RaidCfg Design Document

(Version 1.2)

Sun RaidCfg Project
alias: raidcfg-dev@sun.com

Table of Contents

1.Overview....5
  1.1.Project Description....5
  1.2.Project Roadmap....5
  1.3.Scope....6
  1.4.Definitions and Terminologies....6
  1.5.Competitive Analysis....7
  1.6.Living with Related Projects....8
  1.7.Project Risks....8
  1.8.Interface Commitment Levels....8
2.Technical Description....9
  2.1.Architecture Overview....9
  2.2.Interfaces....12
    2.2.1.Imported Interface....12
    2.2.2.Exported Interfaces....12
  2.3.Privilege Mode....12
  2.4.Internationalization/Localization....13
  2.5.Release Binding....13
  2.6.RAID Configuration Metadata and Operations....14
3.Concepts: RAID Configuration Metadata....15
  3.1.Object Organization: Relationship....16
  3.2.Object Attributes....17
    3.2.1.System Object....18
    3.2.2.Controller Object....18
    3.2.3.Array Object....20
    3.2.4.Disk Object....22
    3.2.5.HSP Object....23
    3.2.6.Array Part Object....24
    3.2.7.Disk Segment Object....25
    3.2.8.Task Object....26
    3.2.9.Property Object....28
4.RaidCfg Common Library APIs....29
  4.1.Introduction to RaidCfg common library....29
    4.1.1.Overview....29
    4.1.2.Multi-thread Support....29
    4.1.3.Object Referring: Object Handle....30
    4.1.4.Error Processing....30
    4.1.5.Functionality....31

4.2.RaidCfg common library APIs....32
4.2.1.raidcfg_errstr()....32
4.2.2.raidcfg_get_controller()....32
4.2.3.raidcfg_get_array()....33
4.2.4.raidcfg_get_disk()....33
4.2.5.raidcfg_open_controller()....34
4.2.6.raidcfg_close_controller()....35
4.2.7.raidcfg_get_type()....35
4.2.8.raidcfg_get_attr()....36
4.2.9.raidcfg_get_container()....36
4.2.10.raidcfg_list_head()....37
4.2.11.raidcfg_list_next()....37
4.2.12.raidcfg_set_attr()....38
4.2.13.raidcfg_set_hsp()....38
4.2.14.raidcfg_unset_hsp()....39
4.2.15.raidcfg_create_array()....40
4.2.16.raidcfg_delete_array()....41
4.2.17.raidcfg_update_fw()....41
5.Plug-in Libraries....42
5.1.Introduction to Plug-in libraries....42
5.1.1.Overview....42
5.1.2.Plug-in Registration....42
5.1.3.Error Processing....43
5.2.Plug-in Functions....44
5.2.1.rdcfg_open_controller()....44
5.2.2.rdcfg_close_controller()....44
5.2.3.rdcfg_compnum()....45
5.2.4.rdcfg_complist()....46
5.2.5.rdcfg_get_attr()....48
5.2.6.rdcfg_set_attr()....49
5.2.7.rdcfg_array_create()....50
5.2.8.rdcfg_array_delete()....51
5.2.9.rdcfg_hsp_bind()....51
5.2.10.rdcfg_hsp_unbind()....52
5.2.11.rdcfg_update_fw()....53
6.End User Interface....53
6.1.Overview....54
6.1.1.Functionalities....54
6.1.2.Compatibility....54
6.2.Man page....55
7.Appendix A: Error Codes....55
8.Appendix B: Array Hierarchical Expression....57

9.Appendix C: API Examples....58
9.1.Example 1: Query all controller information....59
9.2.Example 2: Query all disks for a given controller....60
9.3.Example 3: Query all available arrays for a given controller....60
9.4.Example 4: Creating an single-layer array....61
9.5.Example 5: Deleting an array....63

1. Overview

1.1. Project Description

The purpose of RaidCfg project is to bring up some incremental enhancement over the existing raidctl(1M) utility, which can only support the mpt driver in ON at this time. The enhancement will include support for other RAID controllers, new RAID levels, hot-spare disk management, more detailed RAID configuration information, etc.

1.2. Project Roadmap

The RaidCfg software will support mpt, mptsas and other compatible open source drivers on Solaris x86-32/64 platforms, and mpt, mptsas driver on SPARC platform; it will be capable of performing basic RAID configuration tasks, including creating and deleting RAID arrays, maintaining global/local HSP (hot spare pool) disks, and querying RAID object information. Due to the hardware evolving, the mptsas driver support and Solaris installation support will be implemented in the second phase.

RaidCfg project will simply cover the above basic features, and a future complete RAID solution can be brought up with some enhancement based on RaidCfg. The possible enhancement may cover some of the following:

- More RAID components like add-on cache/battery module support;
- RAID system monitoring and diagnostic features;
- GUI based user interface;
- Support for JumpStart installation;
- Support for soft RAID and/or soft volume configuration;
- RAID system monitoring and diagnostic features;
- Support for more RAID products (like MegaRAID sas, or nVidia RAID support, etc);
- More advanced RAID configuration features (like array extension, data

migration, etc).

1.3. Scope

The purpose of this document is to propose a series of specifications for developing hardware RAID driver configuration interfaces on Solaris, and/or integrating existing RAID controller configuration functionality into Solaris. This document provides an illustration of the architecture of the whole RaidCfg software, and defines a series of interfaces, commands, and parameters so that multiple RAID controller drivers/plug-in modules can co-exist and can be extended to include future descendants.

The defined specifications include the following:

- RaidCfg common library APIs;
- RaidCfg plug-in SPIs;
- raidctl CLI.

1.4. Definitions and Terminologies

RAID	Redundant Array of Inexpensive Disks.
Controller	Logical RAID controller. Usually it’s corresponding to a physical RAID adapter, but some dual channel adapter may appear as 2 controllers to the operating system.
Disk	Physical disk device attached to RAID adapters.
disk Segment	A logical device that stands for a partial space of a physical disk. This is used to draw out a physical disk’s space usage layout.
Array	Can be either top-array or sub-array. An array is formed directly by disk segments or sub-arrays. An array can be addressed by a controller_id/array_id pair.
Top Array	Logical disk device presented to the host level for user data storage. Can be formed directly by disk segments or sub-arrays.
sub-array	A logical device formed by disk segments and used to form an array. A sub-array can not be directly used for data storage.

HSP Disk	A disk that can be swapped into an array to replace a failed disk. Can be divided into local HSP or global HSP disk.
local HSP disk	HSP disk that can be swapped into an associated array to replace a failed disk.
Global HSP disk	HSP disk that can be swapped into any array of the same controller to replace a failed disk.
Metadata	RAID configuration metadata. This comprises all information about the RAID configuration against a controller.
Raidctl	CLI software for hardware RAID configuration work provided by RaidCfg software.
raidctl(1M)	The existing CLI software for hardware RAID configuration over mpt drivers on Solaris platform.
hardware RAID	A kind of RAID implementation that all RAID algorithm is performed by the IOP on the RAID controller.
soft RAID	A kind of RAID implementation that many RAID algorithm is performed by the controller driver, not by on-board IOP processor.

1.5. Competitive Analysis

Several popular operating systems are investigated, including Microsoft Windows, Linux, FreeBSD, and SCO UNIX. On all above operating system platforms, RAID controller drivers and corresponding RAID configuration software tool are provided by vendors, and the RAID configuration software can only support some series of RAID controller adapters coming from the same vendor. For example, Adaptec Inc. provided aaccli RAID configuration software to support their 2200S and 2120S Ultra320 PCI RAID controller adapters on Windows, Linux, FreeBSD, and SCO UNIX platforms, while LSI Logic Inc. provided lsiutil configuration software to support their Fusion-mpt series of RAID controller adapters on Windows, Linux, and FreeBSD platforms.

Compared with the above, RaidCfg project will bring the following to Solaris platform:

- A general CLI utility for hardware RAID configuration; this will be compatible with existing raidctl (1M) utility.
- A set of API functions for hardware RAID configuration that can be consumed by other applications/libraries;
- A set of plug-in SPI interfaces to enable writing specific plug-in modules to

support individual RAID controller drivers with free-style configuration interfaces.

All the above interfaces are extendable with back-wards compatibility.

1.6. Living with Related Projects

None.

1.7. Project Risks

Following are the risks of RaidCfg project:

- RaidCfg provides a set of SPI interfaces so that specific modules could be written to support open source drivers. For RaidCfg is brought to Open Solaris, driver developer from Open Solaris community may simply write free-style drivers.

- One of the goals of RaidCfg project is simply to promote the existing raidctl(1M) utility with aac driver support and we will not provide installation support in the first phase. User must used BIOS to do this before the development work of JumpStart installation support had been finished in the following phase.

1.8. Interface Commitment Levels

Interface Name	Commitment Level	Comments
RaidCfg lib APIs	Project Private	Common library application programming interface definition and parameters

RaidCfg Plug-in SPIs	Project Private	Plug-in service programming interface definition and parameters
RaidCfg CLI	Committed	Raidctl command line syntax

2. Technical Description

This chapter provides an illustration about the architecture of RaidCfg software, and describes the functionalities of RaidCfg software components.

2.1. Architecture Overview

The architecture of Raidcfg is like:

RaidCfg Architecture Diagram

The RaidCfg software is of a multi-tiered architecture. From the bottom to the top, the components consist of plug-in libraries, common library, and lib-client software.

