AIX Version 7.2

Coherent Accelerator Processor Interface (CAPI) programming

Note

Before using this information and the product it supports, read the information in “Notices” on page 29.

This edition applies to AIX Version 7.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this document

You can use the Coherent Accelerator Processor Interface (CAPI) to allow Field Programmable Gate Array (FPGA) based accelerators to access applications (user space) memory directly.

Highlighting

The following highlighting conventions are used in this document:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Bold highlighting also identifies graphical objects, such as buttons, labels, and icons that the you select.
Italics	Identifies parameters for actual names or values that you supply.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or text that you must type.

Case sensitivity in AIX

Everything in the AIX® operating system is case sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type LS, the system responds that the command `is not found`. Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

CAPI programming

You can use the Coherent Accelerator Processor Interface (CAPI) to allow Field Programmable Gate Array (FPGA) based accelerators to access applications (user space) memory directly.

Traditional FPGA-based accelerators perform direct memory access (DMA) transfers in a Peripheral Component Interconnect (PCI) stack to move data between the accelerators and the applications. CAPI provides a general-purpose framework that has a CAPI-based accelerator that can transfer data back and forth from the application memory without the requirement of DMA.

CAPI Flash adapter

Coherent Accelerator Processor Interface (CAPI) provides a high bandwidth, low latency path between external devices, the POWER8® core, and the system's open memory architecture. CAPI adapters are placed in the PCI Express (PCIe) x16 slots, and use the PCIe Gen3 adapter as an underlying transport mechanism.

The CAPI-capable devices can replace either application programs that can run programs running on a POWER8 core or provide custom acceleration implementations. CAPI Flash adapters remove the complexity of the I/O subsystem, so that an accelerator can operate as part of an application. It results in a code path reduction, because applications can interact with the Flash accelerator directly without using the operating system kernel.

CAPI Flash block library

The block library for the Coherent Accelerator Processor Interface (CAPI) Flash adapter provides user space interfaces to CAPI Flash disk at the block or sector level, bypassing the kernel for read and write I/O requests. The block library for the CAPI Flash adapter creates an interface for applications so that the applications need not access the low-level CAPI Flash adapter details.

In AIX operating system, the block library for the CAPI Flash adapter is `libcflsh_block.a`. On Linux® platform, this library is `libcflsh_block.so`.

cblk_init API

Purpose

Initializes the block library for the Coherent Accelerator Processor Interface (CAPI) Flash adapter.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX

int rc = cblk_int(void *arg, int flags)
```

Description

The **`cblk_init`** API initializes the block library for the CAPI Flash adapter. You must call the **`cblk_init`** API before you use any other API in the block library for the CAPI Flash adapter.

Parameters

arg
This parameter is not used currently. It is set to NULL.

flags
Specifies flags for initialization. The default value is 0.

Return values

0
The API completed successfully.

Nonzero value
An error occurred.

cblk_term API

Purpose

Cleans up the resources for the Coherent Accelerator Processor Interface (CAPI) Flash block library when the library is no longer used.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_term(void *arg, int flags)
```

Description

The `cblk_term` API removes the block library for the CAPI Flash adapter when it is not used.

Parameters

arg
This parameter is not used currently (set to NULL).

flags
Specifies flags for initialization. The default value is 0.

Return values

0
The API completed successfully.

Nonzero value
An error occurred.

cblk_open API

Purpose

Opens a collection of contiguous blocks that are called a *chunk* on a Coherent Accelerator Processor Interface (CAPI) Flash device that can complete I/O (read and write) operations. A chunk can be considered as a logical unit number (LUN) that provides access to sectors 0 - *n*-1, where *n* is the size of the chunk in sectors. If virtual LUNs are specified, the chunk is a subset of sectors on a physical LUN.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX

chunk_id_t chunk_id = cblk_open(const char *path, int max_num_requests, int mode,
uint64_t ext_arg, int flags)
```

Description

The **cblk_open** API creates a chunk on a CAPI Flash LUN. This chunk is used for I/O (`cblk_read` or `cblk_write`) requests. The returned `chunk_id` value is assigned to a specific path from a specific

adapter to the calling process. The underlying physical sectors that are used by a chunk are not directly visible to the users of the block layer when the CBLK_OPN_VIRT_LUN flag is set.

When the cblk_open API call completes successfully, a chunk ID that represents the created chunk instance is returned to the calling process that is used for future API calls.

Parameters

path
This parameter identifies the special file name for the CAPI disk. For example, /dev/hdisk1 (AIX) and /dev/sg0 (Linux).

max_num_requests
This parameter indicates the maximum number of commands that can be queued to the adapter for a specific chunk at a specific time. If this value is 0, the block layer chooses a default size. If the value that is specified is too large, the cblk_open request fails with an ENOMEM error value.

mode
This parameter specifies the access mode (O_RDONLY, O_WRONLY, or O_RDWR).

ext_arg
This parameter is not used currently.

flags
This parameter is a collection of the following bit flags:

CBLK_OPN_VIRT_LUN
This flag indicates that a virtual LUN is provisioned on a physical LUN. If this flag is not specified, direct access to the complete physical LUN is provided. This flag is valid only for temporary storage. When the cblk_close API is called, all data sectors for this chunk are released to be used by other operations.

CBLK_OPN_NO_INTRP_THREADS
This flag indicates that the cflash block library does not start any background threads for processing and extracting information about asynchronous completion of I/O requests from the CAPI adapter. The process that uses this library must either call the cblk_aresult library or the cblk_listio library to poll for completion of the I/O operations.

CBLK_OPN_SCRUB_DATA
This flag is valid only when the CBLK_OPN_VIRT_LUN flag is specified. This flag indicates that data on a virtual LUN must be cleared before the LUN can be reused by other operations. This flag is not currently supported for the AIX operating system.

CBLK_OPN_MPIO_FO
This flag is valid only for the AIX operating system. This flag indicates that the cflash block library uses Multipath I/O (MPIO) failover. One path is used for all I/O requests, unless path-specific errors are encountered. If these path errors occur, an alternative path is used, if available. To identify the paths for a CAPI Flash disk, run the **lspath -l hdiskN** command. This flag is not valid if the CBLK_OPN_VIRT_LUN, CBLK_OPN_RESERVE, or CBLK_OPN_FORCED_RESERVE flag is specified.

CBLK_OPN_RESERVE
This flag is valid only for the AIX operating system. This flag indicates that the cflash block library uses reserve policy attribute that is associated with the disk that establishes disk reservations. You cannot use this flag with the CBLK_OPN_MPIO_FO flag.

CBLK_OPN_FORCED_RESERVE
This flag is valid only for the AIX operating system. The behavior of this flag is the same as the CBLK_OPEN_RESERVE flag, except that when the device is opened for the first time, it breaks any unresolved disk reservations. You cannot use this flag with the CBLK_OPN_MPIO_FO flag.

Return values

NULL_CHUNK_ID
An error occurred.

cblk_close API

Purpose
Closes a collection of contiguous blocks that are called a chunk on a Coherent Accelerator Processor Interface (CAPI) Flash memory device that can complete I/O (read and write) operations.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_close(chunk_id_t chunk_id, int flags))
```

