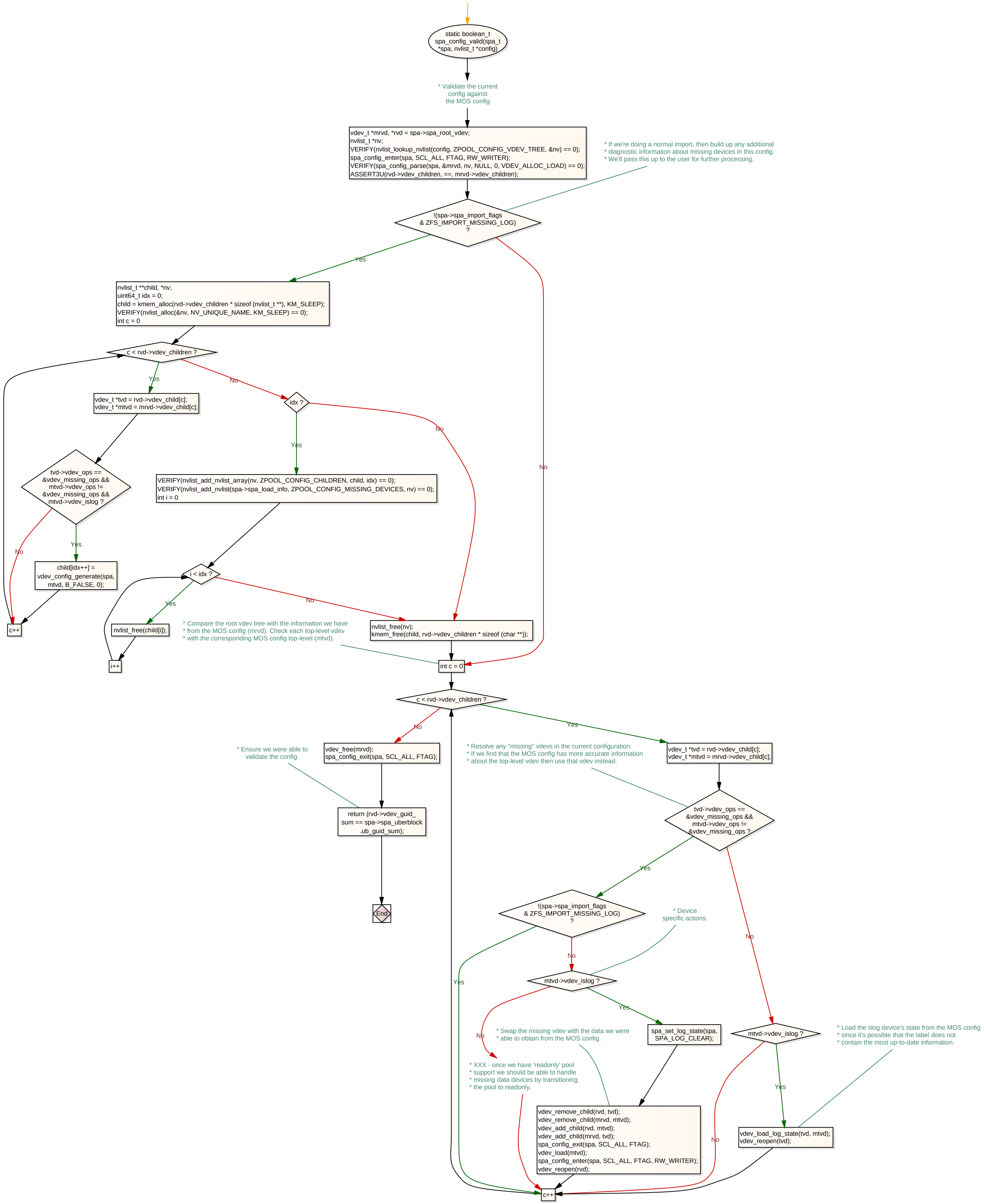

static void
spa_unload(spa_t *spa)
* Opposite of spa_load().
int i;
ASSERT(MUTEX_HELD(&spa_namespace_lock));
* Stop async tasks.
spa_async_suspend(spa);
* Stop syncing.
spa->spa_sync_on ?
Yes
No
txg_sync_stop(spa->spa_dsl_pool);
spa->spa_sync_on = B_FALSE;
* Wait for any
outstanding async
I/O to complete.
spa->spa_async_zio_root
!= NULL ?
Yes
No
(void) zio_wait(spa->spa_async_zio_root);
spa->spa_async_zio_root = NULL;
bpobj_close(&spa->spa_
deferred_bpobj);
* Close the dsl pool.
spa->spa_dsl_pool ?
Yes
No
dsl_pool_close(spa->spa_dsl_pool);
spa->spa_dsl_pool = NULL;
spa->spa_meta_objset = NULL;
ddt_unload(spa);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
* Drop and purge
level 2 cache
spa_l2cache_drop(spa);
* Close all vdevs.
spa->spa_root_vdev ?
Yes
No
vdev_free(spa-
>spa_root_vdev);
ASSERT(spa->spa_root_vdev == NULL);
i = 0
i < spa->spa_spares
.sav_count ?
No
Yes
spa->spa_spares.sav_vdevs
?
vdev_free(spa->spa_spares
.sav_vdevs[i]);
Yes
No
kmem_free(spa->spa_spares.sav_vdevs,
spa->spa_spares.sav_count * sizeof (void *));
spa->spa_spares.sav_vdevs = NULL;
i++
spa->spa_spares
.sav_config ?
Yes
No
nvlist_free(spa->spa_spares.sav_config);
spa->spa_spares.sav_config = NULL;
spa->spa_spares.sav_count = 0;
i = 0
i < spa->spa_
l2cache.sav_count ?
No
Yes
spa->spa_l2cache
.sav_vdevs ?
vdev_clear_stats(spa->spa_l2cache.sav_vdevs[i]);
vdev_free(spa->spa_l2cache.sav_vdevs[i]);
Yes
kmem_free(spa->spa_l2cache.sav_vdevs,
spa->spa_l2cache.sav_count * sizeof (void *));
spa->spa_l2cache.sav_vdevs = NULL;
No
i++
spa->spa_l2cache
.sav_config ?
Yes
nvlist_free(spa->spa_l2cache.sav_config);
spa->spa_l2cache.sav_config = NULL;
No
spa->spa_l2cache.sav_count = 0;
spa->spa_async_suspended = 0;
spa->spa_comment
!= NULL ?
Yes
spa_strfree(spa->spa_comment);
spa->spa_comment = NULL;
No
spa_config_exit(spa,
SCL_ALL, FTAG);
End


static void
spa_load_spares(spa_t
*spa)
* Load (or re-load) the current list of vdevs describing the active spares for
* this pool. When this is called, we have some form of basic information in
* 'spa_spares.sav_config'. We parse this into vdevs, try to open them, and
* then re-generate a more complete list including status information.
nvlist_t **spares;
uint_t nspares;
int i;
vdev_t *vd, *tvd;
ASSERT(spa_config_held(spa,
SCL_ALL, RW_WRITER) == SCL_ALL);
* First, close and free
any existing spare vdevs.
i = 0
i < spa->spa_spares
.sav_count ?
No
Yes
spa->spa_spares.sav_vdevs
?
vd = spa->spa_spares
.sav_vdevs[i];
Undo the call to
spa_activate() below
Yes
No
kmem_free(spa->spa_spares
.sav_vdevs,
spa->spa_spares.sav_count
* sizeof (void *));
(tvd = spa_lookup_by_guid
(spa, vd->vdev_guid,
B_FALSE)) != NULL &&
tvd->vdev_isspare ?
No
Yes
spa_spare_remove(tvd);
spa->spa_spares.sav_
config == NULL ?
Yes
No
nspares = 0;
VERIFY(nvlist_lookup_
nvlist_array(spa->spa_
spares.sav_config,
ZPOOL_CONFIG_SPARES,
&spares, &nspares) == 0);
vdev_close(vd);
vdev_free(vd);
i++
spa->spa_spares.sav_count = (int)nspares;
spa->spa_spares.sav_vdevs = NULL;
nspares == 0 ?
* Construct the array of vdevs, opening them to get status in the
* process. For each spare, there is potentially two different vdev_t
* structures associated with it: one in the list of spares (used only
* for basic validation purposes) and one in the active vdev
* configuration (if it's spared in). During this phase we open and
* validate each vdev on the spare list. If the vdev also exists in the
* active configuration, then we also mark this vdev as an active spare.
No
Yes
spa->spa_spares.sav_vdevs = kmem_alloc(nspares
* sizeof (void *), KM_SLEEP);
i = 0
* Recompute the stashed list of spares, with status information
* this time.
i < spa->spa_spares
.sav_count ?
No
Yes
VERIFY(nvlist_remove(spa->spa_spares.sav_config, ZPOOL_CONFIG_SPARES, DATA_TYPE_NVLIST_ARRAY) == 0);
spares = kmem_alloc(spa->spa_spares.sav_count * sizeof (void *), KM_SLEEP);
i = 0
VERIFY(spa_config_parse(spa, &vd, spares[i], NULL, 0, VDEV_ALLOC_SPARE) == 0);
ASSERT(vd != NULL);
spa->spa_spares.sav_vdevs[i] = vd;
i < spa->spa_spares
.sav_count ?
(tvd = spa_lookup_by_guid
(spa, vd->vdev_guid,
B_FALSE)) != NULL ?
No
Yes
Yes
VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
ZPOOL_CONFIG_SPARES, spares, spa->spa_spares.sav_count) == 0);
i = 0
spares[i] =
vdev_config_generate(spa,
spa->spa_spares.sav_vdevs
[i], B_TRUE,
VDEV_CONFIG_SPARE);
No
!tvd->vdev_isspare ?
Yes
* We only mark the spare active if we were successfully
* able to load the vdev. Otherwise, importing a pool
* with a bad active spare would result in strange
* behavior, because multiple pool would think the spare
* is actively in use.
*
* There is a vulnerability here to an equally bizarre
* circumstance, where a dead active spare is later
* brought back to life (onlined or otherwise). Given
* the rarity of this scenario, and the extra complexity
* it adds, we ignore the possibility.
i++
No
spa_spare_add(tvd);
i < spa->spa_spares
.sav_count ?
No
Yes
kmem_free(spares,
spa->spa_spares.sav_count
* sizeof (void *));
nvlist_free(spares[i]);
!vdev_is_dead(tvd) ?
No
Yes
End
i++
spa_spare_activate(tvd);
vd->vdev_top = vd;
vd->vdev_aux = &spa->spa_spares;
vdev_open(vd) != 0 ?
Yes
No
vdev_validate_aux(vd)
== 0 ?
Yes
No
spa_spare_add(vd);
i++

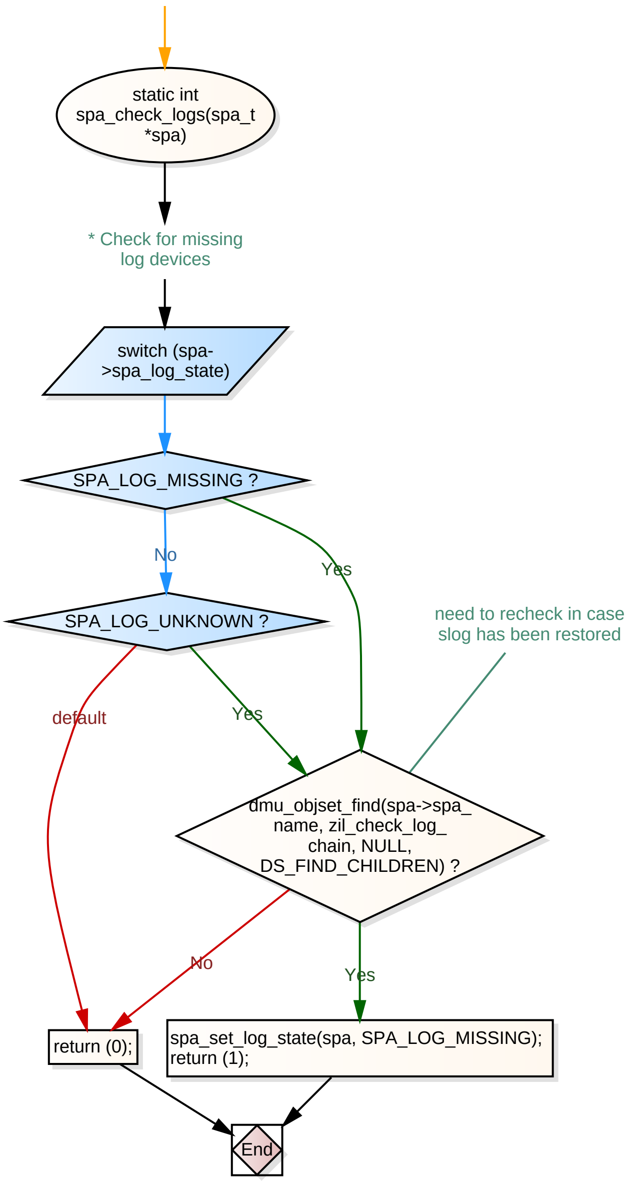

static void spa_load_l2cache(spa_t *spa)
* Load (or re-load) the current list of vdevs describing the active l2cache for
* this pool. When this is called, we have some form of basic information in
* 'spa_l2cache.sav_config'. We parse this into vdevs, try to open them, and
* then re-generate a more complete list including status information.
* Devices which are already active have their details maintained, and are
* not re-opened.
nvlist_t **l2cache;
uint_t nl2cache;
int i, j, oldnvdevs;
uint64_t guid;
vdev_t *vd, **oldvdevs, **newvdevs;
spa_aux_vdev_t *sav = &spa->spa_l2cache;
ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
sav->sav_config != NULL ?
Yes
No
VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
newvdevs = kmem_alloc(nl2cache * sizeof (void *), KM_SLEEP);
nl2cache = 0;
oldvdevs = sav->sav_vdevs;
oldnvdevs = sav->sav_count;
sav->sav_vdevs = NULL;
sav->sav_count = 0;
* Process new
nvlist of vdevs.
i = 0
i < nl2cache ?
Yes
No
VERIFY(nvlist_lookup_uint64(l2cache[i],
ZPOOL_CONFIG_GUID, &guid) == 0);
newvdevs[i] = NULL;
j = 0
j < oldnvdevs ?
Yes
No
vd = oldvdevs[j];
vd != NULL && guid == vd->vdev_guid ?
Yes
No
* Retain previous vdev
for add/remove ops.
j++
newvdevs[i] = vd;
oldvdevs[j] = NULL;
newvdevs[i] == NULL ?
Yes
No
* Create new vdev
VERIFY(spa_config_parse(spa, &vd, l2cache[i],
NULL, 0, VDEV_ALLOC_L2CACHE) == 0);
ASSERT(vd != NULL);
newvdevs[i] = vd;
* Commit this vdev as an l2cache device,
* even if it fails to open.
spa_l2cache_add(vd);
vd->vdev_top = vd;
vd->vdev_aux = sav;
spa_l2cache_activate(vd);
vdev_open(vd) != 0 ?
Yes
No
(void) vdev_validate_aux(vd);
!vdev_is_dead(vd) ?
Yes
No
l2arc_add_vdev(spa, vd);
i++
* Purge vdevs
that were dropped
i = 0
i < oldnvdevs ?
Yes
No
uint64_t pool;
vd = oldvdevs[i];
vd != NULL ?
Yes
No
ASSERT(vd->vdev_isl2cache);
spa_l2cache_exists(vd->vdev_guid, &pool) && pool != 0ULL && l2arc_vdev_present(vd) ?
Yes
No
l2arc_remove_vdev(vd);
vdev_clear_stats(vd);
vdev_free(vd);
i++
oldvdevs ?
Yes
No
kmem_free(oldvdevs, oldnvdevs * sizeof (void *));
sav->sav_config == NULL ?
Yes
No
sav->sav_vdevs = newvdevs;
sav->sav_count = (int)nl2cache;
* Recompute the stashed list of l2cache devices, with status
* information this time.
VERIFY(nvlist_remove(sav->sav_config, ZPOOL_CONFIG_L2CACHE, DATA_TYPE_NVLIST_ARRAY) == 0);
l2cache = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
i = 0
i < sav->sav_count ?
Yes
No
l2cache[i] = vdev_config_generate(spa, sav->sav_vdevs[i], B_TRUE, VDEV_CONFIG_L2CACHE);
i++
VERIFY(nvlist_add_nvlist_array(sav->sav_config, ZPOOL_CONFIG_L2CACHE, l2cache, sav->sav_count) == 0);
out: i = 0
i < sav->sav_count ?
Yes
No
nvlist_free(l2cache[i]);
i++
sav->sav_count ?
Yes
No
kmem_free(l2cache, sav->sav_count * sizeof (void *));
End

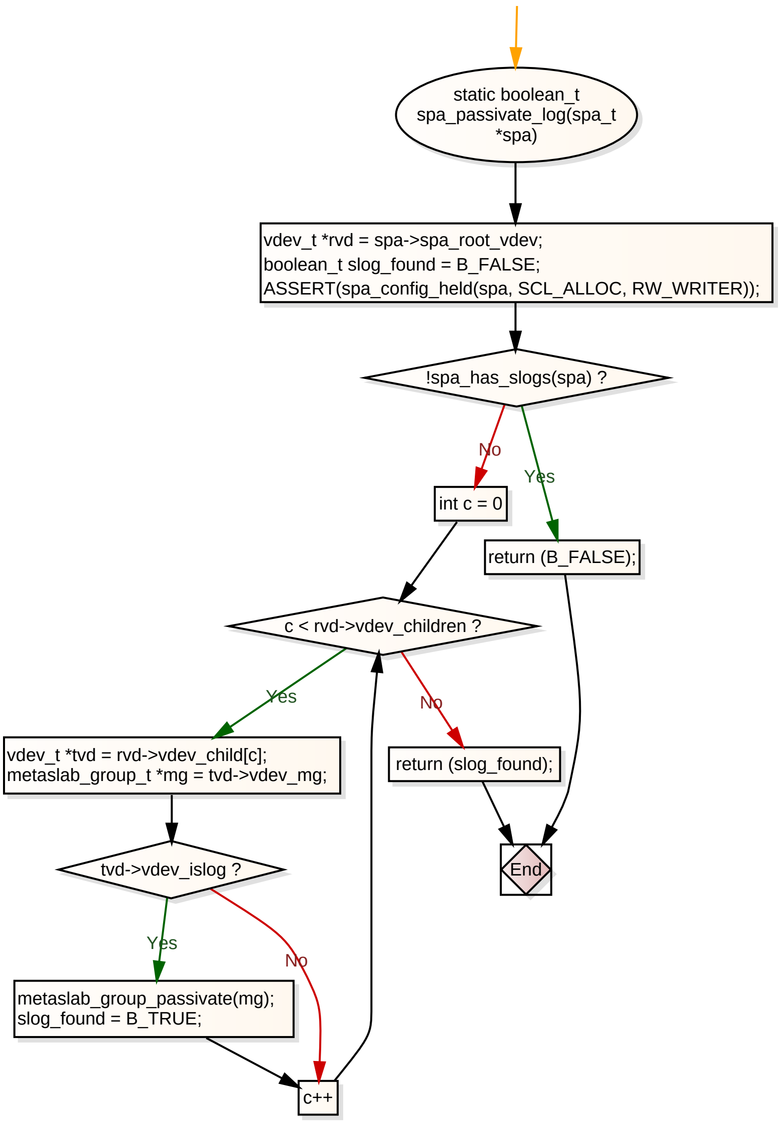

static int
load_nvlist(spa_t *spa,
uint64_t obj,
nvlist_t **value)
dmu_buf_t *db;
char *packed = NULL;
size_t nvsize = 0;
int error;
*value = NULL;
VERIFY(0 == dmu_bonus_hold(spa->spa_meta_objset,
obj, FTAG, &db));
nvsize = *(uint64_t *)db->db_data;
dmu_buf_rele(db, FTAG);
packed = kmem_alloc(nvsize, KM_SLEEP);
error = dmu_read(spa->spa_meta_objset,
obj, 0, nvsize, packed, DMU_READ_PREFETCH);
error == 0 ?
Yes
No
error = nvlist_unpack
(packed, nvsize,
value, 0);
kmem_free(packed, nvsize);
return (error);
End

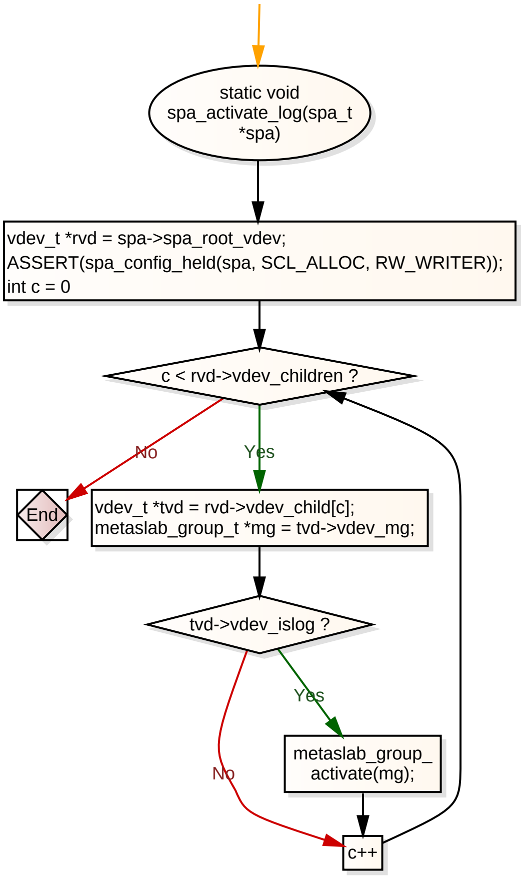

static void spa_check_removed(vdev_t *vd)
* Checks to see if the given vdev could not be opened, in which case we post a
* sysevent to notify the autoreplace code that the device has been removed.
int c = 0
c < vd->vdev_children ?
No
Yes
vd->vdev_ops->vdev_op_leaf && vdev_is_dead(vd) ?
spa_check_removed(vd->vdev_child[c]);
Yes
zfs_post_autoreplace(vd->vdev_spa, vd);
spa_event_notify(vd->vdev_spa, vd, ESC_ZFS_VDEV_CHECK);
c++
No
End

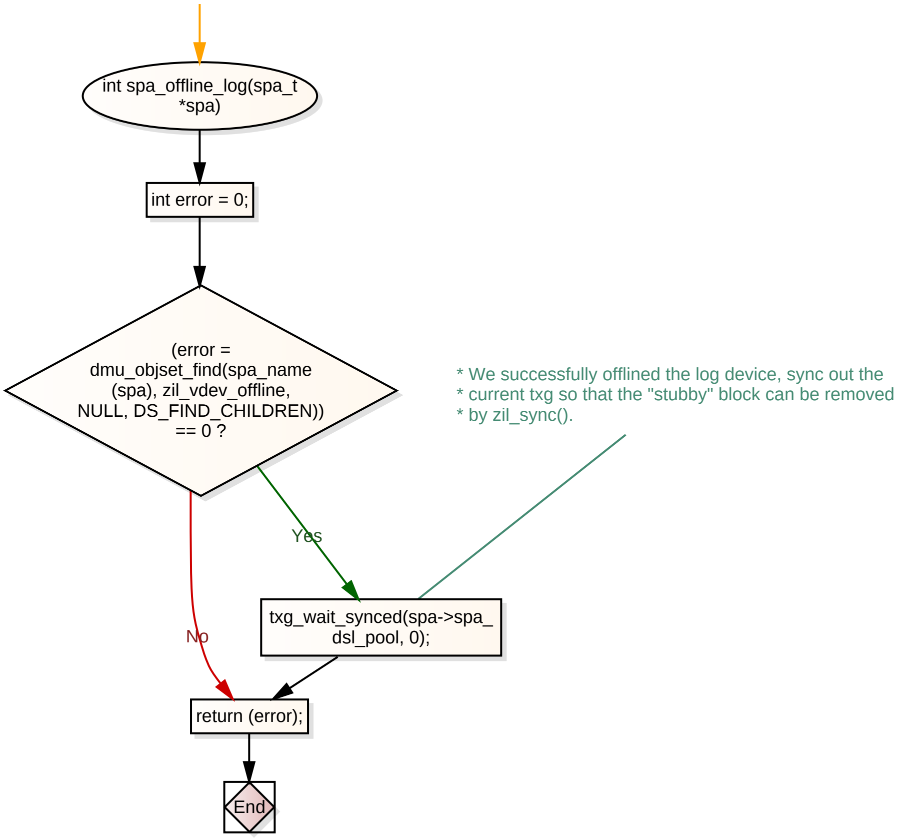


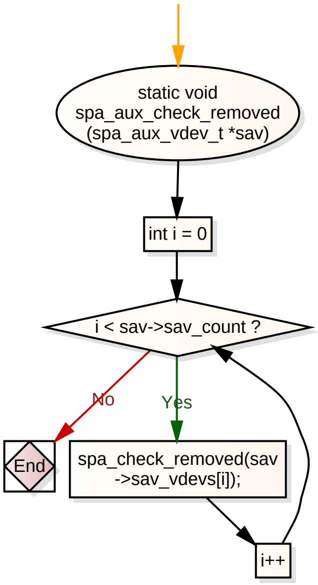
static boolean_t spa_config_valid(spa_t *spa, nvlist_t *config)
* Validate the current config against the MOS config
vdev_t *mrvd, *rvd = spa->spa_root_vdev;
nvlist_t *nv;
VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nv) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
VERIFY(spa_config_parse(spa, &mrvd, nv, NULL, 0, VDEV_ALLOC_LOAD) == 0);
ASSERT3U(rvd->vdev_children, ==, mrvd->vdev_children);
* If we're doing a normal import, then build up any additional
* diagnostic information about missing devices in this config.
* We'll pass this up to the user for further processing.
!(spa->spa_import_flags & ZFS_IMPORT_MISSING_LOG) ?
Yes
No
nvlist_t **child, *nv;
uint64_t idx = 0;
child = kmem_alloc(rvd->vdev_children * sizeof (nvlist_t **), KM_SLEEP);
VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);
int c = 0
c < rvd->vdev_children ?
Yes
No
vdev_t *tvd = rvd->vdev_child[c];
vdev_t *mtvd = mrvd->vdev_child[c];
idx ?
Yes
No
tvd->vdev_ops == &vdev_missing_ops && mtvd->vdev_ops != &vdev_missing_ops && mtvd->vdev_islog ?
VERIFY(nvlist_add_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN, child, idx) == 0);
VERIFY(nvlist_add_nvlist(spa->spa_load_info, ZPOOL_CONFIG_MISSING_DEVICES, nv) == 0);
int i = 0
No
Yes
child[idx++] = vdev_config_generate(spa, mtvd, B_FALSE, 0);
i < idx ?
Yes
No
c++
nvlist_free(child[i]);
* Compare the root vdev tree with the information we have
* from the MOS config (mrvd). Check each top-level vdev
* with the corresponding MOS config top-level (mtvd).
nvlist_free(nv);
kmem_free(child, rvd->vdev_children * sizeof (char **));
i++
int c = 0
c < rvd->vdev_children ?
No
Yes
* Ensure we were able to validate the config.
vdev_free(mrvd);
spa_config_exit(spa, SCL_ALL, FTAG);
* Resolve any "missing" vdevs in the current configuration.
* If we find that the MOS config has more accurate information
* about the top-level vdev then use that vdev instead.
vdev_t *tvd = rvd->vdev_child[c];
vdev_t *mtvd = mrvd->vdev_child[c];
return (rvd->vdev_guid_sum == spa->spa_uberblock.ub_guid_sum);
tvd->vdev_ops == &vdev_missing_ops && mtvd->vdev_ops != &vdev_missing_ops ?
Yes
End
!(spa->spa_import_flags & ZFS_IMPORT_MISSING_LOG) ?
* Device specific actions.
No
No
Yes
mtvd->vdev_islog ?
Yes
No
spa_set_log_state(spa, SPA_LOG_CLEAR);
mtvd->vdev_islog ?
* Load the slog device's state from the MOS config
* since it's possible that the label does not
* contain the most up-to-date information.
* Swap the missing vdev with the data we were
* able to obtain from the MOS config.
* XXX - once we have 'readonly' pool
* support we should be able to handle
* missing data devices by transitioning
* the pool to readonly.
Yes
vdev_remove_child(rvd, tvd);
vdev_remove_child(mrvd, mtvd);
vdev_add_child(rvd, mtvd);
vdev_add_child(mrvd, tvd);
spa_config_exit(spa, SCL_ALL, FTAG);
vdev_load(mtvd);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
vdev_reopen(rvd);
No
vdev_load_log_state(tvd, mtvd);
vdev_reopen(tvd);
c++

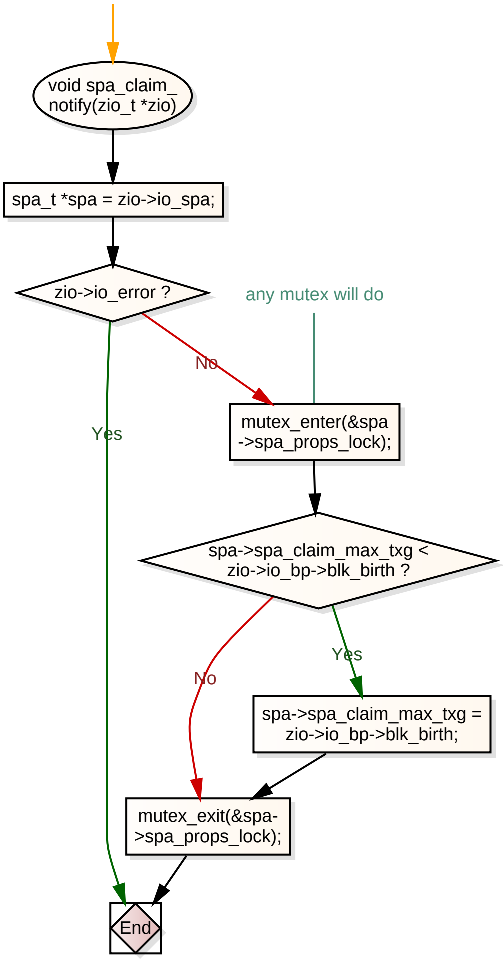

static int spa_check_logs(spa_t *spa)
* Check for missing log devices
switch (spa->spa_log_state)
SPA_LOG_MISSING ?
No
Yes
SPA_LOG_UNKNOWN ?
need to recheck in case slog has been restored
default
Yes
dmu_objset_find(spa->spa_name, zil_check_log_chain, NULL, DS_FIND_CHILDREN) ?
No
Yes
return (0);
spa_set_log_state(spa, SPA_LOG_MISSING);
return (1);
End

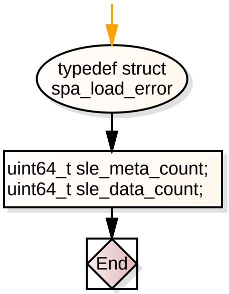

static boolean_t spa_passivate_log(spa_t *spa)
vdev_t *rvd = spa->spa_root_vdev;
boolean_t slog_found = B_FALSE;
ASSERT(spa_config_held(spa, SCL_ALLOC, RW_WRITER));
!spa_has_slogs(spa) ?
No
Yes
int c = 0
return (B_FALSE);
c < rvd->vdev_children ?
Yes
No
vdev_t *tvd = rvd->vdev_child[c];
metaslab_group_t *mg = tvd->vdev_mg;
return (slog_found);
End
tvd->vdev_islog ?
Yes
No
metaslab_group_passivate(mg);
slog_found = B_TRUE;
c++

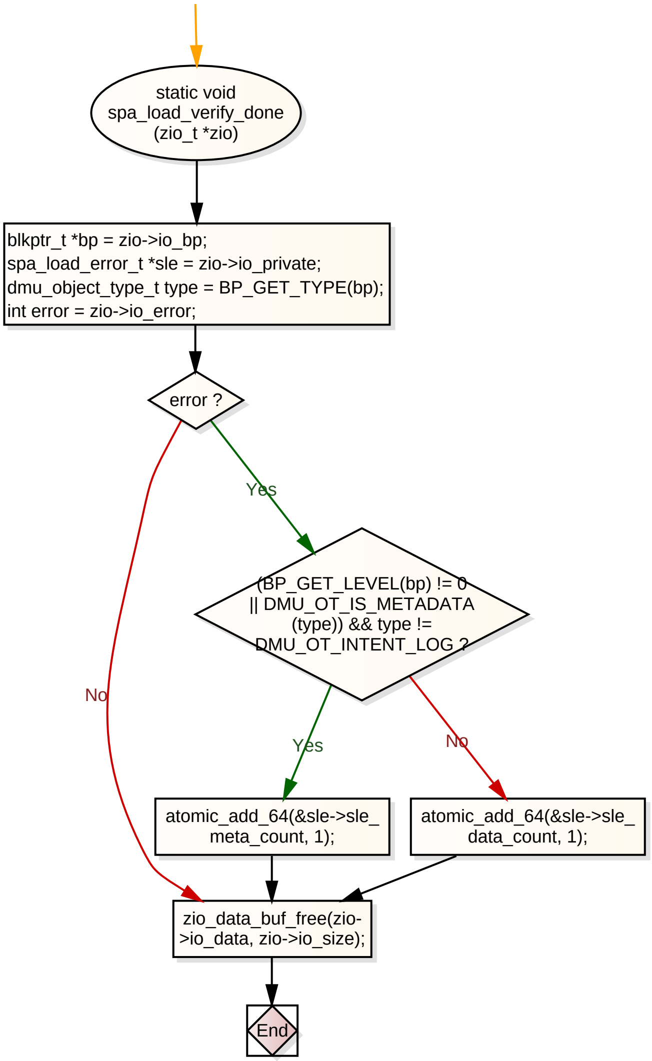

static void spa_activate_log(spa_t *spa)
vdev_t *rvd = spa->spa_root_vdev;
ASSERT(spa_config_held(spa, SCL_ALLOC, RW_WRITER));
int c = 0
c < rvd->vdev_children ?
No
Yes
End
vdev_t *tvd = rvd->vdev_child[c];
metaslab_group_t *mg = tvd->vdev_mg;
tvd->vdev_islog ?
Yes
metaslab_group_activate(mg);
No
c++

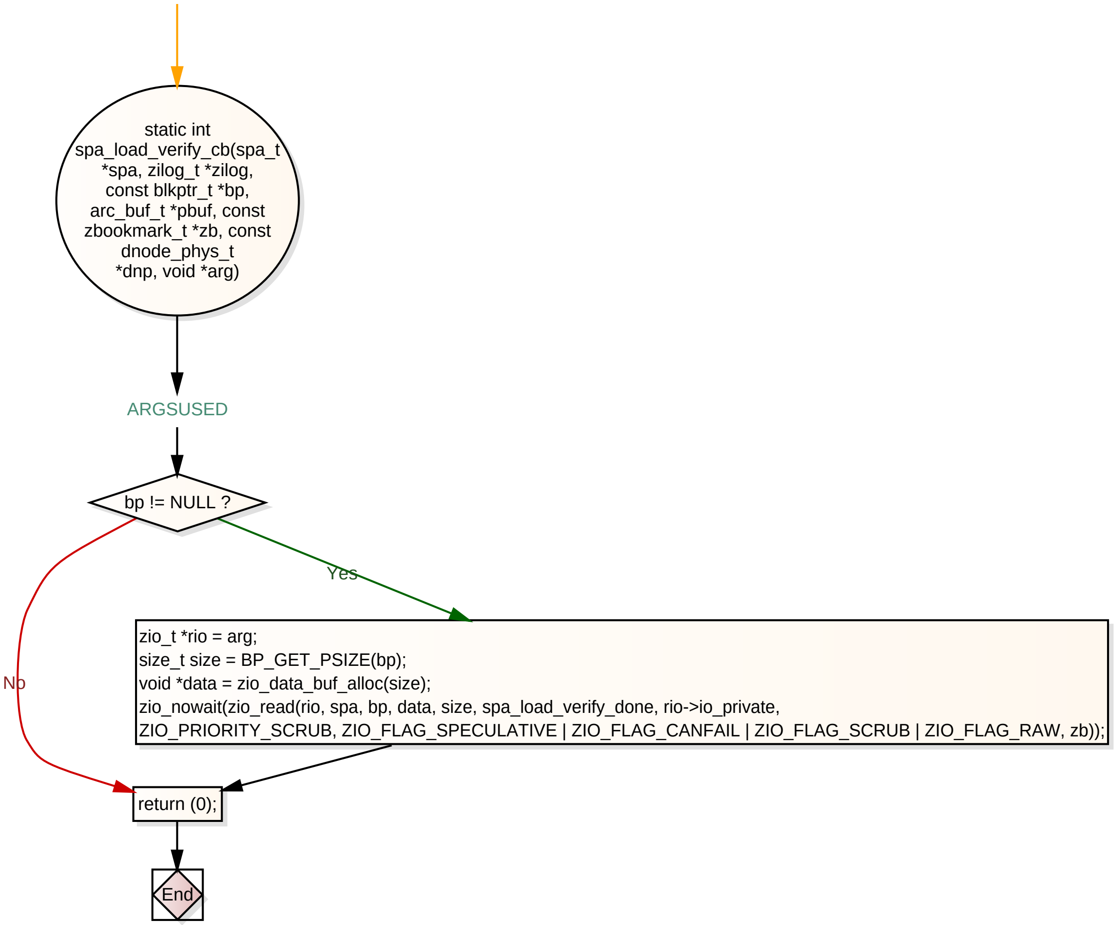

int spa_offline_log(spa_t *spa)
int error = 0;
(error = dmu_objset_find(spa_name (spa), zil_vdev_offline, NULL, DS_FIND_CHILDREN)) == 0 ?
* We successfully offlined the log device, sync out the
* current txg so that the "stubby" block can be removed
* by zil_sync().
Yes
No
txg_wait_synced(spa->spa_dsl_pool, 0);
return (error);
End

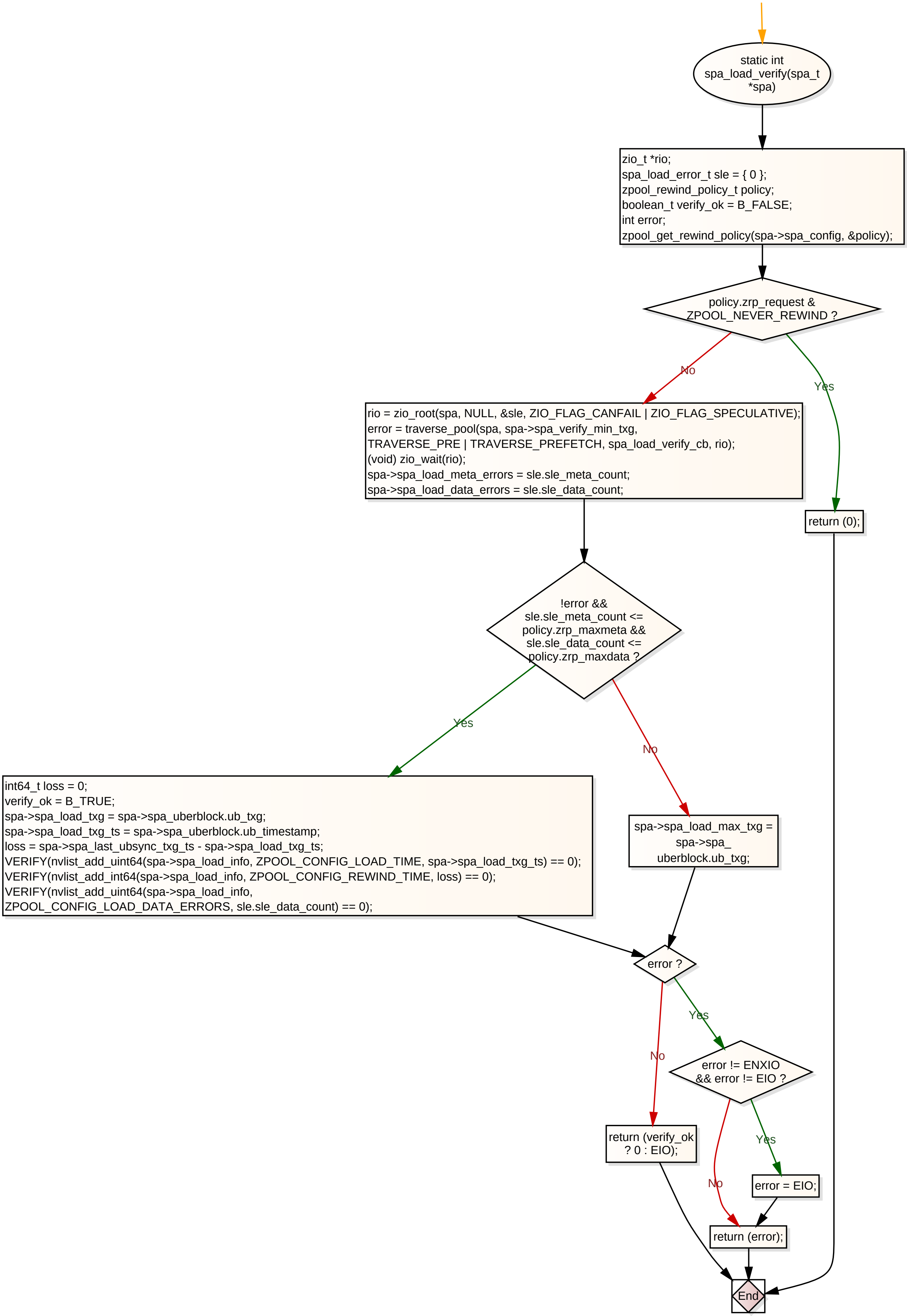

static void
spa_aux_check_removed
(spa_aux_vdev_t *sav)
int i = 0
i < sav->sav_count ?
No
Yes
End
spa_check_removed(sav
->sav_vdevs[i]);
i++