In above diagram, RaidCfg software comprises the following components:

- **Lib-client software: raidctl**

  This is the command line utility that will be used to interpret the user commands, submit request to RaidCfg common library, collect feedback and generate output messages.

  The new raidctl utility is located in /usr/sbin directory, as a replacement to present raidctl(1M) utility. The new raidctl utility will be compatible with present raidctl(1M) utility, with some enhancement such as RAID 5/10/50 and disk segment support, HSP disk configuration, etc.

- **Common library: libraidcfg.so**

  The common library handles all RAID information on local host. It will convert the submitted parameters into internal presentation, determine the request from lib-client software (raidctl) and perform internal RAID processing, and finally call the plug-in SPI functions.

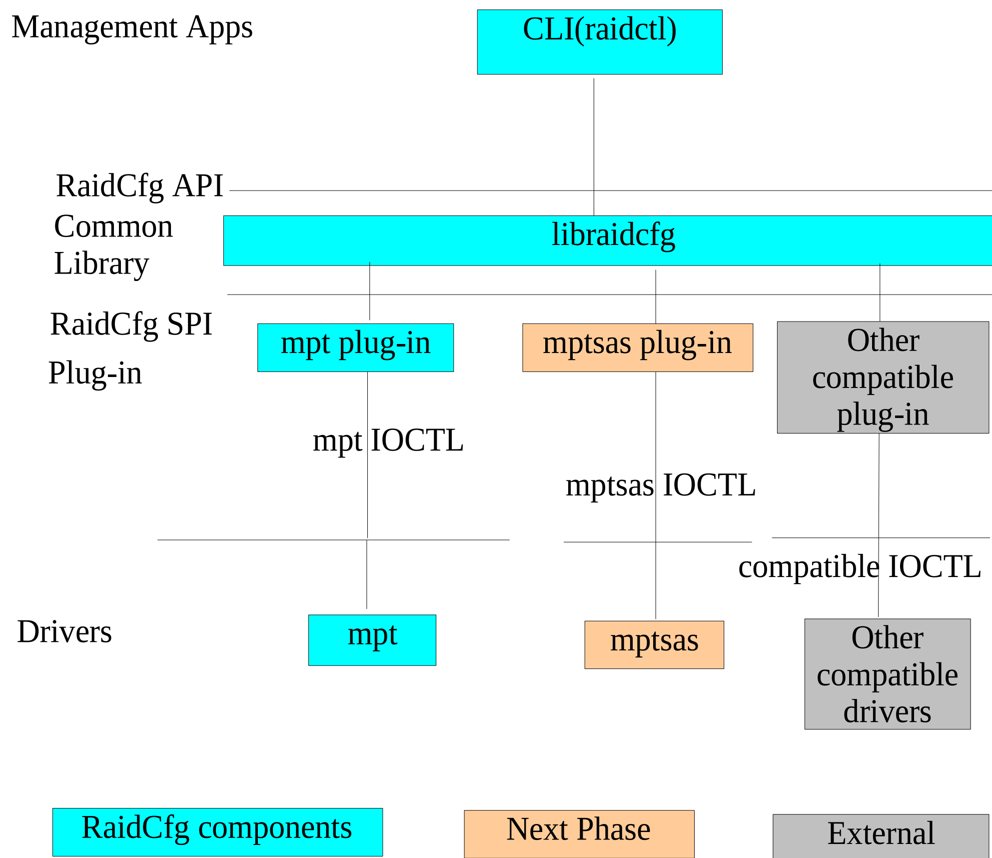
  The common library is located in /usr/lib directory.

- **Plug-in libraries**

  Plug-in libraries are used to convert the requests from common library into RAID configuration IOCTL commands or vice versa. Since the RAID configuration IOCTL interface of controller drivers may vary from one to another, separated plug-in modules must be provided to support drivers with different RAID configuration interface.

  The plug-in libraries are located in /usr/lib/raidcfg directory.

  The controller-specific plug-in modules will be used to transfer data between the common library and drivers with dedicated IOCTL interface. In the first phase RaidCfg will deliver a plug-in module to support mpt driver, and a plug-in module that supports mptsas driver on Solaris will be implemented in the next phase.



- **Drivers**

Drivers will perform concrete operations over the RAID controllers according to the commands issued by Raidcfg software. A dedicated plug-in module must be provided.

RaidCfg put no restriction on the driver's internal implementation, so either hardware RAID or soft RAID controllers can be supported by writing dedicated plug-in module.

2.2. Interfaces

2.2.1. Imported Interface

Following libraries are imported:

- libcfgadm
- libdevinfo

2.2.2. Exported Interfaces

In RaidCfg software, a series of interface have been defined so that each component can communicate with others. The defined interfaces include the following:

- RaidCfg library API
  Application programming interface exported by RaidCfg common library. Used by lib-client software. See chapter 4.

- RaidCfg Plug-in SPI
  Service programming interface. A plug-in module should follow this specification or the module will not be supported by RaidCfg common library. See chapter 5.

- RaidCfg CLI
  Definition of command line syntax. See chapter 6.

2.3. Privilege Mode

RaidCfg software will use privileges to restrict the actions a user can perform. The end user can use RaidCfg software to perform 2 kinds of actions: querying the RAID

configuration or changing it. For querying action, there's no requirement about the privileges; for changing action, the PRIV_SYS_CONFIG privilege is required.

The above privileges should be defined in the File System Management Profile and can be assigned to a user/role as needed. Further more, the HBA driver should check the privilege before conducting any action.

2.4. Internationalization/Localization

The raidctl utility will be an i18N compliant software, so it could be easily extended with other language support; besides of this, the error string return by the common library APIs will also be i18N compliant.

2.5. Release Binding

In the first phase, RaidCfg project will be delivered as a patch containing the following:

raidctl CLI utility

The new raidctl utility is located in /usr/sbin directory, as a replacement to present raidctl(1M) utility.

libraidcfg library

The libraidcfg library is located in /usr/lib directory. It comprises both the common library and the generic plugin module.

Header files:

The following two header files will be provided:

raidcfg.h

Located in /usr/include. This will be used for lib-client software development.

raid_spi.h

Located in /usr/include. This will be used for plug-in software development.

raid_ioctl.h

Located in /usr/include/sys. This will be used for lib-client, plug-in and driver development.

Documents: man pages

Including man page document for raidctl CLI software.

Besides of the above, the patch release will also contain the RaidCfg supported mpt and aac drivers.

2.6. RAID Configuration Metadata and Operations

RaidCfg software is responsible to manipulate the RAID configuration metadata of available RAID controllers on the current host. For hardware controllers, all metadata is originally maintained by individual controllers and stored in the corresponding RAID controller's NVRAM (or disks, or both; and controller is responsible to make the different copy of metadata in NVRAM and in disks consistent), so the metadata is consistent across system boot. RaidCfg software manipulates the metadata through the RAID controller driver's IOCTL interface.

Metadata encompass essential information about all RAID logical devices (we will refer them as RAID objects). It can be divided into 2 kinds of configuration data: the attributes of each RAID object and the relationship between those RAID objects.

RaidCfg software can perform a series of operations over metadata, and these operations are the base of each set of interfaces defined in RaidCfg software. These operations are:

- Querying attributes of any given RAID object; see section 3.2 for detailed description about attributes for each kind of objects.
- Querying relationship of a given RAID object and its associated objects; see section 3.1 for detailed illustration on how RAID objects are organized in RaidCfg software.
- Change a specified attribute of a RAID object. Note that not all attributes of a given RAID object is changeable. Presently only the write_cache_policy attribute of array object is changeable.
- Change the relationship among several RAID objects; usually this kind of operation will also change attributes of some RAID objects. Creating/deleting array are regarded as of this kind of operation; setting/unsetting HSP disks are also regarded as this kind of operations.
- Miscellaneous operations dedicated to specific kind of RAID object, like update the controller's firmware, etc.

The next chapter will provide detailed illustration about how RAID configuration metadata is organized in RaidCfg software; chapter 4, 5, and 6 will provide detailed description about the definition of each interface.

3. Concepts: RAID Configuration Metadata

This chapter provides a detailed description about metadata in RaidCfg software. There are 2 kinds of metadata: RAID object attributes, and the relationship between RAID objects. The original metadata is stored in the RAID controller’s NVRAM (may have backup copies on physical disks) and the controller driver or the corresponding plug-in module is responsible to convert the configuration data into RaidCfg presentation, or vice versa; so it’s un-necessary to store this information in the file system, or anywhere else.

In RaidCfg software, we use object to represent RAID devices, including both logical and physical devices. Presently there are totally 8 kinds of objects defined in RaidCfg software; they are:

- System object;
- Controller object;
- Array object;
- Disk object;
- HSP object;
- Array part object;
- Disk segment object;
- Task object.