Description

The `cblk_close` API releases the blocks that are associated with a chunk to be reused by other operations. Before the blocks can be reused by other operations, the data blocks must be cleared to remove any user data if the `CBLK_OPN_SCRUB_DATA` flag was set in the corresponding `cblk_open` API that returned this `chunk_id` value.

Parameters

chunk_id
Handle for the chunk that is being closed and released for reuse.

flags
Collection of bit flags.

Return values

0
The API completed successfully.

Nonzero value
An error occurred.

cblk_get_lun_size API

Purpose

Returns the size (number of blocks) of the physical logical unit number (LUN) to which a specific chunk is associated.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_get_lun_size(chunk_id_t chunk_id, size_t *size, int flags))
```

Description

The `cblk_get_lun_size` API returns the number of blocks of the physical LUN that is associated with this chunk. To use the `cblk_get_lun_size` service, you must have completed the `cblk_open` API to receive a valid `chunk_id` value.

Parameters

chunk_id
Handle for the chunk for which physical LUN size must be returned.

size
Specifies the total number of 4K blocks for the physical LUN that is associated a specific chunk.

flags
Collection of bit flags.

Return values

0
The API completed successfully.

>0
An error occurred.

cblk_get_size API

Purpose

Returns the size (number of blocks) that is assigned to a specific chunk ID, which is a virtual logical unit number (LUN). That is, the CBLK_OPN_VIRT_LUN flag is specified for the cblk_open call that returned this chunk ID. This service is not valid for LUNs for which the CBLK_OPN_VIRT_LUN flag was not set when the chunks were opened by using the cblk_open API.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_get_size(chunk_id_t chunk_id, size_t *size, int flags))
```

Description

The cblk_get_size service returns the number of blocks allocated to a specific chunk. To use the cblk_get_size service, you must have completed the cblk_open API to receive a valid chunk_id value.

Parameters

chunk_id
Handle for the chunk for which the LUN size must be changed.

size
Specifies the number of 4K blocks for the LUN that is associated with a specific chunk.

flags
Collection of bit flags.

Return values

0
The API completed successfully.

>0
An error occurred.

cblk_set_size API

Purpose

Assigns size (number of blocks) to a specific chunk ID that is a virtual logical unit number (LUN). That is, the CBLK_OPN_VIRT_LUN flag is specified for the cblk_open call that returned this chunk ID. If the blocks are already assigned to this chunk ID, you can increase or decrease the size by specifying a larger