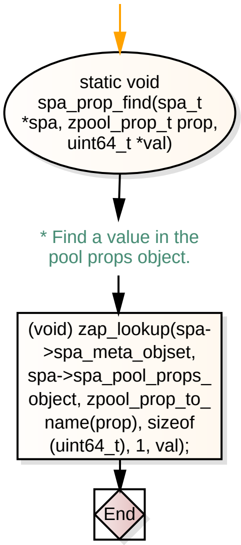

void spa_claim_
notify(zio_t *zio)
spa_t *spa = zio->io_spa;
zio->io_error ?
any mutex will do
No
Yes
mutex_enter(&spa
->spa_props_lock);
spa->spa_claim_max_txg <
zio->io_bp->blk_birth ?
No
Yes
spa->spa_claim_max_txg =
zio->io_bp->blk_birth;
mutex_exit(&spa-
>spa_props_lock);
End


typedef struct spa_load_error
uint64_t sle_meta_count;
uint64_t sle_data_count;
End


static void spa_load_verify_done (zio_t *zio)
blkptr_t *bp = zio->io_bp;
spa_load_error_t *sle = zio->io_private;
dmu_object_type_t type = BP_GET_TYPE(bp);
int error = zio->io_error;
error ?
Yes
No
(BP_GET_LEVEL(bp) != 0 || DMU_OT_IS_METADATA (type)) && type != DMU_OT_INTENT_LOG ?
Yes
No
atomic_add_64(&sle->sle_ meta_count, 1);
atomic_add_64(&sle->sle_ data_count, 1);
zio_data_buf_free(zio- >io_data, zio->io_size);
End


static int spa_load_verify_cb(spa_t *spa, zilog_t *zilog, const blkptr_t *bp, arc_buf_t *pbuf, const zbookmark_t *zb, const dnode_phys_t *dnp, void *arg)
ARGSUSED
bp != NULL ?
Yes
No
zio_t *rio = arg;
size_t size = BP_GET_PSIZE(bp);
void *data = zio_data_buf_alloc(size);
zio_nowait(zio_read(rio, spa, bp, data, size, spa_load_verify_done, rio->io_private,
ZIO_PRIORITY_SCRUB, ZIO_FLAG_SPECULATIVE | ZIO_FLAG_CANFAIL | ZIO_FLAG_SCRUB | ZIO_FLAG_RAW, zb));
return (0);
End

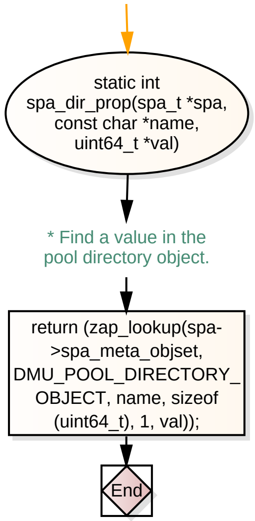

static int
spa_load_verify(spa_t
*spa)
zio_t *rio;
spa_load_error_t sle = { 0 };
zpool_rewind_policy_t policy;
boolean_t verify_ok = B_FALSE;
int error;
zpool_get_rewind_policy(spa->spa_config, &policy);
policy.zrp_request &
ZPOOL_NEVER_REWIND ?
No
Yes
rio = zio_root(spa, NULL, &sle, ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE);
error = traverse_pool(spa, spa->spa_verify_min_txg,
TRAVERSE_PRE | TRAVERSE_PREFETCH, spa_load_verify_cb, rio);
(void) zio_wait(rio);
spa->spa_load_meta_errors = sle.sle_meta_count;
spa->spa_load_data_errors = sle.sle_data_count;
return (0);
!error &&
sle.sle_meta_count <=
policy.zrp_maxmeta &&
sle.sle_data_count <=
policy.zrp_maxdata ?
Yes
No
int64_t loss = 0;
verify_ok = B_TRUE;
spa->spa_load_txg = spa->spa_uberblock.ub_txg;
spa->spa_load_txg_ts = spa->spa_uberblock.ub_timestamp;
loss = spa->spa_last_ubsync_txg_ts - spa->spa_load_txg_ts;
VERIFY(nvlist_add_uint64(spa->spa_load_info, ZPOOL_CONFIG_LOAD_TIME, spa->spa_load_txg_ts) == 0);
VERIFY(nvlist_add_int64(spa->spa_load_info, ZPOOL_CONFIG_REWIND_TIME, loss) == 0);
VERIFY(nvlist_add_uint64(spa->spa_load_info,
ZPOOL_CONFIG_LOAD_DATA_ERRORS, sle.sle_data_count) == 0);
spa->spa_load_max_txg =
spa->spa_
uberblock.ub_txg;
error ?
Yes
No
error != ENXIO
&& error != EIO ?
return (verify_ok
? 0 : EIO);
Yes
No
error = EIO;
return (error);
End

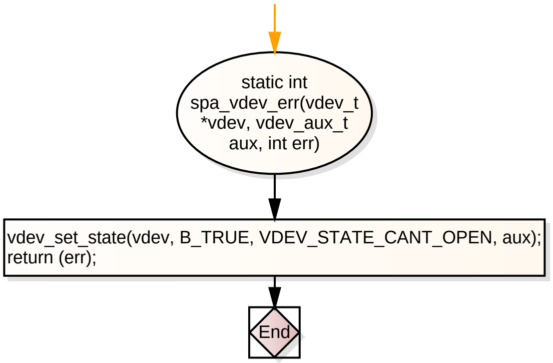


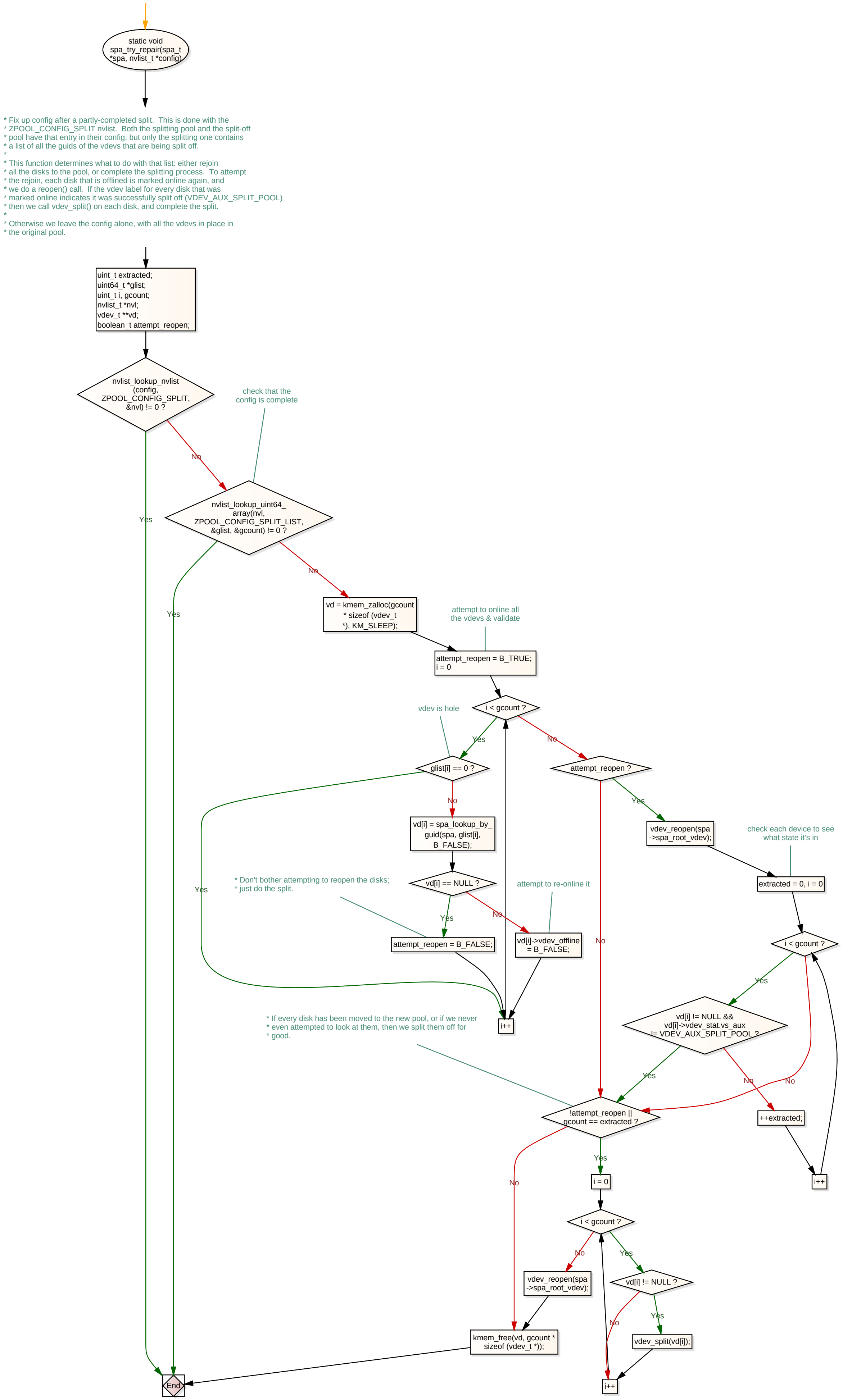
```
static void spa_prop_find(spa_t *spa, zpool_prop_t prop, uint64_t *val)
```

* Find a value in the pool props object.

```
(void) zap_lookup(spa->spa_meta_objset, spa->spa_pool_props_object, zpool_prop_to_name(prop), sizeof (uint64_t), 1, val);
```


static int spa_dir_prop(spa_t *spa, const char *name, uint64_t *val)
* Find a value in the pool directory object.
return (zap_lookup(spa->spa_meta_objset, DMU_POOL_DIRECTORY_OBJECT, name, sizeof (uint64_t), 1, val));
End


static int
spa_vdev_err(vdev_t
*vdev, vdev_aux_t
aux, int err)
vdev_set_state(vdev, B_TRUE, VDEV_STATE_CANT_OPEN, aux);
return (err);
End

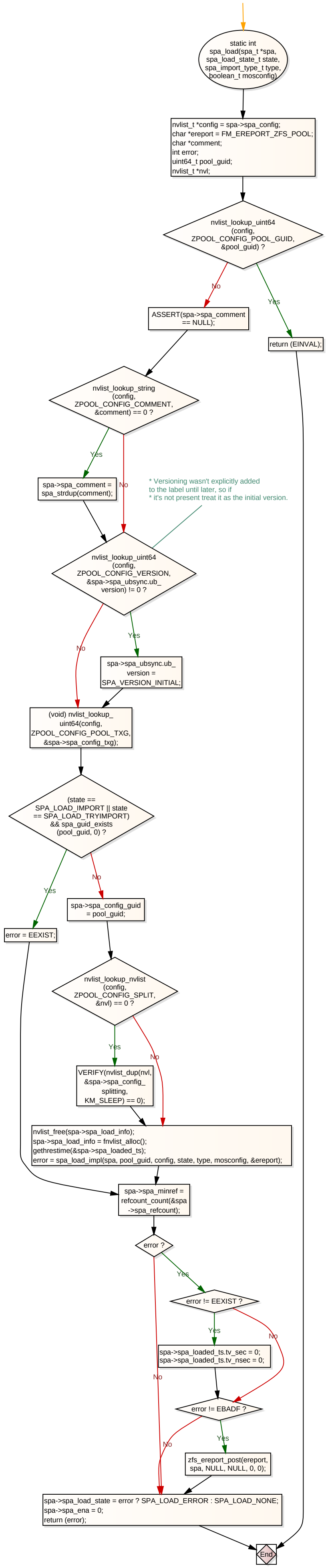


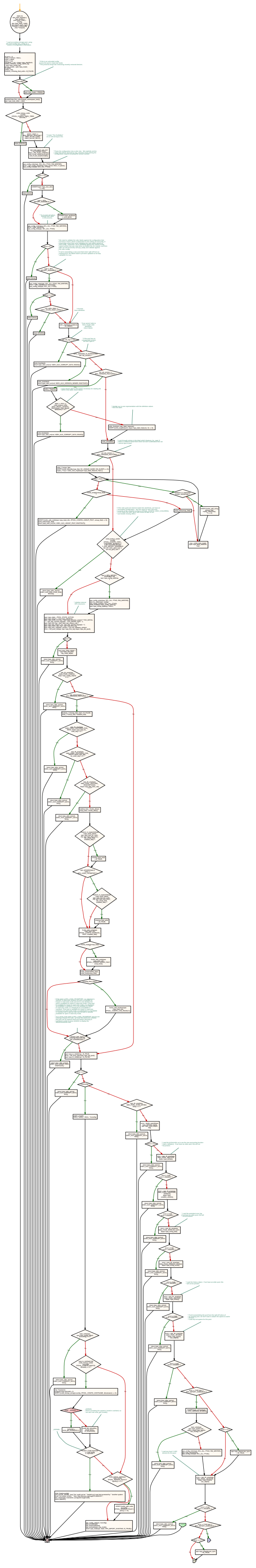
* Fix up config after a partly-completed split. This is done with the
* ZPOOL_CONFIG_SPLIT nvlist. Both the splitting pool and the split-off
* pool have that entry in their config, but only the splitting one contains
* a list of all the guids of the vdevs that are being split off.
*
* This function determines what to do with that list: either rejoin
* all the disks to the pool, or complete the splitting process. To attempt
* the rejoin, each disk that is offlined is marked online again, and
* we do a reopen() call. If the vdev label for every disk that was
* marked online indicates it was successfully split off (VDEV_AUX_SPLIT_POOL)
* then we call vdev_split() on each disk, and complete the split.
*
* Otherwise we leave the config alone, with all the vdevs in place in
* the original pool.

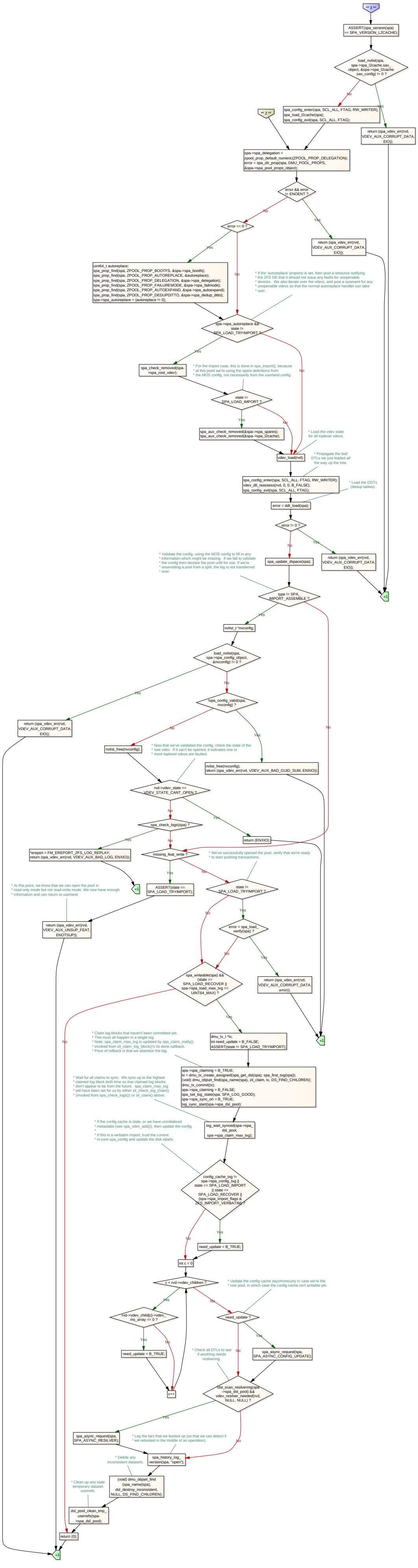
* Don't bother attempting to reopen the disks;
* just do the split.

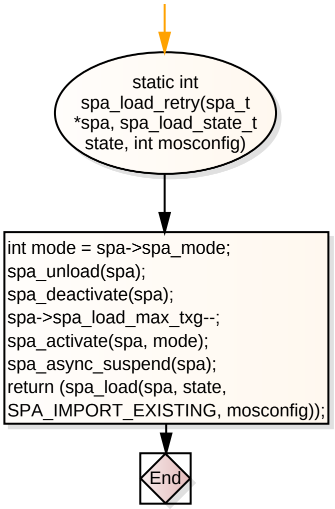
* If every disk has been moved to the new pool, or if we never
* even attempted to look at them, then we split them off for
* good.

static void spa_try_repair(spa_t *spa, nvlist_t *config)
uint_t extracted;
uint64_t *glist;
uint_t i, gcount;
nvlist_t *nvl;
vdev_t **vd;
boolean_t attempt_reopen;
nvlist_lookup_nvlist (config, ZPOOL_CONFIG_SPLIT, &nvl) != 0 ?
check that the config is complete
Yes
No
nvlist_lookup_uint64_ array(nvl, ZPOOL_CONFIG_SPLIT_LIST, &glist, &gcount) != 0 ?
Yes
No
vd = kmem_zalloc(gcount * sizeof (vdev_t *), KM_SLEEP);
attempt to online all the vdevs & validate
attempt_reopen = B_TRUE;
i = 0
i < gcount ?
vdev is hole
Yes
No
glist[i] == 0 ?
Yes
No
vd[i] = spa_lookup_by_ guid(spa, glist[i], B_FALSE);
vd[i] == NULL ?
Yes
No
attempt_reopen = B_FALSE;
attempt to re-online it
vd[i]->vdev_offline = B_FALSE;
i++
attempt_reopen ?
Yes
No
vdev_reopen(spa ->spa_root_vdev);
check each device to see what state it's in
extracted = 0, i = 0
i < gcount ?
Yes
No
vd[i] != NULL && vd[i]->vdev_stat.vs_aux != VDEV_AUX_SPLIT_POOL ?
Yes
No
++extracted;
i++
!attempt_reopen || gcount == extracted ?
Yes
No
i = 0
i < gcount ?
No
Yes
vdev_reopen(spa ->spa_root_vdev);
vd[i] != NULL ?
Yes
No
vdev_split(vd[i]);
i++
kmem_free(vd, gcount * sizeof (vdev_t *));
End


static int
spa_load(spa_t *spa,
spa_load_state_t state,
spa_import_type_t type,
boolean_t mosconfig)
nvlist_t *config = spa->spa_config;
char *ereport = FM_EREPORT_ZFS_POOL;
char *comment;
int error;
uint64_t pool_guid;
nvlist_t *nvl;
nvlist_lookup_uint64
(config,
ZPOOL_CONFIG_POOL_GUID,
&pool_guid) ?
No
Yes
ASSERT(spa->spa_comment
== NULL);
return (EINVAL);
nvlist_lookup_string
(config,
ZPOOL_CONFIG_COMMENT,
&comment) == 0 ?
Yes
No
spa->spa_comment =
spa_strdup(comment);
* Versioning wasn't explicitly added
to the label until later, so if
* it's not present treat it as the initial version.
nvlist_lookup_uint64
(config,
ZPOOL_CONFIG_VERSION,
&spa->spa_ubsync.ub_
version) != 0 ?
No
Yes
spa->spa_ubsync.ub_
version =
SPA_VERSION_INITIAL;
(void) nvlist_lookup_
uint64(config,
ZPOOL_CONFIG_POOL_TXG,
&spa->spa_config_txg);
(state ==
SPA_LOAD_IMPORT || state
== SPA_LOAD_TRYIMPORT)
&& spa_guid_exists
(pool_guid, 0) ?
Yes
No
spa->spa_config_guid
= pool_guid;
error = EEXIST;
nvlist_lookup_nvlist
(config,
ZPOOL_CONFIG_SPLIT,
&nvl) == 0 ?
Yes
No
VERIFY(nvlist_dup(nvl,
&spa->spa_config_
splitting,
KM_SLEEP) == 0);
nvlist_free(spa->spa_load_info);
spa->spa_load_info = fnvlist_alloc();
gethrestime(&spa->spa_loaded_ts);
error = spa_load_impl(spa, pool_guid, config, state, type, mosconfig, &ereport);
spa->spa_minref =
refcount_count(&spa
->spa_refcount);
error ?
Yes
error != EEXIST ?
Yes
No
spa->spa_loaded_ts.tv_sec = 0;
spa->spa_loaded_ts.tv_nsec = 0;
No
error != EBADF ?
Yes
No
zfs_ereport_post(ereport,
spa, NULL, NULL, 0, 0);
spa->spa_load_state = error ? SPA_LOAD_ERROR : SPA_LOAD_NONE;
spa->spa_ena = 0;
return (error);
End

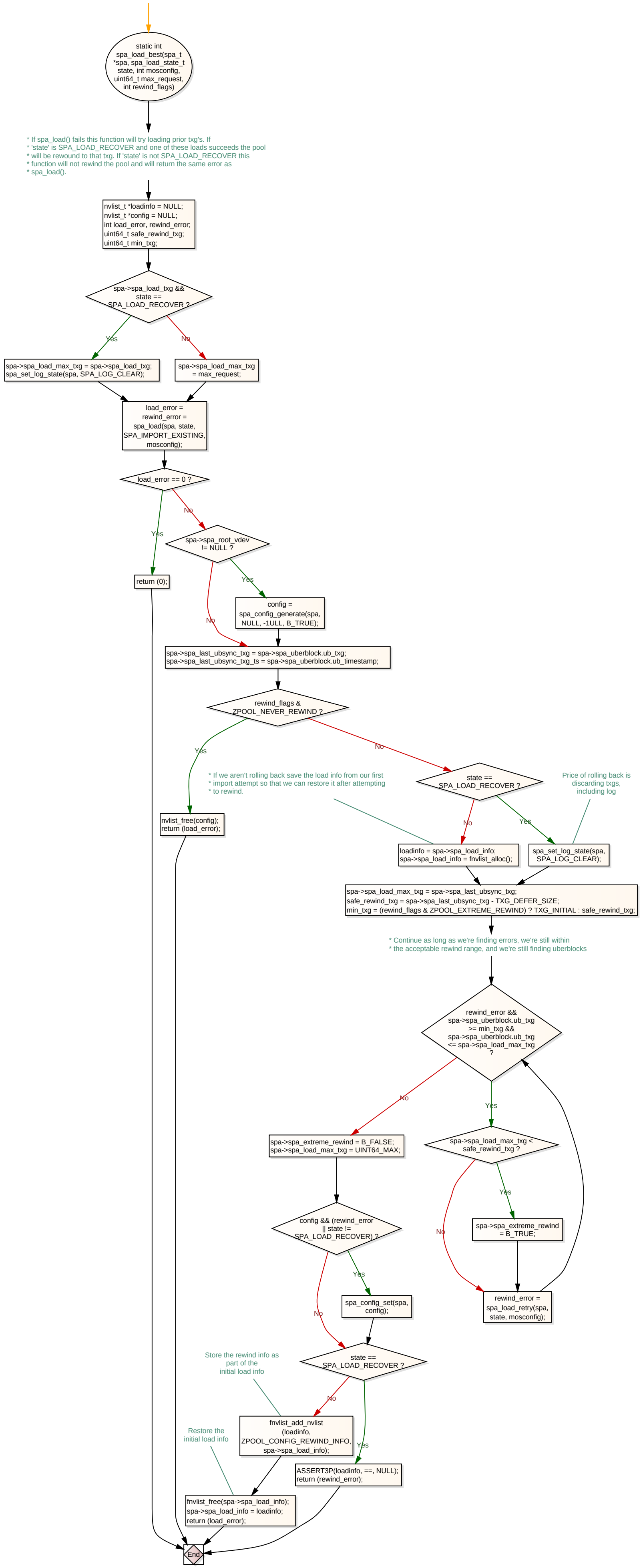

static int spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config, spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig, char **ereport)
* Load an existing storage pool, using the pool's builtin spa_config as a * source of configuration information.
* If this is an untrusted config, access the pool in read-only mode. * This prevents things like resilvering recently removed devices.
mosconfig ?
Yes
No
spa->spa_mode = FREAD;
ASSERT(MUTEX_HELD(&spa_namespace_lock)); spa->spa_load_state = state;
nvlist_lookup_nvlist (config, ZPOOL_CONFIG_VDEV_TREE, &nvroot) ?
return (EINVAL);
parse = (type == SPA_IMPORT_EXISTING ? VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);
* Create "The Godfather" zio to hold all async IOs.
spa->spa_async_zio_root = zio_root(spa, NULL, NULL, ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);
* Parse the configuration into a vdev tree. We explicitly set the * value that will be returned by spa_version() since parsing the * configuration requires knowing the version number.
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER); error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse); spa_config_exit(spa, SCL_ALL, FTAG);
error != 0 ?
return (error);
ASSERT(spa->spa_root_vdev == rvd);
type != SPA_IMPORT_ASSEMBLE ?
ASSERT(spa_guid(spa) == pool_guid);
* Try to open all vdevs, loading each label in the process.
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER); error = vdev_open(rvd); spa_config_exit(spa, SCL_ALL, FTAG);

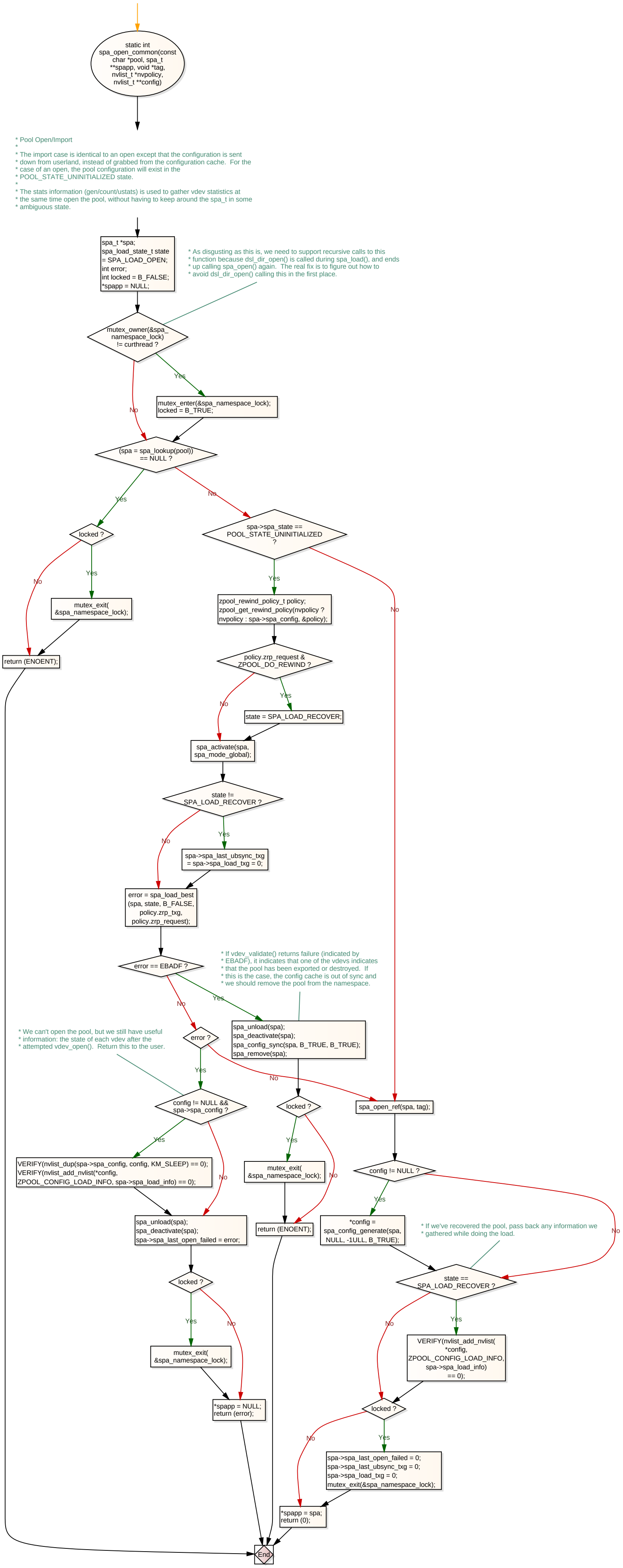

** 3 **
ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
load_nvlist(spa, spa->spa_l2cache.sav_object, &spa->spa_l2cache .sav_config) != 0 ?
No
Yes
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
spa_load_l2cache(spa);
spa_config_exit(spa, SCL_ALL, FTAG);
return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
** 2 **
spa->spa_delegation =
zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
error = spa_dir_prop(spa, DMU_POOL_PROPS,
&spa->spa_pool_props_object);
error && error != ENOENT ?
No
Yes
error == 0 ?
return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
Yes
No
uint64_t autoreplace;
spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO, &spa->spa_dedup_ditto);
spa->spa_autoreplace = (autoreplace != 0);
* If the 'autoreplace' property is set, then post a resource notifying
* the ZFS DE that it should not issue any faults for unopenable
* devices. We also iterate over the vdevs, and post a sysevent for any
* unopenable vdevs so that the normal autoreplace handler can take
* over.
spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT ?
Yes
No
spa_check_removed(spa->spa_root_vdev);
* For the import case, this is done in spa_import(), because
* at this point we're using the spare definitions from
* the MOS config, not necessarily from the userland config.
state != SPA_LOAD_IMPORT ?
Yes
No
spa_aux_check_removed(&spa->spa_spares);
spa_aux_check_removed(&spa->spa_l2cache);
* Load the vdev state
for all toplevel vdevs.
vdev_load(rvd);
* Propagate the leaf
DTLs we just loaded all
the way up the tree.
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
spa_config_exit(spa, SCL_ALL, FTAG);
* Load the DDTs
(dedup tables).
error = ddt_load(spa);
error != 0 ?
No
Yes
spa_update_dspace(spa);
return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
«1
* Validate the config, using the MOS config to fill in any
* information which might be missing. If we fail to validate
* the config then declare the pool unfit for use. If we're
* assembling a pool from a split, the log is not transferred
* over.
type != SPA_IMPORT_ASSEMBLE ?
Yes
No
nvlist_t *nvconfig;
load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0 ?
Yes
No
return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
!spa_config_valid(spa, nvconfig) ?
No
Yes
nvlist_free(nvconfig);
* Now that we've validated the config, check the state of the
* root vdev. If it can't be opened, it indicates one or
* more toplevel vdevs are faulted.
nvlist_free(nvconfig);
return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));
rvd->vdev_state <= VDEV_STATE_CANT_OPEN ?
No
Yes
spa_check_logs(spa) ?
return (ENXIO);
Yes
No
*ereport = FM_EREPORT_ZFS_LOG_REPLAY;
return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
missing_feat_write ?
* We've successfully opened the pool, verify that we're ready
* to start pushing transactions.
Yes
No
«1
ASSERT(state == SPA_LOAD_TRYIMPORT);
state != SPA_LOAD_TRYIMPORT ?
* At this point, we know that we can open the pool in
* read-only mode but not read-write mode. We now have enough
* information and can return to userland.
return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
No
Yes
error = spa_load_verify(spa) ?
No
Yes
spa_writeable(spa) && (state == SPA_LOAD_RECOVER || spa->spa_load_max_txg == UINT64_MAX) ?
return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, error));
Yes
No
* Claim log blocks that haven't been committed yet.
* This must all happen in a single txg.
* Note: spa_claim_max_txg is updated by spa_claim_notify(),
* invoked from zil_claim_log_block()'s i/o done callback.
* Price of rollback is that we abandon the log.
dmu_tx_t *tx;
int need_update = B_FALSE;
ASSERT(state != SPA_LOAD_TRYIMPORT);
«1
* Wait for all claims to sync. We sync up to the highest
* claimed log block birth time so that claimed log blocks
* don't appear to be from the future. spa_claim_max_txg
* will have been set for us by either zil_check_log_chain()
* (invoked from spa_check_logs()) or zil_claim() above.
spa->spa_claiming = B_TRUE;
tx = dmu_tx_create_assigned(spa_get_dsl(spa), spa_first_txg(spa));
(void) dmu_objset_find(spa_name(spa), zil_claim, tx, DS_FIND_CHILDREN);
dmu_tx_commit(tx);
spa->spa_claiming = B_FALSE;
spa_set_log_state(spa, SPA_LOG_GOOD);
spa->spa_sync_on = B_TRUE;
txg_sync_start(spa->spa_dsl_pool);
* If the config cache is stale, or we have uninitialized
* metaslabs (see spa_vdev_add()), then update the config.
*
* If this is a verbatim import, trust the current
* in-core spa_config and update the disk labels.
txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);
config_cache_txg != spa->spa_config_txg || state == SPA_LOAD_IMPORT || state == SPA_LOAD_RECOVER || (spa->spa_import_flags & ZFS_IMPORT_VERBATIM) ?
No
Yes
need_update = B_TRUE;
int c = 0
c < rvd->vdev_children ?
* Update the config cache asychronously in case we're the
* root pool, in which case the config cache isn't writable yet.
Yes
No
rvd->vdev_child[c]->vdev_ms_array == 0 ?
need_update ?
Yes
No
Yes
No
need_update = B_TRUE;
* Check all DTLs to see
if anything needs
resilvering.
spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
c++
!dsl_scan_resilvering(spa->spa_dsl_pool) && vdev_resilver_needed(rvd, NULL, NULL) ?
Yes
No
spa_async_request(spa, SPA_ASYNC_RESILVER);
* Log the fact that we booted up (so that we can detect if
* we rebooted in the middle of an operation).
spa_history_log_version(spa, "open");
* Delete any
inconsistent datasets.
(void) dmu_objset_find (spa_name(spa), dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);
* Clean up any stale
temporary dataset
userrefs.
dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
return (0);
«1



```
static int
spa_load_retry(spa_t
*spa, spa_load_state_t
state, int mosconfig)
```

```
int mode = spa->spa_mode;
spa_unload(spa);
spa_deactivate(spa);
spa->spa_load_max_txg--;
spa_activate(spa, mode);
spa_async_suspend(spa);
return (spa_load(spa, state,
SPA_IMPORT_EXISTING, mosconfig));
```


static int spa_load_best(spa_t *spa, spa_load_state_t state, int mosconfig, uint64_t max_request, int rewind_flags)
* If spa_load() fails this function will try loading prior txg's. If
* 'state' is SPA_LOAD_RECOVER and one of these loads succeeds the pool
* will be rewound to that txg. If 'state' is not SPA_LOAD_RECOVER this
* function will not rewind the pool and will return the same error as
* spa_load().
nvlist_t *loadinfo = NULL;
nvlist_t *config = NULL;
int load_error, rewind_error;
uint64_t safe_rewind_txg;
uint64_t min_txg;
spa->spa_load_txg && state == SPA_LOAD_RECOVER ?
Yes
No
spa->spa_load_max_txg = spa->spa_load_txg;
spa_set_log_state(spa, SPA_LOG_CLEAR);
spa->spa_load_max_txg = max_request;
load_error = rewind_error = spa_load(spa, state, SPA_IMPORT_EXISTING, mosconfig);
load_error == 0 ?
No
Yes
spa->spa_root_vdev != NULL ?
return (0);
Yes
config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
No
spa->spa_last_ubsync_txg = spa->spa_uberblock.ub_txg;
spa->spa_last_ubsync_txg_ts = spa->spa_uberblock.ub_timestamp;
rewind_flags & ZPOOL_NEVER_REWIND ?
Yes
No
* If we aren't rolling back save the load info from our first
* import attempt so that we can restore it after attempting
* to rewind.
state == SPA_LOAD_RECOVER ?
Price of rolling back is discarding txgs, including log
nvlist_free(config);
return (load_error);
No
Yes
loadinfo = spa->spa_load_info;
spa->spa_load_info = fnvlist_alloc();
spa_set_log_state(spa, SPA_LOG_CLEAR);
spa->spa_load_max_txg = spa->spa_last_ubsync_txg;
safe_rewind_txg = spa->spa_last_ubsync_txg - TXG_DEFER_SIZE;
min_txg = (rewind_flags & ZPOOL_EXTREME_REWIND) ? TXG_INITIAL : safe_rewind_txg;
* Continue as long as we're finding errors, we're still within
* the acceptable rewind range, and we're still finding uberblocks
rewind_error && spa->spa_uberblock.ub_txg >= min_txg && spa->spa_uberblock.ub_txg <= spa->spa_load_max_txg ?
No
Yes
spa->spa_extreme_rewind = B_FALSE;
spa->spa_load_max_txg = UINT64_MAX;
spa->spa_load_max_txg < safe_rewind_txg ?
Yes
No
spa->spa_extreme_rewind = B_TRUE;
config && (rewind_error || state != SPA_LOAD_RECOVER) ?
Yes
No
spa_config_set(spa, config);
rewind_error = spa_load_retry(spa, state, mosconfig);
Store the rewind info as part of the initial load info
state == SPA_LOAD_RECOVER ?
No
fnvlist_add_nvlist (loadinfo, ZPOOL_CONFIG_REWIND_INFO, spa->spa_load_info);
Restore the initial load info
Yes
ASSERT3P(loadinfo, ==, NULL);
return (rewind_error);
fnvlist_free(spa->spa_load_info);
spa->spa_load_info = loadinfo;
return (load_error);
End


static int spa_open_common(const char *pool, spa_t **spapp, void *tag, nvlist_t *nvpolicy, nvlist_t **config)
spa_t *spa;
spa_load_state_t state = SPA_LOAD_OPEN;
int error;
int locked = B_FALSE;
*spapp = NULL;
mutex_owner(&spa_namespace_lock) != curthread ?
Yes
mutex_enter(&spa_namespace_lock);
locked = B_TRUE;
No
(spa = spa_lookup(pool)) == NULL ?
Yes
locked ?
Yes
mutex_exit(&spa_namespace_lock);
No
return (ENOENT);
No
spa->spa_state == POOL_STATE_UNINITIALIZED ?
Yes
zpool_rewind_policy_t policy;
zpool_get_rewind_policy(nvpolicy ? nvpolicy : spa->spa_config, &policy);
policy.zrp_request & ZPOOL_DO_REWIND ?
Yes
state = SPA_LOAD_RECOVER;
No
spa_activate(spa, spa_mode_global);
state != SPA_LOAD_RECOVER ?
Yes
spa->spa_last_ubsync_txg = spa->spa_load_txg = 0;
No
error = spa_load_best (spa, state, B_FALSE, policy.zrp_txg, policy.zrp_request);
error == EBADF ?
Yes
spa_unload(spa);
spa_deactivate(spa);
spa_config_sync(spa, B_TRUE, B_TRUE);
spa_remove(spa);
locked ?
Yes
mutex_exit(&spa_namespace_lock);
No
return (ENOENT);
No
error ?
Yes
config != NULL && spa->spa_config ?
Yes
VERIFY(nvlist_dup(spa->spa_config, config, KM_SLEEP) == 0);
VERIFY(nvlist_add_nvlist(*config, ZPOOL_CONFIG_LOAD_INFO, spa->spa_load_info) == 0);
No
spa_unload(spa);
spa_deactivate(spa);
spa->spa_last_open_failed = error;
locked ?
Yes
mutex_exit(&spa_namespace_lock);
No
*spapp = NULL;
return (error);
No
No
spa_open_ref(spa, tag);
config != NULL ?
Yes
*config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
No
state == SPA_LOAD_RECOVER ?
Yes
VERIFY(nvlist_add_nvlist(*config, ZPOOL_CONFIG_LOAD_INFO, spa->spa_load_info) == 0);
No
locked ?
Yes
spa->spa_last_open_failed = 0;
spa->spa_last_ubsync_txg = 0;
spa->spa_load_txg = 0;
mutex_exit(&spa_namespace_lock);
No
*spapp = spa;
return (0);
End

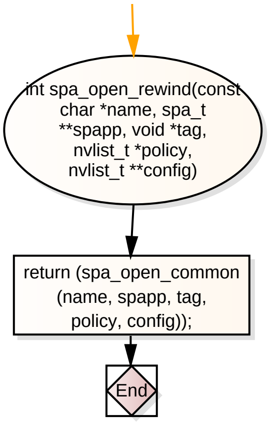
* Pool Open/Import
*
* The import case is identical to an open except that the configuration is sent
* down from userland, instead of grabbed from the configuration cache. For the
* case of an open, the pool configuration will exist in the
* POOL_STATE_UNINITIALIZED state.
*
* The stats information (gen/count/ustats) is used to gather vdev statistics at
* the same time open the pool, without having to keep around the spa_t in some
* ambiguous state.

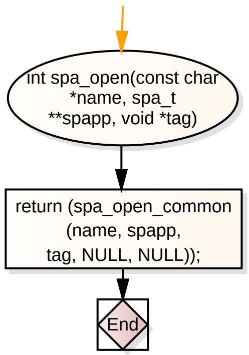
* As disgusting as this is, we need to support recursive calls to this
* function because dsl_dir_open() is called during spa_load(), and ends
* up calling spa_open() again. The real fix is to figure out how to
* avoid dsl_dir_open() calling this in the first place.

* If vdev_validate() returns failure (indicated by
* EBADF), it indicates that one of the vdevs indicates
* that the pool has been exported or destroyed. If
* this is the case, the config cache is out of sync and
* we should remove the pool from the namespace.

* We can't open the pool, but we still have useful
* information: the state of each vdev after the
* attempted vdev_open(). Return this to the user.