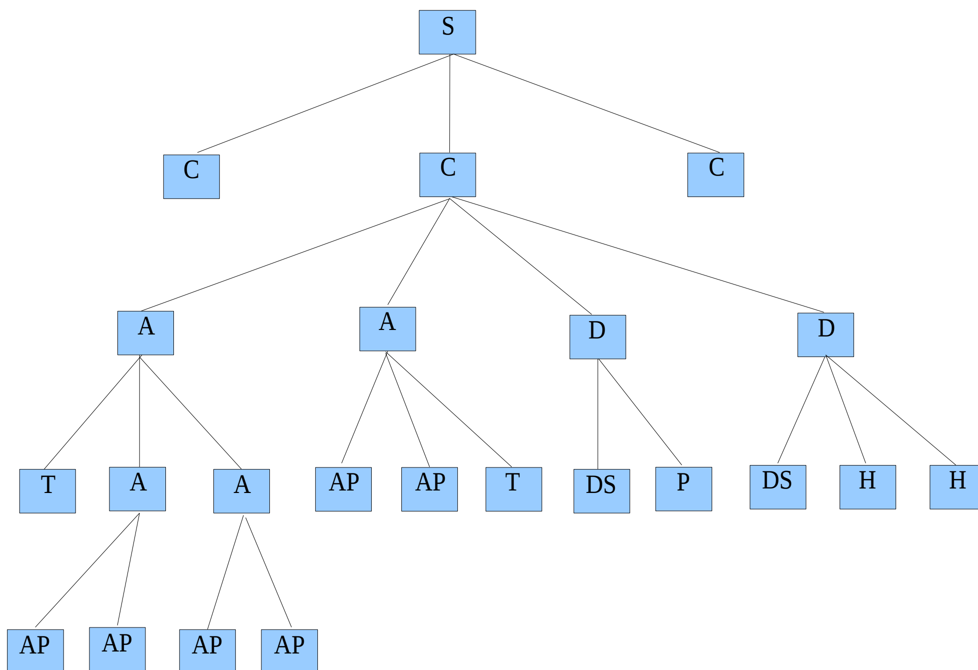
Each object can have a set of associated attributes that implemented as a data structure (section 3.2). All objects are organized as a tree structure via their relationships in RaidCfg software (section 3.1). The common library will invoke the Plug-in SPI interfaces to obtain the metadata, including both the object attributes and their relationships, and build up the internal tree structure; and the lib-client software can use the API functions to manipulate the metadata.

The rest of this chapter provides a detailed illustration about how these RAID configuration data representing the current RAID configuration. Section 3.1 will

describe the relationship between different objects, and section 3.2 will provide detailed description about the attribute of each kind of objects.

3.1. Object Organization: Relationship

All objects are organized as a tree in RaidCfg software, and the object tree is a presentation about the relationships among all objects. A brief chart of object organization is like this:



S: RAID system
C: Controller
A: Array
D: Physical Disk
H: HSP Association
AP: Array Part
DS: Disk Segment
T: Task
P: Property

RAID Object Organization Diagram

In any two objects with direct relationship, following terms are defined:

container
If the two objects are not in same level, the upper one is container;
component
If the two objects are not in same level, the lower one is component;
sibling
If the two objects are in the same level with the same container, they are siblings to each other.

An object can be either a container, or a component, or both.

Objects are organized by the following regulations:

- Only one RAID system object in RaidCfg software; this object stands for the current OS instance.
- RAID system object can only contain controller objects and it has no container. This presents how many and which are the available RAID controllers in current operating system.
- Controller object can contain array and disk objects. This means which physical disks attached to the controller and which are the available arrays.
- Disk object can contain disk segment objects to represent its space usage layout;
- Disk object can contain HSP objects to indicate arrays that use this disk as a HSP disk.
- Disk object can contain Property objects to indicate its property.
- Array object can contain other array objects (sub-array devices) recursively, or array part objects; this means that how disk space are used to form this array.
- Array object can contain a task object to indicate the background activity at a given time.
- Array part object has no components.
- Disk segment object has no components.

3.2. Object Attributes

Basic data types are used to present the attributes of RAID objects. An attribute can be of 8/16/32/64 bit integer number (either signed or un-signed), or a buffer containing

an ascii string.

Not all attributes of an object are valid at any time. Any invalid attribute of integer number (either signed or un-signed) should be set to constant of OBJ_ATTR_NONE, which is defined as following:

```
#define  OBJ_ATTR_NONE   -1
```

For an attribute of string, if it's invalid, the first character should be set to '\0'.

3.2.1.System Object

The system object is the root node of the RAID object tree in RaidCfg software. This object stands for the current OS instance. There's only one system object in RaidCfg software and the corresponding object handle is always 0.

In the RAID object tree, the system object can only have controller objects as its components. This indicates the available RAID controllers in the current OS instance.

The system object has no associated attribute.

3.2.2.Controller Object

A controller object stands for an available RAID controller driver instance that accessible to the current OS instance. It can have array objects and disk objects as its components in RAID object tree; and this indicates the arrays available to host OS and the physical disks attached to this controller.

Normally a controller object is associated with a physical RAID controller device; but for some special adapters, there may be more than one controller objects (for example, LSI 1030 with 2 channels will appears as 2 logical controllers in OS and thus 2 controller objects in RaidCfg).

The attributes of controller object is defined as following:

```
typedef struct {
```

```
    uint32_t controller_id;
    uint32_t max_array_num;
    uint32_t max_seg_per_disk;
    uint32_t connection_type;
    uint64_t capability;
    char fw_version[32];
    char adapter_type[32];
} controller_attr_t;
```

controller_id

Controller device id number. This number is used to address a controller object. In Solaris, each controller should have an entry in directory /dev/cfg, appeared as c*n*, where n is an integer number. **controller_id** should always be equal to *n*.

max_array_num

Maximum number of arrays with current controller. For any given controller, the amount of existing arrays can not exceed this limit.

max_seg_per_disk

Maximum amount of disk segments with a single physical disk. For any physical disk attached to the current controller, the amount of disk segments can not exceed this limit.

connection_type

Indicates the physical interface type. Can be one of the following:

TYPE_SCSI: Parallel SCSI interface.

TYPE_SAS: Serial SCSI interface.

Additional interface type may be added in the future, include SATA, iSCSI, FC, etc.

capability

Bit flags that represent the RAID controller's capabilities. These flags will affect the behavior of RaidCfg common library. Following flags are defined:

CTL_CAP_GLOBAL_HSP

Global HSP disks are supported by this controller.

CTL_CAP_LOCAL_HSP

Local HSP disks are supported by this controller.

CTL_CAP_DISK_VISIBLE

Physical disks attached to this controller are accessible to the host operating system.

CTL_CAP_SMART_ALLOC

Controller can intelligently allocate disk segment to build an array.

CTL_LIMIT_ARRAY_ALIGN

Array_parts belonging to the same array logical device should have the same offset in physical disks.

CTL_CAP_RAID_0

This controller can support RAID 0 arrays.

CTL_CAP_RAID_1
This controller can support RAID 1 arrays.
CTL_CAP_RAID_1E
This controller can support RAID 1E arrays.
CTL_CAP_RAID_5
This controller can support RAID 5 arrays.
CTL_CAP_RAID_10
This controller can support RAID 10 arrays.
CTL_CAP_RAID_50
This controller can support RAID 50 arrays.

fw_version
Version number of the controller's firmware.

adapter_type
Buffer contains a string of product type.

A controller object can be identified by its **controller_id** attribute.

3.2.3.Array Object

An array object can stand for either a top-array logical device, or a sub-array formed another array logical device. For example, a RAID-1 level array is an array object (top-array), while a RAID-10 level array is also an array object (top-array) formed by other 2 RAID-1 level array objects (sub-arrays).

There are some difference between a top-array and a sub-array, as following:

- The top-array is accessible to host OS for data storage, while the sub-array is not;
- A top-array object must be a component of a controller object, while a sub-array must be a component of a top-array object;
- A sub-array object can only have array part objects (see section 3.2.6) as its components in RaidCfg, while a top-array can only have sub-array objects as its components, or only have array part objects as components. This gives information about how an array (either top of sub) is constructed.

The attributes of array object is defined as following:

```
typedef struct {
        uint32_t array_id;
```

```
    union {
        uint64_t reserverd[3];
        struct {
            uint64_t target_id;
            uint64_t lun;
        } idl;
    } tag;
    uint32_t raid_level;
    uint64_t capacity;
    uint32_t stripe_size;
    uint32_t write_policy;
    uint32_t state;
} array_attr_t;
```

array_id

Number used to identify an array. This number is generated by driver or firmware and should be unique within a given controller.

tag

This structure contains a combination of target_id/lun used to address an array device in host OS environment.

raid_level

RAID level of array. Can be one of the following:

- RAID_LEVEL_0
- RAID_LEVEL_1
- RAID_LEVEL_1E
- RAID_LEVEL_5
- RAID_LEVEL_10
- RAID_LEVEL_50

capacity

Capacity of array in bytes.

stripe_size

Size of data stripe used to build the array in bytes. The size must be a multiple of 512. Presently the possible values are: 512 bytes, 1k, 2k, 4k, 8k, 16k, 32k, and 64k, where k stands for 1024 bytes.

Note that for a RAID 1 array, this attribute is in-valid and should be OBJ_ATTR_NONE.

write_policy

Indicates whether cache mechanism is applied to data writing activities of this array.

- CACHE_WR_ON: cache mechanism applied;
- CACHE_WR_OFF: No cache used.