or smaller size. This service is not valid for LUNs for which the CBLK_OPN_VIRT_LUN flag was not set when the chunks were opened by using the cblk_open API.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_set_size(chunk_id_t chunk_id, size_t size, int flags))
```

Description

When you use the virtual LUNs, the cblk_set_size service allocates a number of blocks to a specific chunk. The cblk_set_size API must be called before the cblk_read or cblk_write calls this chunk. To use the cblk_set_size service and to receive a valid chunk_id value, the cblk_open call must be completed.

If blocks were originally assigned to this chunk, which were not reused after the cblk_set_size API allocates the new blocks to the same chunk, and if the CBLK_SCRUB_DATA_FLG flag is set in the **flags** parameter, the original blocks are cleared before they can be reused by other cblk_set_size operations.

After successful completion of the cblk_set_size API, the chunk can have logical block address (LBA) size, in the range 0 - 1, that can be read or written.

Parameters

chunk_id
Handle for the chunk for which the LUN size must be set.

size
Specifies the number of 4K blocks for the LUN that is associated a specific chunk.

flags
Collection of bit flags.

Return values

0
The API completed successfully.

>0
An error occurred.

cblk_get_stats API

Purpose

Returns statistics for a specific chunk ID.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
typedef struct chunk_stats_s {
uint64_t max_transfer_size;    /* Maximum transfer size in */
                               /* blocks of this chunk. */
uint64_t num_reads;            /* Total number of reads issued */
                               /* via cblk_read interface */
uint64_t num_writes;           /* Total number of writes issued */
                               /* via cblk_write interface */
uint64_t num_areads;           /* Total number of async reads */
                               /* issued via cblk_aread interface */
uint64_t num_awrites;          /* Total number of async writes */
                               /* issued via cblk_awrite interface*/
uint32_t num_act_reads;        /* Current number of reads active */
                               /* via cblk_read interface */
```

```
uint32_t num_act_writes;       /* Current number of writes active */
                               /* via cblk_write interface */
uint32_t num_act_areads;       /* Current number of async reads */
                               /* active via cblk_aread interface */
uint32_t num_act_awrites;      /* Current number of async writes */
                               /* active via cblk_awrite interface*/
uint32_t max_num_act_writes;   /* High water mark on the maximum */
                               /* number of writes active at once */
uint32_t max_num_act_reads;    /* High water mark on the maximum */
                               /* number of reads active at once */
uint32_t max_num_act_awrites;  /* High water mark on the maximum */
                               /* number of asyync writes active */
                               /* at once. */
uint32_t max_num_act_areads;   /* High water mark on the maximum */
                               /* number of asyync reads active */
                               /* at once. */
uint64_t num_blocks_read;      /* Total number of blocks read */
uint64_t num_blocks_written;   /* Total number of blocks written */
uint64_t num_errors;           /* Total number of all error */
                               /* responses seen */
uint64_t num_aresult_no_cmplt; /* Number of times cblk_aresult */
                               /* returned with no command */
                               /* completion */
uint64_t num_retries;          /* Total number of all commmand */
                               /* retries. */
uint64_t num_timeouts;         /* Total number of all commmand */
                               /* time-outs. */
uint64_t num_fail_timeouts;    /* Total number of all commmand */
                               /* time-outs that led to a command */
                               /* failure. */
uint64_t num_no_cmds_free;     /* Total number of times we didm't */
                               /* have free command available */
uint64_t num_no_cmd_room ;     /* Total number of times we didm't */
                               /* have room to issue a command to */
                               /* the AFU. */
uint64_t num_no_cmds_free_fail; /* Total number of times we didn't */
                               /* have free command available and */
                               /* failed a request because of this*/
uint64_t num_fc_errors;        /* Total number of all FC */
                               /* error responses seen */
uint64_t num_port0_linkdowns;  /* Total number of all link downs */
                               /* seen on port 0. */
uint64_t num_port1_linkdowns;  /* Total number of all link downs */
                               /* seen on port 1. */
uint64_t num_port0_no_logins;  /* Total number of all no logins */
                               /* seen on port 0. */
uint64_t num_port1_no_logins;  /* Total number of all no logins */
                               /* seen on port 1. */
uint64_t num_port0_fc_errors;  /* Total number of all general FC */
                               /* errors seen on port 0. */
uint64_t num_port1_fc_errors;  /* Total number of all general FC */
                               /* errors seen on port 1. */
uint64_t num_cc_errors;        /* Total number of all check */
                               /* condition responses seen */
uint64_t num_afu_errors;       /* Total number of all AFU error */
                               /* responses seen */
uint64_t num_capi_false_reads; /* Total number of all times */
                               /* poll indicated a read was ready */
                               /* but there was nothing to read. */
uint64_t num_capi_adap_resets; /* Total number of all adapter */
                               /* reset errors. */
uint64_t num_capi_afu_errors;  /* Total number of all */
                               /* CAPI error responses seen */
uint64_t num_capi_afu_intrpts; /* Total number of all */
                               /* CAPI AFU interrupts for command */
                               /* responses seen. */
uint64_t num_capi_unexp_afu_intrpts; /* Total number of all of */
                               /* unexpected AFU interrupts */
uint64_t num_active_threads;   /* Current number of threads */
                               /* running. */
uint64_t max_num_act_threads;  /* Maximum number of threads */
                               /* running simultaneously. */
uint64_t num_cache_hits;       /* Total number of cache hits */
                               /* seen on all reads */
} chunk_stats_t;
int rc = cblk_get_stats(chunk_id_t chunk_id, chunk_stats_t *stats, int flags))
```

Description

The `cblk_get_stats` service returns statistics for a specific chunk ID.

Parameters

chunk_id
Handle for the chunk for which the statistics must be determined.

stats
Specifies the address of the `chunk_stats_t` structure.

flags
Collection of bit flags.

Return values

0
The API completed successfully.

>0
An error occurred.

cblk_read API

Purpose

Reads 4K blocks from the chunk at the specified logical block address (LBA) into the specified buffer. When you use virtual logical unit numbers (LUNs), this LBA is not the same as the LUN's LBA because the chunk does not always start at the LUN's LBA, 0.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_read(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int flags));
```

Description

The `cblk_read` service reads data from the chunk and places that data into the supplied buffer. This call is blocked until the read operation completes with success or error. It means that this call will not return until the read operation completes. In case of a virtual LUN, you must call the `cblk_set_size` API before the `cblk_read`, `cblk_write`, `cblk_aread`, or `cblk_awrite` calls to a specific chunk.

Parameters

chunk_id
Handle for the chunk that is being read.

buf
Specifies the buffer to which data is read into from the chunk.

lba
Specifies the LBA (4K offset) inside the chunk.

nblocks
Specifies the size of the transfer in 4K sectors. For a physical LUN, the upper limit is 16 MB. For a virtual LUN, the upper limit is 4K.

flags
Collection of bit flags.

Return values

-1
Indicates an error. An error number is set for more details.

0
Indicates that no data was read.

***n* > 0**
Indicates that the read operation is successful, where *n* is the number of blocks read.

cblk_write API

Purpose

Writes 4K blocks to the chunk at the specified logical block address (LBA) by using the data from the specified buffer. When you use virtual logical unit numbers (LUNs), this LBA is not the same as the LUN's LBA because the chunk does not start at LBA 0.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
int rc = cblk_write(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int flags));
```

Description

The `cblk_write` API writes data from the specified buffer to the chunk at the specified LBA. The `cblk_write` call is blocked until the write operation completes with success or an error. It means that this call will not return until the write operation completes. In case of a virtual LUN, you must call the `cblk_set_size` API before you call the `cblk_write` API to a specific chunk.