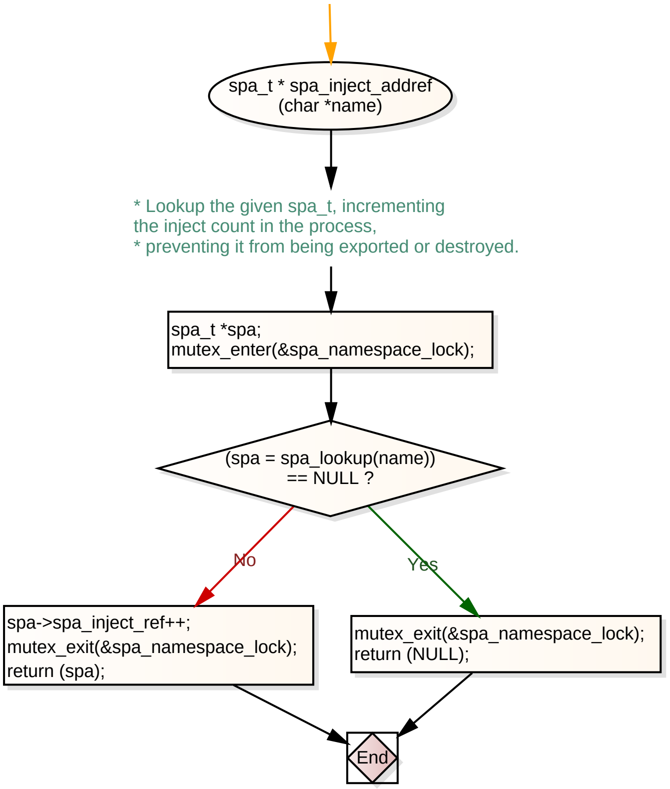
* If we've recovered the pool, pass back any information we
* gathered while doing the load.

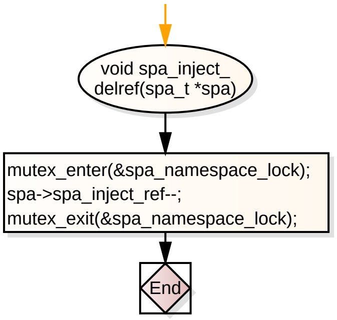

int spa_open_rewind(const char *name, spa_t **spapp, void *tag, nvlist_t *policy, nvlist_t **config)
return (spa_open_common (name, spapp, tag, policy, config));
End

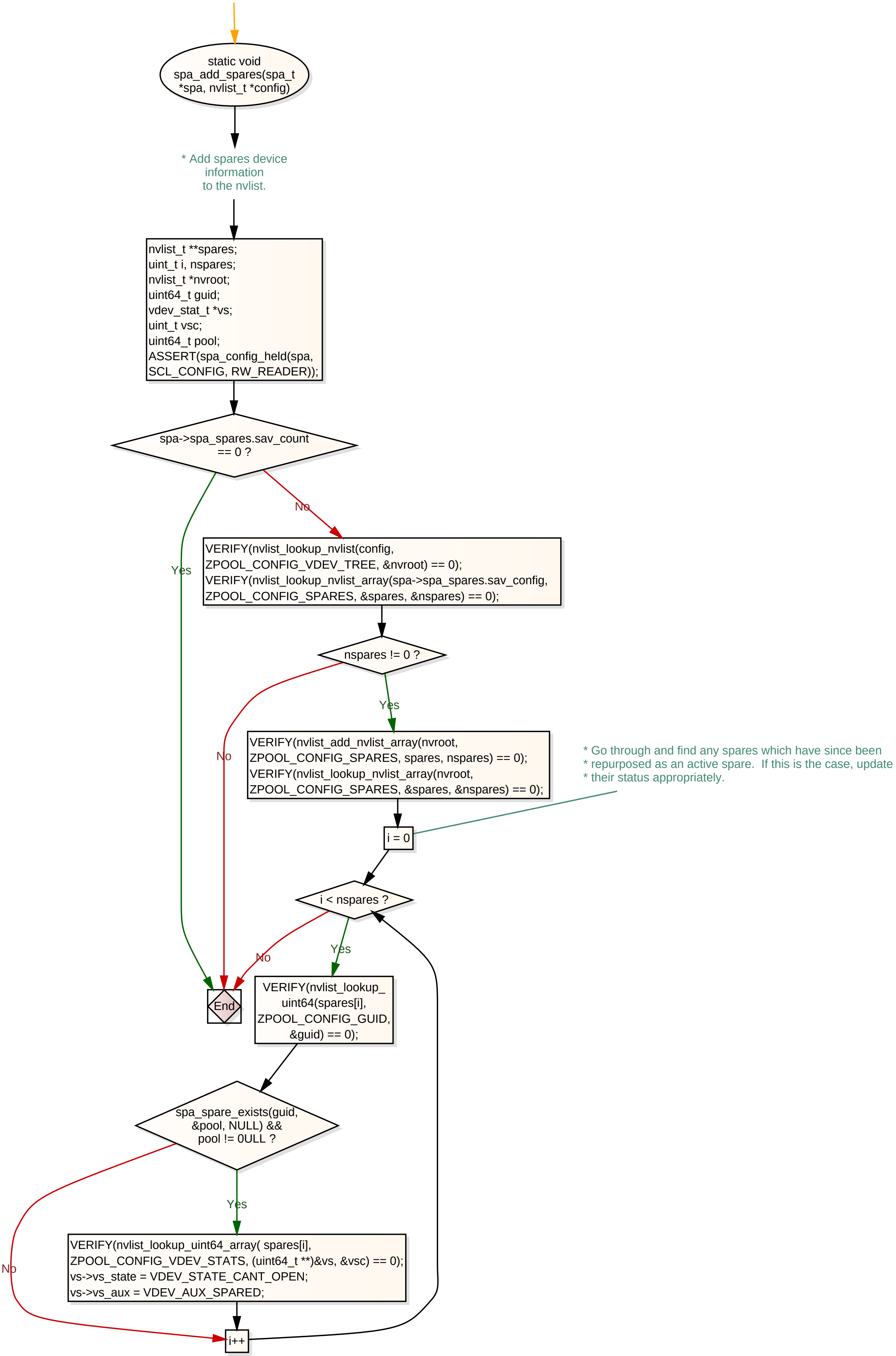


```
int spa_open(const char
*name, spa_t
**spapp, void *tag)
```

```
return (spa_open_common
(name, spapp,
tag, NULL, NULL));
```


spa_t * spa_inject_addref
(char *name)
* Lookup the given spa_t, incrementing
the inject count in the process,
* preventing it from being exported or destroyed.
spa_t *spa;
mutex_enter(&spa_namespace_lock);
(spa = spa_lookup(name))
== NULL ?
No
Yes
spa->spa_inject_ref++;
mutex_exit(&spa_namespace_lock);
return (spa);
mutex_exit(&spa_namespace_lock);
return (NULL);
End


void spa_inject_
delref(spa_t *spa)
mutex_enter(&spa_namespace_lock);
spa->spa_inject_ref--;
mutex_exit(&spa_namespace_lock);
End


static void spa_add_spares(spa_t *spa, nvlist_t *config)
* Add spares device information to the nvlist.
spa->spa_spares.sav_count == 0 ?
Yes
No
nspares != 0 ?
Yes
No
* Go through and find any spares which have since been
* repurposed as an active spare. If this is the case, update
* their status appropriately.
i = 0
i < nspares ?
Yes
No
End
spa_spare_exists(guid, &pool, NULL) && pool != 0ULL ?
Yes
No
i++

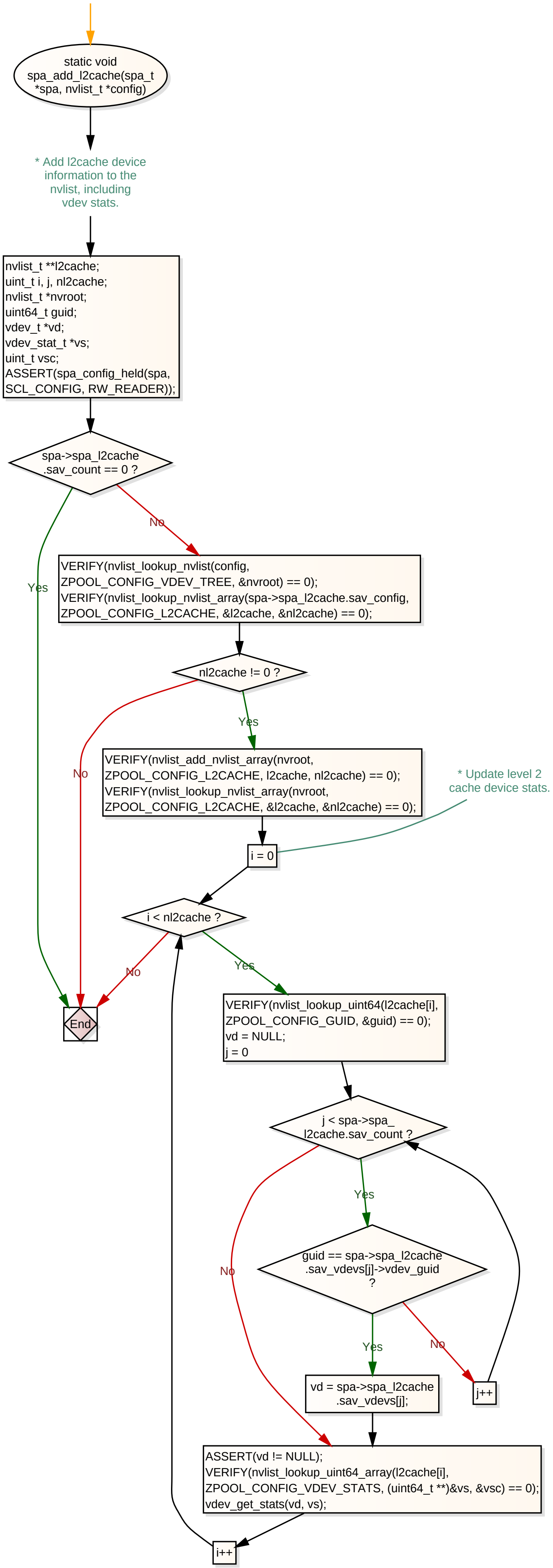
```
nvlist_t **spares;
uint_t i, nspares;
nvlist_t *nvroot;
uint64_t guid;
vdev_stat_t *vs;
uint_t vsc;
uint64_t pool;
ASSERT(spa_config_held(spa,
SCL_CONFIG, RW_READER));
```

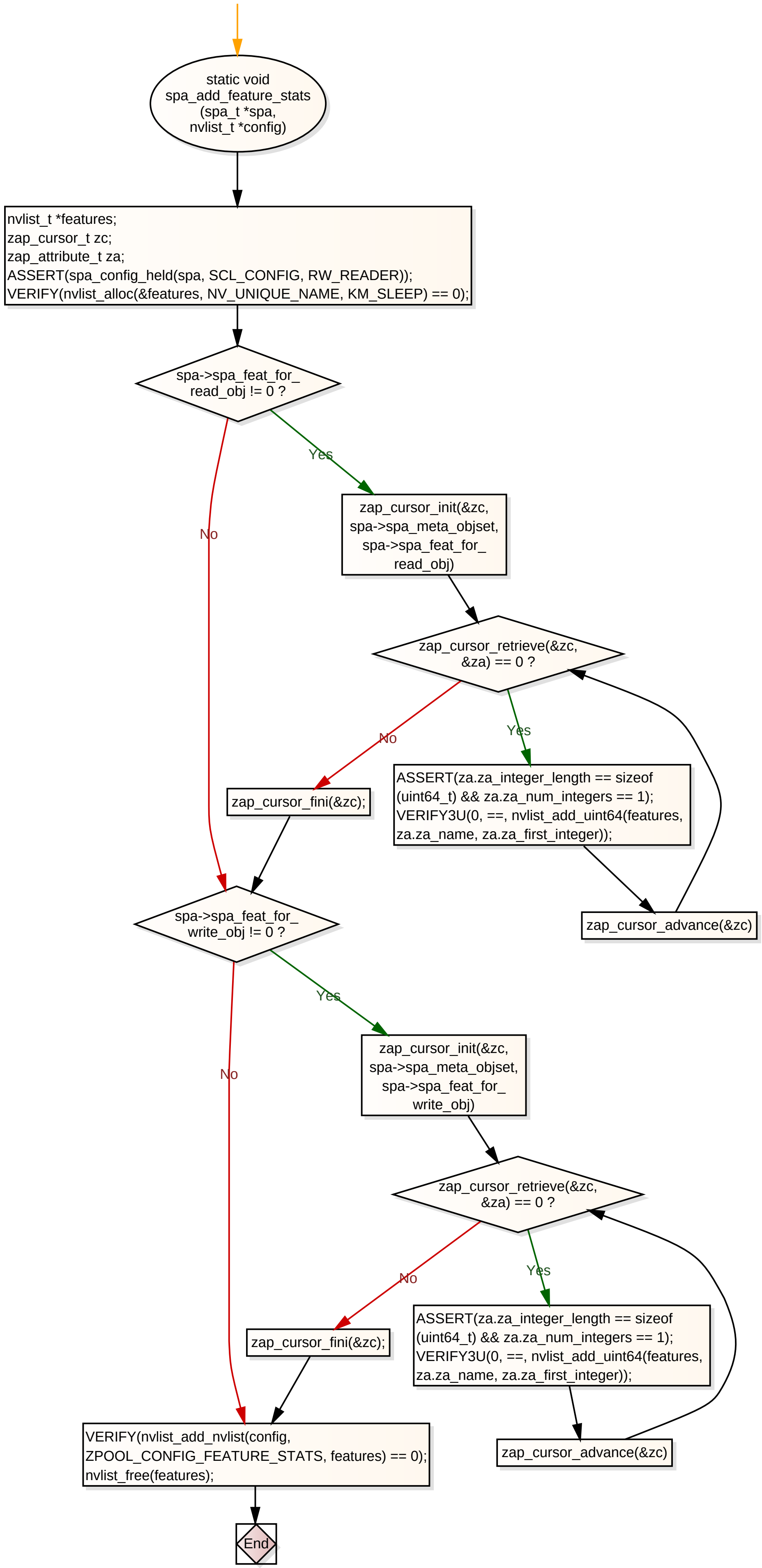
```
VERIFY(nvlist_lookup_nvlist(config,
ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
VERIFY(nvlist_lookup_nvlist_array(spa->spa_spares.sav_config,
ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
```

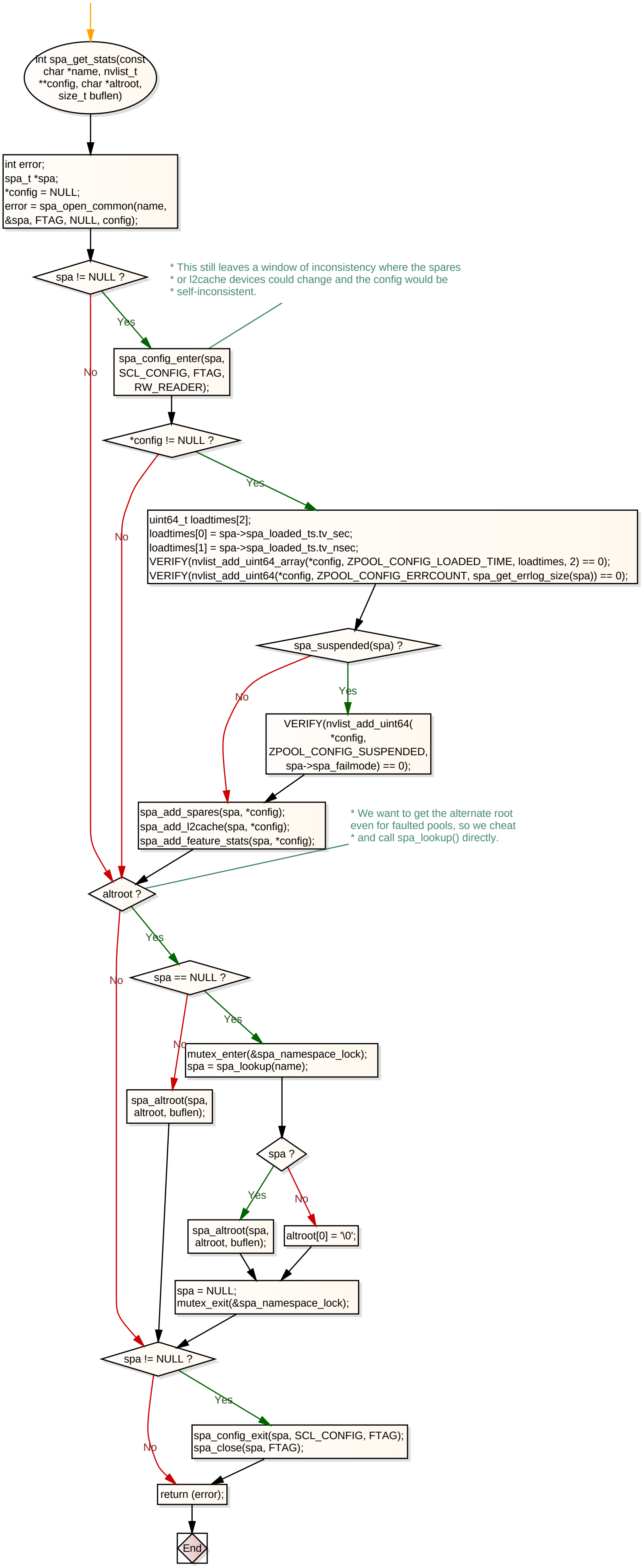
```
VERIFY(nvlist_add_nvlist_array(nvroot,
ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
VERIFY(nvlist_lookup_nvlist_array(nvroot,
ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0);
```

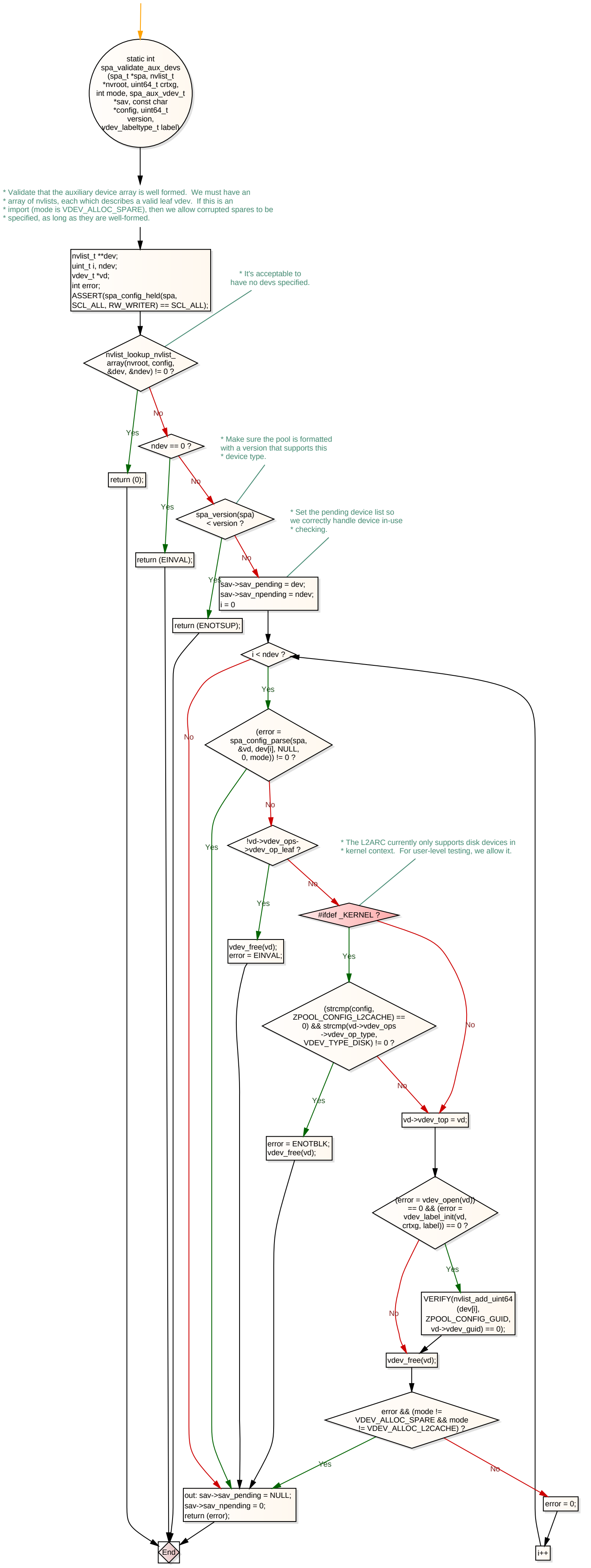
```
VERIFY(nvlist_lookup_
uint64(spares[i],
ZPOOL_CONFIG_GUID,
&guid) == 0);
```

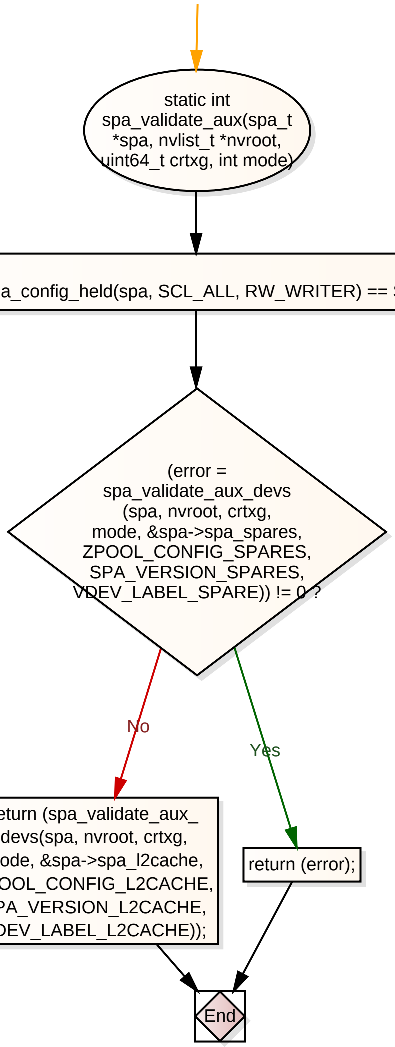
```
VERIFY(nvlist_lookup_uint64_array( spares[i],
ZPOOL_CONFIG_VDEV_STATS, (uint64_t **)&vs, &vsc) == 0);
vs->vs_state = VDEV_STATE_CANT_OPEN;
vs->vs_aux = VDEV_AUX_SPARED;
```

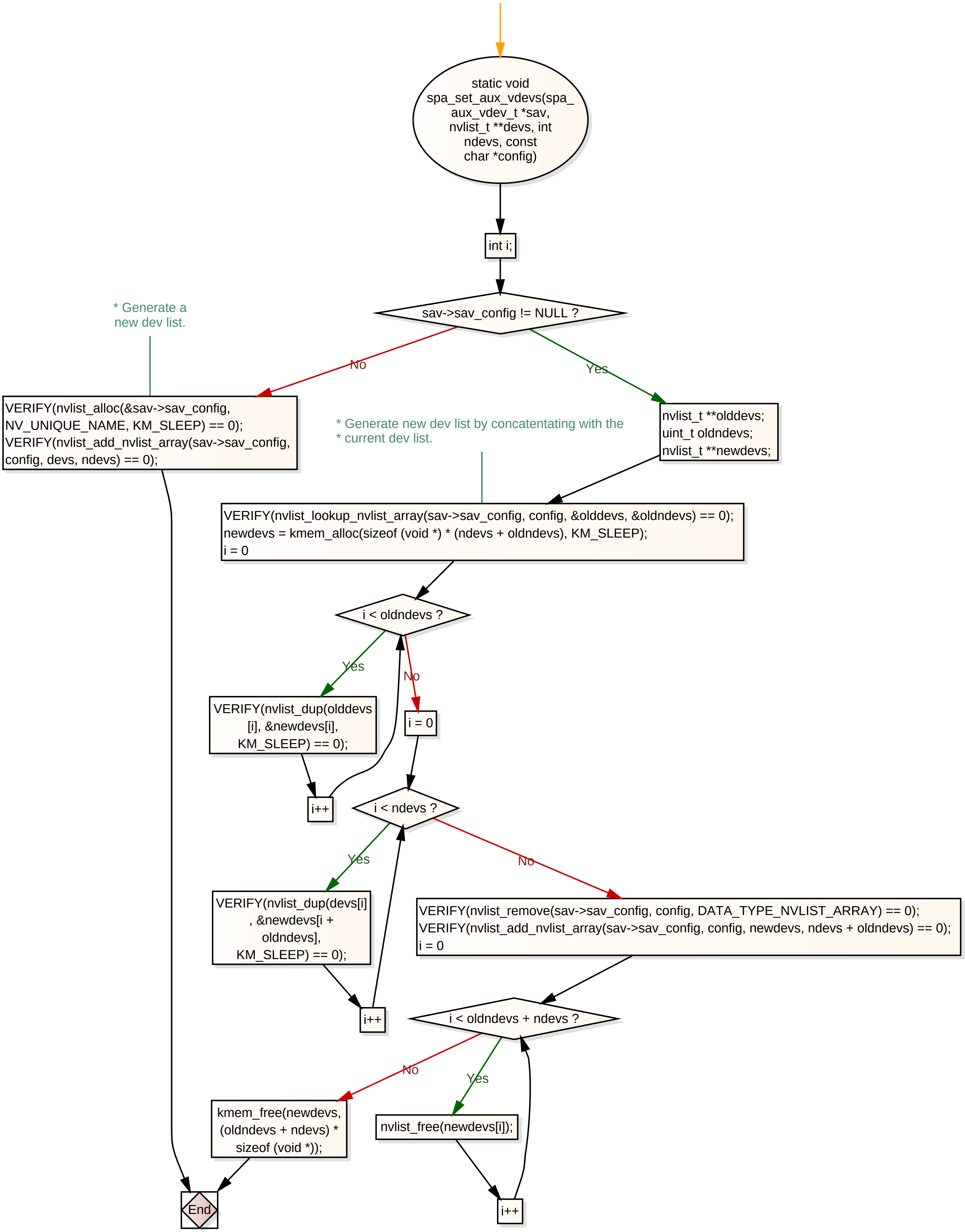

static void spa_add_l2cache(spa_t *spa, nvlist_t *config)
* Add l2cache device information to the nvlist, including vdev stats.
nvlist_t **l2cache;
uint_t i, j, nl2cache;
nvlist_t *nvroot;
uint64_t guid;
vdev_t *vd;
vdev_stat_t *vs;
uint_t vsc;
ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
spa->spa_l2cache .sav_count == 0 ?
No
Yes
VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
VERIFY(nvlist_lookup_nvlist_array(spa->spa_l2cache.sav_config, ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
nl2cache != 0 ?
Yes
No
VERIFY(nvlist_add_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
VERIFY(nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0);
* Update level 2 cache device stats.
i = 0
i < nl2cache ?
No
Yes
End
VERIFY(nvlist_lookup_uint64(l2cache[i], ZPOOL_CONFIG_GUID, &guid) == 0);
vd = NULL;
j = 0
j < spa->spa_ l2cache.sav_count ?
Yes
No
guid == spa->spa_l2cache .sav_vdevs[j]->vdev_guid ?
Yes
No
vd = spa->spa_l2cache .sav_vdevs[j];
j++
ASSERT(vd != NULL);
VERIFY(nvlist_lookup_uint64_array(l2cache[i], ZPOOL_CONFIG_VDEV_STATS, (uint64_t **)&vs, &vsc) == 0);
vdev_get_stats(vd, vs);
i++

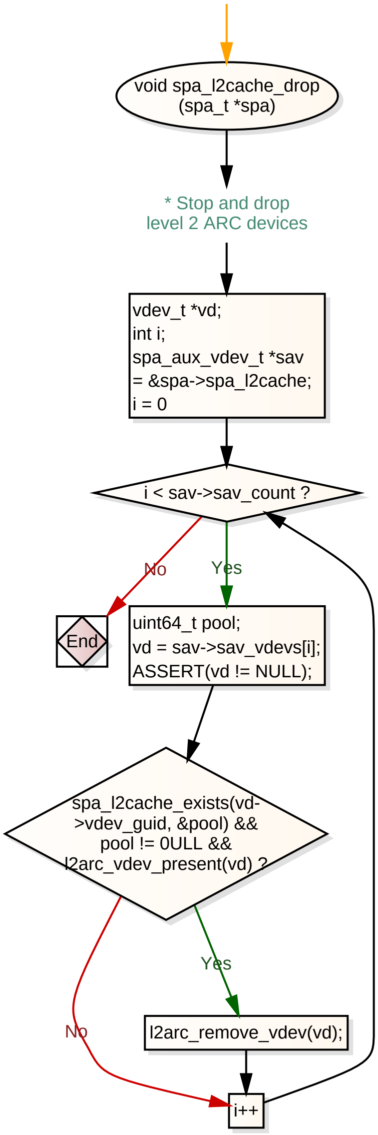

static void
spa_add_feature_stats
(spa_t *spa,
nvlist_t *config)
nvlist_t *features;
zap_cursor_t zc;
zap_attribute_t za;
ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
VERIFY(nvlist_alloc(&features, NV_UNIQUE_NAME, KM_SLEEP) == 0);
spa->spa_feat_for_
read_obj != 0 ?
Yes
No
zap_cursor_init(&zc,
spa->spa_meta_objset,
spa->spa_feat_for_
read_obj)
zap_cursor_retrieve(&zc,
&za) == 0 ?
No
Yes
zap_cursor_fini(&zc);
ASSERT(za.za_integer_length == sizeof
(uint64_t) && za.za_num_integers == 1);
VERIFY3U(0, ==, nvlist_add_uint64(features,
za.za_name, za.za_first_integer));
zap_cursor_advance(&zc)
spa->spa_feat_for_
write_obj != 0 ?
Yes
No
zap_cursor_init(&zc,
spa->spa_meta_objset,
spa->spa_feat_for_
write_obj)
zap_cursor_retrieve(&zc,
&za) == 0 ?
No
Yes
zap_cursor_fini(&zc);
ASSERT(za.za_integer_length == sizeof
(uint64_t) && za.za_num_integers == 1);
VERIFY3U(0, ==, nvlist_add_uint64(features,
za.za_name, za.za_first_integer));
zap_cursor_advance(&zc)
VERIFY(nvlist_add_nvlist(config,
ZPOOL_CONFIG_FEATURE_STATS, features) == 0);
nvlist_free(features);
End

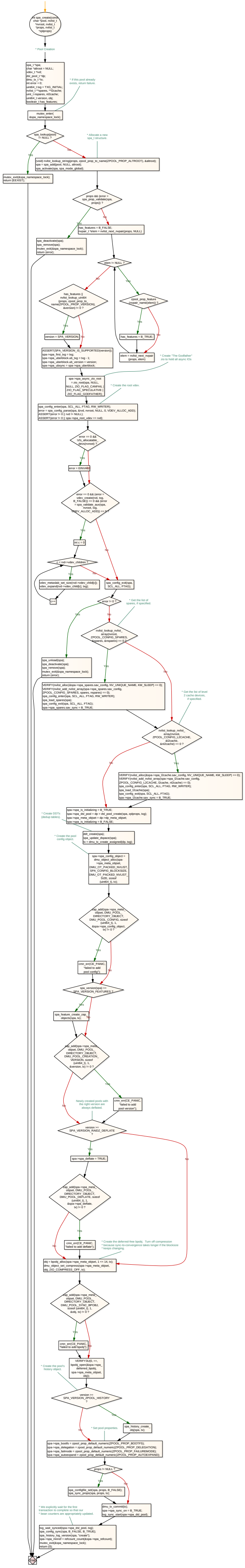

int spa_get_stats(const char *name, nvlist_t **config, char *altroot, size_t buflen)
int error;
spa_t *spa;
*config = NULL;
error = spa_open_common(name, &spa, FTAG, NULL, config);
spa != NULL ?
* This still leaves a window of inconsistency where the spares
* or l2cache devices could change and the config would be
* self-inconsistent.
Yes
No
spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
*config != NULL ?
Yes
No
uint64_t loadtimes[2];
loadtimes[0] = spa->spa_loaded_ts.tv_sec;
loadtimes[1] = spa->spa_loaded_ts.tv_nsec;
VERIFY(nvlist_add_uint64_array(*config, ZPOOL_CONFIG_LOADED_TIME, loadtimes, 2) == 0);
VERIFY(nvlist_add_uint64(*config, ZPOOL_CONFIG_ERRCOUNT, spa_get_errlog_size(spa)) == 0);
spa_suspended(spa) ?
No
Yes
VERIFY(nvlist_add_uint64(*config, ZPOOL_CONFIG_SUSPENDED, spa->spa_failmode) == 0);
spa_add_spares(spa, *config);
spa_add_l2cache(spa, *config);
spa_add_feature_stats(spa, *config);
* We want to get the alternate root
even for faulted pools, so we cheat
* and call spa_lookup() directly.
altroot ?
Yes
No
spa == NULL ?
Yes
No
mutex_enter(&spa_namespace_lock);
spa = spa_lookup(name);
spa_altroot(spa, altroot, buflen);
spa ?
Yes
No
spa_altroot(spa, altroot, buflen);
altroot[0] = '\0';
spa = NULL;
mutex_exit(&spa_namespace_lock);
spa != NULL ?
Yes
No
spa_config_exit(spa, SCL_CONFIG, FTAG);
spa_close(spa, FTAG);
return (error);
End

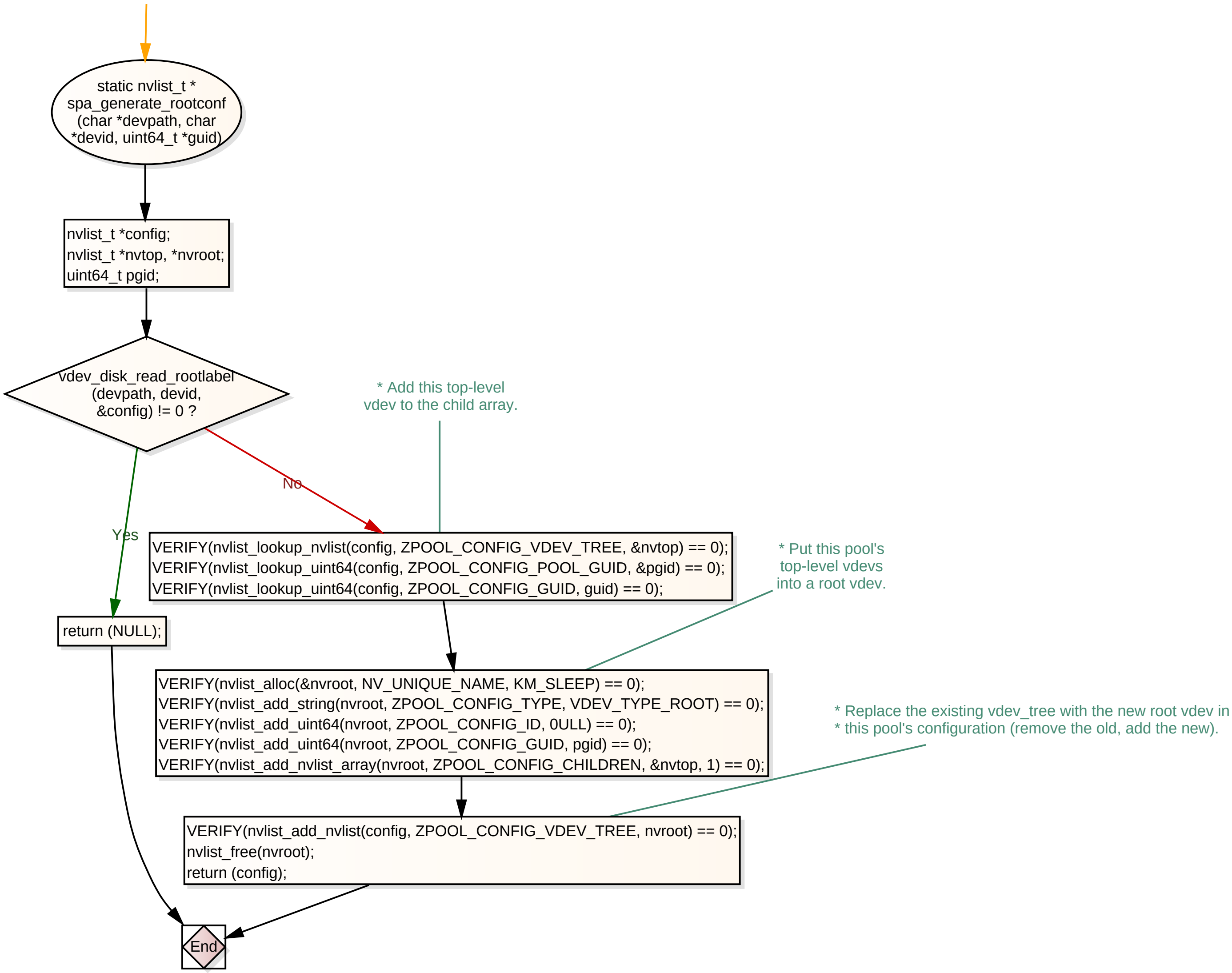

static int spa_validate_aux_devs (spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode, spa_aux_vdev_t *sav, const char *config, uint64_t version, vdev_labeltype_t label)
* Validate that the auxiliary device array is well formed. We must have an
* array of nvlists, each which describes a valid leaf vdev. If this is an
* import (mode is VDEV_ALLOC_SPARE), then we allow corrupted spares to be
* specified, as long as they are well-formed.
nvlist_t **dev;
uint_t i, ndev;
vdev_t *vd;
int error;
ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
* It's acceptable to have no devs specified.
nvlist_lookup_nvlist_array(nvroot, config, &dev, &ndev) != 0 ?
Yes
No
return (0);
ndev == 0 ?
Yes
No
return (EINVAL);
* Make sure the pool is formatted with a version that supports this
* device type.
spa_version(spa) < version ?
Yes
No
return (ENOTSUP);
* Set the pending device list so we correctly handle device in-use
* checking.
sav->sav_pending = dev;
sav->sav_npending = ndev;
i = 0
i < ndev ?
Yes
No
(error = spa_config_parse(spa, &vd, dev[i], NULL, 0, mode)) != 0 ?
Yes
No
!vd->vdev_ops->vdev_op_leaf ?
Yes
No
vdev_free(vd);
error = EINVAL;
* The L2ARC currently only supports disk devices in
* kernel context. For user-level testing, we allow it.
#ifdef _KERNEL ?
Yes
No
(strcmp(config, ZPOOL_CONFIG_L2CACHE) == 0) && strcmp(vd->vdev_ops->vdev_op_type, VDEV_TYPE_DISK) != 0 ?
Yes
No
error = ENOTBLK;
vdev_free(vd);
vd->vdev_top = vd;
(error = vdev_open(vd)) == 0 && (error = vdev_label_init(vd, crtxg, label)) == 0 ?
Yes
No
VERIFY(nvlist_add_uint64 (dev[i], ZPOOL_CONFIG_GUID, vd->vdev_guid) == 0);
vdev_free(vd);
error && (mode != VDEV_ALLOC_SPARE && mode != VDEV_ALLOC_L2CACHE) ?
Yes
No
error = 0;
out: sav->sav_pending = NULL;
sav->sav_npending = 0;
return (error);
End
i++


static int spa_validate_aux(spa_t *spa, nvlist_t *nvroot, uint64_t crtxg, int mode)
int error;
ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
(error = spa_validate_aux_devs (spa, nvroot, crtxg, mode, &spa->spa_spares, ZPOOL_CONFIG_SPARES, SPA_VERSION_SPARES, VDEV_LABEL_SPARE)) != 0 ?
No
Yes
return (spa_validate_aux_devs(spa, nvroot, crtxg, mode, &spa->spa_l2cache, ZPOOL_CONFIG_L2CACHE, SPA_VERSION_L2CACHE, VDEV_LABEL_L2CACHE));
return (error);
End


static void spa_set_aux_vdevs(spa_aux_vdev_t *sav, nvlist_t **devs, int ndevs, const char *config)
int i;
sav->sav_config != NULL ?
No
Yes
* Generate a new dev list.
VERIFY(nvlist_alloc(&sav->sav_config, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_nvlist_array(sav->sav_config, config, devs, ndevs) == 0);
nvlist_t **olddevs;
uint_t oldndevs;
nvlist_t **newdevs;
* Generate new dev list by concatentating with the
* current dev list.
VERIFY(nvlist_lookup_nvlist_array(sav->sav_config, config, &olddevs, &oldndevs) == 0);
newdevs = kmem_alloc(sizeof (void *) * (ndevs + oldndevs), KM_SLEEP);
i = 0
i < oldndevs ?
Yes
No
VERIFY(nvlist_dup(olddevs[i], &newdevs[i], KM_SLEEP) == 0);
i = 0
i++
i < ndevs ?
Yes
No
VERIFY(nvlist_dup(devs[i], &newdevs[i + oldndevs], KM_SLEEP) == 0);
VERIFY(nvlist_remove(sav->sav_config, config, DATA_TYPE_NVLIST_ARRAY) == 0);
VERIFY(nvlist_add_nvlist_array(sav->sav_config, config, newdevs, ndevs + oldndevs) == 0);
i = 0
i++
i < oldndevs + ndevs ?
No
Yes
kmem_free(newdevs, (oldndevs + ndevs) * sizeof (void *));
nvlist_free(newdevs[i]);
End
i++