This attribute can be tuned via a RaidCfg API function after the array's creation.

state

Status of current array logical device. Could be one of the following:

- ARRAY_STATE_OPTIMAL: Array is of normal status.
- ARRAY_STATE_DEGRADED: A part formed this array has been failed but array still accessible. Data can be recovered.
- ARRAY_STATE_FAILED: Some parts formed this array failed and data lost. Data on this array can not be recovered.
- ARRAY_STATE_INACTIVATE: The array is in inactive state.

An array object can be identified by a **controller_id/array_id** attributes pair.

3.2.4.Disk Object

A disk object simply stands for a physical disk device. In RaidCfg software, a disk object must be a component of a controller object (stands for the RAID controller the physical disk is attached to).

A disk object can have disk segment objects (stands for which space of the disk has been occupied by array(s), see section 3.2.7) as its components in RaidCfg; and this make the RaidCfg can draw the layout of current disk and can determine the disk space available for creating new array.

If a disk has been set as a HSP disk (either global or local HSP), the disk object must have a (and only a) HSP object as its component. Note that a HSP disk can not have any disk space occupied by any array; This means a disk object with a HSP object as its component can only have component disk segment objects which representing the reserved disk space (see section 3.2.7).

The attributes of disk object is defined as following:

```
typedef struct {
	uint32_t disk_id;
	union {
		struct {
			uint64_t bus;
			uint64_t target_id;
			uint64_t lun;
		} cidl;
	} tag;
```

```
    uint32_t state;
    uint64_t capacity;
    char vendorid[8];
    char productid[16];
    char revision[4];
} disk_attr_t;
```

disk_id

Number used to identify the physical disk. This number is generated by driver or firmware and should be unique within a given controller.

tag

This field contains information used to address a physical disk device. Since presently RaidCfg can only support SCSI/SAS controllers, physical disks can be addressed by its C.ID.L expression (bus_id: target_id: logical unit number). In the future other presentation can be added, for example, for FC controllers, WWN/GUID expression can be added into this union.

state

Physical disk status. Can be one of the following:

DISK_STATE_GOOD

Disk is of normal status.

DISK_STATE_FAILED

Disk can not take operations.

capacity

Capacity of the physical disk in bytes.

vendorid

Text string representing the vendor of disk.

productid

Text string representing the disk product.

revision

Text string representing the revision of the disk product.

A disk object can be identified by a **controller_id/disk_id** attributes pair.

3.2.5.HSP Object

A HSP object stands for a HSP association between a disk and an array and means a array is using the corresponding disk as a HSP disk.; it must be a component of a disk object and identifies array_id(s) for those arrays who are using the associated physical disk (container disk) as a local HSP disk, or as a global HSP disk. A un-assigned disk

can be converted to a global or a local HSP disk, or vice versa; but a HSP disk (either global or local) can not be converted to another kind of HSP disk.

The attributes of HSP object is defined as following:

```
typedef struct {
        uint32_t type;
        uint32_t associated_id;
} hsp_attr_t;
```

type

Can be one of the following:

- HSP_TYPE_GLOBAL: The disk has been used by all arrays as a global HSP disk;
- HSP_TYPE_LOCAL: The disk has been used as a local HSP disk.

associated_id

If **type** is HSP_TYPE_LOCAL, this field contains the array_id of the array that using this disk as a local HSP disk; otherwise, it should be set to OBJ_ATTR_NONE.

A HSP object can be addressed by a **controller_id/disk_id/associated_id** combination.

3.2.6.Array Part Object

An array part object stands for a partial space of a physical disk, which has been used to build up an array device. An array part object must be a component of the array object (either top-array or sub-array) in RaidCfg. Group of array part objects with the same container array object present how the space coming from different physical disks are used to form the array.

Array part objects within the same container array object must have the same **size** attributes and different **disk_id** attributes. For some specific RAID controllers (for those with the **CTL_LIMIT_ARRAY_ALIGN** bit set in **capability** attribute), the **offset** attributes should also be the same.

Normally, if an array part object existed, there should be a corresponding disk segment object (with identical **offset** and **size** attribute) as a component to the disk object specified by this array part object; but if the array (container to the array part) is

degraded, the corresponding disk segment object may be in invalid status, or simply lost.

The attributes of array part object is defined as following:

```
typedef struct {
    uint32_t disk_id;
    uint32_t state;
    uint64_t offset;
    uint64_t size;
} arraypart_attr_t;
```

disk_id

The device id number of the physical disk which the array part is physically belonging to.

state

Stands for the availability of the disk segment. Can be one of the following:

- ARRAYPART_STATE_GOOD: The array part is of normal status.
- ARRAYPART_STATE_MISSED: Data on this array part is out of sync and need to be recovered.

offset

The start position of the array part, relative to the beginning of the physical disk which the array part is located physically. The number is in bytes.

size

Size of array part. The number is in bytes.

An array part object can be addressed by a **controller_id/disk_id/array_id** combination.

3.2.7.Disk Segment Object

A disk segment object stands for an occupied partial space of a physical disk device. A group of disk segment objects with the same container disk object present the layout of the physical disk usage, and RaidCfg can use this information to determine the available disk space for operation (like creating new array).

There are 3 kinds of disk segment objects defined in RaidCfg, determined by the **state** attribute (see below). Please note for a HSP disk (a disk object that has a child HSP object), it can only have reserved disk segment objects (the **state** attribute of all disk

segments should be set as DISKSEG_STATE_RESERVED).

The attributes of disk segment object is defined as following:

```
typedef struct {
        uint32_t seq_no;
        uint64_t offset;
        uint64_t size;
        uint32_t state;
} diskseg_attr_t;
```

seq_no
Index number of the current disk segment.
A physical disk may have several disk segments, and these disk segments are sorted by the value of **offset** attribute, in ascending order.

offset
The start position of the disk segment, regarding the beginning of the physical disk. The number is in bytes.

size
Size of disk segment. The number is in bytes.

state
Bit flags stand for the availability of the disk segment. Following flags are defined:

- Bit 0: DISKSEG_STATE_RESERVED: Disk segment is for system use and can not be allocated to build an array.
- Bit 1: DISKSEG_STATE_ORPHAN: Disk segment contains data but does not belong to any array. For example, this disk segment space had been used by an array, then the disk had been plug-out and a HSP disk swapped into the array; then re-insert the disk back and the disk segment will not be used any more.
- Bit 2: DISKSEG_STATE_NORMAL: Disk segment is used by an array.

A disk segment object can be addressed by a **controller_id/disk_id/seq_no** combination.

3.2.8. Task Object

A task object stands for a series of asynchronous background activities associated with a raid device (either logical or physical device), which is maintained by driver or

controller firmware and is independent to user-land software. Typically a task will prepare or repair a device for controller's usage (for example, verifying a physical disk, or building an array).

For many hardware RAID products, the controller maintained tasks are consistent across system reboot or even power life cycle; the task maintenance is not a critical requirement for RaidCfg software, since any activities conducted by RaidCfg will simply based on present status. For any given time and a given device (presently can only be an array object), there should only be at most 1 task.

Presently RaidCfg only defined the one kind of task:

- building: re-calculating parity data for a redundant array, and repairing damaged data if bad block detected.

For a given controller, the support for background tasks can be quite different; so the driver/plug-in library is responsible to map the adapter-specific tasks into the above set. For example, for an Adaptec 2120 controller which is supported by AAC driver in Solaris, the driver should map both the REBUILD_RAID5 and REBUILD_MIRROR tasks to an array-rebuilding task in RaidCfg.

The attributes of task object is defined as following:

```
typedef struct {
	uint32_t task_id;
	uint32_t task_func;
	uint32_t task_state;
	uint32_t progress;
} task_attr_t;
```

task_id

This is a unique number identifying a task within a given controller.

The number should be generated by the driver or RAID controller's on-board firmware; the driver or the firmware must ensure this number to be unique for a controller, and should be consistent in the life time of host operating system.

task_func

States the functionality of background task which is related to the current raid device. Currently can only be the following:

- TASK_FUNC_BUILD: Building array task;

task_state

Running state of background task associated with current array. Can be one of the following:

- TASK_STATE_UNKNOWN
- TASK_STATE_TERMINATED
- TASK_STATE_FAILED
- TASK_STATE_DONE
- TASK_STATE_RUNNING
- TASK_STATE_SUSPENDED

task_progress
States what percentage of the current task has been finished.

A task object can be addressed by a **controller_id/task_id** combination.

3.2.9.Property Object

Property object will only associate with a single disk object as a component. The GUID is a property for a physical disk.

The attributes of property object is defined as following:

```
typedef struct {
        uint32_t        prop_id;
        uint32_t        prop_size;
        property_type_t prop_type;
        char            prop[1];
} property_attr_t;
```

prop_id
A unique id number within a controller. It's used to address the property object.

prop_size
Length of buffer specified by the 'prop' field.

prop_type
Specifying what kind of information in buffer 'prop'. Currently can only be PROP_GUID.

prop
Buffer used to store property information.

4. RaidCfg Common Library APIs

4.1. Introduction to RaidCfg common library

4.1.1. Overview

The common library is responsible to query all metadata from drivers and generate a presentation to lib-client software, and will convert requests from lib-client software to plug-in modules/drivers.

The metadata is stored in NVRAM of the hardware RAID controller (or physical disks), not in the host file system or anywhere elsewhere on the host system. This information is consistent across reboot. When the lib-client software (raidctl) submit a request to the common library, the library will query the controller to get the latest metadata and perform the corresponding action; once the action is finished, the common library will release all metadata in memory. This ensures that the common library always manipulate the latest metadata.