Parameters

chunk_id
Handle for the chunk that is being written.

buf
Specifies the buffer of data that is written to the chunk.

lba
Specifies the LBA (4K offset) inside the chunk.

nblocks
Specifies the size of the transfer in 4K sectors. For a physical LUN, the upper limit is 16 MB. For a virtual LUN, the upper limit is 4K.

flags
Collection of bit flags.

Return values

-1
Indicates an error. An error number is set for more details.

0
Indicates that no data was written.

***n* > 0**
Indicates that the write operation is successful, where *n* is the number of blocks written.

cblk_aread API

Purpose

Reads 4K blocks from the chunk at the specified logical block address (LBA) into the specified buffer. When you use virtual logical unit numbers (LUNs), this LBA is not the same as the LUN's LBA because the chunk does not start at LBA 0.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX


typedef enum {
    CBLK_ARW_STATUS_PENDING = 0,     /* Command has not completed */
    CBLK_ARW_STATUS_SUCCESS = 1      /* Command completed successfully */
    CBLK_ARW_STATUS_INVALID = 2      /* Caller's request is invalid */
    CBLK_ARW_STATUS_FAIL = 3         /* Command completed with an error */
} cblk_status_type_t;

typedef struct cblk_arw_status_s {
    cblk_status_type_t status;       /* Status of the command */
                                     /* See errno field for additional */
                                     /* details about the failure */
    size_t blocks_transferred;       /* Number of block transferred by */
                                     /* this reqeuest. */
    int errno;                       /* Errno when status indicates */
                                     /* CBLK_ARW_STAT_FAIL */
} cblk_arw_status_t;

int rc = cblk_aread(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int
*tag, cblk_arw_status_t *status, int flags));
```

Description

The `cblk_aread` service reads data from the chunk and places that data into the supplied buffer. This call is not blocked until the read operation is completed. It means that this call returns immediately after the request is issued, before the read operation might be complete. A subsequent `cblk_aresult` call must be invoked to poll on completion. In case of a virtual LUN, you must call the `cblk_set_size` API before you call the `cblk_aread` API.

Parameters

chunk_id
Handle for the chunk that is being read.

buf
Specifies the buffer to which data is read into from the chunk.

lba
Specifies the LBA (4K offset) inside the chunk.

nblocks
Specifies the size of the transfer in 4K sectors. For a physical LUN, the upper limit is 16 MB. For a virtual LUN, the upper limit is 4K.

tag
Specifies the returned identifier so that you can uniquely identify each command that was issued.

status
Specifies the address that is provided by the calling process, which is updated by the `capiblock` library when the `cblk_aread` API completes. Applications can use the polling process for the **`status`** argument instead of using the `cblk_aresult` service.

The CAPI adapter cannot update this field directly. Software threads are required to update the `status` parameter. This field is not used if the `CBLK_OPN_NO_INTRP_THREADS` flag was specified for the `cblk_open` API that returned this `chunk_id` value.

flags
Collection of the following bit flags:

CBLK_ARW_WAIT_CMD_FLAGS
Blocks the `cblk_aread` service until a free command is available to issue the request. Otherwise, this service can return a value of -1 with an error value of EWOULDBLOCK (if there is no free command currently available).

CBLK_ARW_USER_TAG_FLAGS
Indicates that the calling process is specifying a user-defined tag for this request. Then, the caller must use this tag with the `cblk_aresult` API and set its `CBLK_ARESULT_USER_TAG` flag.

CBLK_ARW_USER_STATUS_FLAG
Indicates that the calling process set the **`status`** parameter that will be updated when the command completes.

Return values

-1
Indicates an error. An error number is set for more details.

0
Indicates that this API is successful.

***n* > 0**
Indicates that the read operation is complete (possibly from cache), where *n* is the number of blocks read.

cblk_awrite API

Purpose

Writes 4K blocks to the chunk at the specified logical block address (LBA) by using the data from the specified buffer. When you use virtual logical unit numbers (LUNs), this LBA is not the same as the LUN's LBA because the chunk does not start at LBA 0.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX

typedef enum {
       CBLK_ARW_STAT_NOT_ISSUED = 0  /* Command has not been issued */
       CBLK_ARW_STAT_PENDING = 1     /* Command has not completed */
       CBLK_ARW_STAT_SUCCESS = 2     /* Command completed successfully */
       CBLK_ARW_STAT_FAIL = 3        /* Command completed with error */
} cblk_status_type_t;

typedef struct cblk_arw_status_s {
       cblk_status_type_t status;    /* Status of command */
                                     /* See errno field for additional */
                                     /* details about the failure */
       size_t blocks_transferred;    /* Number of block transferred by */
                                     /* this reqeuest. */
       int errno;                    /* Errno when status indicates */
                                     /* CBLK_ARW_STAT_FAIL */
} cblk_arw_status_t;

int rc = cblk_awrite(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int
*tag, cblk_arw_status_t *status, int flags));
```

Description

The `cblk_awrite` API writes data from the specified buffer to the chunk at the specified LBA. This call is not blocked until the write operation is completed. It means that this call returns immediately after the request is issued, before the write operation might be complete. A subsequent `cblk_aresult` call must

be invoked to poll on completion. In case of a virtual LUN, you must call the `cblk_set_size` API before you call the `cblk_awrite` API.

Parameters

chunk_id
Handle for the chunk that is being written.

buf
Specifies the buffer of data that is written to the chunk.

lba
Specifies the LBA (4K offset) inside the chunk.

nblocks
Specifies the size of the transfer in 4K sectors. For a physical LUN, the upper limit is 16 MB. For a virtual LUN, the upper limit is 4K.

tag
Specifies the returned identifier so that you can uniquely identify each command that was issued.

status
Specifies the address that is provided by the calling process, which the `capiblock` library updates when the `cblk_aread` API completes. The `cblk_aread` API can be used by an application in place of using the `cblk_aresult` service.