void spa_l2cache_drop (spa_t *spa)
* Stop and drop level 2 ARC devices
vdev_t *vd;
int i;
spa_aux_vdev_t *sav = &spa->spa_l2cache;
i = 0
i < sav->sav_count ?
No
Yes
End
uint64_t pool;
vd = sav->sav_vdevs[i];
ASSERT(vd != NULL);
spa_l2cache_exists(vd->vdev_guid, &pool) && pool != 0ULL && l2arc_vdev_present(vd) ?
Yes
No
l2arc_remove_vdev(vd);
i++


int spa_create(const char *pool, nvlist_t *nvroot, nvlist_t *props, nvlist_t *zplprops)
* Pool Creation
spa_t *spa;
char *altroot = NULL;
vdev_t *rvd;
dsl_pool_t *dp;
dmu_tx_t *tx;
int error = 0;
uint64_t txg = TXG_INITIAL;
nvlist_t **spares, **l2cache;
uint_t nspares, nl2cache;
uint64_t version, obj;
boolean_t has_features;
* If this pool already exists, return failure.
mutex_enter(&spa_namespace_lock);
spa_lookup(pool) != NULL ?
Yes
No
* Allocate a new spa_t structure.
mutex_exit(&spa_namespace_lock);
return (EEXIST);
(void) nvlist_lookup_string(props, zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
spa = spa_add(pool, NULL, altroot);
spa_activate(spa, spa_mode_global);
props && (error = spa_prop_validate(spa, props)) ?
Yes
No
spa_deactivate(spa);
spa_remove(spa);
mutex_exit(&spa_namespace_lock);
return (error);
has_features = B_FALSE;
nvpair_t *elem = nvlist_next_nvpair(props, NULL)
elem != NULL ?
No
Yes
has_features || nvlist_lookup_uint64 (props, zpool_prop_to_ name(ZPOOL_PROP_VERSION), &version) != 0 ?
zpool_prop_feature (nvpair_name(elem)) ?
Yes
No
Yes
No
version = SPA_VERSION;
has_features = B_TRUE;
ASSERT(SPA_VERSION_IS_SUPPORTED(version));
spa->spa_first_txg = txg;
spa->spa_uberblock.ub_txg = txg - 1;
spa->spa_uberblock.ub_version = version;
spa->spa_ubsync = spa->spa_uberblock;
elem = nvlist_next_nvpair (props, elem)
* Create "The Godfather" zio to hold all async IOs
spa->spa_async_zio_root = zio_root(spa, NULL, NULL, ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);
* Create the root vdev.
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, VDEV_ALLOC_ADD);
ASSERT(error != 0 || rvd != NULL);
ASSERT(error != 0 || spa->spa_root_vdev == rvd);
error == 0 && !zfs_allocatable_ devs(nvroot) ?
Yes
No
error = EINVAL;
error == 0 && (error = vdev_create(rvd, txg, B_FALSE)) == 0 && (error = spa_validate_aux(spa, nvroot, txg, VDEV_ALLOC_ADD)) == 0 ?
Yes
No
int c = 0
c < rvd->vdev_children ?
Yes
No
vdev_metaslab_set_size(rvd->vdev_child[c]);
vdev_expand(rvd->vdev_child[c], txg);
c++
spa_config_exit(spa, SCL_ALL, FTAG);
error != 0 ?
* Get the list of spares, if specified.
Yes
No
nvlist_lookup_nvlist_ array(nvroot, ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0 ?
spa_unload(spa);
spa_deactivate(spa);
spa_remove(spa);
mutex_exit(&spa_namespace_lock);
return (error);
Yes
No
VERIFY(nvlist_alloc(&spa->spa_spares.sav_config, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config, ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
spa_load_spares(spa);
spa_config_exit(spa, SCL_ALL, FTAG);
spa->spa_spares.sav_sync = B_TRUE;
* Get the list of level 2 cache devices, if specified.
nvlist_lookup_nvlist_ array(nvroot, ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0 ?
No
Yes
VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config, ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
spa_load_l2cache(spa);
spa_config_exit(spa, SCL_ALL, FTAG);
spa->spa_l2cache.sav_sync = B_TRUE;
spa->spa_is_initializing = B_TRUE;
spa->spa_dsl_pool = dp = dsl_pool_create(spa, zplprops, txg);
spa->spa_meta_objset = dp->dp_meta_objset;
spa->spa_is_initializing = B_FALSE;
* Create DDTs (dedup tables).
ddt_create(spa);
spa_update_dspace(spa);
tx = dmu_tx_create_assigned(dp, txg);
* Create the pool config object.
spa->spa_config_object = dmu_object_alloc(spa->spa_meta_objset, DMU_OT_PACKED_NVLIST, SPA_CONFIG_BLOCKSIZE, DMU_OT_PACKED_NVLIST_ SIZE, sizeof (uint64_t), tx);
zap_add(spa->spa_meta_ objset, DMU_POOL_ DIRECTORY_OBJECT, DMU_POOL_CONFIG, sizeof (uint64_t), 1, &spa->spa_config_object, tx) != 0 ?
Yes
No
cmn_err(CE_PANIC, "failed to add pool config");
spa_version(spa) >= SPA_VERSION_FEATURES ?
Yes
No
spa_feature_create_zap_ objects(spa, tx);
zap_add(spa->spa_meta_ objset, DMU_POOL_ DIRECTORY_OBJECT, DMU_POOL_CREATION_ VERSION, sizeof (uint64_t), 1, &version, tx) != 0 ?
No
Yes
Newly created pools with the right version are always deflated.
cmn_err(CE_PANIC, "failed to add pool version");
version >= SPA_VERSION_RAIDZ_DEFLATE ?
Yes
No
spa->spa_deflate = TRUE;
zap_add(spa->spa_meta_ objset, DMU_POOL_ DIRECTORY_OBJECT, DMU_POOL_DEFLATE, sizeof (uint64_t), 1, &spa->spa_deflate, tx) != 0 ?
Yes
No
cmn_err(CE_PANIC, "failed to add deflate");
* Create the deferred-free bpobj. Turn off compression
* because sync-to-convergence takes longer if the blocksize
* keeps changing.
obj = bpobj_alloc(spa->spa_meta_objset, 1 << 14, tx);
dmu_object_set_compress(spa->spa_meta_objset, obj, ZIO_COMPRESS_OFF, tx);
zap_add(spa->spa_meta_ objset, DMU_POOL_ DIRECTORY_OBJECT, DMU_POOL_SYNC_BPOBJ, sizeof (uint64_t), 1, &obj, tx) != 0 ?
Yes
No
cmn_err(CE_PANIC, "failed to add bpobj");
VERIFY3U(0, ==, bpobj_open(&spa->spa_ deferred_bpobj, spa->spa_meta_objset, obj));
* Create the pool's history object.
version >= SPA_VERSION_ZPOOL_HISTORY ?
Yes
No
spa_history_create_ obj(spa, tx);
* Set pool properties.
spa->spa_bootfs = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
spa->spa_failmode = zpool_prop_default_numeric(ZPOOL_PROP_FAILUREMODE);
spa->spa_autoexpand = zpool_prop_default_numeric(ZPOOL_PROP_AUTOEXPAND);
props != NULL ?
Yes
No
spa_configfile_set(spa, props, B_FALSE);
spa_sync_props(spa, props, tx);
dmu_tx_commit(tx);
spa->spa_sync_on = B_TRUE;
txg_sync_start(spa->spa_dsl_pool);
* We explicitly wait for the first transaction to complete so that our
* bean counters are appropriately updated.
txg_wait_synced(spa->spa_dsl_pool, txg);
spa_config_sync(spa, B_FALSE, B_TRUE);
spa_history_log_version(spa, "create");
spa->spa_minref = refcount_count(&spa->spa_refcount);
mutex_exit(&spa_namespace_lock);
return (0);
End



```
static nvlist_t *
spa_generate_rootconf
(char *devpath, char
*devid, uint64_t *guid)
```

```
nvlist_t *config;
nvlist_t *nvtop, *nvroot;
uint64_t pgid;
```

```
vdev_disk_read_rootlabel
(devpath, devid,
&config) != 0 ?
```

```
return (NULL);
```

```
VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvtop) == 0);
VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID, &pgid) == 0);
VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID, guid) == 0);
```

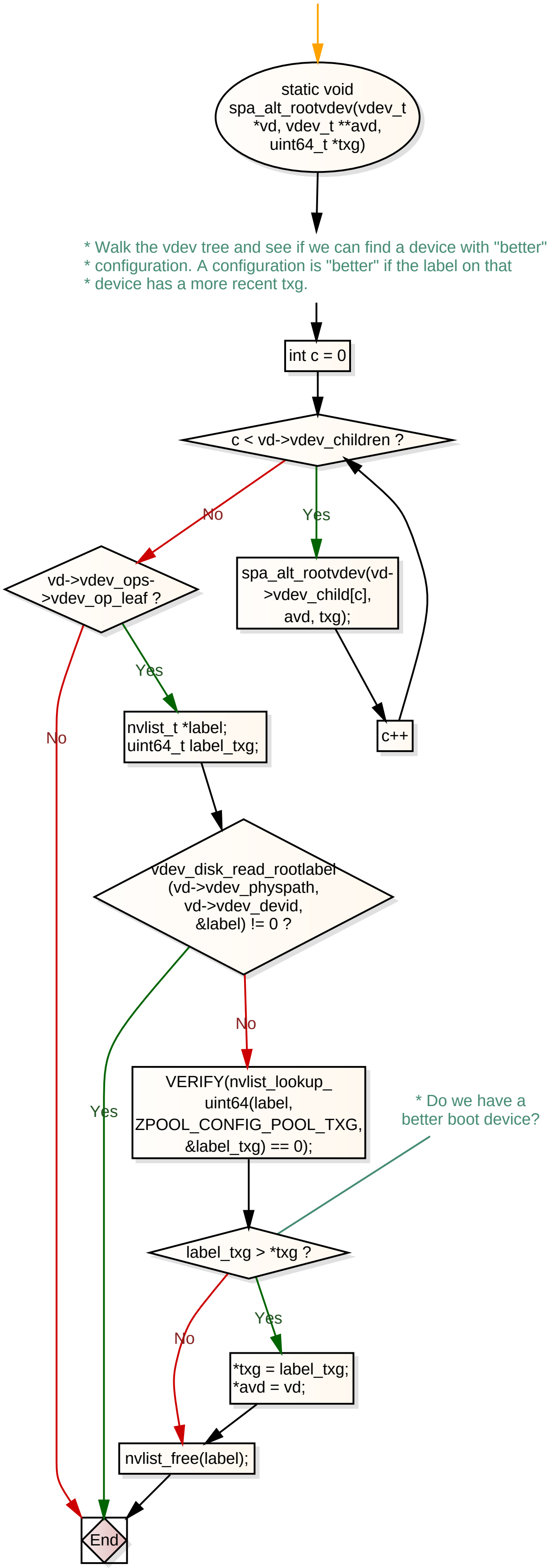
* Add this top-level
vdev to the child array.

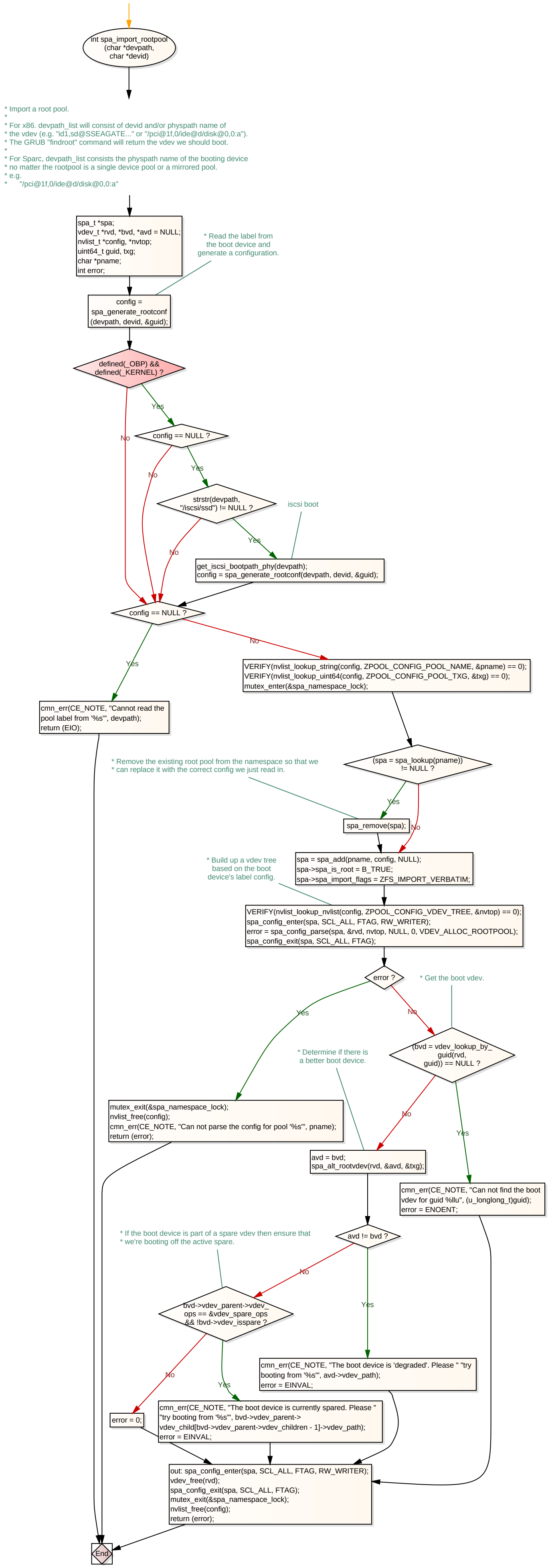
```
VERIFY(nvlist_alloc(&nvroot, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_string(nvroot, ZPOOL_CONFIG_TYPE, VDEV_TYPE_ROOT) == 0);
VERIFY(nvlist_add_uint64(nvroot, ZPOOL_CONFIG_ID, 0ULL) == 0);
VERIFY(nvlist_add_uint64(nvroot, ZPOOL_CONFIG_GUID, pgid) == 0);
VERIFY(nvlist_add_nvlist_array(nvroot, ZPOOL_CONFIG_CHILDREN, &nvtop, 1) == 0);
```

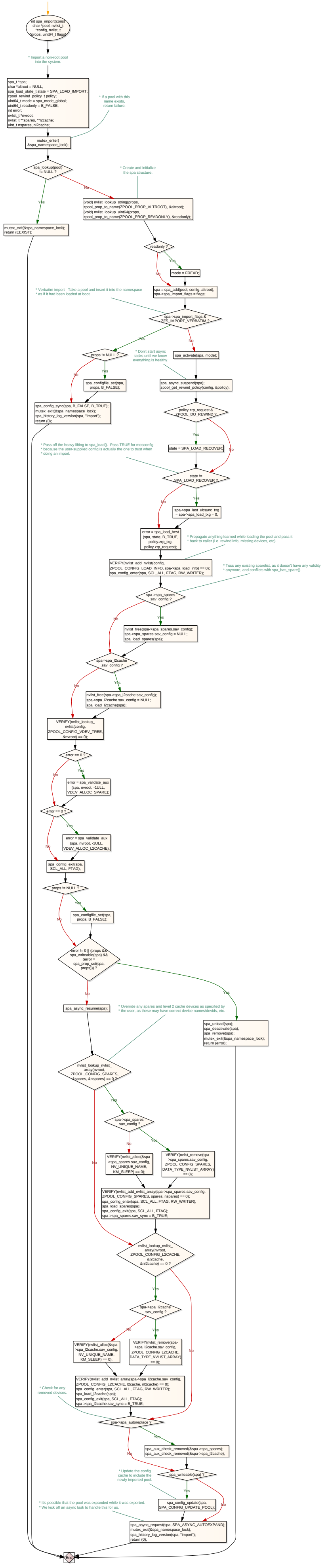
* Put this pool's
top-level vdevs
into a root vdev.

```
VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, nvroot) == 0);
nvlist_free(nvroot);
return (config);
```

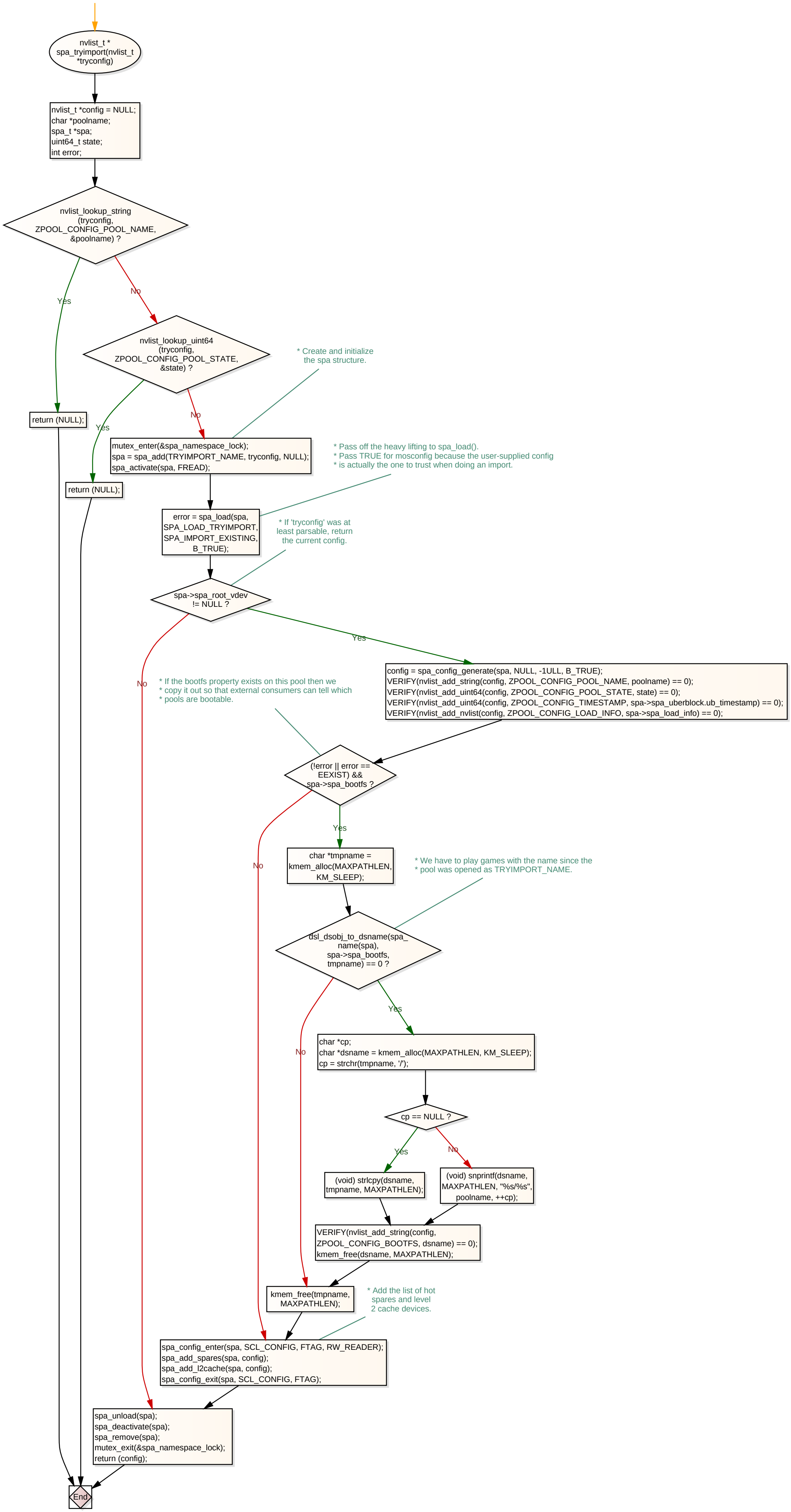
* Replace the existing vdev_tree with the new root vdev in
* this pool's configuration (remove the old, add the new).

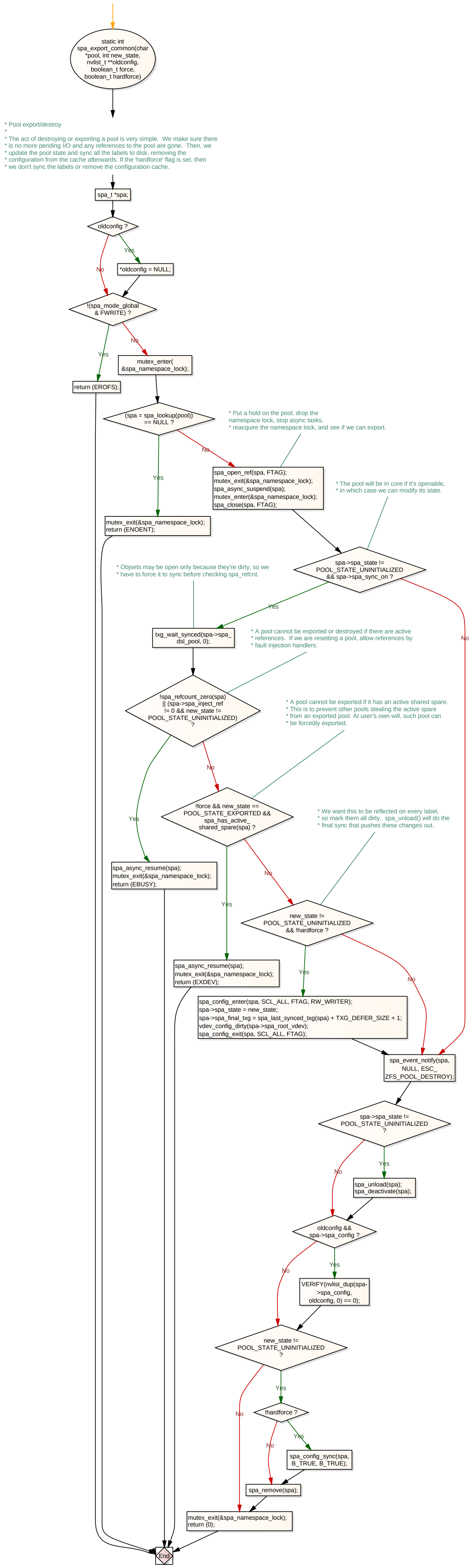

static void spa_alt_rootvdev(vdev_t *vd, vdev_t **avd, uint64_t *txg)
* Walk the vdev tree and see if we can find a device with "better"
* configuration. A configuration is "better" if the label on that
* device has a more recent txg.
int c = 0
c < vd->vdev_children ?
No
Yes
vd->vdev_ops->vdev_op_leaf ?
spa_alt_rootvdev(vd->vdev_child[c], avd, txg);
Yes
No
nvlist_t *label;
uint64_t label_txg;
c++
vdev_disk_read_rootlabel (vd->vdev_physpath, vd->vdev_devid, &label) != 0 ?
No
Yes
VERIFY(nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_TXG, &label_txg) == 0);
* Do we have a better boot device?
label_txg > *txg ?
Yes
No
*txg = label_txg;
*avd = vd;
nvlist_free(label);
End


int spa_import_rootpool
(char *devpath,
char *devid)
* Import a root pool.
*
* For x86. devpath_list will consist of devid and/or physpath name of
* the vdev (e.g. "id1,sd@SSEAGATE..." or "/pci@1f,0/ide@d/disk@0,0:a").
* The GRUB "findroot" command will return the vdev we should boot.
*
* For Sparc, devpath_list consists the physpath name of the booting device
* no matter the rootpool is a single device pool or a mirrored pool.
* e.g.
*   "/pci@1f,0/ide@d/disk@0,0:a"
spa_t *spa;
vdev_t *rvd, *bvd, *avd = NULL;
nvlist_t *config, *nvtop;
uint64_t guid, txg;
char *pname;
int error;
* Read the label from
the boot device and
generate a configuration.
config =
spa_generate_rootconf
(devpath, devid, &guid);
defined(_OBP) &&
defined(_KERNEL) ?
Yes
No
config == NULL ?
Yes
No
strstr(devpath,
"/iscsi/ssd") != NULL ?
iscsi boot
Yes
No
get_iscsi_bootpath_phy(devpath);
config = spa_generate_rootconf(devpath, devid, &guid);
config == NULL ?
No
Yes
VERIFY(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME, &pname) == 0);
VERIFY(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG, &txg) == 0);
mutex_enter(&spa_namespace_lock);
cmn_err(CE_NOTE, "Cannot read the
pool label from '%s'", devpath);
return (EIO);
* Remove the existing root pool from the namespace so that we
* can replace it with the correct config we just read in.
(spa = spa_lookup(pname))
!= NULL ?
Yes
No
spa_remove(spa);
spa = spa_add(pname, config, NULL);
spa->spa_is_root = B_TRUE;
spa->spa_import_flags = ZFS_IMPORT_VERBATIM;
* Build up a vdev tree
based on the boot
device's label config.
VERIFY(nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvtop) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
error = spa_config_parse(spa, &rvd, nvtop, NULL, 0, VDEV_ALLOC_ROOTPOOL);
spa_config_exit(spa, SCL_ALL, FTAG);
error ?
* Get the boot vdev.
Yes
No
(bvd = vdev_lookup_by_
guid(rvd,
guid)) == NULL ?
* Determine if there is
a better boot device.
mutex_exit(&spa_namespace_lock);
nvlist_free(config);
cmn_err(CE_NOTE, "Can not parse the config for pool '%s'", pname);
return (error);
No
Yes
avd = bvd;
spa_alt_rootvdev(rvd, &avd, &txg);
cmn_err(CE_NOTE, "Can not find the boot
vdev for guid %llu", (u_longlong_t)guid);
error = ENOENT;
* If the boot device is part of a spare vdev then ensure that
* we're booting off the active spare.
avd != bvd ?
No
Yes
bvd->vdev_parent->vdev_
ops == &vdev_spare_ops
&& !bvd->vdev_isspare ?
cmn_err(CE_NOTE, "The boot device is 'degraded'. Please " "try
booting from '%s'", avd->vdev_path);
error = EINVAL;
No
Yes
error = 0;
cmn_err(CE_NOTE, "The boot device is currently spared. Please "
"try booting from '%s'", bvd->vdev_parent->
vdev_child[bvd->vdev_parent->vdev_children - 1]->vdev_path);
error = EINVAL;
out: spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
vdev_free(rvd);
spa_config_exit(spa, SCL_ALL, FTAG);
mutex_exit(&spa_namespace_lock);
nvlist_free(config);
return (error);
End



```
int spa_import(const char *pool, nvlist_t *config, nvlist_t *props, uint64_t flags)

* Import a non-root pool
into the system.

spa_t *spa;
char *altroot = NULL;
spa_load_state_t state = SPA_LOAD_IMPORT;
zpool_rewind_policy_t policy;
uint64_t mode = spa_mode_global;
uint64_t readonly = B_FALSE;
int error;
nvlist_t *nvroot;
nvlist_t **spares, **l2cache;
uint_t nspares, nl2cache;

* If a pool with this
name exists,
return failure.

mutex_enter(
&spa_namespace_lock);

spa_lookup(pool)
!= NULL ?

mutex_exit(&spa_namespace_lock);
return (EEXIST);

* Create and initialize
the spa structure.

(void) nvlist_lookup_string(props,
zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
(void) nvlist_lookup_uint64(props,
zpool_prop_to_name(ZPOOL_PROP_READONLY), &readonly);

readonly ?

mode = FREAD;

spa = spa_add(pool, config, altroot);
spa->spa_import_flags = flags;

* Verbatim import - Take a pool and insert it into the namespace
* as if it had been loaded at boot.

spa->spa_import_flags &
ZFS_IMPORT_VERBATIM ?

props != NULL ?

spa_configfile_set(spa,
props, B_FALSE);

spa_config_sync(spa, B_FALSE, B_TRUE);
mutex_exit(&spa_namespace_lock);
spa_history_log_version(spa, "import");
return (0);

spa_activate(spa, mode);

* Don't start async
tasks until we know
everything is healthy.

spa_async_suspend(spa);
zpool_get_rewind_policy(config, &policy);

policy.zrp_request &
ZPOOL_DO_REWIND ?

state = SPA_LOAD_RECOVER;

* Pass off the heavy lifting to spa_load().  Pass TRUE for mosconfig
* because the user-supplied config is actually the one to trust when
* doing an import.

state !=
SPA_LOAD_RECOVER ?

spa->spa_last_ubsync_txg
= spa->spa_load_txg = 0;

error = spa_load_best
(spa, state, B_TRUE,
policy.zrp_txg,
policy.zrp_request);

* Propagate anything learned while loading the pool and pass it
* back to caller (i.e. rewind info, missing devices, etc).

VERIFY(nvlist_add_nvlist(config,
ZPOOL_CONFIG_LOAD_INFO, spa->spa_load_info) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

* Toss any existing sparelist, as it doesn't have any validity
* anymore, and conflicts with spa_has_spare().

spa->spa_spares
.sav_config ?

nvlist_free(spa->spa_spares.sav_config);
spa->spa_spares.sav_config = NULL;
spa_load_spares(spa);

spa->spa_l2cache
.sav_config ?

nvlist_free(spa->spa_l2cache.sav_config);
spa->spa_l2cache.sav_config = NULL;
spa_load_l2cache(spa);

VERIFY(nvlist_lookup_
nvlist(config,
ZPOOL_CONFIG_VDEV_TREE,
&nvroot) == 0);

error == 0 ?

error = spa_validate_aux
(spa, nvroot, -1ULL,
VDEV_ALLOC_SPARE);

error == 0 ?

error = spa_validate_aux
(spa, nvroot, -1ULL,
VDEV_ALLOC_L2CACHE);

spa_config_exit(spa,
SCL_ALL, FTAG);

props != NULL ?

spa_configfile_set(spa,
props, B_FALSE);

error != 0 || (props &&
spa_writeable(spa) &&
(error =
spa_prop_set(spa,
props))) ?

spa_unload(spa);
spa_deactivate(spa);
spa_remove(spa);
mutex_exit(&spa_namespace_lock);
return (error);

spa_async_resume(spa);

* Override any spares and level 2 cache devices as specified by
* the user, as these may have correct device names/devids, etc.

nvlist_lookup_nvlist_
array(nvroot,
ZPOOL_CONFIG_SPARES,
&spares, &nspares) == 0 ?

spa->spa_spares
.sav_config ?

VERIFY(nvlist_alloc(&spa-
>spa_spares.sav_config,
NV_UNIQUE_NAME,
KM_SLEEP) == 0);

VERIFY(nvlist_remove(spa-
>spa_spares.sav_config,
ZPOOL_CONFIG_SPARES,
DATA_TYPE_NVLIST_ARRAY)
== 0);

VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
spa_load_spares(spa);
spa_config_exit(spa, SCL_ALL, FTAG);
spa->spa_spares.sav_sync = B_TRUE;

nvlist_lookup_nvlist_
array(nvroot,
ZPOOL_CONFIG_L2CACHE,
&l2cache,
&nl2cache) == 0 ?

spa->spa_l2cache
.sav_config ?

VERIFY(nvlist_alloc(&spa-
>spa_l2cache.sav_config,
NV_UNIQUE_NAME,
KM_SLEEP) == 0);

VERIFY(nvlist_remove(spa-
>spa_l2cache.sav_config,
ZPOOL_CONFIG_L2CACHE,
DATA_TYPE_NVLIST_ARRAY)
== 0);

VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
spa_load_l2cache(spa);
spa_config_exit(spa, SCL_ALL, FTAG);
spa->spa_l2cache.sav_sync = B_TRUE;

* Check for any
removed devices.

spa->spa_autoreplace ?

spa_aux_check_removed(&spa->spa_spares);
spa_aux_check_removed(&spa->spa_l2cache);

spa_writeable(spa) ?

* Update the config
cache to include the
newly-imported pool.

spa_config_update(spa,
SPA_CONFIG_UPDATE_POOL);

* It's possible that the pool was expanded while it was exported.
* We kick off an async task to handle this for us.

spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);
mutex_exit(&spa_namespace_lock);
spa_history_log_version(spa, "import");
return (0);
```


nvlist_t * spa_tryimport(nvlist_t *tryconfig)
nvlist_t *config = NULL;
char *poolname;
spa_t *spa;
uint64_t state;
int error;
nvlist_lookup_string (tryconfig, ZPOOL_CONFIG_POOL_NAME, &poolname) ?
Yes
No
return (NULL);
nvlist_lookup_uint64 (tryconfig, ZPOOL_CONFIG_POOL_STATE, &state) ?
Yes
No
return (NULL);
* Create and initialize the spa structure.
mutex_enter(&spa_namespace_lock);
spa = spa_add(TRYIMPORT_NAME, tryconfig, NULL);
spa_activate(spa, FREAD);
* Pass off the heavy lifting to spa_load().
* Pass TRUE for mosconfig because the user-supplied config
* is actually the one to trust when doing an import.
error = spa_load(spa, SPA_LOAD_TRYIMPORT, SPA_IMPORT_EXISTING, B_TRUE);
* If 'tryconfig' was at least parsable, return the current config.
spa->spa_root_vdev != NULL ?
Yes
No
config = spa_config_generate(spa, NULL, -1ULL, B_TRUE);
VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME, poolname) == 0);
VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE, state) == 0);
VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_TIMESTAMP, spa->spa_uberblock.ub_timestamp) == 0);
VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_LOAD_INFO, spa->spa_load_info) == 0);
* If the bootfs property exists on this pool then we
* copy it out so that external consumers can tell which
* pools are bootable.
(!error || error == EEXIST) && spa->spa_bootfs ?
Yes
No
char *tmpname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
* We have to play games with the name since the
* pool was opened as TRYIMPORT_NAME.
dsl_dsobj_to_dsname(spa_name(spa), spa->spa_bootfs, tmpname) == 0 ?
Yes
No
char *cp;
char *dsname = kmem_alloc(MAXPATHLEN, KM_SLEEP);
cp = strchr(tmpname, '/');
cp == NULL ?
Yes
No
(void) strlcpy(dsname, tmpname, MAXPATHLEN);
(void) snprintf(dsname, MAXPATHLEN, "%s/%s", poolname, ++cp);
VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_BOOTFS, dsname) == 0);
kmem_free(dsname, MAXPATHLEN);
kmem_free(tmpname, MAXPATHLEN);
* Add the list of hot spares and level 2 cache devices.
spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
spa_add_spares(spa, config);
spa_add_l2cache(spa, config);
spa_config_exit(spa, SCL_CONFIG, FTAG);
spa_unload(spa);
spa_deactivate(spa);
spa_remove(spa);
mutex_exit(&spa_namespace_lock);
return (config);
End



static int spa_export_common(char *pool, int new_state, nvlist_t **oldconfig, boolean_t force, boolean_t hardforce)

* Pool export/destroy
*
* The act of destroying or exporting a pool is very simple. We make sure there
* is no more pending I/O and any references to the pool are gone. Then, we
* update the pool state and sync all the labels to disk, removing the
* configuration from the cache afterwards. If the 'hardforce' flag is set, then
* we don't sync the labels or remove the configuration cache.

spa_t *spa;

oldconfig ?

*oldconfig = NULL;

!(spa_mode_global & FWRITE) ?

return (EROFS);

mutex_enter(&spa_namespace_lock);

(spa = spa_lookup(pool)) == NULL ?

* Put a hold on the pool, drop the
namespace lock, stop async tasks,
* reacquire the namespace lock, and see if we can export.

mutex_exit(&spa_namespace_lock);
return (ENOENT);

spa_open_ref(spa, FTAG);
mutex_exit(&spa_namespace_lock);
spa_async_suspend(spa);
mutex_enter(&spa_namespace_lock);
spa_close(spa, FTAG);

* The pool will be in core if it's openable,
* in which case we can modify its state.

spa->spa_state != POOL_STATE_UNINITIALIZED && spa->spa_sync_on ?

* Objsets may be open only because they're dirty, so we
* have to force it to sync before checking spa_refcnt.

txg_wait_synced(spa->spa_ dsl_pool, 0);

* A pool cannot be exported or destroyed if there are active
* references. If we are resetting a pool, allow references by
* fault injection handlers.

!spa_refcount_zero(spa) || (spa->spa_inject_ref != 0 && new_state != POOL_STATE_UNINITIALIZED) ?

* A pool cannot be exported if it has an active shared spare.
* This is to prevent other pools stealing the active spare
* from an exported pool. At user's own will, such pool can
* be forcedly exported.

!force && new_state == POOL_STATE_EXPORTED && spa_has_active_ shared_spare(spa) ?

spa_async_resume(spa);
mutex_exit(&spa_namespace_lock);
return (EBUSY);

* We want this to be reflected on every label,
* so mark them all dirty. spa_unload() will do the
* final sync that pushes these changes out.

new_state != POOL_STATE_UNINITIALIZED && !hardforce ?

spa_async_resume(spa);
mutex_exit(&spa_namespace_lock);
return (EXDEV);

spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
spa->spa_state = new_state;
spa->spa_final_txg = spa_last_synced_txg(spa) + TXG_DEFER_SIZE + 1;
vdev_config_dirty(spa->spa_root_vdev);
spa_config_exit(spa, SCL_ALL, FTAG);

spa_event_notify(spa, NULL, ESC_ ZFS_POOL_DESTROY);

spa->spa_state != POOL_STATE_UNINITIALIZED ?

spa_unload(spa);
spa_deactivate(spa);

oldconfig && spa->spa_config ?

VERIFY(nvlist_dup(spa->spa_config, oldconfig, 0) == 0);

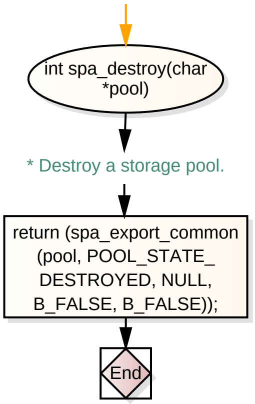
new_state != POOL_STATE_UNINITIALIZED ?

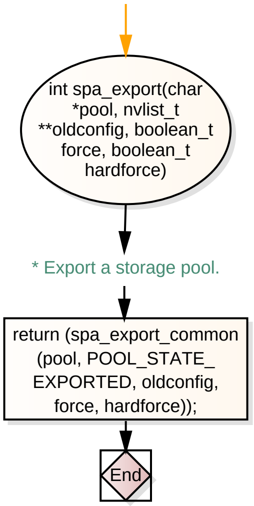
!hardforce ?