4.1.2. Multi-thread Support

The RaidCfg common library is multi-thread safe; a global mutex has been provided to make sure that all actions passed to the common library will be serialized.

4.1.3.Object Referring: Object Handle

In RaidCfg common library, all RAID devices are represented as objects, and these objects are organized as a tree; each object has a handle associated with it, and the handles are exported to lib-client software, so the lib-client software can use object handles to access RAID objects.

Object handles are consistent across the corresponding objects' life time (from the opening of controller to the close of controller, or the deleting of the object). It’s unnecessary to open an object to get a handle, or to release a handle explicitly, except for the controller object. For a given controller, once it has been opened, when an object (as a direct or in-direct component of the controller) is accessed for the first time, RaidCfg common library will assign a handle to this object, and the handle will exist until the corresponding RAID object is deleted (for example, deleting an array) or the corresponding controller is closed.

A reason of using object handles in RaidCfg APIs is to simplify the maintenance of diverse objects. For example, an array can be addressed by a combination of controller_id/array_id, while an array part can be addressed by a combination of controller_id/array_id/disk_id; and a disk segments object can be addressed by a combination of controller_id/disk_id/offset. It could be easier for the lib-client software to maintain the uniformed handles, instead of diverse device_id numbers (or other parameters).

4.1.4.Error Processing

For each API function except raidcfg_errstr(), the return value is a integer number. If no error occurs, the return value should be greater or equal to 0; if less than 0, a error occurred and the return value is the error code. For the definition of error code, see Appendix A.

For each API function that will modify the metadata, the common library will check the passed parameters before conduct the plug-in functions; but because of the variety of the supported hardware, some additional verification over the parameters may be conducted by hardware-dependent plug-in modules and RaidCfg can not define all possible error code. So, for these API functions, a character string pointer should be passed and will hold the hardware-dependent error text string, which is returned by

individual plug-in modules. If no error occurred, this plug-in error string pointer will be set to NULL.

4.1.5.Functionality

The API functions provided by RaidCfg common library will cover the following functionality:

- Query associated attributes for a given RAID object;
- Query the relationship of a given RAID object;
- Change specified attribute of a given RAID object;
- Change the relationship (organization) of a series of RAID objects;
- Perform miscellaneous operation against an object.

These API functions exported by RaidCfg common library are user land interfaces that can be consumed by lib-client software. The prototype of these functions are (with a prefix of raidcfg_) defined in /usr/include/sys/raidcfg.h head file.

Based on the above defined functionalities, RaidCfg provides the following API functions:

- Maintain handles for RAID object

raidcfg_get_controller()
raidcfg_get_array()
raidcfg_get_disk()
raidcfg_open_controller();
raidcfg_close_controller();

- Query associated attributes with a given RAID object

raidcfg_get_type()
raidcfg_get_attr()

- Query the relationship of a given RAID object

raidcfg_get_container()
raidcfg_list_head()
raidcfg_list_next()

- Change specified attribute of a given RAID object

raidcfg_set_attr()

- Change the relationship (organization) of a series of RAID objects

raidcfg_set_hsp()
raidcfg_unset_hsp()
raidcfg_create_array()
raidcfg_delete_array()

- Perform miscellaneous operation against an object

raidcfg_update_fw();

In addition, following types are defined in /usr/include/sys/raidcfg.h:

```
typedef int raid_obj_type_id_t;
```

4.2. RaidCfg common library APIs

4.2.1. raidcfg_errstr()

This function will return a pointer to a string which contains explanation of passed error code.

```
char *raidcfg_errstr(int err_code);
```

err_code

Error number returned by other API functions. For the definition of error code, see Appendix A.

If the passed parameter is not a valid error number, this function will return NULL.

4.2.2. raidcfg_get_controller()

This function will return a handle of the specified controller. Note that even if the controller is not opened, this operation will still succeed.

int raidcfg_get_controller(uint32_t controller_id);

controller_id
RAID controller id number. See section 3.2.2 for detailed description.

If it’s greater than 0, it’s the handle of specified controller object; if less than 0, it’s the error code.

The return value can not be 0.

4.2.3.raidcfg_get_array()

This function will return a handle of the specified array. Note that the specified controller should be opened or this function will return with error.

int raidcfg_get_array(int controller_handle, uint32_t target_id, uint64_t lun);

controller_handle
RAID controller's handle.
target_id/lun
The combination of target_id/lun numbers used to address an array. See section 3.2.3 for details.

If it’s greater than 0, it’s the handle of specified array object; if less than 0, it’s the error code.

The return value can not be 0.

4.2.4.raidcfg_get_disk()

This function will return a handle of the specified physical disk. Note that the specified controller should be opened or this function will return with error.

int raidcfg_get_disk(int controller_handle, union disk_tag tag);

controller_handle
RAID controller's handle.
tag
Tag information used to address a physical disk device. The definition is as following:

```
union disk_tag_t {
        struct {
                uint32_t bus;
                uint32_t target_id;
                uint64_t lun;
        } cidl;
};
```

If it’s greater than 0, it’s the handle of specified disk object; if less than 0, it’s the error code.

The return value can not be 0.

4.2.5.raidcfg_open_controller()

This function will open the specified controller exclusively. This is a pre-requisite to access any object as a direct or indirect component of this controller.

```
int raidcfg_open_controller(int controller_handle, char **err_str);
```

controller_handle
Controller object handle.
err_str
Address of hardware-dependent error string buffer pointer.

The return value can not be great than 0; if it equals 0, the controller is successfully opened and can not be accessed by other processes. If it’s less than 0, an error occurred and the return value is the error code.

4.2.6.raidcfg_close_controller()

This function will close the specified controller so the controller can be used by other process. Note that all object handles related to this controller will be invalid, except the controller's handle.

```
int raidcfg_close_controller(int controller_handle, char **err_str);
```

controller_handle
Controller object handle.
err_str
Address of hardware-dependent error string buffer pointer.

The return value can not be great than 0; if it equals 0, the controller is successfully closed. If it’s less than 0, an error occurred and the return value is the error code.

4.2.7.raidcfg_get_type()

This function returns the type information of specified object.

```
int raidcfg_get_type(int obj_handle);
```

obj_handle
The component object’s handle.

If the return value is great than or equal to 0, it’s the type of the object specified by the handle. Can be one of the following:

- OBJ_TYPE_SYSTEM,
- OBJ_TYPE_CONTROLLER
- OBJ_TYPE_ARRAY
- OBJ_TYPE_DISK
- OBJ_TYPE_HSP
- OBJ_TYPE_ARRAY_PART
- OBJ_TYPE_DISK_SEG
- OBJ_TYPE_TASK
- OBJ_TYPE_PROP

If it’s less than 0, an error occurred and the return value is the error code.

4.2.8.raidcfg_get_attr()

This function will dump out the specified object’s attributes to the buffer. For detailed information about object attributes, see section 3.2. Note that for a controller device, if it’s not opened, this function will still succeed, but all attributes except the **controller_id** will be invalid, since the other attributes are coming from the driver. For other kind of objects, this function returns an error code of ERR_DRIVER_CLOSED.

```
int raidcfg_get_attr(int obj_handle, void *attr);
```

obj_handle
Handle of the target object;
attr
Pointer to buffer used to hold attribute information of specified object.

This function returns 0 when successfully completed; if the return value is less than 0, an error occurred and the return value is the error code. The return value can not be greater than 0.

4.2.9.raidcfg_get_container()

This function returns the handle of the container object of the specified object.

```
int raidcfg_get_container(int obj_handle);
```

obj_handle
The component object’s handle.

If the return value is greater than or equal to 0, it’s the handle of specified disk object; if less than 0, it’s the error code.

4.2.10.raidcfg_list_head()

Returns handle of the first component object of given type belonging to the specified container object.

```
int raidcfg_list_head(int obj_handle, raid_obj_type_id_t type);
```

obj_handle
Container object's handle;
type
Component object type. Can be one of the following:

- OBJ_TYPE_SYSTEM,
- OBJ_TYPE_CONTROLLER
- OBJ_TYPE_ARRAY
- OBJ_TYPE_DISK
- OBJ_TYPE_HSP
- OBJ_TYPE_ARRAY_PART
- OBJ_TYPE_DISK_SEG
- OBJ_TYPE_TASK
- OBJ_TYPE_PROP

If the return value is greater than 0, it's the handle of the first component object of the specified type; if equals to 0, the specified object has no component of the specified type; if less than 0, it's the error code.

4.2.11.raidcfg_list_next()

Returns handle of a sibling object of the same type with the specified object.

```
int raidcfg_list_next(int obj_handle);
```

obj_handle
Object's handle.

If the return value is greater than 0, it's the handle of the next object of the same type, which has the same container object with the specified object; if equals to 0, the

specified object was the last object of the specified type; if less than 0, it's the error code.