The CAPI adapter cannot update this field directly. It requires software threads to update the status region. This field is not used if the `CBLK_OPN_NO_INTRP_THREADS` flag was specified for the `cblk_open` API that returned this `chunk_id` value.

flags
Collection of the following bit flags:

CBLK_ARW_WAIT_CMD_FLAGS
Blocks the `cblk_aread` service to wait for a free command to issue the request. Otherwise, this service can return a value of -1 with an error value of EWOULDBLOCK (if there is no free command currently available).

CBLK_ARW_USER_TAG_FLAGS
Indicates that the calling process is specifying a user-defined tag for this request. Then, the calling process must use this tag with the `cblk_aresult` API and set its `CBLK_ARESULT_USER_TAG` flag.

CBLK_ARW_USER_STATUS_FLAG
Indicates that the calling process set the **`status`** parameter that will be updated when the command completes.

Return values

-1
Indicates an error. An error number is set for more details.

0
Indicates that this API is successfully issued.

***n* > 0**
Indicates that the read operation is complete, where *n* is the number of blocks written.

cblk_aresult API

Purpose

Returns status and completion information for asynchronous requests.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
rc = cblk_aresult(chunk_id_t chunk_id, int *tag, uint64_t *status, int flags);
```

Description

The `cblk_aresult` API returns the status of the pending requests that are issued by using the `cblk_aread` or `cblk_awrite` APIs. If these pending requests are complete, this API returns the completion information.

Parameters

chunk_id
Handle for the chunk that is being written.

tag
Pointer to tag the calling process that it is waiting for the request completion. If the `CBLK_ARESULT_NEXT_TAG` flag is set, this field returns the tag for the next asynchronous request completion.

status
The pointer to the status. The status is returned when a request completes.

flags
Specifies the following flags to the `cblk_aresult` API:

CBLK_ARESULT_BLOCKING
Specify this flag if you want the `cblk_aresult` API to be blocked until a command completes (provided active commands exist). If the `CBLK_ARESULT_NEXT_TAG` flag is specified, this call returns after any asynchronous I/O request completes.

CBLK_ARESULT_USER_TAG
Specify this flag to check the status of an asynchronous request that was issued with a user-specified tag.

CBLK_ARESULT_NEXT_TAG
Specify this flag if you want the `cblk_aresult` API to return when the next active asynchronous command completes.

Return values

-1
Indicates an error. An error number is set for more details.

0
Indicates that this API is successfully issued.

***n* > 0**
Indicates that the request is complete, where *n* is the number of blocks that are read or written.

cblk_clone_after_fork API

Purpose

Designates a child process to access the same virtual logical unit number (LUN) as the parent process. This service is valid only for the Linux platform.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX
rc = cblk_clone_after_fork(chunk_id_t chunk_id, int mode, int flags);
```

Description

The `cblk_clone_after_fork` service designates a child process to access data from the parent process. The child process must perform this operation immediately after the **`fork()`** system call by using the parent's chunk ID to access that storage. If the child process does not perform this operation, the child process will not have any access to the parent's chunk IDs. This service is not valid for physical LUNs.

Note: This service is valid only for the Linux platform.

Parameters

chunk_id
Handle for the chunk that is in use by the parent process. If this call returns successfully, this chunk ID can also be used by the child process.

mode
Specifies the access mode for the child process (`O_RDONLY`, `O_WRONLY`, or `O_RDWR`).

Note: The child processes cannot have greater access than the parent process. The descendant processes can have less access.

flags
This parameter is a bit flag that is specified by the calling process.

Return values

0
Indicates that the request completed successfully.

-1
Indicates an error. An error number is set for more details.

cblk_listio API

Purpose

Issues multiple I/O requests to Coherent Accelerator Processor Interface (CAPI) Flash disk with a single call and waits for the completion of multiple I/O requests from a CAPI Flash disk.

Syntax

```
#include <capiblock.h> for Linux or <sys/capiblock.h> for AIX

typedef struct cblk_io {
      uchar version;                /* Version of the structure */
#define CBLK_IO_VERSION_0 "I"       /* Initial version 0 */
      int flags;                    /* Flags for request */
#define CBLK_IO_USER_TAG 0x0001     /* Caller is specifying a user defined */
                                    /* tag. */
#define CBLK_IO_USER_STATUS 0x0002  /* Caller is specifying a status location */
                                    /* to be updated */
#define CBLK_IO_PRIORITY_REQ 0x0004 /* This is (high) priority request that */
                                    /* must be expediated vs non-priority */
                                    /* requests */
      uchar request_type;           /* Type of request */
#define CBLK_IO_TYPE_READ 0x01      /* Read data request */
#define CBLK_IO_TYPE_WRITE 0x02     /* Write data request */
      void *buf;                    /* Data buffer for the request */
      offset_t lba;                 /* Starting Logical block address for */
                                    /* the request. */
      size_t nblocks;               /* Size of request based on number of */
                                    /* blocks. */
      int tag;                      /* Tag for the request. */
      cblk_arw_status_t stat;       /* Status of the request */
} cblk_io_t

int rc = cblk_listio(chunk_id_t chunk_id,cblk_io_t *issue_io_list[],int
```

```
issue_io_items,cblk_io_t *pending_io_list[],int pending_io_items, cblk_io_t
*wait_io_list[], int wait_items, cblk_io_t *completion_io_list[],int
*completion_items, uint64_t timeout, int flags));
```

Description

The `cblk_listio` service provides an interface to issue multiple I/O requests with a single call and poll for completion of multiple I/O requests by using a single call. The individual requests are specified by the `cblk_io_t` type, which includes a data buffer, starting logical block address (LBA), and a transfer size in 4K blocks.