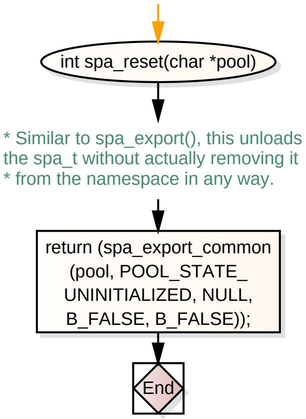
spa_config_sync(spa, B_TRUE, B_TRUE);

spa_remove(spa);

mutex_exit(&spa_namespace_lock);
return (0);


int spa_destroy(char *pool)
* Destroy a storage pool.
return (spa_export_common (pool, POOL_STATE_ DESTROYED, NULL, B_FALSE, B_FALSE));
End

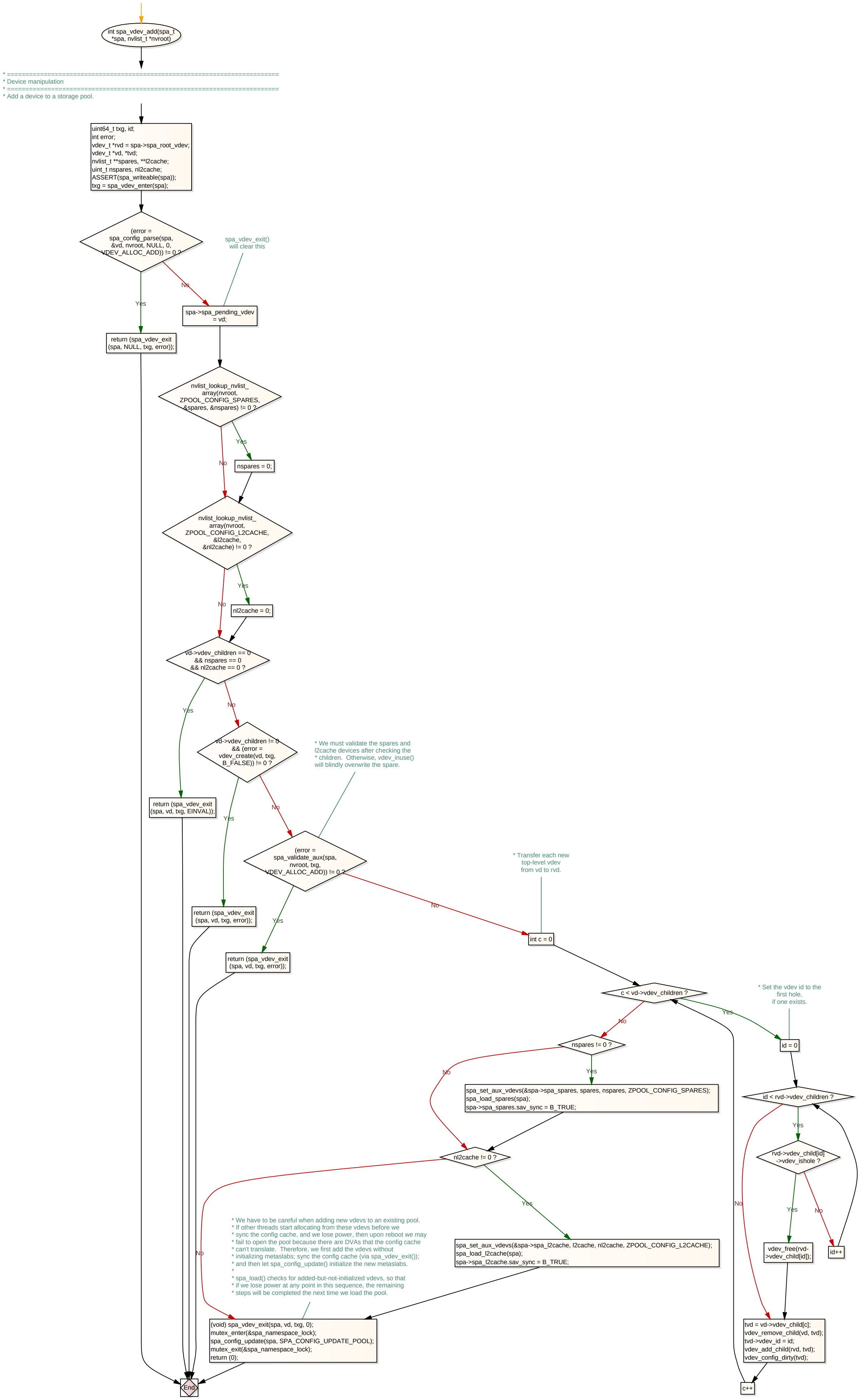

int spa_export(char *pool, nvlist_t **oldconfig, boolean_t force, boolean_t hardforce)
* Export a storage pool.
return (spa_export_common (pool, POOL_STATE_ EXPORTED, oldconfig, force, hardforce));
End



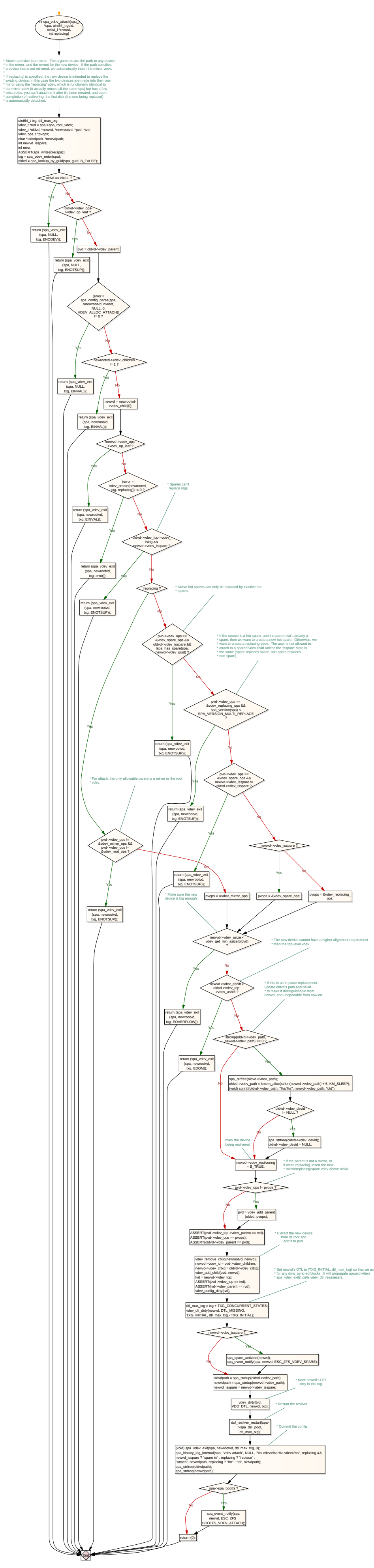
```
int spa_reset(char *pool)
```

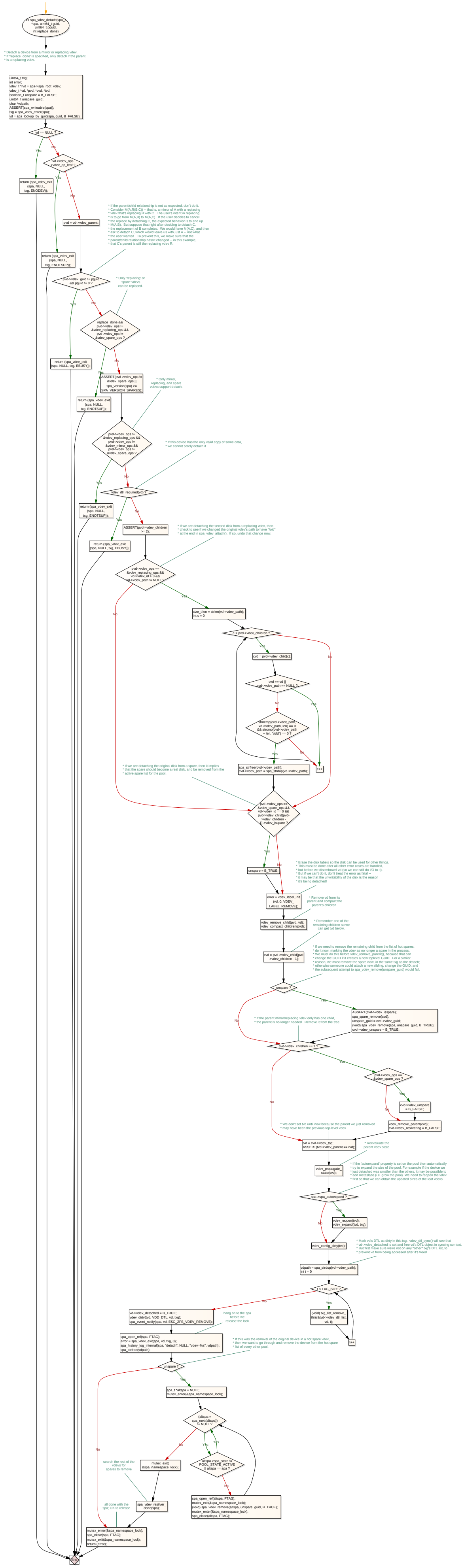
* Similar to spa_export(), this unloads the spa_t without actually removing it
* from the namespace in any way.

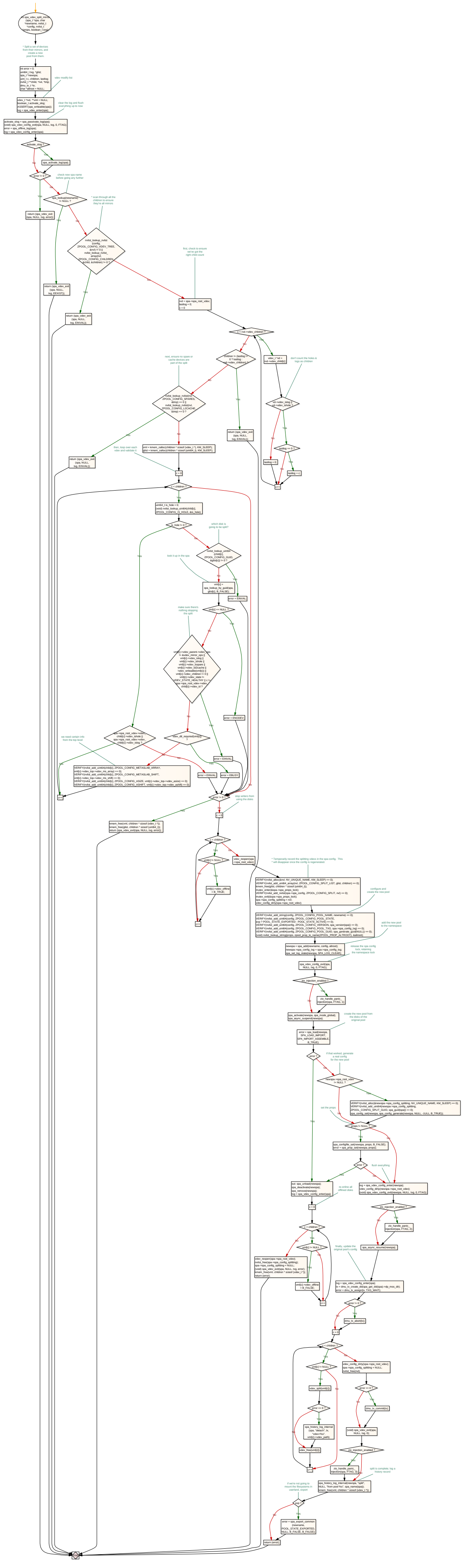
```
return (spa_export_common
(pool, POOL_STATE_
UNINITIALIZED, NULL,
B_FALSE, B_FALSE));
```

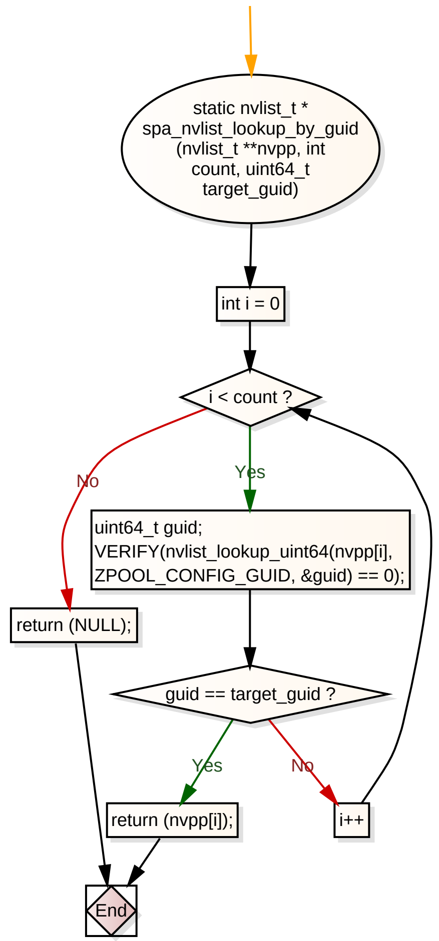


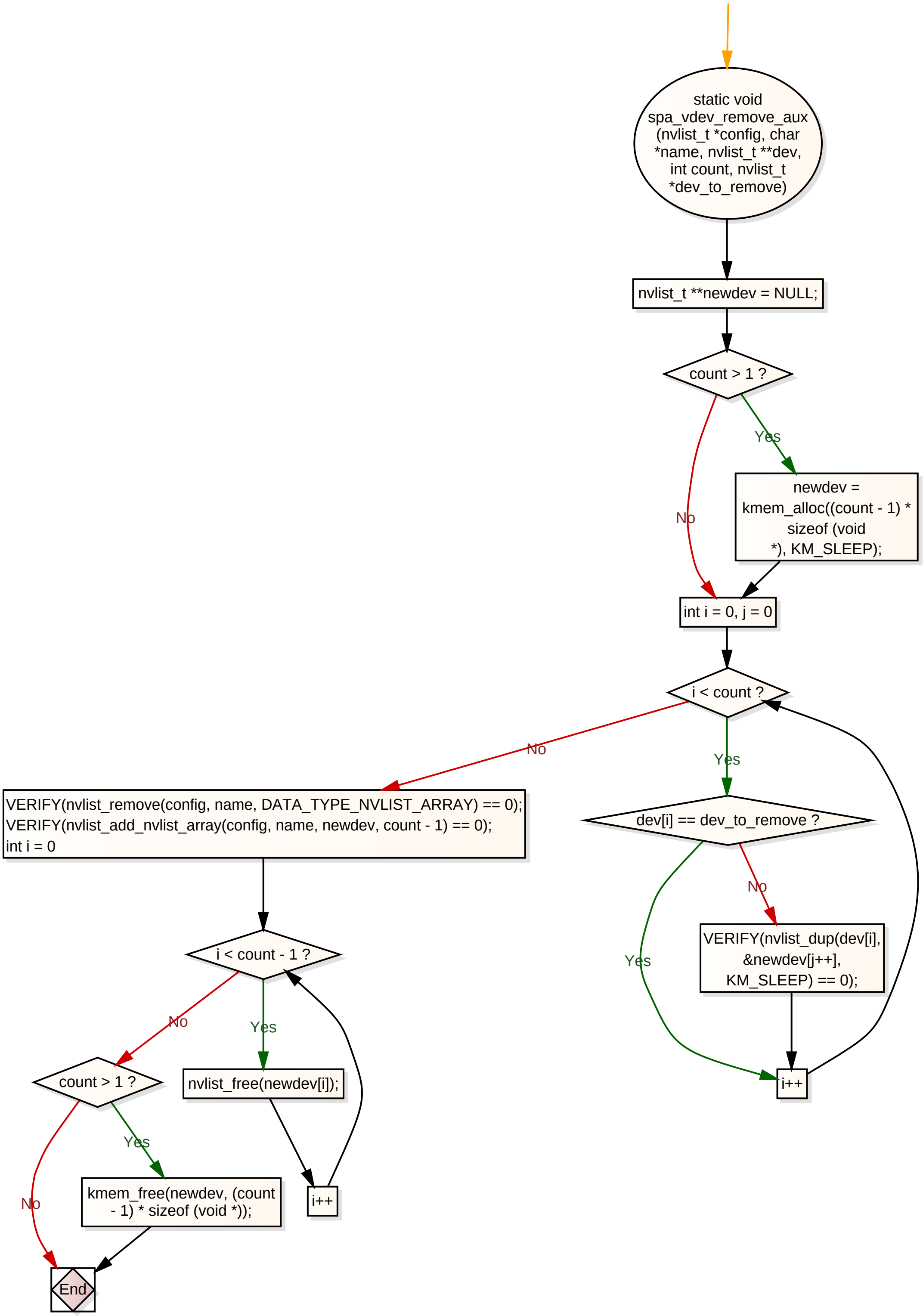
```
int spa_vdev_add(spa_t
*spa, nvlist_t *nvroot)

* ========================================================================
* Device manipulation
* ========================================================================
* Add a device to a storage pool.

uint64_t txg, id;
int error;
vdev_t *rvd = spa->spa_root_vdev;
vdev_t *vd, *tvd;
nvlist_t **spares, **l2cache;
uint_t nspares, nl2cache;
ASSERT(spa_writeable(spa));
txg = spa_vdev_enter(spa);

(error =
spa_config_parse(spa,
&vd, nvroot, NULL, 0,
VDEV_ALLOC_ADD)) != 0 ?

return (spa_vdev_exit
(spa, NULL, txg, error));

spa_vdev_exit()
will clear this

spa->spa_pending_vdev
= vd;

nvlist_lookup_nvlist_
array(nvroot,
ZPOOL_CONFIG_SPARES,
&spares, &nspares) != 0 ?

nspares = 0;

nvlist_lookup_nvlist_
array(nvroot,
ZPOOL_CONFIG_L2CACHE,
&l2cache,
&nl2cache) != 0 ?

nl2cache = 0;

vd->vdev_children == 0
&& nspares == 0
&& nl2cache == 0 ?

return (spa_vdev_exit
(spa, vd, txg, EINVAL));

vd->vdev_children != 0
&& (error =
vdev_create(vd, txg,
B_FALSE)) != 0 ?

return (spa_vdev_exit
(spa, vd, txg, error));

* We must validate the spares and
l2cache devices after checking the
* children.  Otherwise, vdev_inuse()
will blindly overwrite the spare.

(error =
spa_validate_aux(spa,
nvroot, txg,
VDEV_ALLOC_ADD)) != 0 ?

return (spa_vdev_exit
(spa, vd, txg, error));

* Transfer each new
top-level vdev
from vd to rvd.

int c = 0

c < vd->vdev_children ?

nspares != 0 ?

spa_set_aux_vdevs(&spa->spa_spares, spares, nspares, ZPOOL_CONFIG_SPARES);
spa_load_spares(spa);
spa->spa_spares.sav_sync = B_TRUE;

nl2cache != 0 ?

spa_set_aux_vdevs(&spa->spa_l2cache, l2cache, nl2cache, ZPOOL_CONFIG_L2CACHE);
spa_load_l2cache(spa);
spa->spa_l2cache.sav_sync = B_TRUE;

* We have to be careful when adding new vdevs to an existing pool.
* If other threads start allocating from these vdevs before we
* sync the config cache, and we lose power, then upon reboot we may
* fail to open the pool because there are DVAs that the config cache
* can't translate.  Therefore, we first add the vdevs without
* initializing metaslabs; sync the config cache (via spa_vdev_exit());
* and then let spa_config_update() initialize the new metaslabs.
*
* spa_load() checks for added-but-not-initialized vdevs, so that
* if we lose power at any point in this sequence, the remaining
* steps will be completed the next time we load the pool.

(void) spa_vdev_exit(spa, vd, txg, 0);
mutex_enter(&spa_namespace_lock);
spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
mutex_exit(&spa_namespace_lock);
return (0);

* Set the vdev id to the
first hole,
if one exists.

id = 0

id < rvd->vdev_children ?

rvd->vdev_child[id]
->vdev_ishole ?

vdev_free(rvd-
>vdev_child[id]);

id++

tvd = vd->vdev_child[c];
vdev_remove_child(vd, tvd);
tvd->vdev_id = id;
vdev_add_child(rvd, tvd);
vdev_config_dirty(tvd);

c++

End
```

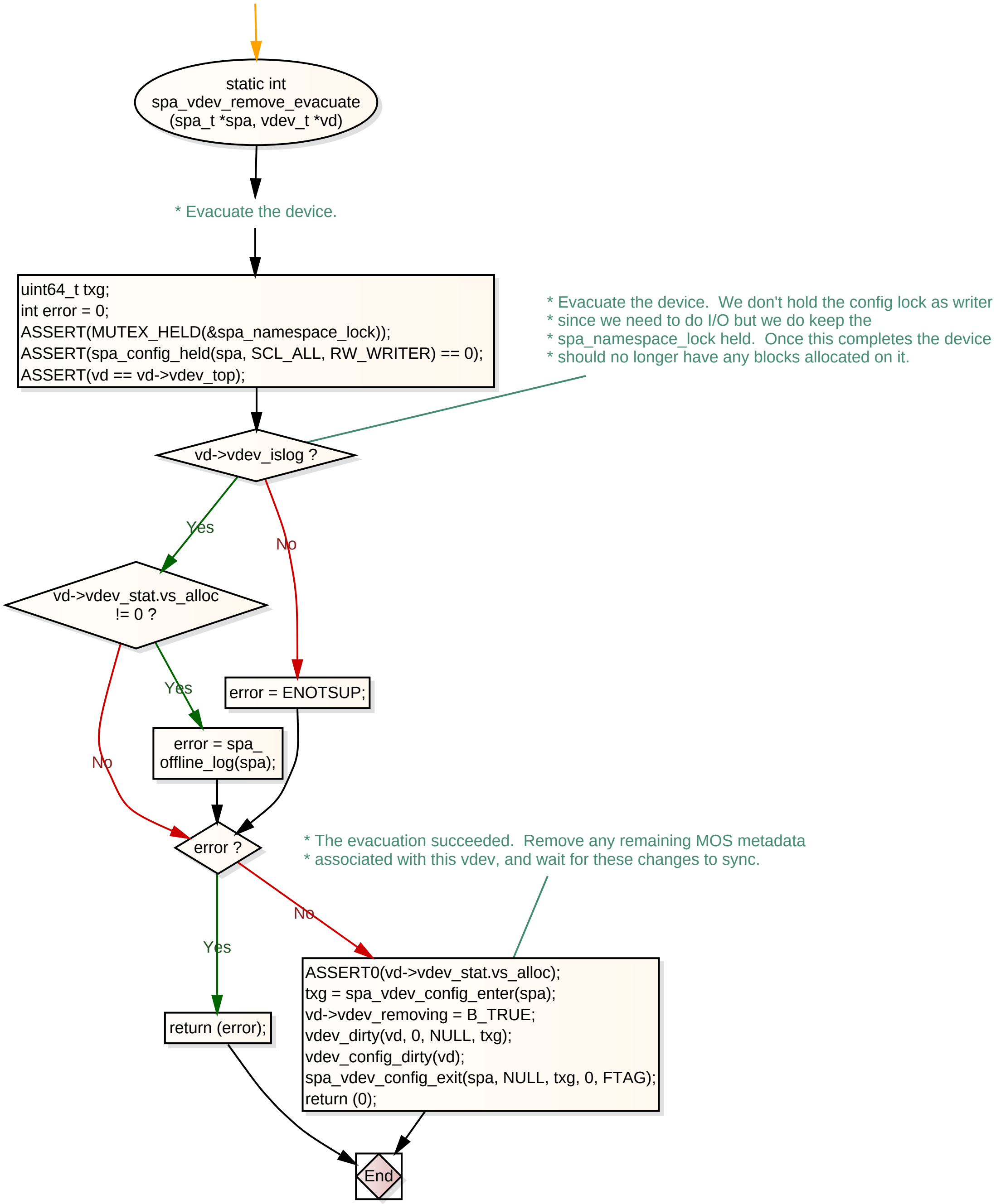

int spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot, int replacing)
* Attach a device to a mirror. The arguments are the path to any device
* in the mirror, and the nvroot for the new device. If the path specifies
* a device that is not mirrored, we automatically insert the mirror vdev.
*
* If 'replacing' is specified, the new device is intended to replace the
* existing device; in this case the two devices are made into their own
* mirror using the 'replacing' vdev, which is functionally identical to
* the mirror vdev (it actually reuses all the same ops) but has a few
* extra rules: you can't attach to it after it's been created, and upon
* completion of resilvering, the first disk (the one being replaced)
* is automatically detached.
uint64_t txg, dtl_max_txg;
vdev_t *rvd = spa->spa_root_vdev;
vdev_t *oldvd, *newvd, *newrootvd, *pvd, *tvd;
vdev_ops_t *pvops;
char *oldvdpath, *newvdpath;
int newvd_isspare;
int error;
ASSERT(spa_writeable(spa));
txg = spa_vdev_enter(spa);
oldvd = spa_lookup_by_guid(spa, guid, B_FALSE);
oldvd == NULL ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, ENODEV));
!oldvd->vdev_ops->vdev_op_leaf ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, ENOTSUP));
pvd = oldvd->vdev_parent;
(error = spa_config_parse(spa, &newrootvd, nvroot, NULL, 0, VDEV_ALLOC_ATTACH)) != 0 ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, EINVAL));
newrootvd->vdev_children != 1 ?
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, EINVAL));
newvd = newrootvd->vdev_child[0];
!newvd->vdev_ops->vdev_op_leaf ?
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, EINVAL));
(error = vdev_create(newrootvd, txg, replacing)) != 0 ?
* Spares can't replace logs
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, error));
oldvd->vdev_top->vdev_islog && newvd->vdev_isspare ?
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, ENOTSUP));
!replacing ?
* Active hot spares can only be replaced by inactive hot
* spares.
Yes
No
pvd->vdev_ops == &vdev_spare_ops && oldvd->vdev_isspare && !spa_has_spare(spa, newvd->vdev_guid) ?
* If the source is a hot spare, and the parent isn't already a
* spare, then we want to create a new hot spare. Otherwise, we
* want to create a replacing vdev. The user is not allowed to
* attach to a spared vdev child unless the 'isspare' state is
* the same (spare replaces spare, non-spare replaces
* non-spare).
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, ENOTSUP));
pvd->vdev_ops == &vdev_replacing_ops && spa_version(spa) < SPA_VERSION_MULTI_REPLACE ?
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, ENOTSUP));
pvd->vdev_ops == &vdev_spare_ops && newvd->vdev_isspare != oldvd->vdev_isspare ?
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, ENOTSUP));
* For attach, the only allowable parent is a mirror or the root
* vdev.
pvd->vdev_ops != &vdev_mirror_ops && pvd->vdev_ops != &vdev_root_ops ?
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, ENOTSUP));
newvd->vdev_isspare ?
Yes
No
pvops = &vdev_mirror_ops;
pvops = &vdev_spare_ops;
pvops = &vdev_replacing_ops;
* Make sure the new device is big enough.
newvd->vdev_asize < vdev_get_min_asize(oldvd) ?
* The new device cannot have a higher alignment requirement
* than the top-level vdev.
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, EOVERFLOW));
newvd->vdev_ashift > oldvd->vdev_top->vdev_ashift ?
* If this is an in-place replacement, update oldvd's path and devid
* to make it distinguishable from newvd, and unopenable from now on.
Yes
No
return (spa_vdev_exit (spa, newrootvd, txg, EDOM));
strcmp(oldvd->vdev_path, newvd->vdev_path) == 0 ?
Yes
No
spa_strfree(oldvd->vdev_path);
oldvd->vdev_path = kmem_alloc(strlen(newvd->vdev_path) + 5, KM_SLEEP);
(void) sprintf(oldvd->vdev_path, "%s/%s", newvd->vdev_path, "old");
oldvd->vdev_devid != NULL ?
Yes
No
spa_strfree(oldvd->vdev_devid);
oldvd->vdev_devid = NULL;
mark the device being resilvered
newvd->vdev_resilvering = B_TRUE;
* If the parent is not a mirror, or if we're replacing, insert the new
* mirror/replacing/spare vdev above oldvd.
pvd->vdev_ops != pvops ?
Yes
No
pvd = vdev_add_parent (oldvd, pvops);
ASSERT(pvd->vdev_top->vdev_parent == rvd);
ASSERT(pvd->vdev_ops == pvops);
ASSERT(oldvd->vdev_parent == pvd);
* Extract the new device from its root and add it to pvd.
vdev_remove_child(newrootvd, newvd);
newvd->vdev_id = pvd->vdev_children;
newvd->vdev_crtxg = oldvd->vdev_crtxg;
vdev_add_child(pvd, newvd);
tvd = newvd->vdev_top;
ASSERT(pvd->vdev_top == tvd);
ASSERT(tvd->vdev_parent == rvd);
vdev_config_dirty(tvd);
* Set newvd's DTL to [TXG_INITIAL, dtl_max_txg) so that we account
* for any dmu_sync-ed blocks. It will propagate upward when
* spa_vdev_exit() calls vdev_dtl_reassess().
dtl_max_txg = txg + TXG_CONCURRENT_STATES;
vdev_dtl_dirty(newvd, DTL_MISSING, TXG_INITIAL, dtl_max_txg - TXG_INITIAL);
newvd->vdev_isspare ?
Yes
No
spa_spare_activate(newvd);
spa_event_notify(spa, newvd, ESC_ZFS_VDEV_SPARE);
oldvdpath = spa_strdup(oldvd->vdev_path);
newvdpath = spa_strdup(newvd->vdev_path);
newvd_isspare = newvd->vdev_isspare;
* Mark newvd's DTL dirty in this txg.
vdev_dirty(tvd, VDD_DTL, newvd, txg);
* Restart the resilver
dsl_resilver_restart(spa->spa_dsl_pool, dtl_max_txg);
* Commit the config
(void) spa_vdev_exit(spa, newrootvd, dtl_max_txg, 0);
spa_history_log_internal(spa, "vdev attach", NULL, "%s vdev=%s %s vdev=%s", replacing && newvd_isspare ? "spare in" : replacing ? "replace" : "attach", newvdpath, replacing ? "for" : "to", oldvdpath);
spa_strfree(oldvdpath);
spa_strfree(newvdpath);
spa->spa_bootfs ?
Yes
No
spa_event_notify(spa, newvd, ESC_ZFS_BOOTFS_VDEV_ATTACH);
return (0);
End

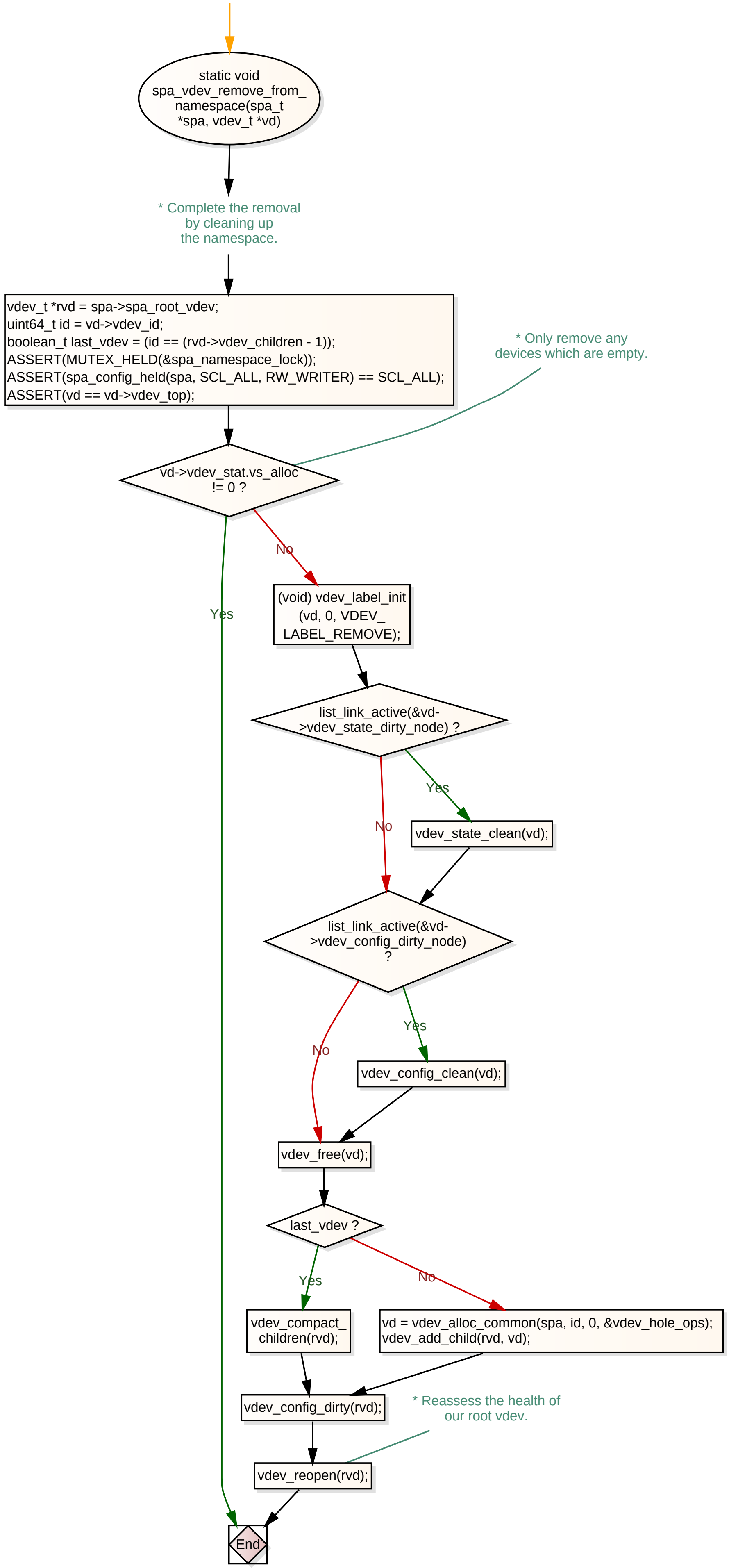

int spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid, int replace_done)
* Detach a device from a mirror or replacing vdev.
* If 'replace_done' is specified, only detach if the parent
* is a replacing vdev.
uint64_t txg;
int error;
vdev_t *rvd = spa->spa_root_vdev;
vdev_t *vd, *pvd, *cvd, *tvd;
boolean_t unspare = B_FALSE;
uint64_t unspare_guid;
char *vdpath;
ASSERT(spa_writeable(spa));
txg = spa_vdev_enter(spa);
vd = spa_lookup_by_guid(spa, guid, B_FALSE);
vd == NULL ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, ENODEV));
!vd->vdev_ops->vdev_op_leaf ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, ENOTSUP));
pvd = vd->vdev_parent;
* If the parent/child relationship is not as expected, don't do it.
* Consider M(A,R(B,C)) -- that is, a mirror of A with a replacing
* vdev that's replacing B with C. The user's intent in replacing
* is to go from M(A,B) to M(A,C). If the user decides to cancel
* the replace by detaching C, the expected behavior is to end up
* M(A,B). But suppose that right after deciding to detach C,
* the replacement of B completes. We would have M(A,C), and then
* ask to detach C, which would leave us with just A -- not what
* the user wanted. To prevent this, we make sure that the
* parent/child relationship hasn't changed -- in this example,
* that C's parent is still the replacing vdev R.
pvd->vdev_guid != pguid && pguid != 0 ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, EBUSY));
* Only 'replacing' or 'spare' vdevs can be replaced.
replace_done && pvd->vdev_ops != &vdev_replacing_ops && pvd->vdev_ops != &vdev_spare_ops ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, ENOTSUP));
ASSERT(pvd->vdev_ops != &vdev_spare_ops || spa_version(spa) >= SPA_VERSION_SPARES);
* Only mirror, replacing, and spare vdevs support detach.
pvd->vdev_ops != &vdev_replacing_ops && pvd->vdev_ops != &vdev_mirror_ops && pvd->vdev_ops != &vdev_spare_ops ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, ENOTSUP));
* If this device has the only valid copy of some data,
* we cannot safely detach it.
vdev_dtl_required(vd) ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, EBUSY));
ASSERT(pvd->vdev_children >= 2);
* If we are detaching the second disk from a replacing vdev, then
* check to see if we changed the original vdev's path to have "/old"
* at the end in spa_vdev_attach(). If so, undo that change now.
pvd->vdev_ops == &vdev_replacing_ops && vd->vdev_id > 0 && vd->vdev_path != NULL ?
Yes
No
size_t len = strlen(vd->vdev_path);
int c = 0
c < pvd->vdev_children ?
Yes
No
cvd = pvd->vdev_child[c];
cvd == vd || cvd->vdev_path == NULL ?
Yes
No
strncmp(cvd->vdev_path, vd->vdev_path, len) == 0 && strcmp(cvd->vdev_path + len, "/old") == 0 ?
Yes
No
spa_strfree(cvd->vdev_path);
cvd->vdev_path = spa_strdup(vd->vdev_path);
c++
* If we are detaching the original disk from a spare, then it implies
* that the spare should become a real disk, and be removed from the
* active spare list for the pool.
pvd->vdev_ops == &vdev_spare_ops && vd->vdev_id == 0 && pvd->vdev_child[pvd->vdev_children - 1]->vdev_isspare ?
Yes
No
unspare = B_TRUE;
* Erase the disk labels so the disk can be used for other things.
* This must be done after all other error cases are handled,
* but before we disembowel vd (so we can still do I/O to it).
* But if we can't do it, don't treat the error as fatal --
* it may be that the unwritability of the disk is the reason
* it's being detached!
error = vdev_label_init (vd, 0, VDEV_LABEL_REMOVE);
* Remove vd from its parent and compact the parent's children.
vdev_remove_child(pvd, vd);
vdev_compact_children(pvd);
* Remember one of the remaining children so we can get tvd below.
cvd = pvd->vdev_child[pvd->vdev_children - 1];
* If we need to remove the remaining child from the list of hot spares,
* do it now, marking the vdev as no longer a spare in the process.
* We must do this before vdev_remove_parent(), because that can
* change the GUID if it creates a new toplevel GUID. For a similar
* reason, we must remove the spare now, in the same txg as the detach;
* otherwise someone could attach a new sibling, change the GUID, and
* the subsequent attempt to spa_vdev_remove(unspare_guid) would fail.
unspare ?
Yes
No
ASSERT(cvd->vdev_isspare);
spa_spare_remove(cvd);
unspare_guid = cvd->vdev_guid;
(void) spa_vdev_remove(spa, unspare_guid, B_TRUE);
cvd->vdev_unspare = B_TRUE;
* If the parent mirror/replacing vdev only has one child,
* the parent is no longer needed. Remove it from the tree.
pvd->vdev_children == 1 ?
Yes
No
pvd->vdev_ops == &vdev_spare_ops ?
Yes
No
cvd->vdev_unspare = B_FALSE;
vdev_remove_parent(cvd);
cvd->vdev_resilvering = B_FALSE;
* We don't set tvd until now because the parent we just removed
* may have been the previous top-level vdev.
tvd = cvd->vdev_top;
ASSERT(tvd->vdev_parent == rvd);
* Reevaluate the parent vdev state.
vdev_propagate_state(cvd);
* If the 'autoexpand' property is set on the pool then automatically
* try to expand the size of the pool. For example if the device we
* just detached was smaller than the others, it may be possible to
* add metaslabs (i.e. grow the pool). We need to reopen the vdev
* first so that we can obtain the updated sizes of the leaf vdevs.
spa->spa_autoexpand ?
Yes
No
vdev_reopen(tvd);
vdev_expand(tvd, txg);
vdev_config_dirty(tvd);
* Mark vd's DTL as dirty in this txg. vdev_dtl_sync() will see that
* vd->vdev_detached is set and free vd's DTL object in syncing context.
* But first make sure we're not on any *other* txg's DTL list, to
* prevent vd from being accessed after it's freed.
vdpath = spa_strdup(vd->vdev_path);
int t = 0
t < TXG_SIZE ?
Yes
No
(void) txg_list_remove_this(&tvd->vdev_dtl_list, vd, t);
t++
vd->vdev_detached = B_TRUE;
vdev_dirty(tvd, VDD_DTL, vd, txg);
spa_event_notify(spa, vd, ESC_ZFS_VDEV_REMOVE);
hang on to the spa before we release the lock
spa_open_ref(spa, FTAG);
error = spa_vdev_exit(spa, vd, txg, 0);
spa_history_log_internal(spa, "detach", NULL, "vdev=%s", vdpath);
spa_strfree(vdpath);
* If this was the removal of the original device in a hot spare vdev,
* then we want to go through and remove the device from the hot spare
* list of every other pool.
unspare ?
Yes
No
spa_t *altspa = NULL;
mutex_enter(&spa_namespace_lock);
(altspa = spa_next(altspa)) != NULL ?
Yes
No
altspa->spa_state != POOL_STATE_ACTIVE || altspa == spa ?
Yes
No
spa_open_ref(altspa, FTAG);
mutex_exit(&spa_namespace_lock);
(void) spa_vdev_remove(altspa, unspare_guid, B_TRUE);
mutex_enter(&spa_namespace_lock);
spa_close(altspa, FTAG);
mutex_exit(&spa_namespace_lock);
search the rest of the vdevs for spares to remove
spa_vdev_resilver_done(spa);
all done with the spa; OK to release
mutex_enter(&spa_namespace_lock);
spa_close(spa, FTAG);
mutex_exit(&spa_namespace_lock);
return (error);
End