4.2.12.raidcfg_set_attr()

This function is used to set/apply the attributes of the specified array.

```
int raidcfg_set_attr(int obj_handle, uint32_t set_cmd, void *value, char **err_str);
```

obj_handle
Handle to the target array.

set_cmd
Sub-command to indicate which attribute will be set; currently it can be SET_CACHE_WR_PLY, used to set **write_policy** attribute of array object, and SET_ACTIVATION_PLY used to set activation attribute of array object.

value
Pointer of value that will be set; for SET_CACHE_WR_PLY sub-command, it can be either CACHE_WR_ON or CACHE_WR_OFF (see section 3.2.3). For some controllers that provided more options, the driver/plug-in can choose the default setting if **value** is CACHE_WR_ON; the final write_policy is controller specific and user should refer the driver/controller manual; for SET_ACTIVATION_PLY sub-command, it can be ARRAY_ACT_ACTIVATE (see section 3.2.3).

err_str
Address of hardware-dependent error string buffer pointer.

The return value can not be greater than 0; if it equals to 0, the operation is successfully accomplished; if less than 0, it's the error code.

4.2.13.raidcfg_set_hsp()

This function will convert a set of un-assigned physical disks as either local or global HSP disks. Note that it can not convert a local HSP disk to a global HSP disk, or vice versa.

```
int raidcfg_set_hsp(struct raidcfg_hsp_relations_t *handles, char **err_str);
```

handles

Each element contains pair of array handle and disk handle, means the specified physical disk will be assigned as a local HSP disk to the corresponding array, or as will be assigned as a global HSP, depending on **array_handle** value.

```
struct raidcfg_hsp_relation_t {
        int array_handle;
        int disk_handle;
  };
```

err_str

Address of hardware-dependent error string buffer pointer.

If the **array_handle** equals to OBJ_ATTR_NONE, the corresponding disk will be set as a global HSP disk.

The return value can not be greater than 0; if it equals to 0, the operation is successfully accomplished; if less than 0, it’s the error code.

4.2.14.raidcfg_unset_hsp()

This function will convert a set of HSP disks to un-assigned disks.

```
int raidcfg_unset_hsp(struct raidcfg_hsp_relations_t *handles, char **err_str);
```

handles

Each element contains pair of array handle and disk handle, means the specified physical disk will no longer be a local HSP disk or a global HSP disk, depending on **array_handle** value.

For information about the definition of the structure, see the previous section (raidcfg_set_hsp()).

When the specified disk is a global HSP disk, if **array_handle** equals to OBJ_ATTR_NONE, this disk will be converted into a un-assigned disk; if **array_handle** doesn’t equal to OBJ_ATTR_NONE, the function will return with error.

err_str

Address of hardware-dependent error string buffer pointer.

The return value can not be greater than 0; if it equals to 0, the operation is successfully accomplished; if less than 0, it's the error code.

4.2.15.raidcfg_create_array()

This function is used to create an array and return the handle of the newly created array object.

int raidcfg_create_array(int num_of_comps, int *disk_handles, uint32_t raid_level, uint64_t *size, uint32_t stripe_size, char **err_str);

num_of_comps
Size of array specified by **disk_handles**;

disk_handles
Handle of disks used to build the specified array. These handles should be organized in array hierarchical expression (see Appendix B).

raid_level
Raid level of the array that will be created; for the possible values, see section 3.2.3.

size
Pointer to capacity of the array that will be created. The value is in bytes. If the pointed value is set to OBJ_ATTR_NONE, this function will not create the volume but fill the address with a maximum size of the volume that can be created by the specified disks. If the pointed value is 0, this function will try to create an array with maximum capacity.

stripe_size
Size of data stripe used to build the array, in bytes; for the possible values, see section 3.2.3. If the passed value is 0, this function will let the controller driver to pick up a default value.

Note that if the **raid_level** had been set to RAID_LEVEL_1, this parameter will be ignored.

err_str
Address of hardware-dependent error string buffer pointer.

If the return value is greater than or equal to 0, the action is successfully completed and the return value is the handle of the newly created array; If it's less than 0 then an error occurred and the return value is the error code.

4.2.16.raidcfg_delete_array()

Delete the array specified by the handle.

```
int raidcfg_delete_array(int array_handle, char **err_str);
```

array_handle

Handle of the array which will be deleted. Note that the array must be a top-array, not a sub-array.

err_str

Address of hardware-dependent error string buffer pointer.

The return value can not be greater than 0; if it equals to 0, the operation is successfully accomplished; if less than 0, it’s the error code.

4.2.17.raidcfg_update_fw()

Update the firmware of the specified controller.

```
int raidcfg_update_fw(int controller_handle, char *file, char **err_str);
```

controller_handle

Handle of the controller.

file

Path name of firmware image file.

err_str

Address of hardware-dependent error string buffer pointer.

The return value can not be greater than 0; if it equals to 0, the operation is successfully accomplished; if less than 0, it’s the error code.

5. Plug-in Libraries

5.1. Introduction to Plug-in libraries

5.1.1. Overview

The controller-specific plug-in libraries are located in directory /usr/lib/raidcfg. A controller-specific plug-in library will convert the requests from RaidCfg common library to driver-specific IOCTL commands, or vice versa. Each controller-specific plug-in library should have a name as *xxx*, where *xxx* is the name of the driver.

In RaidCfg software, all controller drivers should be cfgadm compliant; this means each driver should have an entry in directory /dev/cfg, appearing as c*n*, where *n* is an integer number (see section 3.2.2). The common library will enumerate all these entries and for each entry, common library will try to detect if the controller is a RAID controller supported by RaidCfg (see following description for the progress).

When the RaidCfg software is going to access a RAID controller driver for the first time, the common library searches the /usr/lib/raidcfg directory for the corresponding controller-specific plug-in library and load it into memory and, if the plug-in module didn’t follow this specification, the module will be unloaded by the common library. If the RaidCfg common library failed to load the controller-specific plug-in library, or the plug-in module can not be found, which means the controller is not supported by RaidCfg software.

5.1.2. Plug-in Registration

When the RaidCfg common library loads a plug-in library into memory, it will check the version of the plug-in module, then search for a series of exported functions and fill up an internal data structure, and finally register this internal data structure into a

list. During the plug-in module loading process, the common library will not invoke any function exported by the plug-in module, and the plug-in module is responsible to complete all necessary initializing work by itself. And when the plug-in module is unloaded, the plug-in module should free all allocated resource.

For each plug-in module, the following variable should be exported:

```
uint32_t rdcfg_version;
```

This variable is used to indicate the version number of the plug-in spec the current module is following; presently the possible value is RDCFG_PLUGIN_V1.

Each plug-in module should also export the following functions:

```
int rdcfg_open_controller();
int rdcfg_close_controller();
int rdcfg_compnum();
int rdcfg_complist();
int rdcfg_get_attr();
int rdcfg_array_create();
int rdcfg_array_delete();

int rdcfg_set_attr();
int rdcfg_hsp_bind();
int rdcfg_hsp_unbind();
int rdcfg_update_fw();
```

Among the above defined functions, the last 4 are optional functions and the others are mandatory. Each plug-in module should implement all the mandatory functions, or the plug-in will not be loaded by common-lib.

5.1.3.Error Processing

For each exported plug-in function, the return value is a integer number; if no error occurs, the return value should be greater or equal to 0; if less than 0, a error occurred and the return value is the error code. For the definition of error code, see Appendix A.

For each plug-in exported function that will modify metadata, the address of a string pointer will be passed by the common library, and the plug-in module can fill this

with a pointer to a text string for error description, in case of any error occurred. This is not mandatory, and typically the plug-in function should return a pre-defined error code to address a problem; but if the plug-in function have to perform some additional verification over the parameters, and it can not accurately map the hardware dependent error to a pre-defined error code, it can pass a text message to the common library using this. In this case, typically the error code should be ERR_PLUG_IN.

5.2. Plug-in Functions

5.2.1. rdcfg_open_controller()

This function is used to exclusively open a controller minor node for configuration work.

```
int rdcfg_open_controller(uint32_t controller_id, char **err_ptr);
```

controller_id
Controller device id number.

err_ptr
Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.2. rdcfg_close_controller()

This function is used to close a controller minor node.

```
int rdcfg_close_controller(uint32_t controller_id, char **err_ptr);
```

controller_id
Controller device id number.

err_ptr

Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.3.rdcfg_compnum()

This function returns an integer number representing how many components of given type belonging to the specified container are available.

int rdcfg_compnum(uint32_t controller_id, uint32_t container_id, raid_obj_type_id_t container_type, raid_obj_type_id_t component_type);

controller_id

Controller device id number.

container_id

The meaning of the field depends on **container_type**.

Container_type	**container_id**
RAID_OBJ_TYPE_CONTROLLER	Not used;
RAID_OBJ_TYPE_ARRAY	The array_id;
RAID_OBJ_TYPE_DISK	The disk_id.

container_type

Specify the raid device type identified by combination of **controller_id** and **container_id**.

Could be one of the following:

- RAID_OBJ_TYPE_CONTROLLER
- RAID_OBJ_TYPE_ARRAY
- RAID_OBJ_TYPE_DISK

component_type

Type of component objects that will be queried. Possible value depends on

container_type.

container_type	component_type
RAID_OBJ_TYPE_CONTROLLER	RAID_OBJ_TYPE_ARRAY, RAID_OBJ_TYPE_DISK,
RAID_OBJ_TYPE_ARRAY	RAID_OBJ_TYPE_ARRAY, RAID_OBJ_TYPE_ARRAYPART, RAID_OBJ_TYPE_TASK
RAID_OBJ_TYPE_DISK	RAID_OBJ_TYPE_DISKSEG, RAID_OBJ_TYPE_HSP RAID_OBJ_TYPE_PROP

For objects of the given type that can not be the component of container, this function must return 0.