This service can update the I/O requests that are associated with the `cblk_io_t` type (that is, update status, tags, and flags based on disposition of the I/O request).

This service cannot be used to check for the completion of I/O requests issued through the `cblk_aread` or `cblk_awrite` APIs.

Parameters

chunk_id
Handle for the chunk that is associated with the I/O requests.

issue_io_list
This parameter specifies an array of I/O requests to issue to CAPI Flash disks. Each individual array element of type `cblk_io_t` specifies an individual I/O request that contains the data buffer, starting LBA, and a transfer size in 4K blocks. These array elements can be updated by this API to indicate completion status and tags. The status field of the individual `cblk_io_t` array elements are initialized by this API. If the **issue_io_list** parameter is null, this API can be used to wait for completion of other requests that are issued by the previous `cblk_listio` calls by setting the **pending_io_list** parameter.

issue_io_items
Specifies the number of array elements in the **issue_io_list** array.

pending_io_list
Specifies an array of I/O requests that were issued through a previous `cblk_listio` request. You can use the **pending_io_list** parameter to poll for I/O request completion, without waiting for completion of all the requests (that is, setting the **completion_io_list** parameter).

pending_io_items
Specifies the number of array elements in the **pending_io_list** array.

wait_io_list
Specifies the array of I/O requests, for which the `cblk_listio` service is blocked until the I/O requests complete. These I/O requests must also be specified in either the **issue_io_list** parameter or the **pending_io_list** parameter. If an I/O request in the **issue_io_list** array fails to be issued because of invalid settings by the calling process or no resources, that I/O request's elements in the `io_list` is updated to indicate this failure (status is set as CBLK_ARW_STAT_NOT_ISSUED) and the `cblk_listio` API does not wait for that I/O request completion. Thus, all I/O requests in the **wait_io_list** array that completed will have a status of CBLK_ARW_STAT_SUCCESS or CBLK_ARW_STAT_FAIL. The status is not updated for I/O requests that are not complete.

wait_items
Specifies the number of array elements in the **wait_io_list** array.

completion_io_list
This parameter is set by the calling process to an initialized (zeroed) array of I/O requests and the **completion_items** parameter is set to the number of array elements in the array. When the `cblk_listio` API returns, the array contains I/O requests that are specified in the **issue_io_list** and **pending_io_list** parameters that were completed by the CAPI device but not specified in the **wait_io_list** parameter. If an I/O request in the `io_list` array fails to be issued because of invalid settings by the calling process or no resources, that I/O request's element is not copied to the **completion_io_list** parameter and its status in the `io_list` array is updated to indicate this

failure (status is set as CBLK_ARW_STAT_NOT_ISSUED). Thus, all I/O requests that are returned in this list will have a status of CBLK_ARW_STAT_SUCCESS or CBLK_ARW_STAT_FAIL.

completion_items
This parameter is set by the calling process to the address of the number of array elements that this API placed in the **completion_io_list** parameter. When this API returns, the value of this parameter is updated to the number of I/O requests that is placed in the **completion_io_list** parameter.

timeout
Specifies the timeout value in microseconds to wait for all I/O requests in the **wait_io_list** parameter. This parameter is valid only if the **wait_io_list** parameter is not null. If any of the I/O requests in the **wait_io_list** parameter do not complete within the timeout value, this API returns a value of -1 and sets the error number to a value ETIMEDOUT (when this error occurs, some commands might have completed in the **wait_io_list** parameter). Thus, the calling process must check each request in the **wait_io_list** parameter to determine which requests are completed. The calling process must remove the completed items from the **pending_io_list** parameter before the next invocation of this API. A timeout value of 0 indicates that this API is blocked until requests in the **wait_io_list** parameter are completed.

flags
Specifies the following bit flag:

CBLK_LISTIO_WAIT_ISSUE_CMD
Blocks the `cblk_listio` API until a free command is available to issue all requests even if the timeout value is exceeded and the `CBLK_LISTIO_WAIT_CMD_FLAG` flag is set. Otherwise, this service can return a value of -1 with an error value of EWOULDBLOCK if free commands are currently not available (for this situation, some commands might have successfully queued to the issued list. The calling process must examine the individual I/O requests in the **issue_io_list** parameter to determine which requests failed.)

Return values

-1
Error and error number are set for more details.

0
This API completed successfully without any errors.

CAPI Flash key-value library

The key-value library provides an interface to Coherent Accelerator Processor Interface (CAPI) Flash devices to store, retrieve, and manage arrays. The key-value library maps the key-value semantics to the CAPI Flash block library.

In the AIX operating system, the key-value library is `libarkdb.a`. On the Linux platform, this library is `libarkdb.so`.

ark_create API

Purpose
Creates a key-value store instance.

Syntax

```
int ark_create(path, ark, flags)
char * file;
ARK ** handle;
uint64_t flags;
```

Description

The `ark_create` API creates a key-value store instance on the host system.

The **path** parameter can be used to specify the special file (for example, the `/dev/sdx` file for the Linux platform or the `/dev/hdiskx` file for AIX operating system) that represents the physical logical unit number (LUN) created on the Flash storage. If the **path** parameter is not a special file, the API assumes that the file must be used for the key-value store. If the file does not exist, the file is created. If the **path** parameter is NULL, memory is used for the key-value store.

The **flags** parameter indicates the properties of the key-value store. If you want to specify a special file for the physical LUN, you can specify whether to use the existing key-value store in the physical LUN or to create the key-value store in a virtual LUN. By default, the entire physical LUN is used for the key-value store. If a virtual LUN is required, the `ARK_KV_VIRTUAL_LUN` bit flag must be set in the **flags** parameter.