int spa_vdev_split_mirror (spa_t *spa, char *newname, nvlist_t *config, nvlist_t *props, boolean_t exp)
* Split a set of devices from their mirrors, and create a new pool from them.
int error = 0;
uint64_t txg, *glist;
spa_t *newspa;
uint_t c, children, lastlog;
nvlist_t **child, *nvl, *tmp;
dmu_tx_t *tx;
char *altroot = NULL;
vdev modify list
vdev_t *rvd, **vml = NULL;
boolean_t activate_slog;
ASSERT(spa_writeable(spa));
txg = spa_vdev_enter(spa);
clear the log and flush everything up to now
activate_slog = spa_passivate_log(spa);
(void) spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
error = spa_offline_log(spa);
txg = spa_vdev_config_enter(spa);
activate_slog ?
spa_activate_log(spa);
error != 0 ?
check new spa name before going any further
spa_lookup(newname) != NULL ?
return (spa_vdev_exit (spa, NULL, txg, error));
* scan through all the children to ensure they're all mirrors
nvlist_lookup_nvlist (config, ZPOOL_CONFIG_VDEV_TREE, &nvl) != 0 || nvlist_lookup_nvlist_ array(nvl, ZPOOL_CONFIG_CHILDREN, &child, &children) != 0 ?
return (spa_vdev_exit (spa, NULL, txg, EEXIST));
return (spa_vdev_exit (spa, NULL, txg, EINVAL));
first, check to ensure we've got the right child count
rvd = spa->spa_root_vdev;
lastlog = 0;
c = 0
c < rvd->vdev_children ?
vdev_t *vd = rvd->vdev_child[c];
don't count the holes & logs as children
children != (lastlog != 0 ? lastlog : rvd->vdev_children) ?
vd->vdev_islog || vd->vdev_ishole ?
lastlog == 0 ?
lastlog = 0;
lastlog = c;
c++
next, ensure no spare or cache devices are part of the split
nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_SPARES, &tmp) == 0 || nvlist_lookup_nvlist(nvl, ZPOOL_CONFIG_L2CACHE, &tmp) == 0 ?
return (spa_vdev_exit (spa, NULL, txg, EINVAL));
return (spa_vdev_exit (spa, NULL, txg, EINVAL));
then, loop over each vdev and validate it
vml = kmem_zalloc(children * sizeof (vdev_t *), KM_SLEEP);
glist = kmem_zalloc(children * sizeof (uint64_t), KM_SLEEP);
c = 0
c < children ?
uint64_t is_hole = 0;
(void) nvlist_lookup_uint64(child[c], ZPOOL_CONFIG_IS_HOLE, &is_hole);
is_hole != 0 ?
which disk is going to be split?
nvlist_lookup_uint64 (child[c], ZPOOL_CONFIG_GUID, &glist[c]) != 0 ?
look it up in the spa
vml[c] = spa_lookup_by_guid(spa, glist[c], B_FALSE);
error = EINVAL;
vml[c] == NULL ?
make sure there's nothing stopping the split
vml[c]->vdev_parent->vdev_ops != &vdev_mirror_ops || vml[c]->vdev_islog || vml[c]->vdev_ishole || vml[c]->vdev_isspare || vml[c]->vdev_isl2cache || !vdev_writeable(vml[c]) || vml[c]->vdev_children != 0 || vml[c]->vdev_state != VDEV_STATE_HEALTHY || c != spa->spa_root_vdev->vdev child[c]->vdev_id ?
error = ENODEV;
spa->spa_root_vdev->vdev_ child[c]->vdev_ishole || spa->spa_root_vdev->vdev_ child[c]->vdev_islog ?
vdev_dtl_required(vml[c]) ?
error = EINVAL;
we need certain info from the top level
VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_ARRAY, vml[c]->vdev_top->vdev_ms_array) == 0);
VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_METASLAB_SHIFT, vml[c]->vdev_top->vdev_ms_shift) == 0);
VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASIZE, vml[c]->vdev_top->vdev_asize) == 0);
VERIFY(nvlist_add_uint64(child[c], ZPOOL_CONFIG_ASHIFT, vml[c]->vdev_top->vdev_ashift) == 0);
error = EINVAL;
error = EBUSY;
c++
error != 0 ?
stop writers from using the disks
kmem_free(vml, children * sizeof (vdev_t *));
kmem_free(glist, children * sizeof (uint64_t));
return (spa_vdev_exit(spa, NULL, txg, error));
c = 0
c < children ?
vml[c] != NULL ?
vml[c]->vdev_offline = B_TRUE;
c++
vdev_reopen(spa ->spa_root_vdev);
* Temporarily record the splitting vdevs in the spa config. This
* will disappear once the config is regenerated.
VERIFY(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_uint64_array(nvl, ZPOOL_CONFIG_SPLIT_LIST, glist, children) == 0);
kmem_free(glist, children * sizeof (uint64_t));
mutex_enter(&spa->spa_props_lock);
VERIFY(nvlist_add_nvlist(spa->spa_config, ZPOOL_CONFIG_SPLIT, nvl) == 0);
mutex_exit(&spa->spa_props_lock);
spa->spa_config_splitting = nvl;
vdev_config_dirty(spa->spa_root_vdev);
configure and create the new pool
VERIFY(nvlist_add_string(config, ZPOOL_CONFIG_POOL_NAME, newname) == 0);
VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_STATE,
exp ? POOL_STATE_EXPORTED : POOL_STATE_ACTIVE) == 0);
VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_VERSION, spa_version(spa)) == 0);
VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_TXG, spa->spa_config_txg) == 0);
VERIFY(nvlist_add_uint64(config, ZPOOL_CONFIG_POOL_GUID, spa_generate_guid(NULL)) == 0);
(void) nvlist_lookup_string(props, zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
add the new pool to the namespace
newspa = spa_add(newname, config, altroot);
newspa->spa_config_txg = spa->spa_config_txg;
spa_set_log_state(newspa, SPA_LOG_CLEAR);
release the spa config lock, retaining the namespace lock
spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
zio_injection_enabled ?
zio_handle_panic_ injection(spa, FTAG, 1);
spa_activate(newspa, spa_mode_global);
spa_async_suspend(newspa);
create the new pool from the disks of the original pool
error = spa_load(newspa, SPA_LOAD_IMPORT, SPA_IMPORT_ASSEMBLE, B_TRUE);
error ?
if that worked, generate a real config for the new pool
newspa->spa_root_vdev != NULL ?
VERIFY(nvlist_alloc(&newspa->spa_config_splitting, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_uint64(newspa->spa_config_splitting,
ZPOOL_CONFIG_SPLIT_GUID, spa_guid(spa)) == 0);
spa_config_set(newspa, spa_config_generate(newspa, NULL, -1ULL, B_TRUE));
set the props
props != NULL ?
spa_configfile_set(newspa, props, B_FALSE);
error = spa_prop_set(newspa, props);
error ?
flush everything
out: spa_unload(newspa);
spa_deactivate(newspa);
spa_remove(newspa);
txg = spa_vdev_config_enter(spa);
re-online all offlined disks
txg = spa_vdev_config_enter(newspa);
vdev_config_dirty(newspa->spa_root_vdev);
(void) spa_vdev_config_exit(newspa, NULL, txg, 0, FTAG);
zio_injection_enabled ?
zio_handle_panic_ injection(spa, FTAG, 2);
spa_async_resume(newspa);
c = 0
c < children ?
vml[c] != NULL ?
vdev_reopen(spa->spa_root_vdev);
nvlist_free(spa->spa_config_splitting);
spa->spa_config_splitting = NULL;
(void) spa_vdev_exit(spa, NULL, txg, error);
kmem_free(vml, children * sizeof (vdev_t *));
return (error);
vml[c]->vdev_offline = B_FALSE;
c++
finally, update the original pool's config
txg = spa_vdev_config_enter(spa);
tx = dmu_tx_create_dd(spa_get_dsl(spa)->dp_mos_dir);
error = dmu_tx_assign(tx, TXG_WAIT);
error != 0 ?
dmu_tx_abort(tx);
c = 0
c < children ?
vml[c] != NULL ?
vdev_config_dirty(spa->spa_root_vdev);
spa->spa_config_splitting = NULL;
nvlist_free(nvl);
vdev_split(vml[c]);
error == 0 ?
error == 0 ?
dmu_tx_commit(tx);
spa_history_log_internal (spa, "detach", tx, "vdev=%s", vml[c]->vdev_path);
vdev_free(vml[c]);
(void) spa_vdev_exit(spa, NULL, txg, 0);
c++
zio_injection_enabled ?
zio_handle_panic_ injection(spa, FTAG, 3);
split is complete; log a history record
spa_history_log_internal(newspa, "split", NULL, "from pool %s", spa_name(spa));
kmem_free(vml, children * sizeof (vdev_t *));
if we're not going to mount the filesystems in userland, export
exp ?
error = spa_export_common (newname, POOL_STATE_EXPORTED, NULL, B_FALSE, B_FALSE);
return (error);
Yes
No


static nvlist_t *
spa_nvlist_lookup_by_guid
(nvlist_t **nvpp, int
count, uint64_t
target_guid)
int i = 0
i < count ?
No
Yes
uint64_t guid;
VERIFY(nvlist_lookup_uint64(nvpp[i],
ZPOOL_CONFIG_GUID, &guid) == 0);
return (NULL);
guid == target_guid ?
Yes
No
return (nvpp[i]);
i++
End


static void spa_vdev_remove_aux (nvlist_t *config, char *name, nvlist_t **dev, int count, nvlist_t *dev_to_remove)
nvlist_t **newdev = NULL;
count > 1 ?
Yes
newdev = kmem_alloc((count - 1) * sizeof (void *), KM_SLEEP);
No
int i = 0, j = 0
i < count ?
Yes
dev[i] == dev_to_remove ?
No
VERIFY(nvlist_dup(dev[i], &newdev[j++], KM_SLEEP) == 0);
Yes
i++
No
VERIFY(nvlist_remove(config, name, DATA_TYPE_NVLIST_ARRAY) == 0);
VERIFY(nvlist_add_nvlist_array(config, name, newdev, count - 1) == 0);
int i = 0
i < count - 1 ?
Yes
nvlist_free(newdev[i]);
i++
No
count > 1 ?
Yes
kmem_free(newdev, (count - 1) * sizeof (void *));
No
End


static int
spa_vdev_remove_evacuate
(spa_t *spa, vdev_t *vd)
* Evacuate the device.
uint64_t txg;
int error = 0;
ASSERT(MUTEX_HELD(&spa_namespace_lock));
ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
ASSERT(vd == vd->vdev_top);
* Evacuate the device. We don't hold the config lock as writer
* since we need to do I/O but we do keep the
* spa_namespace_lock held. Once this completes the device
* should no longer have any blocks allocated on it.
vd->vdev_islog ?
Yes
No
vd->vdev_stat.vs_alloc
!= 0 ?
Yes
No
error = ENOTSUP;
error = spa_
offline_log(spa);
error ?
* The evacuation succeeded. Remove any remaining MOS metadata
* associated with this vdev, and wait for these changes to sync.
No
Yes
ASSERT0(vd->vdev_stat.vs_alloc);
txg = spa_vdev_config_enter(spa);
vd->vdev_removing = B_TRUE;
vdev_dirty(vd, 0, NULL, txg);
vdev_config_dirty(vd);
spa_vdev_config_exit(spa, NULL, txg, 0, FTAG);
return (0);
return (error);
End



```
static void
spa_vdev_remove_from_
namespace(spa_t
*spa, vdev_t *vd)
```

* Complete the removal
by cleaning up
the namespace.

```
vdev_t *rvd = spa->spa_root_vdev;
uint64_t id = vd->vdev_id;
boolean_t last_vdev = (id == (rvd->vdev_children - 1));
ASSERT(MUTEX_HELD(&spa_namespace_lock));
ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);
ASSERT(vd == vd->vdev_top);
```

* Only remove any
devices which are empty.

```
vd->vdev_stat.vs_alloc
!= 0 ?
```

```
(void) vdev_label_init
(vd, 0, VDEV_
LABEL_REMOVE);
```

```
list_link_active(&vd-
>vdev_state_dirty_node) ?
```

```
vdev_state_clean(vd);
```

```
list_link_active(&vd-
>vdev_config_dirty_node)
?
```

```
vdev_config_clean(vd);
```

```
vdev_free(vd);
```

```
last_vdev ?
```

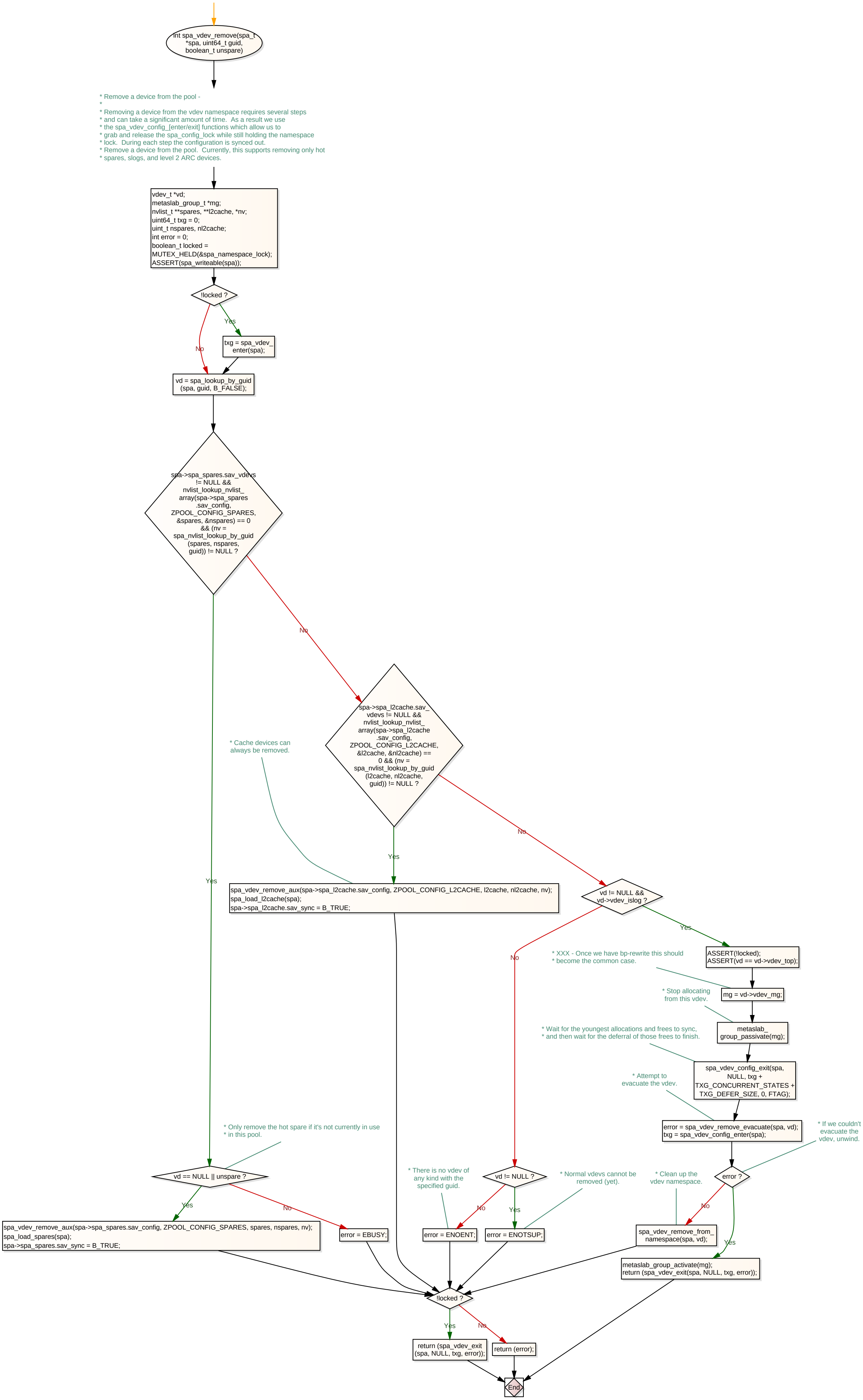
```
vdev_compact_
children(rvd);
```

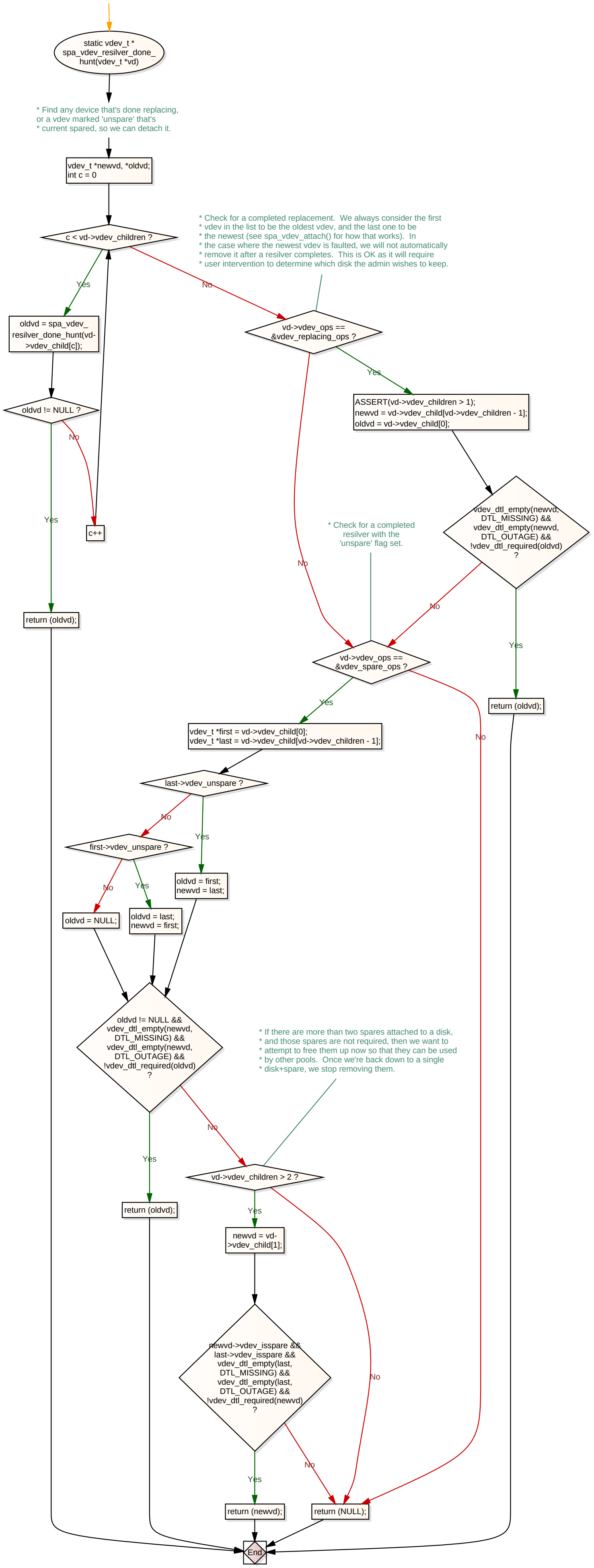
```
vd = vdev_alloc_common(spa, id, 0, &vdev_hole_ops);
vdev_add_child(rvd, vd);
```

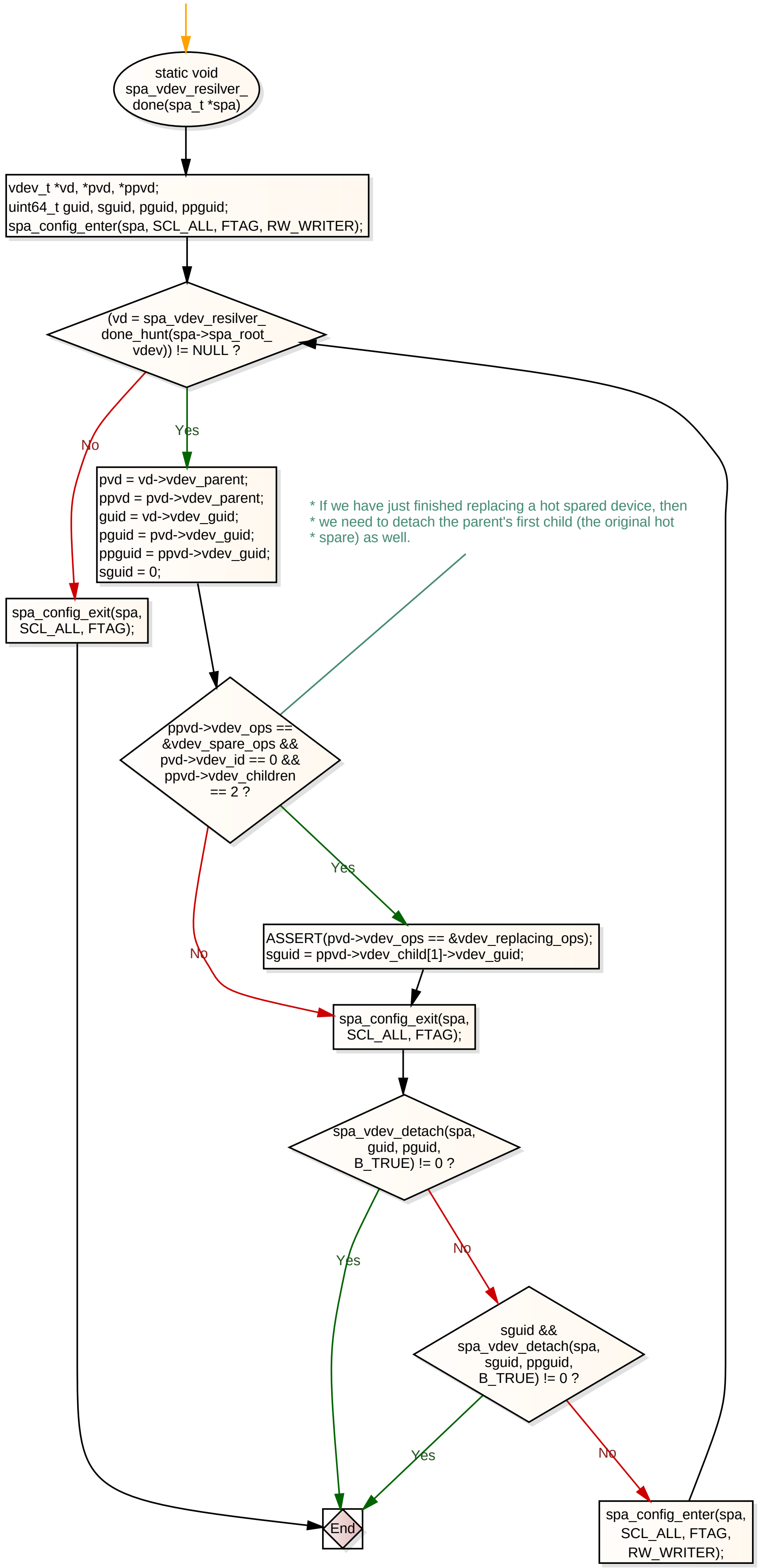
```
vdev_config_dirty(rvd);
```

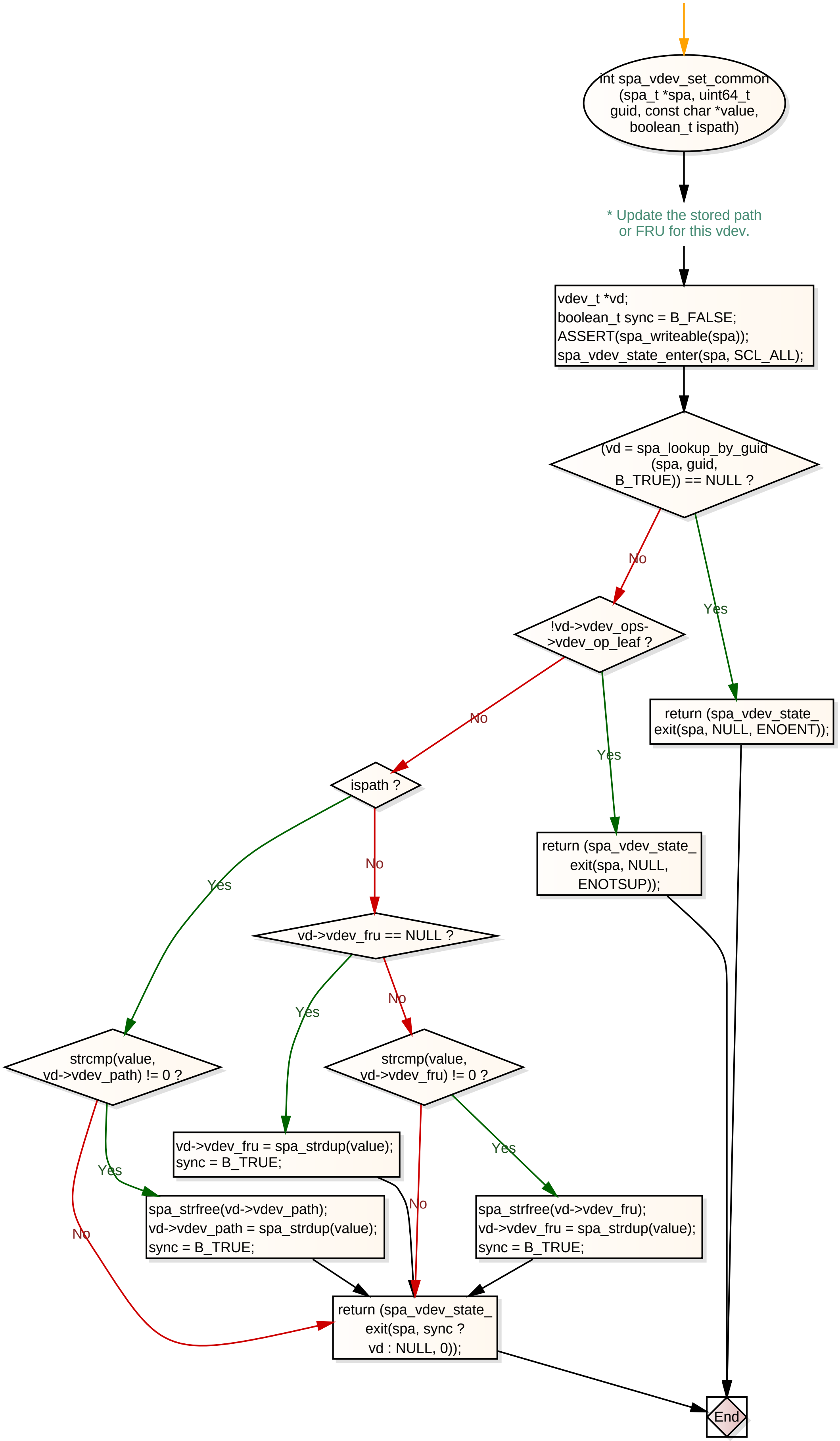
* Reassess the health of
our root vdev.

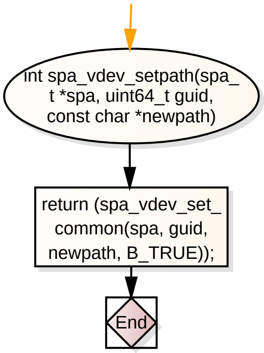
```
vdev_reopen(rvd);
```

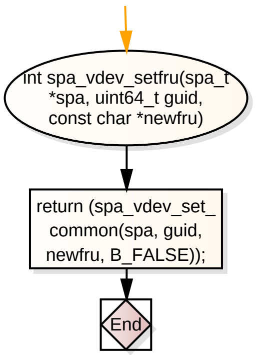

int spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare)
* Remove a device from the pool -
*
* Removing a device from the vdev namespace requires several steps
* and can take a significant amount of time. As a result we use
* the spa_vdev_config_[enter/exit] functions which allow us to
* grab and release the spa_config_lock while still holding the namespace
* lock. During each step the configuration is synced out.
* Remove a device from the pool. Currently, this supports removing only hot
* spares, slogs, and level 2 ARC devices.
vdev_t *vd;
metaslab_group_t *mg;
nvlist_t **spares, **l2cache, *nv;
uint64_t txg = 0;
uint_t nspares, nl2cache;
int error = 0;
boolean_t locked =
MUTEX_HELD(&spa_namespace_lock);
ASSERT(spa_writeable(spa));
!locked ?
Yes
No
txg = spa_vdev_enter(spa);
vd = spa_lookup_by_guid (spa, guid, B_FALSE);
spa->spa_spares.sav_vdevs != NULL && nvlist_lookup_nvlist_array(spa->spa_spares .sav_config, ZPOOL_CONFIG_SPARES, &spares, &nspares) == 0 && (nv = spa_nvlist_lookup_by_guid (spares, nspares, guid)) != NULL ?
No
spa->spa_l2cache.sav_vdevs != NULL && nvlist_lookup_nvlist_array(spa->spa_l2cache .sav_config, ZPOOL_CONFIG_L2CACHE, &l2cache, &nl2cache) == 0 && (nv = spa_nvlist_lookup_by_guid (l2cache, nl2cache, guid)) != NULL ?
* Cache devices can always be removed.
Yes
No
Yes
spa_vdev_remove_aux(spa->spa_l2cache.sav_config, ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache, nv);
spa_load_l2cache(spa);
spa->spa_l2cache.sav_sync = B_TRUE;
vd != NULL && vd->vdev_islog ?
Yes
No
ASSERT(!locked);
ASSERT(vd == vd->vdev_top);
* XXX - Once we have bp-rewrite this should
* become the common case.
mg = vd->vdev_mg;
* Stop allocating from this vdev.
metaslab_group_passivate(mg);
* Wait for the youngest allocations and frees to sync,
* and then wait for the deferral of those frees to finish.
spa_vdev_config_exit(spa, NULL, txg + TXG_CONCURRENT_STATES + TXG_DEFER_SIZE, 0, FTAG);
* Attempt to evacuate the vdev.
error = spa_vdev_remove_evacuate(spa, vd);
txg = spa_vdev_config_enter(spa);
* If we couldn't evacuate the vdev, unwind.
* Only remove the hot spare if it's not currently in use
* in this pool.
vd == NULL || unspare ?
Yes
No
* There is no vdev of any kind with the specified guid.
vd != NULL ?
No
Yes
* Normal vdevs cannot be removed (yet).
* Clean up the vdev namespace.
error ?
No
Yes
spa_vdev_remove_aux(spa->spa_spares.sav_config, ZPOOL_CONFIG_SPARES, spares, nspares, nv);
spa_load_spares(spa);
spa->spa_spares.sav_sync = B_TRUE;
error = EBUSY;
error = ENOENT;
error = ENOTSUP;
spa_vdev_remove_from_namespace(spa, vd);
metaslab_group_activate(mg);
return (spa_vdev_exit(spa, NULL, txg, error));
!locked ?
Yes
No
return (spa_vdev_exit (spa, NULL, txg, error));
return (error);
End

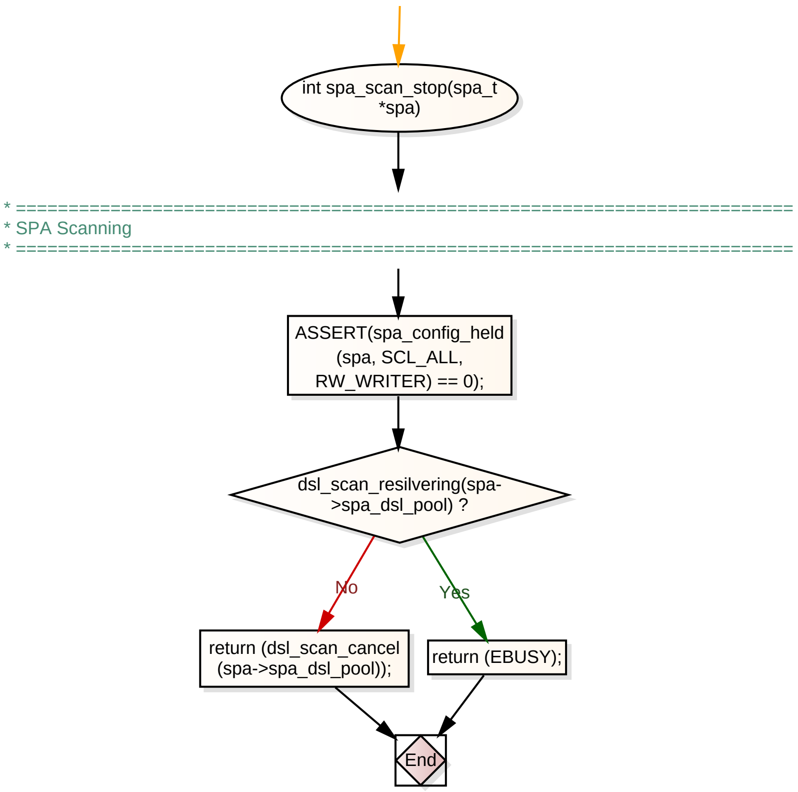

static vdev_t * spa_vdev_resilver_done_ hunt(vdev_t *vd)
* Find any device that's done replacing,
or a vdev marked 'unspare' that's
* current spared, so we can detach it.
vdev_t *newvd, *oldvd;
int c = 0
c < vd->vdev_children ?
Yes
No
* Check for a completed replacement. We always consider the first
* vdev in the list to be the oldest vdev, and the last one to be
* the newest (see spa_vdev_attach() for how that works). In
* the case where the newest vdev is faulted, we will not automatically
* remove it after a resilver completes. This is OK as it will require
* user intervention to determine which disk the admin wishes to keep.
oldvd = spa_vdev_ resilver_done_hunt(vd- >vdev_child[c]);
vd->vdev_ops == &vdev_replacing_ops ?
Yes
ASSERT(vd->vdev_children > 1);
newvd = vd->vdev_child[vd->vdev_children - 1];
oldvd = vd->vdev_child[0];
oldvd != NULL ?
No
c++
Yes
vdev_dtl_empty(newvd, DTL_MISSING) && vdev_dtl_empty(newvd, DTL_OUTAGE) && !vdev_dtl_required(oldvd) ?
* Check for a completed resilver with the 'unspare' flag set.
No
No
return (oldvd);
vd->vdev_ops == &vdev_spare_ops ?
Yes
Yes
return (oldvd);
vdev_t *first = vd->vdev_child[0];
vdev_t *last = vd->vdev_child[vd->vdev_children - 1];
No
last->vdev_unspare ?
No
first->vdev_unspare ?
Yes
oldvd = first;
newvd = last;
No
Yes
oldvd = NULL;
oldvd = last;
newvd = first;
oldvd != NULL && vdev_dtl_empty(newvd, DTL_MISSING) && vdev_dtl_empty(newvd, DTL_OUTAGE) && !vdev_dtl_required(oldvd) ?
* If there are more than two spares attached to a disk,
* and those spares are not required, then we want to
* attempt to free them up now so that they can be used
* by other pools. Once we're back down to a single
* disk+spare, we stop removing them.
No
Yes
vd->vdev_children > 2 ?
return (oldvd);
Yes
newvd = vd- >vdev_child[1];
newvd->vdev_isspare && last->vdev_isspare && vdev_dtl_empty(last, DTL_MISSING) && vdev_dtl_empty(last, DTL_OUTAGE) && !vdev_dtl_required(newvd) ?
No
No
Yes
return (newvd);
return (NULL);
End

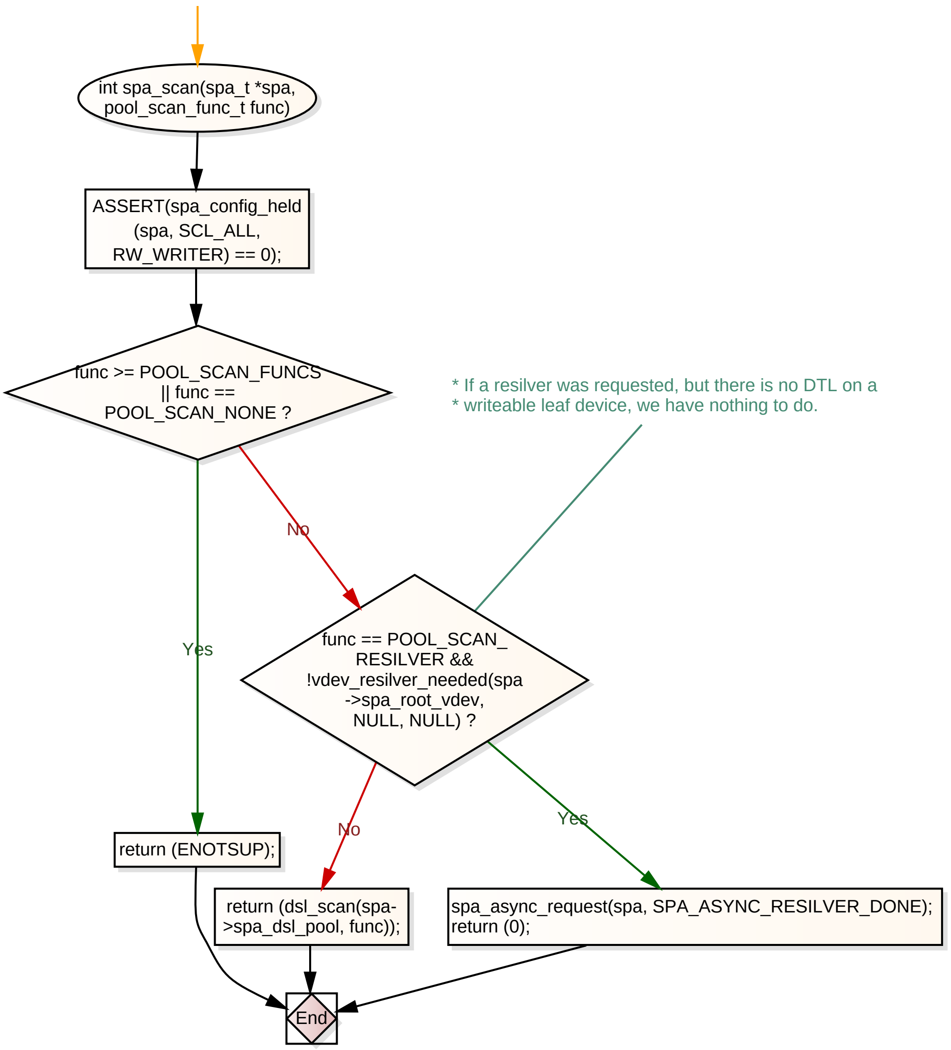

static void spa_vdev_resilver_done(spa_t *spa)
vdev_t *vd, *pvd, *ppvd;
uint64_t guid, sguid, pguid, ppguid;
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
(vd = spa_vdev_resilver_done_hunt(spa->spa_root_vdev)) != NULL ?
Yes
No
pvd = vd->vdev_parent;
ppvd = pvd->vdev_parent;
guid = vd->vdev_guid;
pguid = pvd->vdev_guid;
ppguid = ppvd->vdev_guid;
sguid = 0;
* If we have just finished replacing a hot spared device, then
* we need to detach the parent's first child (the original hot
* spare) as well.
spa_config_exit(spa, SCL_ALL, FTAG);
ppvd->vdev_ops == &vdev_spare_ops && pvd->vdev_id == 0 && ppvd->vdev_children == 2 ?
Yes
No
ASSERT(pvd->vdev_ops == &vdev_replacing_ops);
sguid = ppvd->vdev_child[1]->vdev_guid;
spa_config_exit(spa, SCL_ALL, FTAG);
spa_vdev_detach(spa, guid, pguid, B_TRUE) != 0 ?
Yes
No
sguid && spa_vdev_detach(spa, sguid, ppguid, B_TRUE) != 0 ?
Yes
No
End
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

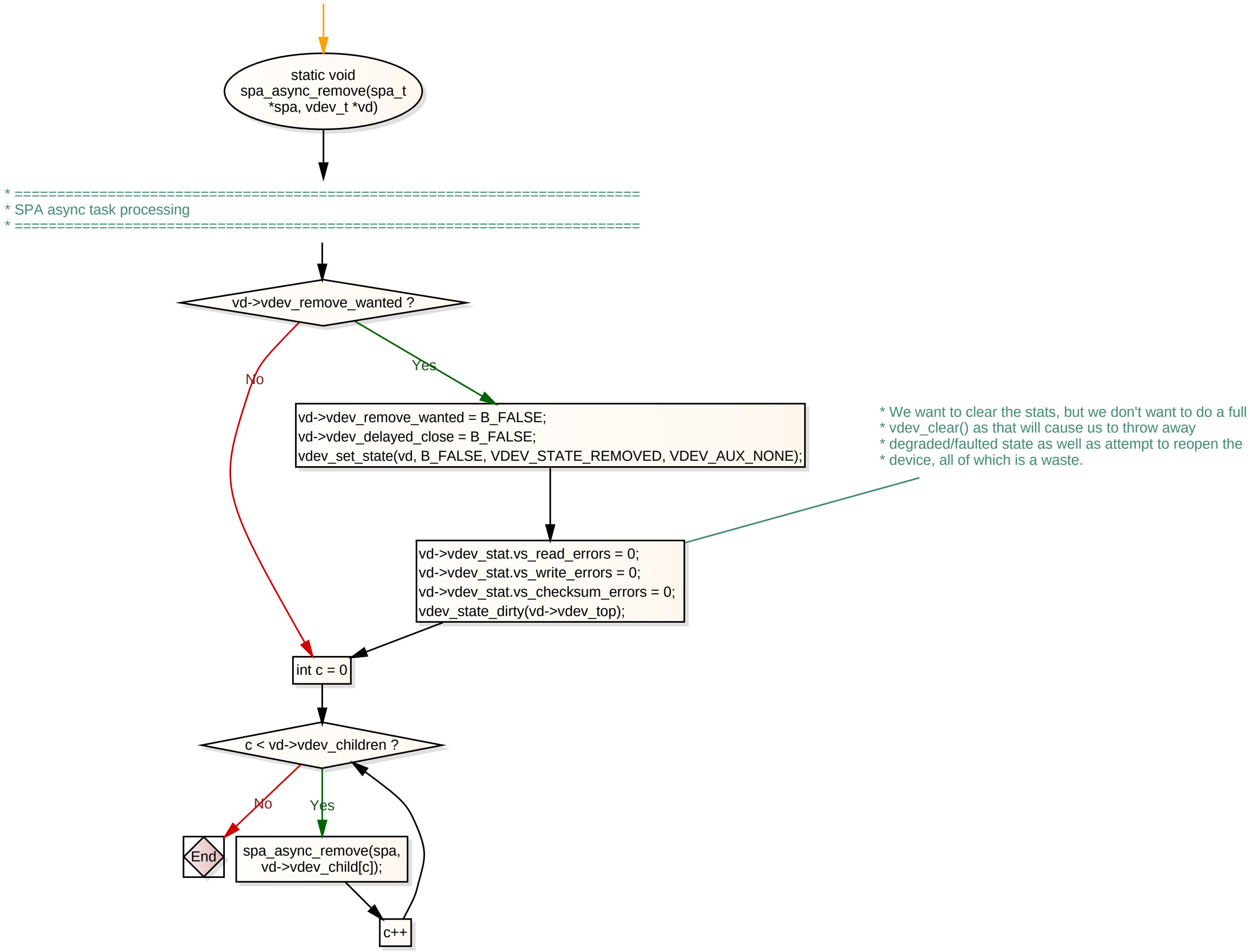

int spa_vdev_set_common (spa_t *spa, uint64_t guid, const char *value, boolean_t ispath)
* Update the stored path or FRU for this vdev.
vdev_t *vd;
boolean_t sync = B_FALSE;
ASSERT(spa_writeable(spa));
spa_vdev_state_enter(spa, SCL_ALL);
(vd = spa_lookup_by_guid (spa, guid, B_TRUE)) == NULL ?
No
Yes
!vd->vdev_ops- >vdev_op_leaf ?
return (spa_vdev_state_ exit(spa, NULL, ENOENT));
No
Yes
ispath ?
return (spa_vdev_state_ exit(spa, NULL, ENOTSUP));
Yes
No
vd->vdev_fru == NULL ?
Yes
No
strcmp(value, vd->vdev_path) != 0 ?
strcmp(value, vd->vdev_fru) != 0 ?
vd->vdev_fru = spa_strdup(value);
sync = B_TRUE;
Yes
Yes
No
No
spa_strfree(vd->vdev_path);
vd->vdev_path = spa_strdup(value);
sync = B_TRUE;
spa_strfree(vd->vdev_fru);
vd->vdev_fru = spa_strdup(value);
sync = B_TRUE;
return (spa_vdev_state_ exit(spa, sync ? vd : NULL, 0));
End