This function returns 0 or a positive number on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.4.rdcfg_complist()

This function will query out the device id numbers of the components of the given type belonging to the specified container, and set the corresponding fields of passed attribute data structures with the device id numbers.

```
int rdcfg_complist(uint32_t controller_id, uint32_t container_id, raid_obj_type_id_t container_type, raid_obj_type_id_t comp_type, int comp_num, uint32_t *ids);
```

controller_id

Controller device id number.

container_id

The meaning of the field depends on **container_type**.

container_type	container_id
RAID_OBJ_TYPE_CONTROLLER	Not used;
RAID_OBJ_TYPE_ARRAY	The array_id;
RAID_OBJ_TYPE_DISK	The disk_id.

container_type

Specify the raid device type identified by combination of **controller_id** and **container_id**.

Could be one of the following:

- RAID_OBJ_TYPE_CONTROLLER
- RAID_OBJ_TYPE_ARRAY
- RAID_OBJ_TYPE_DISK

comp_type

Type of component objects that will be queried. Possible value depends on **container_type.**

container_type	comp_type
RAID_OBJ_TYPE_CONTROLLER	RAID_OBJ_TYPE_ARRAY, RAID_OBJ_TYPE_DISK,
RAID_OBJ_TYPE_ARRAY	RAID_OBJ_TYPE_ARRAY, RAID_OBJ_TYPE_ARRAYPART, RAID_OBJ_TYPE_TASK
RAID_OBJ_TYPE_DISK	RAID_OBJ_TYPE_DISKSEG, RAID_OBJ_TYPE_HSP RAID_OBJ_TYPE_PROP

comp_num

Number of components that will be queried;

ids

Buffer that will contain the id numbers of queried component objects.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.5.rdcfg_get_attr()

This function will query out the attributes of the specified RAID object and fulfill the passed data structure with these attributes.

int rdcfg_get_attr(uint32_t controller_id, uint32_t device_id, uint32_t index_id, raid_obj_type_id_t type, void *attr);

controller_id

Controller device id number.

device_id and **index_id**

Contents depends on **type** field.

Type	device_id	index_id
RAID_OBJ_TYPE_CONTROLLER	Not used	Not used
RAID_OBJ_TYPE_ARRAY	array_id	Not used
RAID_OBJ_TYPE_DISK	disk_id	Not used
RAID_OBJ_TYPE_HSP	disk_id	array_id
RAID_OBJ_TYPE_ARRAYPART	array_id	disk_id
RAID_OBJ_TYPE_DISKSEG	disk_id	seq_no number
RAID_OBJ_TYPE_TASK	task_id	Not used
RAID_OBJ_TYPE_PROP	disk_id	prop_id

type

Can be one of the following:

- RAID_OBJ_TYPE_CONTROLLER
- RAID_OBJ_TYPE_ARRAY
- RAID_OBJ_TYPE_DISK
- RAID_OBJ_TYPE_HSP
- RAID_OBJ_TYPE_ARRAYPART
- RAID_OBJ_TYPE_DISKSEG

- RAID_OBJ_TYPE_TASK
- RAID_OBJ_TYPE_PROP

attr

Buffer that will contain the object attributes. Depends on type.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.6.rdcfg_set_attr()

This function will set the attributes of the specified device.

int rdcfg_set_attr(uint32_t controller_id, uint32_t device_id, uint32_t cmd, uint32_t *value, char **err_ptr);

controller_id

Controller device id number.

device_id

Since presently only SET_CACHE_WR_PLY and SET_CACHE_RD_PLY sub-commands are defined, this field must contain the **array_id** for the target array..

cmd

The sub-command number used to identify the action. Currently only the following command had been defined:

- SET_CACHE_WR_PLY
- SET_ACTIVATION_PLY

value

Pointer to the parameter that will be used by sub-command.

For SET_CACHE_WR_PLY sub-command, following are possible values:

- CACHE_WR_OFF
- CACHE_WR_ON

For SET_ACTIVATION_PLY sub-command, following are possible values:

- ARRAY_ACT_ACTIVATE

err_ptr

Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.7.rdcfg_array_create()

This function will create an array belonging to the specified controller, or calculating the maximum capacity of the volume that can be created using the specified physical disks.

int rdcfg_array_create(uint32_t controller_id, array_attr_t *attr, int num, arraypart_attr_t *arraypart_attrs, char **err_ptr);

controller_id
Controller device id number.

attr
Pointer to array data structure; this data structure contains the attributes (stripe_size, read_policy, write_policy, raid_level, capacity) required for the array creation. The **array_id** field will be replaced with array_id of newly created array if operation succeeded.
If the **capacity** field has been set to OBJ_NONE_ATTR, this function will not create the target volume but will calculate the maximum capacity of the volume that can be created using the specified disks; and the **capacity** field should be set the value of the capacity that been calculated out by this function.

num
Indicates how many elements in the list pointed by **arraypart_attrs** are available.

arraypart_attrs
Array part objects that will be used to build the array. These array part objects must be organized in array hierarchical expression (see Appendix B).
If the offset attribute has been set to OBJ_ATTR_NONE, the driver/firmware should allocate disk space by itself and create the array.
If the size attribute has been set to OBJ_ATTR_NONE, the whole disk will be used to create the array.

err_ptr
Address of hardware-dependent error string buffer pointer.

If succeed, this function will fill the **array_id** field of pointed array data structure with the array id number of newly created array, and return 0; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

Note this function has a special feature: it’s also used to calculate the maximum capacity of the target volume; if this feature is not supported, this function must return

ERR_OP_NO_IMPL, and the common-lib will try to calculate the maximum capacity by itself.

5.2.8.rdcfg_array_delete()

This function will delete the specified array and release occupied disk space.

int rdcfg_array_delete(uint32_t controller_id, uint32_t array_id, char **err_ptr);

controller_id
Controller device id number.
Array_id
Array logical device id number.
err_ptr
Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.9.rdcfg_hsp_bind()

This function will set the specified physical disk(s) as HSP disk(s) for the specified array(s).

int rdcfg_hsp_bind(uint32_t controller_id, hsp_relation_t *hsp_relations, char **err_ptr);

controller_fd
File handle of the opened controller device.
hsp_relations
Pointer to a data structure array with each element defined as following:

```
typedef struct {
        uint32_t array_id;
        uint32_t disk_id;
} hsp_relation_t;
```

Each element indicates that the specified disk will be a HSP disk for the specified array. If the **array_id** equals to OBJ_ATTR_NONE, the corresponding disk will be set as a global HSP disk.

err_ptr

Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.10.rdcfg_hsp_unbind()

This function will convert the specified HSP disk(s) to un-assigned disk(s).

```
int rdcfg_hsp_unbind(uint32_t controller_id, hsp_relation_t *hsp_relations, char **err_ptr);
```

controller_id

Controller device id number.

hsp_relations

Pointer to a data structure array with each element defined as following:

```
typedef struct {
        uint32_t array_id;
        uint32_t disk_id;
  } hsp_relation_t;
```

Each element indicates that the specified disk will not be a HSP disk for the specified array. If the **array_id** equals to OBJ_ATTR_NONE, the corresponding disk will be converted into a un-assigned disk.

err_ptr

Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

5.2.11.rdcfg_update_fw()

This function will update the firmware of the controller.

int rdcfg_update_fw(uint32_t controller_id, char *buf, uint32_size, char **err_ptr);

controller_id
Controller device id number.
buf
Pointer for buffer containing the firmware image.
size
Size of the buffer in bytes.
err_ptr
Address of hardware-dependent error string buffer pointer.

This function returns 0 on successful completion; if return value is less than 0, the function failed and the return value contains error code, which is defined in Appendix A.

6. End User Interface

Based on the common library, RaidCfg will provide a CLI utility located in /usr/sbin. For compatibility reason, this utility had been named as raidctl, and will be compatible with existing raidctl(1M) utility in Solaris.

6.1. Overview

6.1.1. Functionalities

raidctl is a CLI tool capable of:

- Querying information available RAID controllers in current system;
- Querying information of physically attached disks;
- Querying information about available RAID arrays in current system;
- Creating and deleting array of several RAID levels, including 0, 1, 5, 10, and 50;
- Convert physical disks into a global HSP disks, or vice versa;
- Set physical disks as local HSP disks to specified array, or vice versa;
- Downloading firmware to specific controllers.

6.1.2. Compatibility

The raidctl utility from RaidCfg project will be compatible to existing raidctl(1M) utility in Solaris. This means the raidctl utility can interpret the same CLI options and parameters with existing raidctl(1M) utility, and the definition of return code are the same. The newly-added features will not affect the behavior of existing features.

However, since the raidctl utility has introduced some enhanced features, the output will be optimized for more detailed information. The raidctl utility will follow the same layout style of existing one to minimize the transferring effort, and corresponding training courses will be provided.

For detailed information about raidctl options, please refer the man page (section 6.2).

6.2. Man page

See the accompanied man page document.