A key-value store that is configured to use the entire physical LUN can be persisted. You can shut down a key-value store instance by using the persistence (that is, saving the current state as data) of a key-value store, and then you can open the same physical LUN and load the previous key-value store instance in the same state as it was when it closed. To configure a key-value store instance to be persisted when the key-value store instance is shut down (`ark_delete`), set the `ARK_KV_PERSIST_STORE` bit in the **flags** parameter. By default, a key-value store instance is not configured to be persisted. To load the persisted key-value store instance that is stored on the physical LUN, set the `ARK_KV_PERSIST_LOAD` bit in the **flags** parameter. By default, the persisted instance, if present, is not loaded and is overwritten by any new persisted data.

Only those key-value stores can be persisted that are stored on physical LUNs.

Upon successful completion, the **handle** parameter represents the newly created key-value store instance that is used for future API calls.

Parameters

path
Specifies the CAPI adapter, file, or memory for the key-value store.

ark
Specifies the handle representing the key-value store.

flags
Collection of the following bit flags to determine the properties of the key-value store:

ARK_KV_VIRTUAL_LUN
Specifies the key-value store to use a virtual LUN created from the physical LUN represented by the special file.

ARK_KV_PERSIST_STORE
Configures the key-value store instance to be persisted upon shutdown of the key-value store instance. You can shut down or delete a key-value store instance by using the `ark_delete` API.

ARK_KV_PERSIST_LOAD
Loads the stored configuration if persistence data is present on the physical LUN.

Return values

0
Indicates successful completion. The **handle** parameter points to the newly created key-value store instance.

EINVAL
Invalid value for one of the parameters.

ENOSPC
Not enough memory or Flash storage.

ENOTREADY
System is not ready for key-value store configuration.

ark_delete API

Purpose
Deletes a key-value store instance.

Syntax

```
int ark_delete(ark)
ARK *ark;
```

Description

The `ark_delete` API deletes a key-value store instance that is specified by the **ark** parameter on the host system. On successful completion, all associated memory and storage resources are released. And, if the ARK instance is configured to persist, the configuration is persisted so that the instance can be loaded later.

Parameters

ark
A handle that represents the key-value store instance.

Return values

Upon successful completion, the `ark_delete` API will clean and remove all resources associated with the key-value store instance and return 0. If unsuccessful, the `ark_delete` API will return one of the following non-zero error code:

0
The API completed successfully. All resources associated with the key-value store instance are removed.

EINVAL
Key-value store handle is not valid.

Nonzero value
An error occurred and the API did not complete successfully.

ark_set, ark_set_async_cb API

Purpose
Writes a key-value pair.

Syntax

```
int ark_set(ark, klen, key, vlen, val, res)
int ark_set_async_cb(ark, klen, key, vlen, val, callback, dt)

ARK * ark;
uint64_t klen;
void * key;
uint64_t vlen;
void * val;
void *(*callback)(int errcode, uint64_t dt, uint64_t res);
uint64_t dt;
```

Description

The `ark_set` API stores the key and value into the store for the key-value store instance that is represented by the **ark** parameter. The `ark_set_async_cb` API operates in an asynchronous mode, in

which the API immediately returns to the calling process and the operation is scheduled to run. After the operation is performed, the `callback` function is called to notify the calling process about the operation completion.

For a key-value store instance, if the key is present, the stored value is replaced with the `val` value.

On successful completion, the key-value pair is written in the store and the number of bytes written to the key-value store is returned to the calling process through the **`res`** parameter.

Parameters

ark
Indicates a handle that represents the connection for the key-value store instance.

key
Specifies the key for the key-value pair.

klen
Indicates the length of the key in bytes.

val
Specifies the value for the key-value pair.

vlen
Indicates the length of the value in bytes.

res
Indicates the number of bytes that are written to the key-value store on successful completion of the I/O operation.

callback
Specifies the function to call on completion of the I/O operation.

dt
Indicates a 64-bit value to tag an asynchronous API call.

Return values

On successful completion, the `ark_set` and `ark_set_async_cb` APIs write the key-value in the store associated with the key-value store instance and return the number of bytes written. The return value of the `ark_set` API indicates the status of the operation. The return value of the `ark_set_async_cb` API indicates whether the asynchronous operation was accepted or rejected. The status is stored in the **`errcode`** parameter when the `callback` function is run. If the API is unsuccessful, the `ark_set` and `ark_set_async_cb` APIs return one of the following nonzero error codes:

EINVAL
Invalid parameters.

ENOSPC
Not enough space is remaining in the key-value store.

ark_get, ark_get_async_cb API

Purpose

Retrieves a value for a specific key.

Syntax

```
int ark_get(ark, klen, key, vbuflen, vbuf, voff, res)
int ark_get_async_cb(ark, klen, key, vbuflen, vbuf, voff, callback, dt)

ARK * ark;
uint64_t klen;
void * key;
uint64_t vbuflen;
void * vbuf;
```

```
uint64_t voff;
void *(*callback)(int errcode, uint64_t dt, uint64_t res);
uint64_t dt;
```

Description

The `ark_get` and `ark_get_async_cb` APIs query the key-value store associated with the **ark** parameter for a specific **key** parameter. If the key is found, the key's value is returned in the **vbuf** parameter with a maximum of **vbuflen** bytes written in the key-value store starting at the **voff** offset parameter in the key's value. The `ark_get_async_cb` API operates in an asynchronous mode, in which the API immediately returns to the calling process and the retrieval operation is scheduled to run. After the operation is completed, the `callback` function is called to notify the calling process about the completion of the operation.