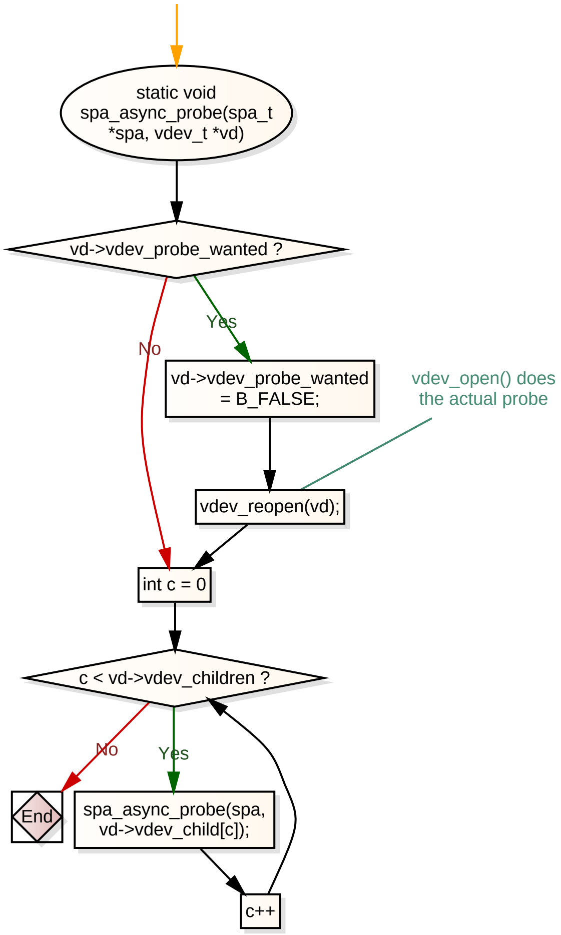

int spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath)
return (spa_vdev_set_common(spa, guid, newpath, B_TRUE));
End

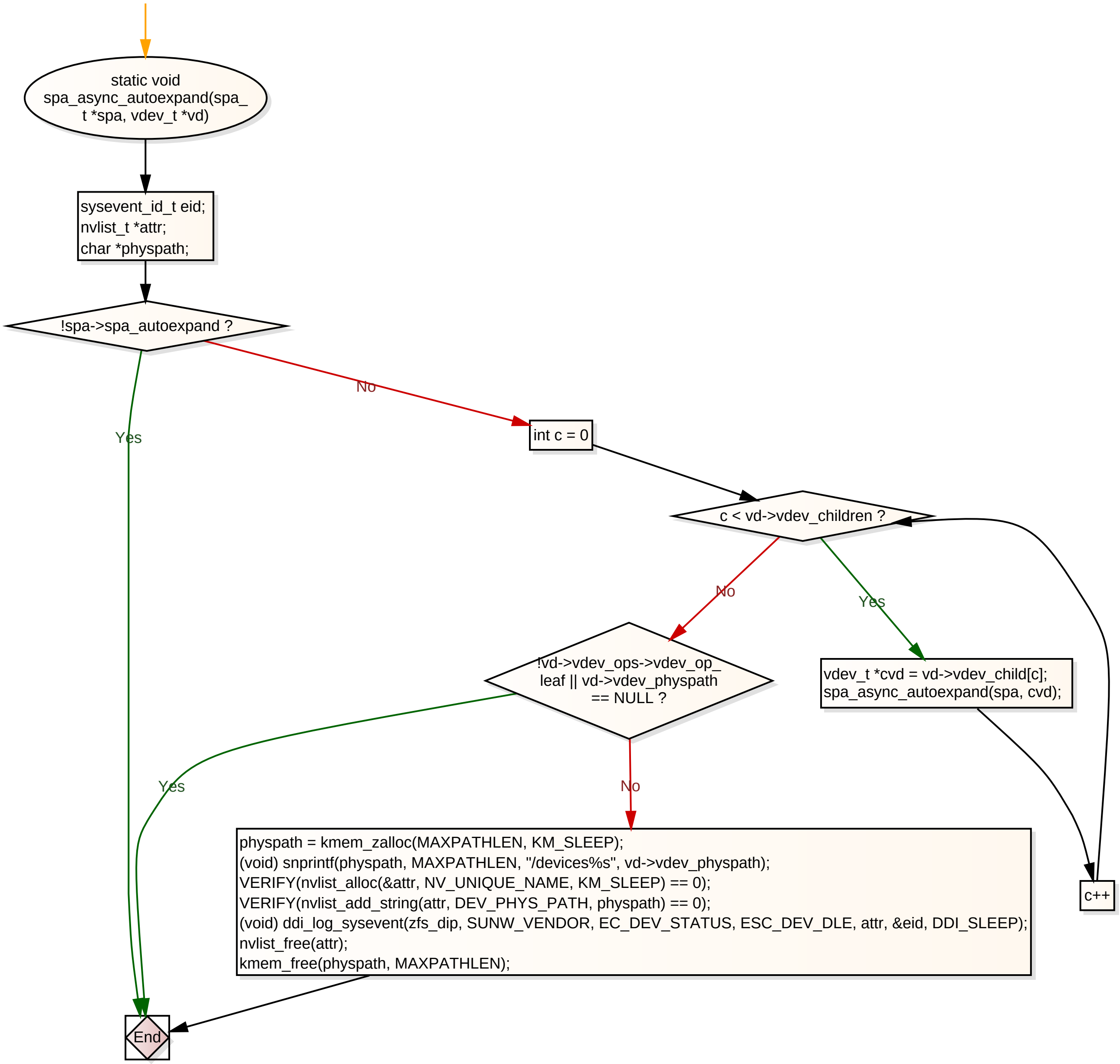

int spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru)
return (spa_vdev_set_common(spa, guid, newfru, B_FALSE));
End

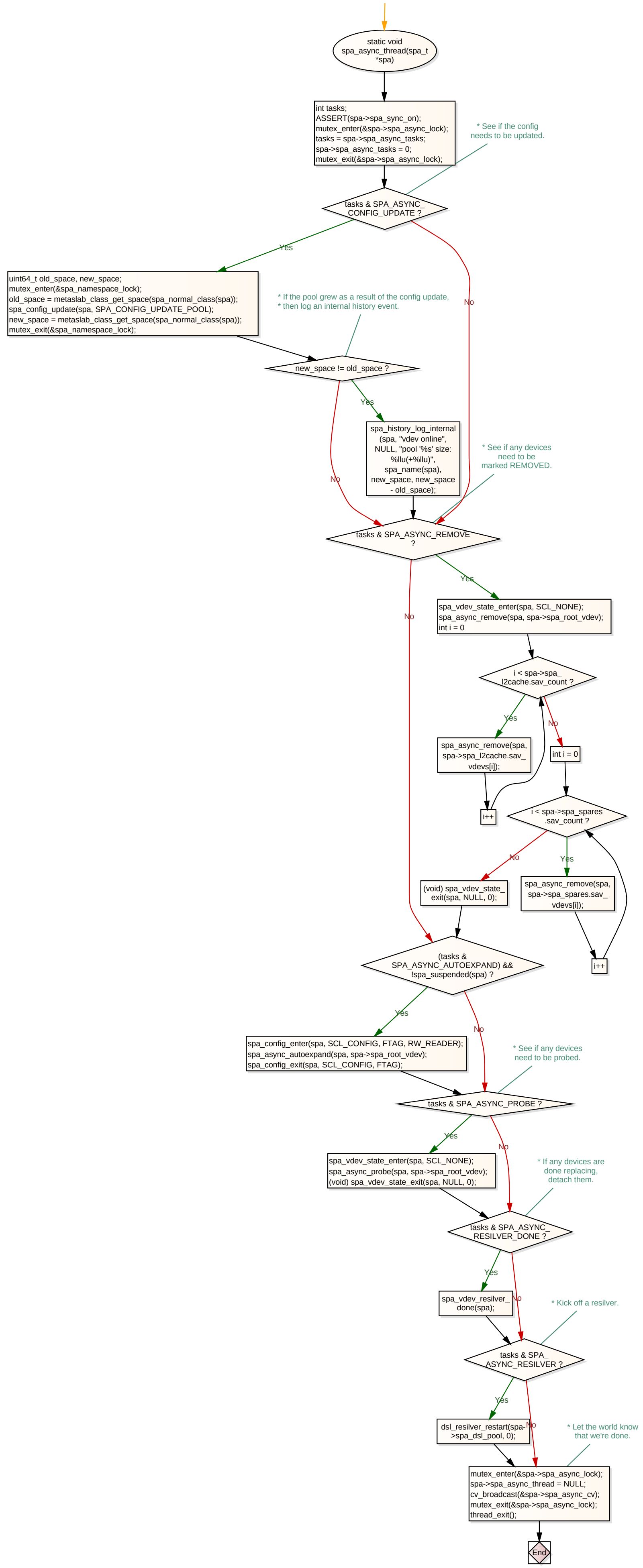

int spa_scan_stop(spa_t *spa)
* ==
* SPA Scanning
* ==
ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == 0);
dsl_scan_resilvering(spa->spa_dsl_pool) ?
No
Yes
return (dsl_scan_cancel(spa->spa_dsl_pool));
return (EBUSY);
End

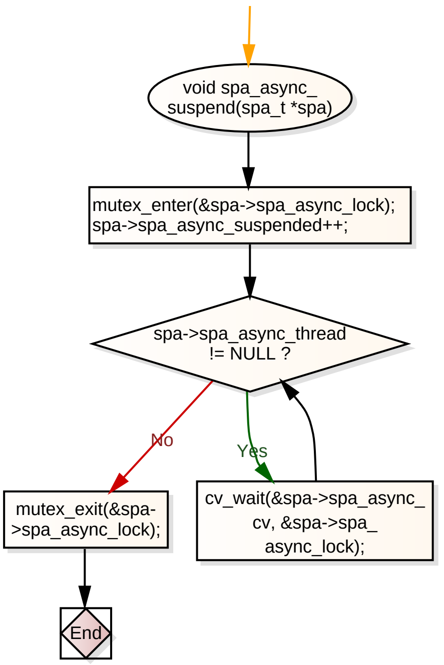

int spa_scan(spa_t *spa, pool_scan_func_t func)
ASSERT(spa_config_held (spa, SCL_ALL, RW_WRITER) == 0);
func >= POOL_SCAN_FUNCS || func == POOL_SCAN_NONE ?
* If a resilver was requested, but there is no DTL on a
* writeable leaf device, we have nothing to do.
No
Yes
func == POOL_SCAN_ RESILVER && !vdev_resilver_needed(spa ->spa_root_vdev, NULL, NULL) ?
No
Yes
return (ENOTSUP);
return (dsl_scan(spa- >spa_dsl_pool, func));
spa_async_request(spa, SPA_ASYNC_RESILVER_DONE);
return (0);
End

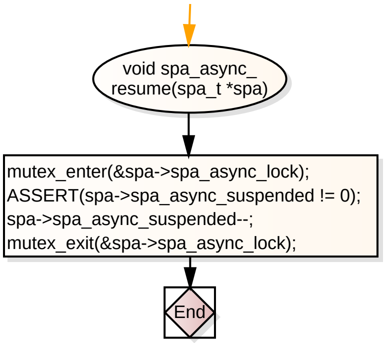

static void spa_async_remove(spa_t *spa, vdev_t *vd)
* ==
* SPA async task processing
* ==
vd->vdev_remove_wanted ?
No
Yes
vd->vdev_remove_wanted = B_FALSE;
vd->vdev_delayed_close = B_FALSE;
vdev_set_state(vd, B_FALSE, VDEV_STATE_REMOVED, VDEV_AUX_NONE);
* We want to clear the stats, but we don't want to do a full
* vdev_clear() as that will cause us to throw away
* degraded/faulted state as well as attempt to reopen the
* device, all of which is a waste.
vd->vdev_stat.vs_read_errors = 0;
vd->vdev_stat.vs_write_errors = 0;
vd->vdev_stat.vs_checksum_errors = 0;
vdev_state_dirty(vd->vdev_top);
int c = 0
c < vd->vdev_children ?
No
Yes
End
spa_async_remove(spa, vd->vdev_child[c]);
c++

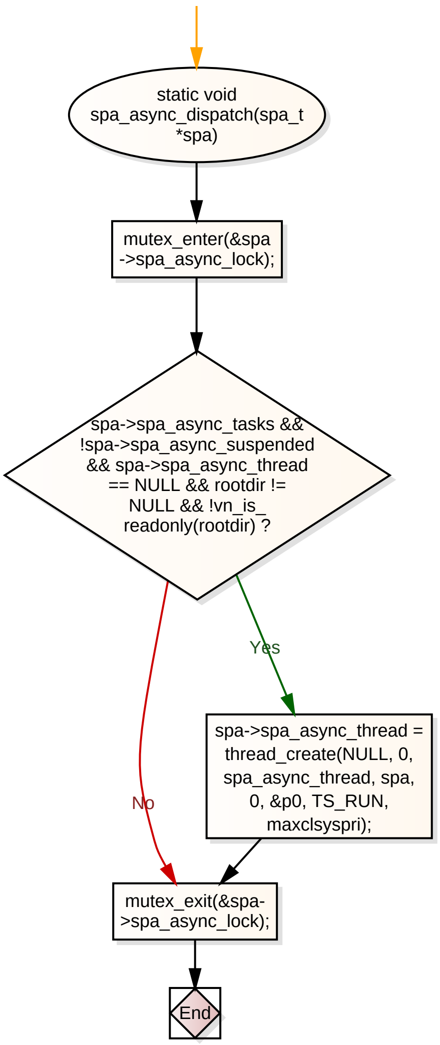

static void spa_async_probe(spa_t *spa, vdev_t *vd)
vd->vdev_probe_wanted ?
No
Yes
vd->vdev_probe_wanted = B_FALSE;
vdev_open() does the actual probe
vdev_reopen(vd);
int c = 0
c < vd->vdev_children ?
No
Yes
End
spa_async_probe(spa, vd->vdev_child[c]);
c++

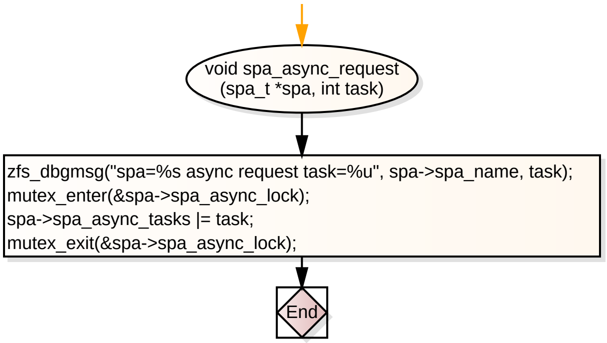

static void spa_async_autoexpand(spa_t *spa, vdev_t *vd)
sysevent_id_t eid;
nvlist_t *attr;
char *physpath;
!spa->spa_autoexpand ?
Yes
No
int c = 0
c < vd->vdev_children ?
No
Yes
!vd->vdev_ops->vdev_op_leaf || vd->vdev_physpath == NULL ?
vdev_t *cvd = vd->vdev_child[c];
spa_async_autoexpand(spa, cvd);
Yes
No
physpath = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
(void) snprintf(physpath, MAXPATHLEN, "/devices%s", vd->vdev_physpath);
VERIFY(nvlist_alloc(&attr, NV_UNIQUE_NAME, KM_SLEEP) == 0);
VERIFY(nvlist_add_string(attr, DEV_PHYS_PATH, physpath) == 0);
(void) ddi_log_sysevent(zfs_dip, SUNW_VENDOR, EC_DEV_STATUS, ESC_DEV_DLE, attr, &eid, DDI_SLEEP);
nvlist_free(attr);
kmem_free(physpath, MAXPATHLEN);
c++
End


static void
spa_async_thread(spa_t
*spa)
int tasks;
ASSERT(spa->spa_sync_on);
mutex_enter(&spa->spa_async_lock);
tasks = spa->spa_async_tasks;
spa->spa_async_tasks = 0;
mutex_exit(&spa->spa_async_lock);
* See if the config
needs to be updated.
tasks & SPA_ASYNC_
CONFIG_UPDATE ?
Yes
No
uint64_t old_space, new_space;
mutex_enter(&spa_namespace_lock);
old_space = metaslab_class_get_space(spa_normal_class(spa));
spa_config_update(spa, SPA_CONFIG_UPDATE_POOL);
new_space = metaslab_class_get_space(spa_normal_class(spa));
mutex_exit(&spa_namespace_lock);
* If the pool grew as a result of the config update,
* then log an internal history event.
new_space != old_space ?
Yes
No
spa_history_log_internal
(spa, "vdev online",
NULL, "pool '%s' size:
%llu(+%llu)",
spa_name(spa),
new_space, new_space
- old_space);
* See if any devices
need to be
marked REMOVED.
tasks & SPA_ASYNC_REMOVE
?
Yes
No
spa_vdev_state_enter(spa, SCL_NONE);
spa_async_remove(spa, spa->spa_root_vdev);
int i = 0
i < spa->spa_
l2cache.sav_count ?
Yes
No
spa_async_remove(spa,
spa->spa_l2cache.sav_
vdevs[i]);
i++
int i = 0
i < spa->spa_spares
.sav_count ?
No
Yes
(void) spa_vdev_state_
exit(spa, NULL, 0);
spa_async_remove(spa,
spa->spa_spares.sav_
vdevs[i]);
i++
(tasks &
SPA_ASYNC_AUTOEXPAND) &&
!spa_suspended(spa) ?
Yes
No
spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
spa_async_autoexpand(spa, spa->spa_root_vdev);
spa_config_exit(spa, SCL_CONFIG, FTAG);
* See if any devices
need to be probed.
tasks & SPA_ASYNC_PROBE ?
Yes
No
spa_vdev_state_enter(spa, SCL_NONE);
spa_async_probe(spa, spa->spa_root_vdev);
(void) spa_vdev_state_exit(spa, NULL, 0);
* If any devices are
done replacing,
detach them.
tasks & SPA_ASYNC_
RESILVER_DONE ?
Yes
No
spa_vdev_resilver_
done(spa);
* Kick off a resilver.
tasks & SPA_
ASYNC_RESILVER ?
Yes
No
dsl_resilver_restart(spa-
>spa_dsl_pool, 0);
* Let the world know
that we're done.
mutex_enter(&spa->spa_async_lock);
spa->spa_async_thread = NULL;
cv_broadcast(&spa->spa_async_cv);
mutex_exit(&spa->spa_async_lock);
thread_exit();
End

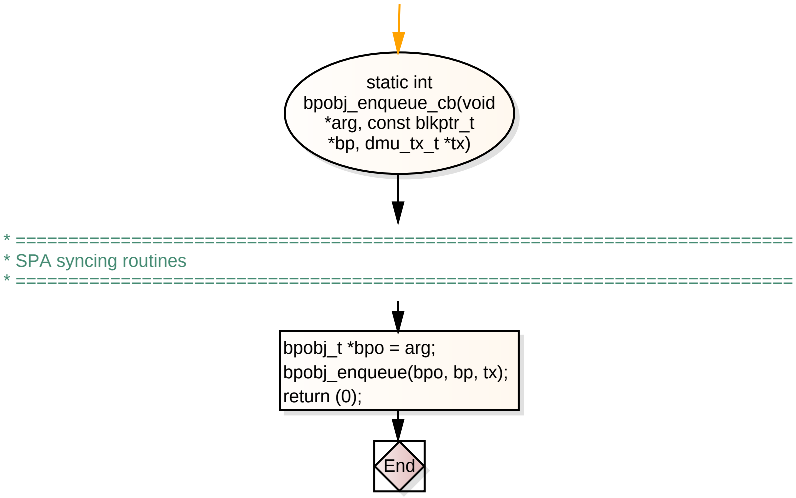

void spa_async_
suspend(spa_t *spa)
mutex_enter(&spa->spa_async_lock);
spa->spa_async_suspended++;
spa->spa_async_thread
!= NULL ?
No
Yes
mutex_exit(&spa-
>spa_async_lock);
cv_wait(&spa->spa_async_
cv, &spa->spa_
async_lock);
End


void spa_async_
resume(spa_t *spa)
mutex_enter(&spa->spa_async_lock);
ASSERT(spa->spa_async_suspended != 0);
spa->spa_async_suspended--;
mutex_exit(&spa->spa_async_lock);
End


static void spa_async_dispatch(spa_t *spa)
mutex_enter(&spa ->spa_async_lock);
spa->spa_async_tasks && !spa->spa_async_suspended && spa->spa_async_thread == NULL && rootdir != NULL && !vn_is_ readonly(rootdir) ?
Yes
spa->spa_async_thread = thread_create(NULL, 0, spa_async_thread, spa, 0, &p0, TS_RUN, maxclsyspri);
No
mutex_exit(&spa- >spa_async_lock);
End



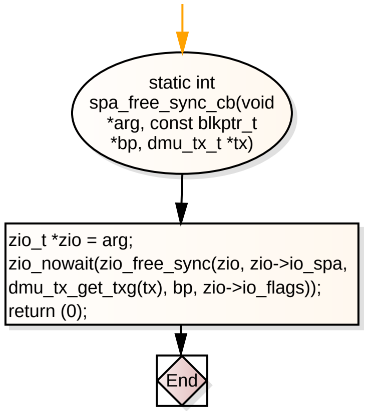
```
zfs_dbgmsg("spa=%s async request task=%u", spa->spa_name, task);
mutex_enter(&spa->spa_async_lock);
spa->spa_async_tasks |= task;
mutex_exit(&spa->spa_async_lock);
```



static int bpobj_enqueue_cb(void *arg, const blkptr_t *bp, dmu_tx_t *tx)
End

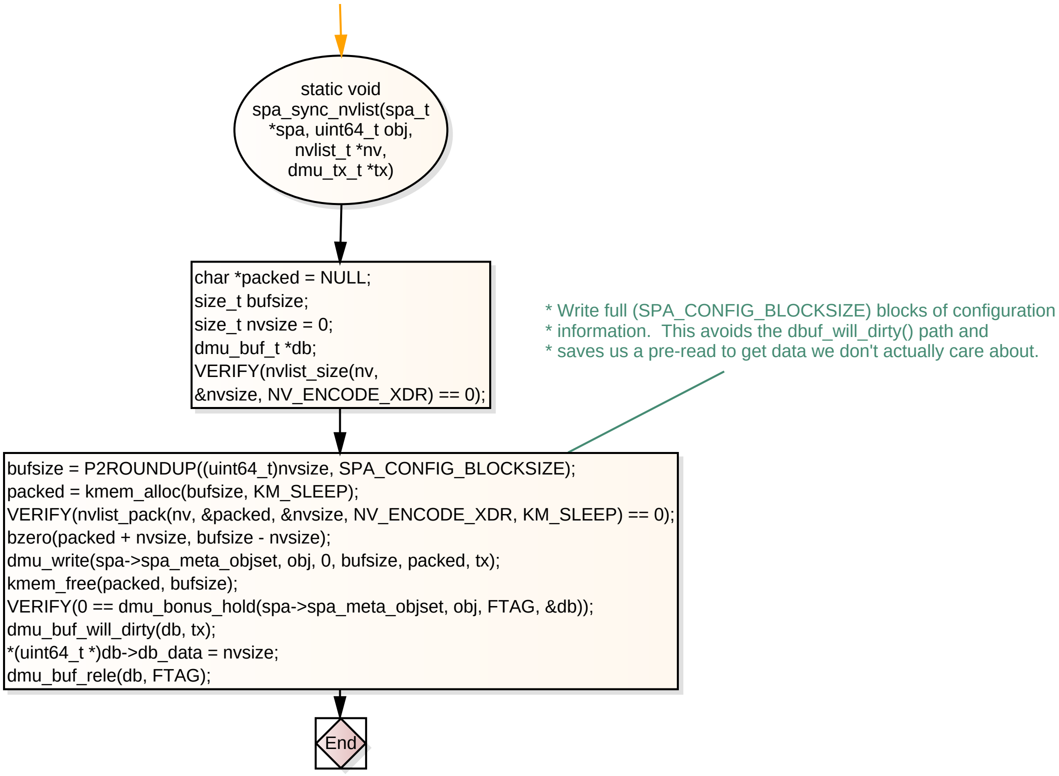
```
* ========================================================================
* SPA syncing routines
* ========================================================================
```

```
bpobj_t *bpo = arg;
bpobj_enqueue(bpo, bp, tx);
return (0);
```



```
static int
spa_free_sync_cb(void
*arg, const blkptr_t
*bp, dmu_tx_t *tx)
```

```
zio_t *zio = arg;
zio_nowait(zio_free_sync(zio, zio->io_spa,
dmu_tx_get_txg(tx), bp, zio->io_flags));
return (0);
```

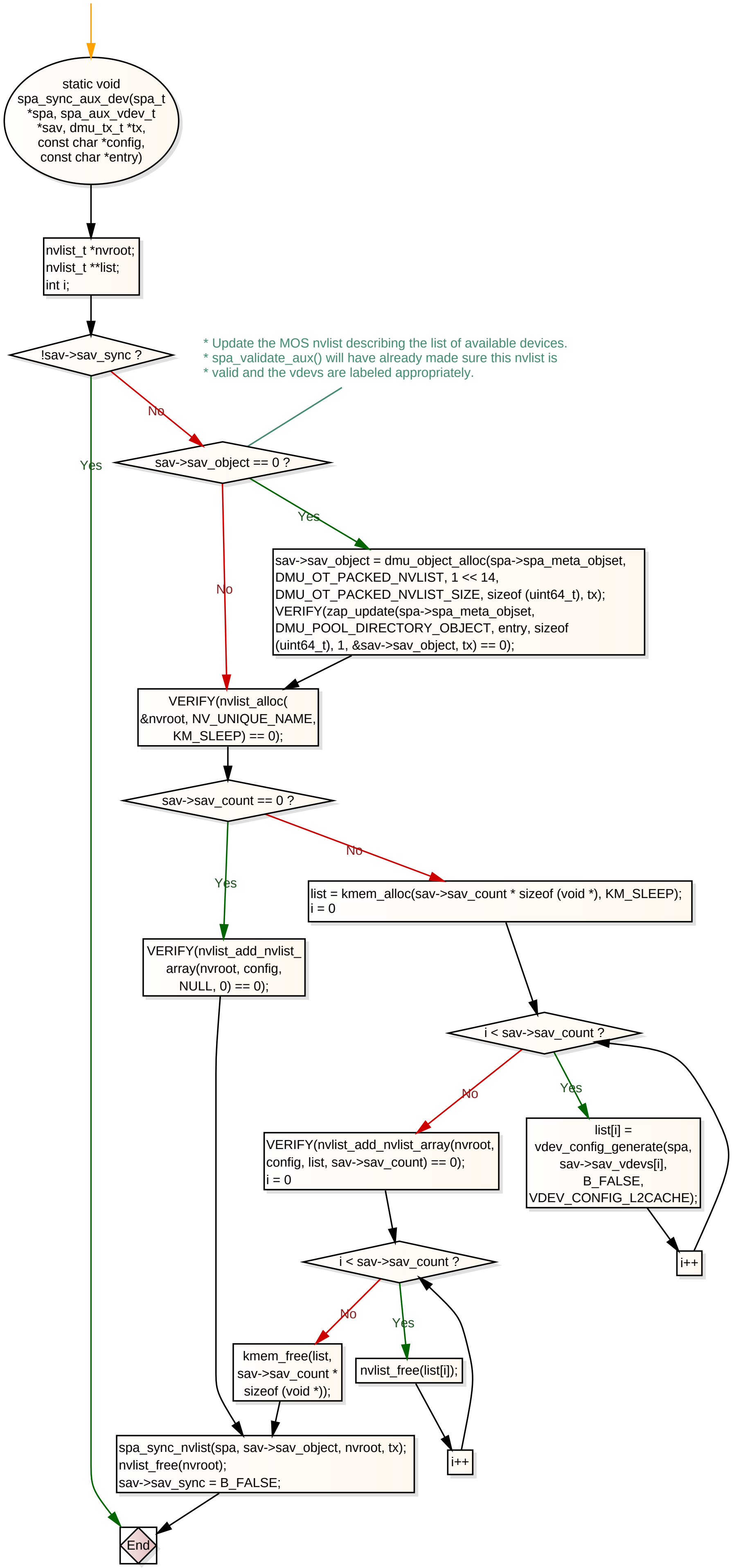


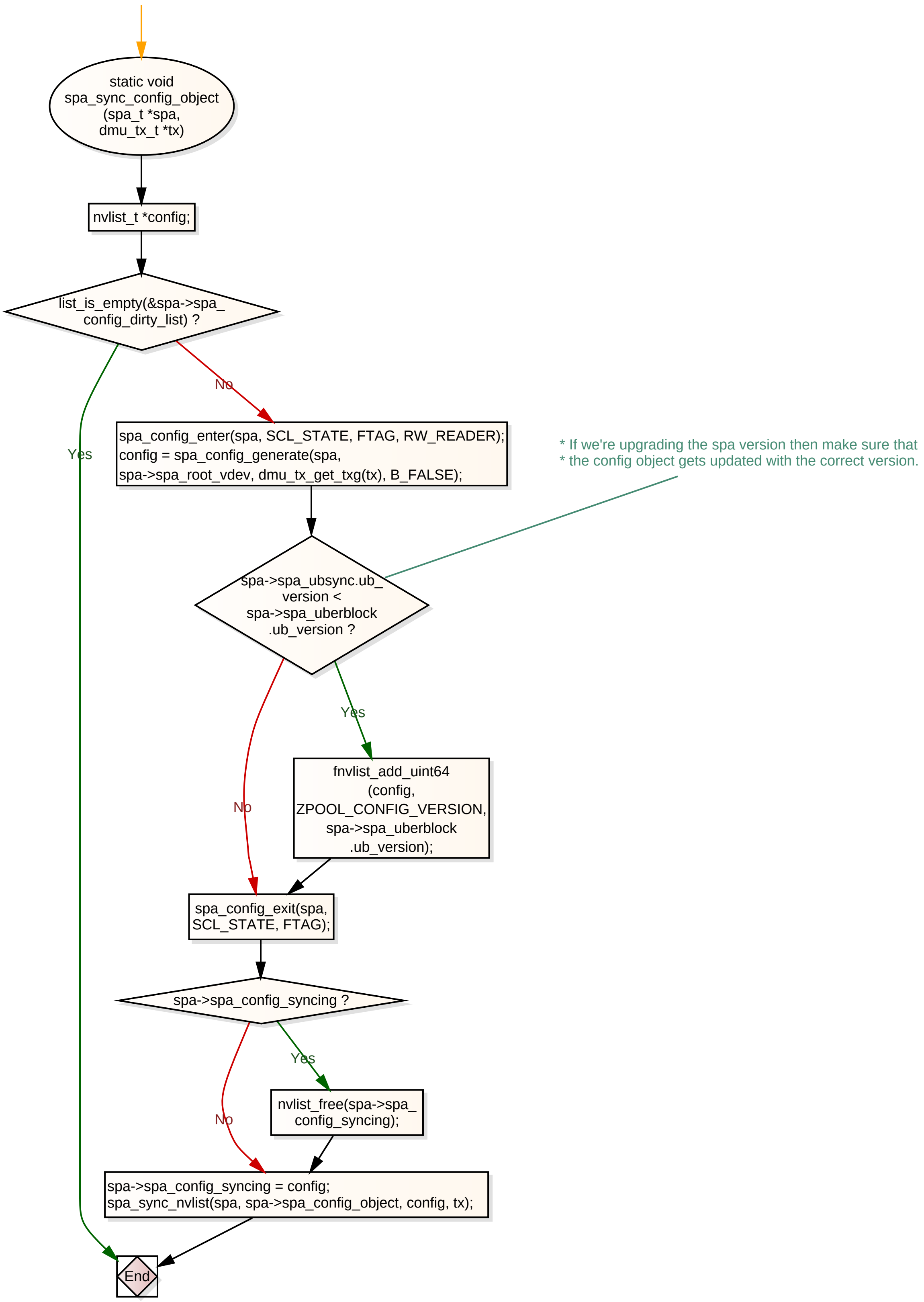
```
static void
spa_sync_nvlist(spa_t
*spa, uint64_t obj,
nvlist_t *nv,
dmu_tx_t *tx)
```

```
char *packed = NULL;
size_t bufsize;
size_t nvsize = 0;
dmu_buf_t *db;
VERIFY(nvlist_size(nv,
&nvsize, NV_ENCODE_XDR) == 0);
```

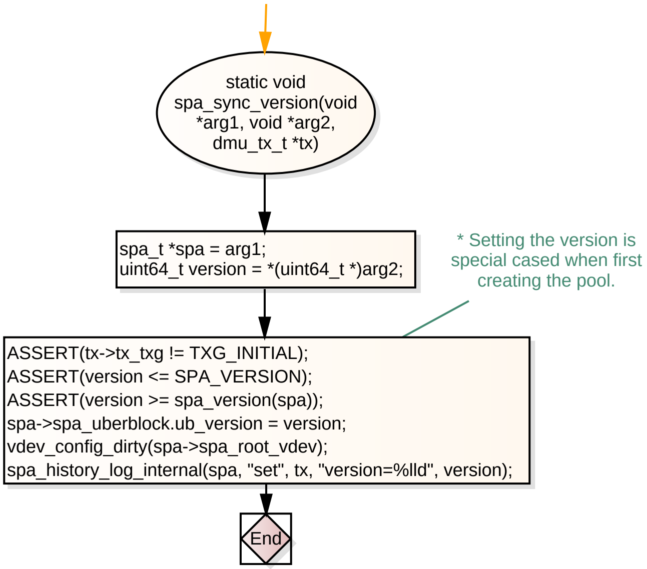
```
* Write full (SPA_CONFIG_BLOCKSIZE) blocks of configuration
* information.  This avoids the dbuf_will_dirty() path and
* saves us a pre-read to get data we don't actually care about.
```

```
bufsize = P2ROUNDUP((uint64_t)nvsize, SPA_CONFIG_BLOCKSIZE);
packed = kmem_alloc(bufsize, KM_SLEEP);
VERIFY(nvlist_pack(nv, &packed, &nvsize, NV_ENCODE_XDR, KM_SLEEP) == 0);
bzero(packed + nvsize, bufsize - nvsize);
dmu_write(spa->spa_meta_objset, obj, 0, bufsize, packed, tx);
kmem_free(packed, bufsize);
VERIFY(0 == dmu_bonus_hold(spa->spa_meta_objset, obj, FTAG, &db));
dmu_buf_will_dirty(db, tx);
*(uint64_t *)db->db_data = nvsize;
dmu_buf_rele(db, FTAG);
```


static void spa_sync_aux_dev(spa_t *spa, spa_aux_vdev_t *sav, dmu_tx_t *tx, const char *config, const char *entry)
nvlist_t *nvroot;
nvlist_t **list;
int i;
!sav->sav_sync ?
* Update the MOS nvlist describing the list of available devices.
* spa_validate_aux() will have already made sure this nvlist is
* valid and the vdevs are labeled appropriately.
No
Yes
sav->sav_object == 0 ?
Yes
No
sav->sav_object = dmu_object_alloc(spa->spa_meta_objset,
DMU_OT_PACKED_NVLIST, 1 << 14,
DMU_OT_PACKED_NVLIST_SIZE, sizeof (uint64_t), tx);
VERIFY(zap_update(spa->spa_meta_objset,
DMU_POOL_DIRECTORY_OBJECT, entry, sizeof
(uint64_t), 1, &sav->sav_object, tx) == 0);
VERIFY(nvlist_alloc(&nvroot, NV_UNIQUE_NAME, KM_SLEEP) == 0);
sav->sav_count == 0 ?
No
Yes
list = kmem_alloc(sav->sav_count * sizeof (void *), KM_SLEEP);
i = 0
VERIFY(nvlist_add_nvlist_array(nvroot, config, NULL, 0) == 0);
i < sav->sav_count ?
No
Yes
VERIFY(nvlist_add_nvlist_array(nvroot,
config, list, sav->sav_count) == 0);
i = 0
list[i] = vdev_config_generate(spa, sav->sav_vdevs[i], B_FALSE, VDEV_CONFIG_L2CACHE);
i++
i < sav->sav_count ?
No
Yes
kmem_free(list, sav->sav_count * sizeof (void *));
nvlist_free(list[i]);
i++
spa_sync_nvlist(spa, sav->sav_object, nvroot, tx);
nvlist_free(nvroot);
sav->sav_sync = B_FALSE;
End



* If we're upgrading the spa version then make sure that
* the config object gets updated with the correct version.

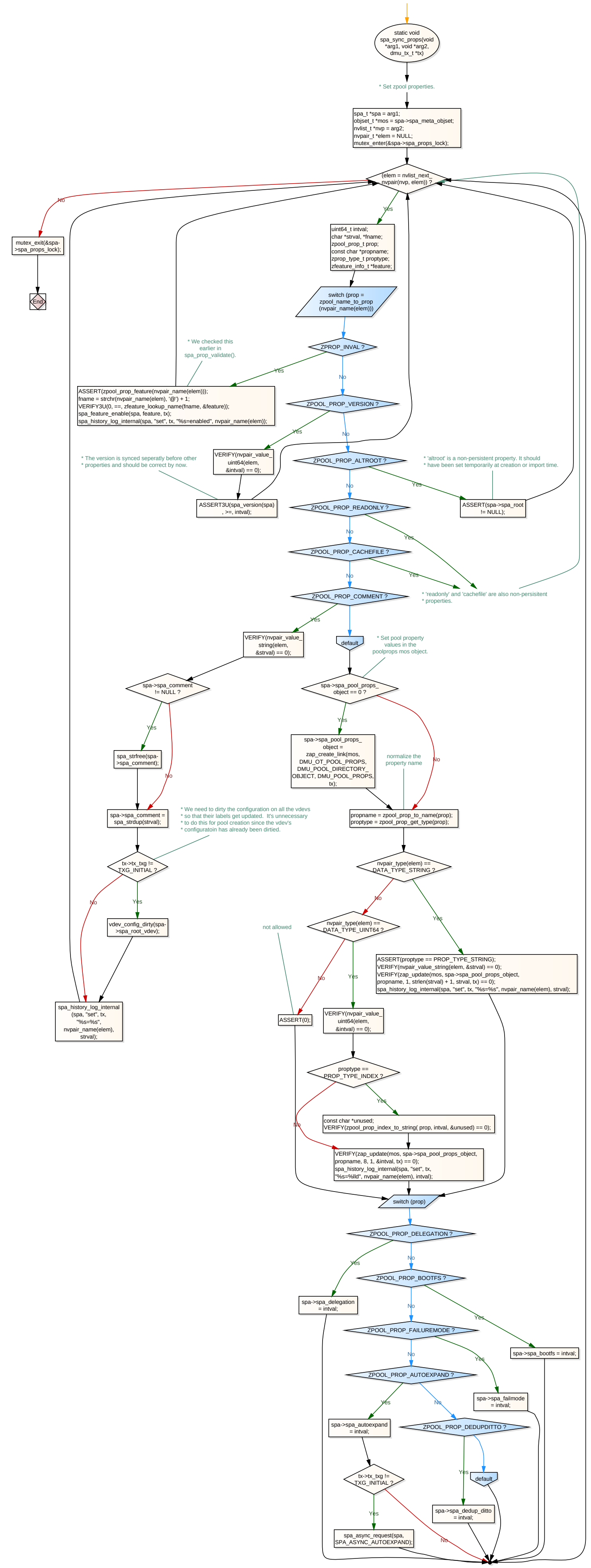


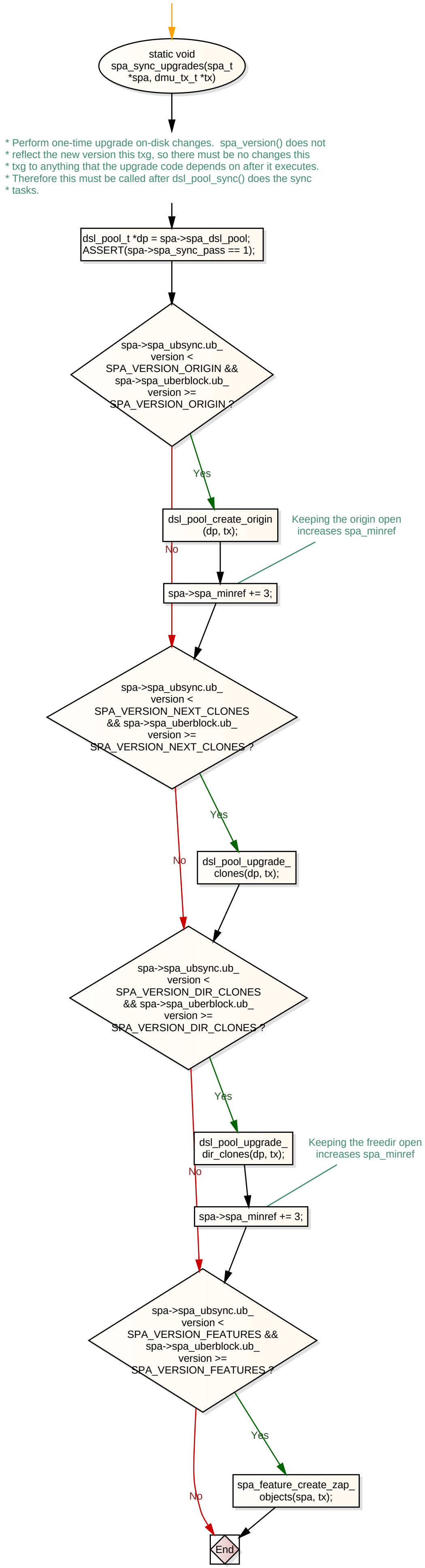
static void spa_sync_version(void *arg1, void *arg2, dmu_tx_t *tx)
End

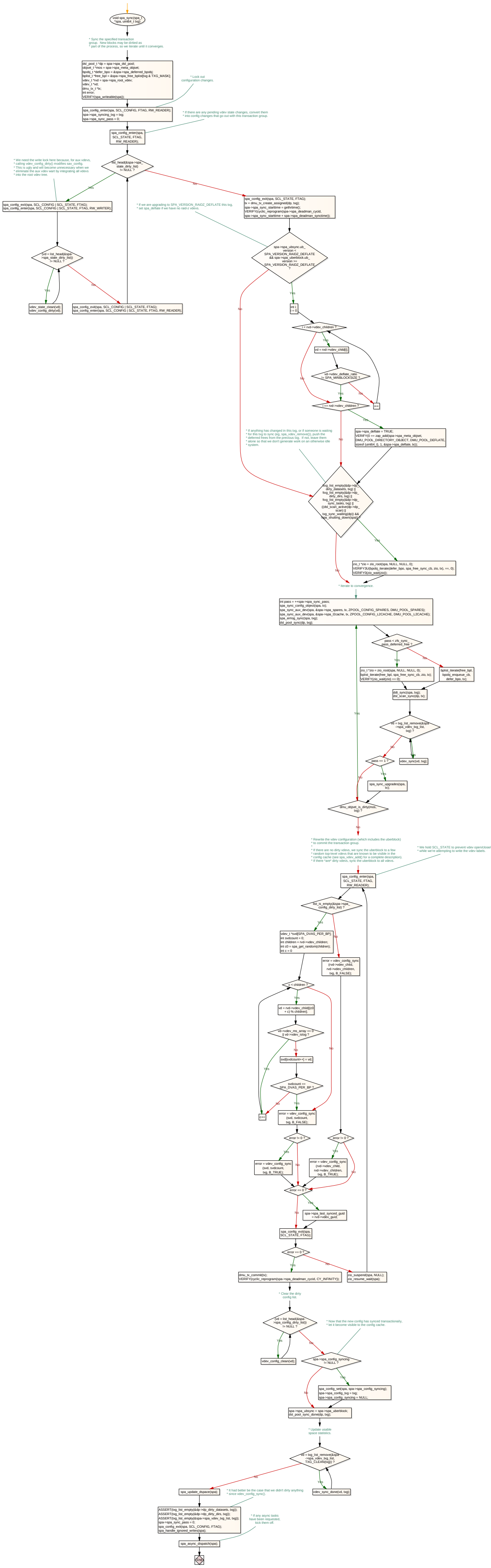
```
spa_t *spa = arg1;
uint64_t version = *(uint64_t *)arg2;
```

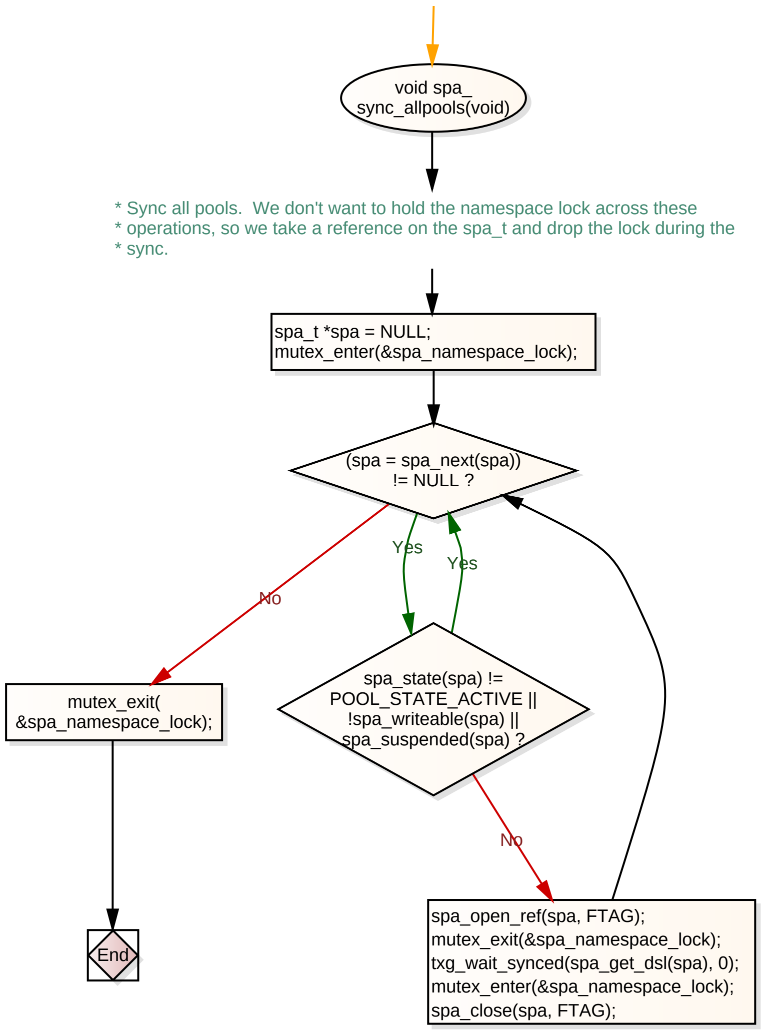
* Setting the version is special cased when first creating the pool.