7. Appendix A: Error Codes

A set of error codes has been defined in /usr/include/sys/raidcfg.h; this definition is used by all components of RaidCfg software. The defined error codes and their meaning are as following:

Error Code	Description
SUCCESS	Operation succeeded.
ERR_DRIVER_NOT_FOUND	Controller device file can not be found.
ERR_DRIVER_OPEN	Can not open controller device file.
ERR_DRIVER_LOCK	Controller is opened by another process.
ERR_DRIVER_CLOSED	Controller is not opened by current process.
ERR_DRIVER_ACROSS	Operation across multiple controllers.
ERR_ARRAY_LEVEL	Operation not support by this level of RAID volume.
ERR_ARRAY_SIZE	Capacity of array exceeded limitation.
ERR_ARRAY_STRIPE_SIZE	Illegal stripe size.
ERR_ARRAY_CACHE_POLICY	Illegal cache policy.
ERR_ARRAY_IN_USE	Array is in use and can not be deleted.
ERR_ARRAY_TASK	Array has background task and can not perform current operation.
ERR_ARRAY_CONFIG	Configuration over device node failed.
ERR_ARRAY_DISKNUM	Number of disks mis-match the array.

ERR_ARRAY_LAYOUT	Illegal array layout.
ERR_ARRAY_AMOUNT	Too many arrays.
ERR_DISK_STATE	Disk status is not OK for current operation.
ERR_DISK_SPACE	No enough disk space.
ERR_DISK_SEG_AMOUNT	Too many segments within a physical disk.
ERR_DISK_NOT_EMPTY	Disk has been used and can not perform required action.
ERR_DISK_TASK	Disk has background task and can not perform current operation.
ERR_TASK_STATE	Task state is not OK for current operation.
ERR_OP_ILLEGAL	Illegal operation.
ERR_OP_NO_IMPL	Operation is not implemented by plug-in or driver.
ERR_OP_FAILED	Operation in plug-in or driver failed.
ERR_DEVICE_NOENT	Device not found.
ERR_DEVICE_TYPE	Illegal type of device.
ERR_DEVICE_DUP	Device record duplicated.
ERR_DEVICE_OVERFLOW	Too many devices.
ERR_DEVICE_UNCLEAN	Device pool is not clean.
ERR_DEVICE_INVALID	Device record is invalid.
ERR_NOMEM	Can not allocate more memory space.
ERR_PRIV	No privilege.
ERR_PLUG_IN	Plug-in encounters an error.

8. Appendix B: Array Hierarchical Expression

This is a general expression about how disks are organized to form an array, especially for multi-level array (like RAID 10 and RAID 50). It's used to organize the parameters in API functions, SPI functions and IOCTL interfaces.

The basic idea is that since an array (either top-array or sub-array) is formed by a group of other objects (e.g, sub-arrays, array parts), we can bracket these objects as a group; for a multi-level array, this can be done recursively and thus we can use this method to represent the organization of any given array in RaidCfg software.

For example, for the API interface, we can specify an array formed by disk A, B and C like (A B C), and an RAID 10 array formed by disk A, B, C and D as ((A B)(C D)).

For RaidCfg common library API, plug-in SPI and IOCTL interfaces, two constants of RAID_SEP_BEGIN and RAID_SEP_END are defined to group array components, as following:

```
#define RAID_SEP_BEGIN   -1
#define RAID_SEP_END-2
```

The expression of array follows these regulations:

- Array layout is expressed as components embraced by RAID_SEP_BEGIN and RAID_SEP_END in common library API, plug-in SPI and IOCTL interfaces;
- For API interface, components are disk handles; and for SPI and IOCTL interfaces, components are array parts that will form the array.
- All disks or array parts should be in the same level in depth.

Following are examples for RAID 5 and RAID 10 regarding RaidCfg API:

RAID5
RAID_SEP_BEGIN
DISK1
DISK2
DISK3
RAID_SEP_END

RAID10
RAID_SEP_BEGIN
RAID_SEP_BEGIN
DISK1
DISK2
RAID_SEP_END
RAID_SEP_BEGIN
DISK1
DISK2
RAID_SEP_END
RAID_SEP_END

Array Hierarchical Expression

For plug-in SPI or IOCTL interfaces, the expression is similar to the above; the only difference is that, with RaidCfg API, each element in the above diagram is a **disk_id** representing the involved disks; while with the plug-in SPI and IOCTL interface, element is an array part data structure representing disk space used to build the array (note that for the separator element, **disk_id** field should be set to RAID_SEP_BEGIN or RAID_SEP_END).

9. Appendix C: API Examples

Following are examples about RaidCfg API routines. Too simplify the code, we only provide error-processing sample code with the first example and omitted it with other examples.

Variables in *Italic* style are regarded as passed parameters.

9.1. Example 1: Query all controller information

Parameters:

None

```
int handle;
int ret;
char *err, *hardware_err;
raidcfg_controller_t attr;

...
handle = raidcfg_list_head(OBJ_SYSTEM, OBJ_TYPE_CONTROLLER);
while (handle > 0) {
    ret = raidcfg_open_controller(handle, &hardware_err);
    if (ret < 0) {
        err = raidcfg_errstr(ret);
        print_err(err, hardware_err);
        break;
    }

    if ((ret = raidcfg_get_attr(handle, &attr)) < 0) {
        err = raidcfg_errstr(ret);
        print_err(err, NULL);
        raidcfg_close_controller(handle, &hardware_err);
        break;
    }

    ...

    raidcfg_close_controller(handle, &hardware_err);
    handle = raidcfg_list_next(handle);
}

...
```

9.2. Example 2: Query all disks for a given controller

Parameters:

controller_id: specify the target controller that will be queried.

```
int controller_handle, disk_handle;
int ret;
char *err, *hardware_err;
raidcfg_disk_t attr;

...
controller_handle = raidcfg_get_controller(controller_id);
if (controller_handle < 0)
	return;

if ((ret = raidcfg_open_controller(controller_handle, &hardware_err)) < 0)
	return;

disk_handle = raidcfg_list_head(controller_handle, OBJ_TYPE_DISK);
while (disk_handle > 0) {
	if ((ret = raidcfg_get_attr(disk_handle, &attr)) < 0)
		break;

	...

	disk_handle = raidcfg_list_next(disk_handle);
}

raidcfg_close_controller(controller_handle, &hardware_err);
```

9.3. Example 3: Query all available arrays for a given controller

Parameters:

controller_id: specify the target controller that will be queried.

```
int controller_handle, array_handle;
int ret;
char *err, *hardware_err;
raidcfg_array_t attr;

...
controller_handle = raidcfg_get_controller(controller_id);
if (controller_handle < 0)
	return;

if ((ret = raidcfg_open_controller(controller_handle, &hardware_err)) < 0)
	return;

array_handle = raidcfg_list_head(controller_handle, OBJ_TYPE_ARRAY);
while (array_handle > 0) {
	if ((ret = raidcfg_get_attr(array_handle, &attr)) < 0)
		break;

	...

	array_handle = raidcfg_list_next(array_handle);
}

raidcfg_close_controller(controller_handle, &hardware_err);
```

9.4. Example 4: Creating an single-layer array

Parameters:

controller_id: specify the target controller that will be queried;
disk_num: amount of disks that will be used to create the array;
disk_tags: buffer contains disk_tags (including bus, target_id, and lun number) for involved disks;
array_size: total capacity of the array that will be created;
raid_level: RAID level of the target array;
stripe_size: stripe size that will be used for the target array.

```
int controller_handle, array_handle, *disk_handles;
int ret, i;
union disk_tag_t tag;
char *err, *hardware_err;

...
controller_handle = raidcfg_get_controller(controller_id);
if (controller_handle < 0)
        return;

if ((ret = raidcfg_open_controller(controller_handle, &hardware_err)) < 0)
        return;

disk_handles = calloc(disk_num + 2, sizeof (int));
if (disk_handles == NULL)
        return;

for (i = 1; i < disk_num + 2; ++ i) {
        tag.cidl.bus = disk_tags[i-1].bus;
        tag.cidl.target_id = disk_tags[i-1].target_id;
        tag.cidl.lun = disk_tags[i-1].lun;
        disk_handles[i] = raidcfg_get_disk(controller_handle, tag);

        if (disk_handles[i] < 0) {
                raidcfg_close_controller(controller_handle, &hardware_err);
                return;
        }
}

disk_handles[0] = RAID_SEP_BEGIN;
disk_handles[disk_num + 1] = RAID_SEP_END;

array_handle = raidcfg_array_create(disk_num + 2, disk_handles,
        raid_level, &array_size, stripe_size, &hardware_err);

if (array_handle < 0) {
        err = raidcfg_errstr(array_handle);
        print_err(err, hardware_err);
        return;
}

...
```

9.5. Example 5: Deleting an array

Parameters:

controller_id: specify the target controller that will be queried;

array: specify the target array that will be deleted. This contains the target_id and lun of the array.

```
int controller_handle, array_handle;
int ret;
char *err, *hardware_err;

...
controller_handle = raidcfg_get_controller(controller_id);
if (controller_handle < 0)
	return;

if ((ret = raidcfg_open_controller(controller_handle, &hardware_err)) < 0)
	return;

array_handle = raidcfg_get_array(controller_handle, array.target_id, array.lun);
if (array_handle < 0) {
	raidcfg_close_controller(controller_handle, &hardware_err);
	return;
}

ret = raidcfg_array_delete(array_handle, &hardware_err);

if (array_handle < 0) {
	err = raidcfg_errstr(array_handle);
	print_err(err, hardware_err);
	return;
}

...
```