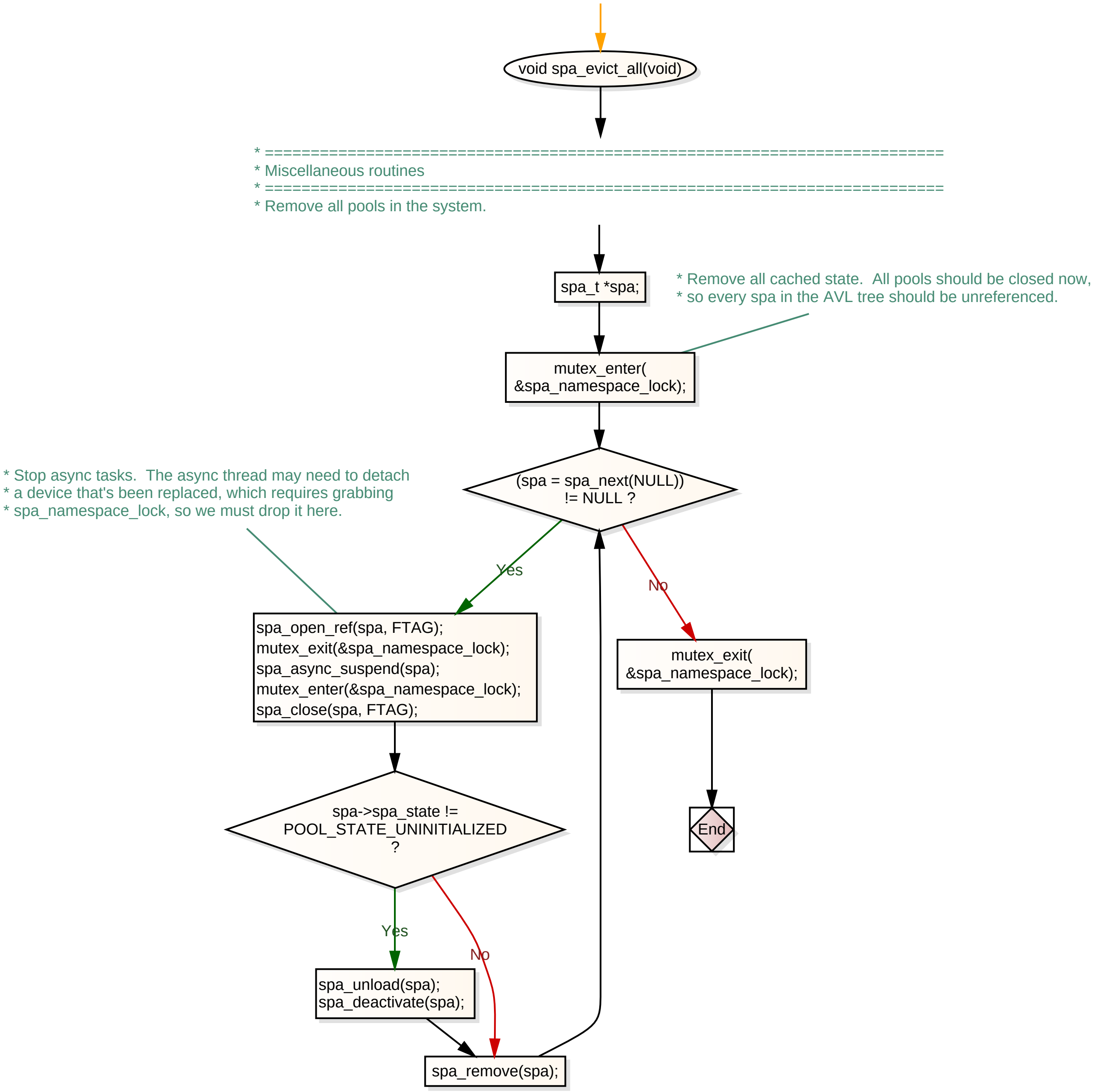
```
ASSERT(tx->tx_txg != TXG_INITIAL);
ASSERT(version <= SPA_VERSION);
ASSERT(version >= spa_version(spa));
spa->spa_uberblock.ub_version = version;
vdev_config_dirty(spa->spa_root_vdev);
spa_history_log_internal(spa, "set", tx, "version=%lld", version);
```

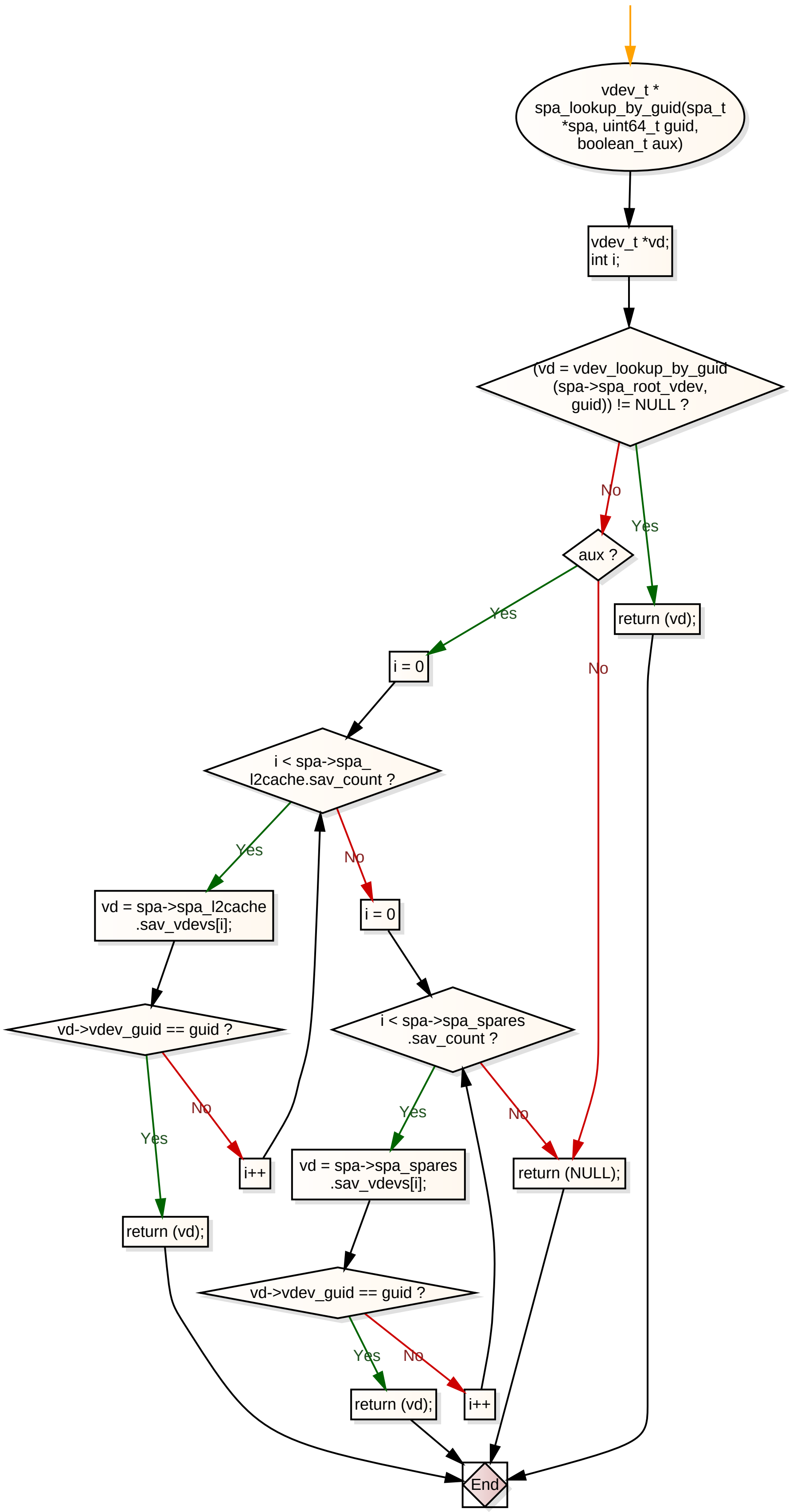

static void spa_sync_props(void *arg1, void *arg2, dmu_tx_t *tx)
* Set zpool properties.
spa_t *spa = arg1;
objset_t *mos = spa->spa_meta_objset;
nvlist_t *nvp = arg2;
nvpair_t *elem = NULL;
mutex_enter(&spa->spa_props_lock);
(elem = nvlist_next_nvpair(nvp, elem)) ?
No
mutex_exit(&spa->spa_props_lock);
End
Yes
uint64_t intval;
char *strval, *fname;
zpool_prop_t prop;
const char *propname;
zprop_type_t proptype;
zfeature_info_t *feature;
switch (prop = zpool_name_to_prop (nvpair_name(elem)))
ZPROP_INVAL ?
* We checked this earlier in spa_prop_validate().
Yes
ASSERT(zpool_prop_feature(nvpair_name(elem)));
fname = strchr(nvpair_name(elem), '@') + 1;
VERIFY3U(0, ==, zfeature_lookup_name(fname, &feature));
spa_feature_enable(spa, feature, tx);
spa_history_log_internal(spa, "set", tx, "%s=enabled", nvpair_name(elem));
No
ZPOOL_PROP_VERSION ?
Yes
VERIFY(nvpair_value_uint64(elem, &intval) == 0);
* The version is synced seperatly before other
* properties and should be correct by now.
ASSERT3U(spa_version(spa), >=, intval);
No
ZPOOL_PROP_ALTROOT ?
* 'altroot' is a non-persistent property. It should
* have been set temporarily at creation or import time.
Yes
ASSERT(spa->spa_root != NULL);
No
ZPOOL_PROP_READONLY ?
Yes
No
ZPOOL_PROP_CACHEFILE ?
Yes
* 'readonly' and 'cachefile' are also non-persisitent
* properties.
No
ZPOOL_PROP_COMMENT ?
Yes
VERIFY(nvpair_value_string(elem, &strval) == 0);
default
* Set pool property values in the poolprops mos object.
spa->spa_comment != NULL ?
Yes
spa_strfree(spa->spa_comment);
No
spa->spa_comment = spa_strdup(strval);
* We need to dirty the configuration on all the vdevs
* so that their labels get updated. It's unnecessary
* to do this for pool creation since the vdev's
* configuratoin has already been dirtied.
tx->tx_txg != TXG_INITIAL ?
No
Yes
vdev_config_dirty(spa->spa_root_vdev);
spa_history_log_internal (spa, "set", tx, "%s=%s", nvpair_name(elem), strval);
spa->spa_pool_props_object == 0 ?
Yes
spa->spa_pool_props_object = zap_create_link(mos, DMU_OT_POOL_PROPS, DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_PROPS, tx);
normalize the property name
No
propname = zpool_prop_to_name(prop);
proptype = zpool_prop_get_type(prop);
nvpair_type(elem) == DATA_TYPE_STRING ?
No
Yes
nvpair_type(elem) == DATA_TYPE_UINT64 ?
not allowed
ASSERT(proptype == PROP_TYPE_STRING);
VERIFY(nvpair_value_string(elem, &strval) == 0);
VERIFY(zap_update(mos, spa->spa_pool_props_object, propname, 1, strlen(strval) + 1, strval, tx) == 0);
spa_history_log_internal(spa, "set", tx, "%s=%s", nvpair_name(elem), strval);
No
Yes
ASSERT(0);
VERIFY(nvpair_value_uint64(elem, &intval) == 0);
proptype == PROP_TYPE_INDEX ?
Yes
const char *unused;
VERIFY(zpool_prop_index_to_string(prop, intval, &unused) == 0);
No
VERIFY(zap_update(mos, spa->spa_pool_props_object, propname, 8, 1, &intval, tx) == 0);
spa_history_log_internal(spa, "set", tx, "%s=%lld", nvpair_name(elem), intval);
switch (prop)
ZPOOL_PROP_DELEGATION ?
Yes
No
spa->spa_delegation = intval;
ZPOOL_PROP_BOOTFS ?
Yes
No
spa->spa_bootfs = intval;
ZPOOL_PROP_FAILUREMODE ?
Yes
No
spa->spa_failmode = intval;
ZPOOL_PROP_AUTOEXPAND ?
Yes
No
spa->spa_autoexpand = intval;
ZPOOL_PROP_DEDUPDITTO ?
Yes
default
tx->tx_txg != TXG_INITIAL ?
Yes
No
spa_async_request(spa, SPA_ASYNC_AUTOEXPAND);
spa->spa_dedup_ditto = intval;

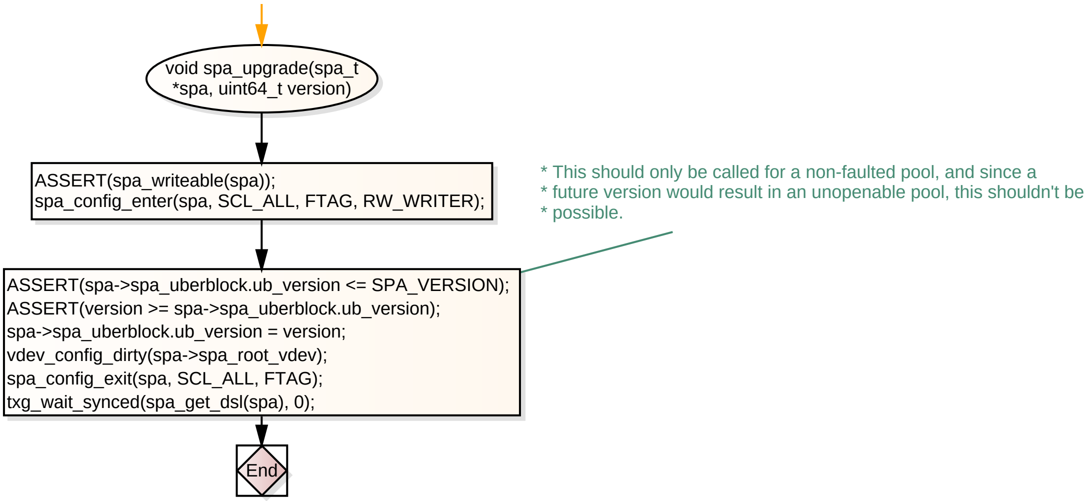

static void spa_sync_upgrades(spa_t *spa, dmu_tx_t *tx)
* Perform one-time upgrade on-disk changes. spa_version() does not
* reflect the new version this txg, so there must be no changes this
* txg to anything that the upgrade code depends on after it executes.
* Therefore this must be called after dsl_pool_sync() does the sync
* tasks.
dsl_pool_t *dp = spa->spa_dsl_pool;
ASSERT(spa->spa_sync_pass == 1);
spa->spa_ubsync.ub_version < SPA_VERSION_ORIGIN && spa->spa_uberblock.ub_version >= SPA_VERSION_ORIGIN ?
Yes
dsl_pool_create_origin (dp, tx);
Keeping the origin open increases spa_minref
No
spa->spa_minref += 3;
spa->spa_ubsync.ub_version < SPA_VERSION_NEXT_CLONES && spa->spa_uberblock.ub_version >= SPA_VERSION_NEXT_CLONES ?
Yes
dsl_pool_upgrade_clones(dp, tx);
No
spa->spa_ubsync.ub_version < SPA_VERSION_DIR_CLONES && spa->spa_uberblock.ub_version >= SPA_VERSION_DIR_CLONES ?
Yes
dsl_pool_upgrade_dir_clones(dp, tx);
Keeping the freedir open increases spa_minref
No
spa->spa_minref += 3;
spa->spa_ubsync.ub_version < SPA_VERSION_FEATURES && spa->spa_uberblock.ub_version >= SPA_VERSION_FEATURES ?
Yes
spa_feature_create_zap_objects(spa, tx);
No
End


void spa_sync(spa_t *spa, uint64_t txg)
* Sync the specified transaction group. New blocks may be dirtied as
* part of the process, so we iterate until it converges.
dsl_pool_t *dp = spa->spa_dsl_pool;
objset_t *mos = spa->spa_meta_objset;
bpobj_t *defer_bpo = &spa->spa_deferred_bpobj;
bplist_t *free_bpl = &spa->spa_free_bplist[txg & TXG_MASK];
vdev_t *rvd = spa->spa_root_vdev;
vdev_t *vd;
dmu_tx_t *tx;
int error;
VERIFY(spa_writeable(spa));
* Lock out configuration changes.
spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
spa->spa_syncing_txg = txg;
spa->spa_sync_pass = 0;
* If there are any pending vdev state changes, convert them
* into config changes that go out with this transaction group.
spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
list_head(&spa->spa_state_dirty_list) != NULL ?
* We need the write lock here because, for aux vdevs,
* calling vdev_config_dirty() modifies sav_config.
* This is ugly and will become unnecessary when we
* eliminate the aux vdev wart by integrating all vdevs
* into the root vdev tree.
Yes
spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_WRITER);
(vd = list_head(&spa->spa_state_dirty_list)) != NULL ?
Yes
vdev_state_clean(vd);
vdev_config_dirty(vd);
No
spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_READER);
No
spa_config_exit(spa, SCL_STATE, FTAG);
tx = dmu_tx_create_assigned(dp, txg);
spa->spa_sync_starttime = gethrtime();
VERIFY(cyclic_reprogram(spa->spa_deadman_cycid,
spa->spa_sync_starttime + spa->spa_deadman_synctime));
* If we are upgrading to SPA_VERSION_RAIDZ_DEFLATE this txg,
* set spa_deflate if we have no raid-z vdevs.
spa->spa_ubsync.ub_version < SPA_VERSION_RAIDZ_DEFLATE && spa->spa_uberblock.ub_version >= SPA_VERSION_RAIDZ_DEFLATE ?
Yes
int i;
i = 0
i < rvd->vdev_children ?
Yes
vd = rvd->vdev_child[i];
vd->vdev_deflate_ratio != SPA_MINBLOCKSIZE ?
Yes
No
i++
No
i == rvd->vdev_children ?
Yes
spa->spa_deflate = TRUE;
VERIFY(0 == zap_add(spa->spa_meta_objset,
DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
sizeof (uint64_t), 1, &spa->spa_deflate, tx));
No
No
* If anything has changed in this txg, or if someone is waiting
* for this txg to sync (eg, spa_vdev_remove()), push the
* deferred frees from the previous txg. If not, leave them
* alone so that we don't generate work on an otherwise idle
* system.
!txg_list_empty(&dp->dp_dirty_datasets, txg) || !txg_list_empty(&dp->dp_dirty_dirs, txg) || !txg_list_empty(&dp->dp_sync_tasks, txg) || ((dsl_scan_active(dp->dp_scan) || txg_sync_waiting(dp)) && !spa_shutting_down(spa)) ?
Yes
zio_t *zio = zio_root(spa, NULL, NULL, 0);
VERIFY3U(bpobj_iterate(defer_bpo, spa_free_sync_cb, zio, tx), ==, 0);
VERIFY0(zio_wait(zio));
No
* Iterate to convergence.
int pass = ++spa->spa_sync_pass;
spa_sync_config_object(spa, tx);
spa_sync_aux_dev(spa, &spa->spa_spares, tx, ZPOOL_CONFIG_SPARES, DMU_POOL_SPARES);
spa_sync_aux_dev(spa, &spa->spa_l2cache, tx, ZPOOL_CONFIG_L2CACHE, DMU_POOL_L2CACHE);
spa_errlog_sync(spa, txg);
dsl_pool_sync(dp, txg);
pass < zfs_sync_pass_deferred_free ?
Yes
zio_t *zio = zio_root(spa, NULL, NULL, 0);
bplist_iterate(free_bpl, spa_free_sync_cb, zio, tx);
VERIFY(zio_wait(zio) == 0);
No
bplist_iterate(free_bpl, bpobj_enqueue_cb, defer_bpo, tx);
ddt_sync(spa, txg);
dsl_scan_sync(dp, tx);
vd = txg_list_remove(&spa->spa_vdev_txg_list, txg) ?
Yes
vdev_sync(vd, txg);
No
pass == 1 ?
Yes
spa_sync_upgrades(spa, tx);
No
dmu_objset_is_dirty(mos, txg) ?
Yes
No
* Rewrite the vdev configuration (which includes the uberblock)
* to commit the transaction group.
*
* If there are no dirty vdevs, we sync the uberblock to a few
* random top-level vdevs that are known to be visible in the
* config cache (see spa_vdev_add() for a complete description).
* If there *are* dirty vdevs, sync the uberblock to all vdevs.
* We hold SCL_STATE to prevent vdev open/close/etc.
* while we're attempting to write the vdev labels.
spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
list_is_empty(&spa->spa_config_dirty_list) ?
Yes
vdev_t *svd[SPA_DVAS_PER_BP];
int svdcount = 0;
int children = rvd->vdev_children;
int c0 = spa_get_random(children);
int c = 0
c < children ?
Yes
vd = rvd->vdev_child[(c0 + c) % children];
vd->vdev_ms_array == 0 || vd->vdev_islog ?
Yes
No
svd[svdcount++] = vd;
svdcount == SPA_DVAS_PER_BP ?
Yes
No
c++
No
error = vdev_config_sync(svd, svdcount, txg, B_FALSE);
error != 0 ?
Yes
error = vdev_config_sync(svd, svdcount, txg, B_TRUE);
No
No
error = vdev_config_sync(rvd->vdev_child, rvd->vdev_children, txg, B_FALSE);
error != 0 ?
Yes
error = vdev_config_sync(rvd->vdev_child, rvd->vdev_children, txg, B_TRUE);
No
error == 0 ?
Yes
spa->spa_last_synced_guid = rvd->vdev_guid;
No
spa_config_exit(spa, SCL_STATE, FTAG);
error == 0 ?
Yes
dmu_tx_commit(tx);
VERIFY(cyclic_reprogram(spa->spa_deadman_cycid, CY_INFINITY));
No
zio_suspend(spa, NULL);
zio_resume_wait(spa);
* Clear the dirty config list.
(vd = list_head(&spa->spa_config_dirty_list)) != NULL ?
Yes
vdev_config_clean(vd);
No
* Now that the new config has synced transactionally,
* let it become visible to the config cache.
spa->spa_config_syncing != NULL ?
Yes
spa_config_set(spa, spa->spa_config_syncing);
spa->spa_config_txg = txg;
spa->spa_config_syncing = NULL;
No
spa->spa_ubsync = spa->spa_uberblock;
dsl_pool_sync_done(dp, txg);
* Update usable space statistics.
vd = txg_list_remove(&spa->spa_vdev_txg_list, TXG_CLEAN(txg)) ?
Yes
vdev_sync_done(vd, txg);
No
spa_update_dspace(spa);
* It had better be the case that we didn't dirty anything
* since vdev_config_sync().
ASSERT(txg_list_empty(&dp->dp_dirty_datasets, txg));
ASSERT(txg_list_empty(&dp->dp_dirty_dirs, txg));
ASSERT(txg_list_empty(&spa->spa_vdev_txg_list, txg));
spa->spa_sync_pass = 0;
spa_config_exit(spa, SCL_CONFIG, FTAG);
spa_handle_ignored_writes(spa);
* If any async tasks have been requested, kick them off.
spa_async_dispatch(spa);
End


void spa_
sync_allpools(void)
* Sync all pools. We don't want to hold the namespace lock across these
* operations, so we take a reference on the spa_t and drop the lock during the
* sync.
spa_t *spa = NULL;
mutex_enter(&spa_namespace_lock);
(spa = spa_next(spa))
!= NULL ?
Yes
Yes
No
mutex_exit(
&spa_namespace_lock);
spa_state(spa) !=
POOL_STATE_ACTIVE ||
!spa_writeable(spa) ||
spa_suspended(spa) ?
No
End
spa_open_ref(spa, FTAG);
mutex_exit(&spa_namespace_lock);
txg_wait_synced(spa_get_dsl(spa), 0);
mutex_enter(&spa_namespace_lock);
spa_close(spa, FTAG);


void spa_evict_all(void)
* ==
* Miscellaneous routines
* ==
* Remove all pools in the system.
spa_t *spa;
* Remove all cached state. All pools should be closed now,
* so every spa in the AVL tree should be unreferenced.
mutex_enter(
&spa_namespace_lock);
(spa = spa_next(NULL))
!= NULL ?
* Stop async tasks. The async thread may need to detach
* a device that's been replaced, which requires grabbing
* spa_namespace_lock, so we must drop it here.
Yes
No
spa_open_ref(spa, FTAG);
mutex_exit(&spa_namespace_lock);
spa_async_suspend(spa);
mutex_enter(&spa_namespace_lock);
spa_close(spa, FTAG);
mutex_exit(
&spa_namespace_lock);
spa->spa_state !=
POOL_STATE_UNINITIALIZED
?
End
Yes
No
spa_unload(spa);
spa_deactivate(spa);
spa_remove(spa);


vdev_t * spa_lookup_by_guid(spa_t *spa, uint64_t guid, boolean_t aux)
vdev_t *vd;
int i;
(vd = vdev_lookup_by_guid (spa->spa_root_vdev, guid)) != NULL ?
No
Yes
aux ?
return (vd);
Yes
No
i = 0
i < spa->spa_ l2cache.sav_count ?
Yes
No
vd = spa->spa_l2cache .sav_vdevs[i];
i = 0
vd->vdev_guid == guid ?
i < spa->spa_spares .sav_count ?
No
Yes
Yes
No
i++
vd = spa->spa_spares .sav_vdevs[i];
return (NULL);
return (vd);
vd->vdev_guid == guid ?
Yes
No
return (vd);
i++
End

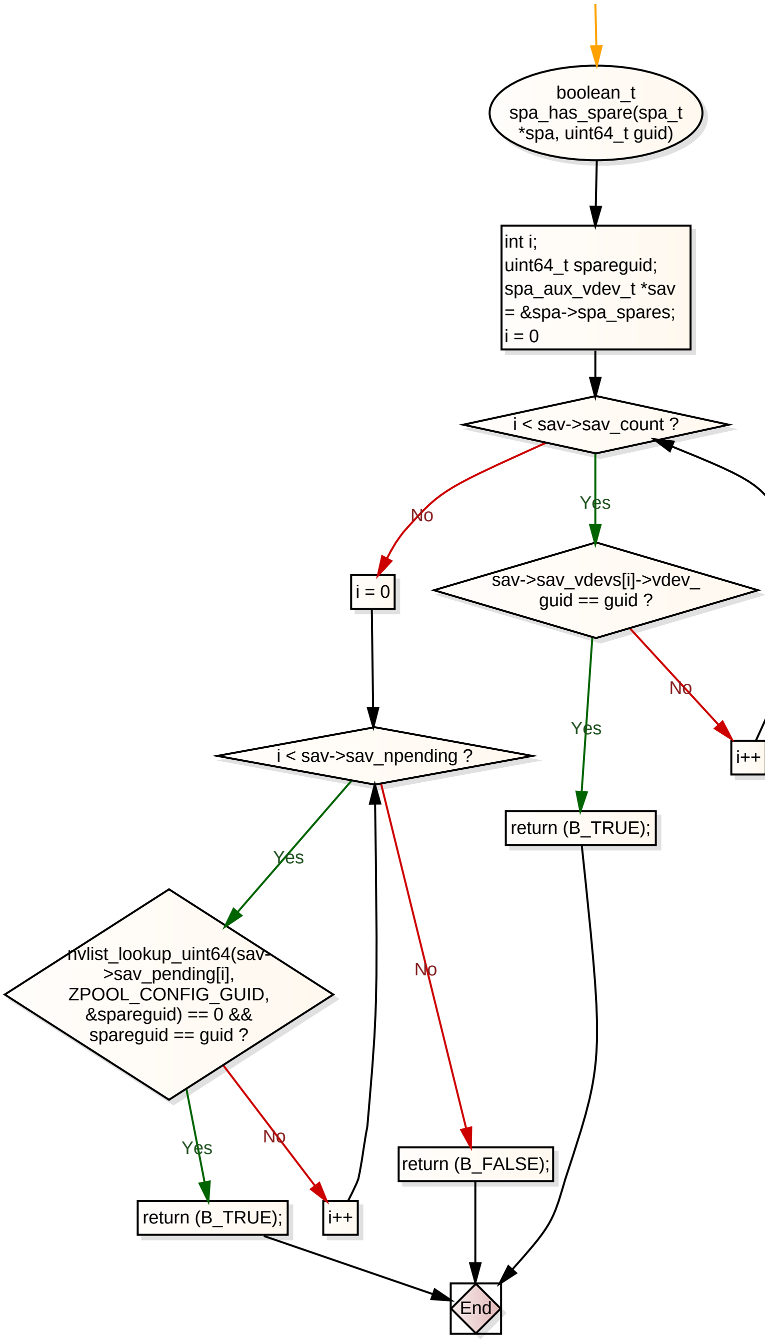


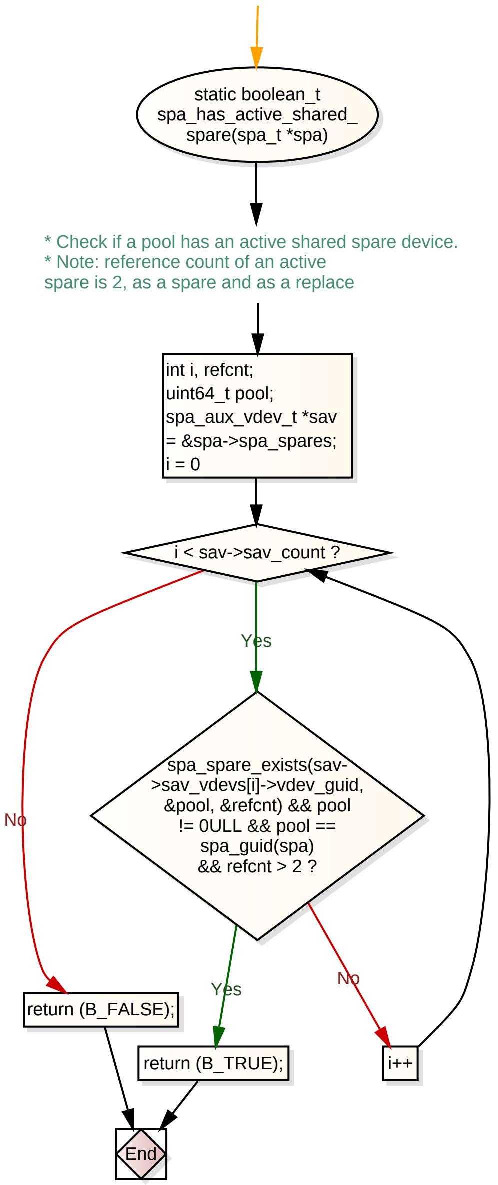
```
void spa_upgrade(spa_t *spa, uint64_t version)
```

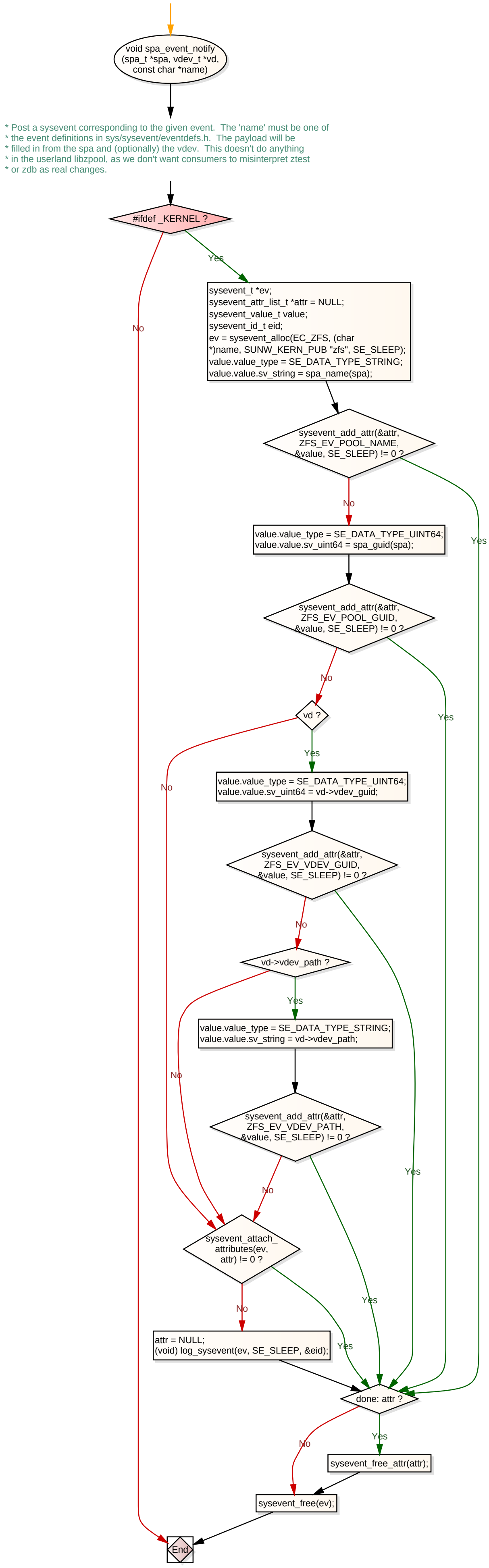
```
ASSERT(spa_writeable(spa));
spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
```

```
ASSERT(spa->spa_uberblock.ub_version <= SPA_VERSION);
ASSERT(version >= spa->spa_uberblock.ub_version);
spa->spa_uberblock.ub_version = version;
vdev_config_dirty(spa->spa_root_vdev);
spa_config_exit(spa, SCL_ALL, FTAG);
txg_wait_synced(spa_get_dsl(spa), 0);
```

```
* This should only be called for a non-faulted pool, and since a
* future version would result in an unopenable pool, this shouldn't be
* possible.
```


boolean_t spa_has_spare(spa_t *spa, uint64_t guid)
int i;
uint64_t spareguid;
spa_aux_vdev_t *sav = &spa->spa_spares;
i = 0
i < sav->sav_count ?
No
Yes
i = 0
sav->sav_vdevs[i]->vdev_guid == guid ?
No
Yes
i++
return (B_TRUE);
i < sav->sav_npending ?
Yes
No
nvlist_lookup_uint64(sav->sav_pending[i], ZPOOL_CONFIG_GUID, &spareguid) == 0 && spareguid == guid ?
Yes
No
return (B_FALSE);
return (B_TRUE);
i++
End


static boolean_t spa_has_active_shared_ spare(spa_t *spa)
* Check if a pool has an active shared spare device.
* Note: reference count of an active spare is 2, as a spare and as a replace
int i, refcnt;
uint64_t pool;
spa_aux_vdev_t *sav = &spa->spa_spares;
i = 0
i < sav->sav_count ?
Yes
No
spa_spare_exists(sav->sav_vdevs[i]->vdev_guid, &pool, &refcnt) && pool != 0ULL && pool == spa_guid(spa) && refcnt > 2 ?
Yes
No
return (B_FALSE);
return (B_TRUE);
i++
End


void spa_event_notify
(spa_t *spa, vdev_t *vd,
const char *name)
* Post a sysevent corresponding to the given event. The 'name' must be one of
* the event definitions in sys/sysevent/eventdefs.h. The payload will be
* filled in from the spa and (optionally) the vdev. This doesn't do anything
* in the userland libzpool, as we don't want consumers to misinterpret ztest
* or zdb as real changes.
#ifdef _KERNEL ?
Yes
No
sysevent_t *ev;
sysevent_attr_list_t *attr = NULL;
sysevent_value_t value;
sysevent_id_t eid;
ev = sysevent_alloc(EC_ZFS, (char
*)name, SUNW_KERN_PUB "zfs", SE_SLEEP);
value.value_type = SE_DATA_TYPE_STRING;
value.value.sv_string = spa_name(spa);
sysevent_add_attr(&attr,
ZFS_EV_POOL_NAME,
&value, SE_SLEEP) != 0 ?
No
Yes
value.value_type = SE_DATA_TYPE_UINT64;
value.value.sv_uint64 = spa_guid(spa);
sysevent_add_attr(&attr,
ZFS_EV_POOL_GUID,
&value, SE_SLEEP) != 0 ?
No
Yes
vd ?
Yes
No
value.value_type = SE_DATA_TYPE_UINT64;
value.value.sv_uint64 = vd->vdev_guid;
sysevent_add_attr(&attr,
ZFS_EV_VDEV_GUID,
&value, SE_SLEEP) != 0 ?
No
vd->vdev_path ?
Yes
No
value.value_type = SE_DATA_TYPE_STRING;
value.value.sv_string = vd->vdev_path;
sysevent_add_attr(&attr,
ZFS_EV_VDEV_PATH,
&value, SE_SLEEP) != 0 ?
Yes
No
sysevent_attach_
attributes(ev,
attr) != 0 ?
Yes
No
Yes
attr = NULL;
(void) log_sysevent(ev, SE_SLEEP, &eid);
done: attr ?
No
Yes
sysevent_free_attr(attr);
sysevent_free(ev);
End