If the API is successful, the length of the key's value is stored in the **res** parameter of the `callback` function.

Parameters

ark
Indicates a handle that represents the connection for the key-value store instance.

key
Specifies the key for the key-value pair.

klen
Indicates the length of the key in bytes.

vbuf
Specifies the buffer to store the key's value for the key-value pair.

vbuflen
Specifies the length of the **vbuf** buffer.

voff
Specifies the offset value in the key to start the read operation.

res
Stores the size of the key in bytes if the `ark_get` API completes successfully.

callback
Specifies the `callback` function to be called when the I/O operation completes.

dt
Specifies a 64-bit value to tag an asynchronous API call.

Return values

On successful completion, the `ark_get` and `ark_get_async_cb` APIs return 0. The return value of the `ark_get` API indicates the status of the operation. The return value of the `ark_get_async_cb` API indicates whether the asynchronous operation was accepted or rejected. The status of the asynchronous API is stored in the **errcode** parameter of the `callback` function. If unsuccessful, the `ark_get` and `ark_get_async_cb` APIs return one of the following nonzero error codes:

EINVAL
Invalid parameters.

ENOENT
Key not found.

ENOSPC
Not enough space in the memory buffer to store the key's value.

ark_del, ark_del_async_cb API

Purpose

Deletes the value associated with a specific key.

Syntax

```
int ark_del(ark, klen, key, res)
int ark_del_async_cb(ark, klen, key, callback, dt)

ARK * ark
uint64_t klen;
void * key;
void *(*callback)(int errcode, uint64_t dt, uint64_t res);
uint64_t dt;
```

Description

The `ark_del` and `ark_del_async_cb` APIs query the key-value store associated with the **handle** parameter for a specific **key** parameter. If the key is found, the `ark_del` API deletes the value from the key-value store. The `ark_del_async_cb` API operates in an asynchronous mode, in which the API immediately returns to the calling process and the removal operation is scheduled to run. After the operation is completed, the `callback` function is called to notify the calling process about the operation completion.

If the API is successful, the length of the key's value is returned to the calling process in the **res** parameter of the `callback` function.

Parameters

ark
Indicates a handle that represents the connection for the key-value store instance.

key
Specifies the key for a key-value pair.

klen
Indicates the length of the key in bytes.

res
Stores the size of the key in bytes if this API completes successfully.

callback
Specifies the `callback` function to be called when the I/O operation is completed.

dt
Specifies a 64-bit value to tag an asynchronous API call.

Return values

On successful completion, the `ark_del` and `ark_del_async_cb` APIs return the value of 0. The return value of the `ark_del` API indicates the status of the operation. The return value of the `ark_del_async_cb` API indicates whether the asynchronous operation was accepted or rejected. The status of the asynchronous API is stored in the **errcode** parameter of the `callback` function. If unsuccessful, the `ark_del` and `ark_del_async_cb` APIs return one of the following non-zero error codes:

EINVAL
Invalid parameters.

ENOENT
Key not found.

ark_exists, ark_exists_async_cb API

Purpose

Queries the key-value store to check whether a specific key is present.

Syntax

```
int ark_exist(ark, klen, key, res)
int ark_exist_async_cb(ark, klen, key, callback, dt)

ARK * ark
uint64_t klen;
void * key;
void *(*callback)(int errcode, uint64_t dt, uint64_t res);
uint64_t dt;
```

Description

The `ark_exists` and `ark_exists_async_cb` APIs query the key-value store associated with the **`ark`** parameter for a specific **key** parameter. If the key is found, the `ark_exist` API returns the size of the value in bytes in the **`res`** parameter. The key and its value are not altered. The `ark_exists_async_cb` API operates in an asynchronous mode, in which the API immediately returns to the calling process and the querying operation is scheduled to run. After the operation is run, the `callback` function is called to notify the calling process about the operation completion.

Parameters

ark
Indicates a handle that represents the connection for the key-value store instance.

key
Specifies the key for a key-value pair.

klen
Indicates the length of the key in bytes.

res
Stores the size of the key in bytes if the API completes successfully.

callback
Specifies the `callback` function to be called on completion of the I/O operation.

dt
Specifies a 64-bit value to tag an asynchronous API call.

Return values

On successful completion, the `ark_exists` and `ark_exists_async_cb` APIs return the value of 0. The return value of the `ark_exists` API indicates the status of the operation. The return value of the `ark_exists_async_cb` API indicates whether the asynchronous operation was accepted or rejected. The status of the asynchronous API is stored in the **`errcode`** parameter of the `callback` function. If unsuccessful, the `ark_exists` and `ark_exists_async_cb` APIs return one of the following non-zero error codes:

EINVAL
Invalid parameters.

ENOENT
Key not found.

ark_first API

Purpose

Returns the first key that is found in the key-value store and returns the handle to iterate through the key-value store.

Syntax

```
ARI*ark_first(ark, kbuflen, klen, kbuf)
ARK * ark
uint64_t buflen;
int64_t *klen;
void * kbuf;
```

Description

The `ark_first` API returns the first key found in the key-value store in the **kbuf** buffer and the size of the key in the **klen** parameter, when the key size (**klen**) is less than the **kbuf** size (**kbuflen**).

If this API completes successfully, an iterator handle is returned to the calling process that must be used to retrieve the next key in the key-value store by calling the `ark_next` API.

Parameters

ark
Indicates a handle that represents the connection for the key-value store instance.

kbuflen
Indicates the length of the **kbuf** parameter.

klen
Specifies the size of the key returned in the **kbuf** parameter.

kbuf
Specifies the buffer to hold the key.

Return values

On successful completion, the `ark_first` API returns a handle that must be used to iterate through the key-value store on subsequent calls by using the `ark_next` API. If unsuccessful, the `ark_first` API returns NULL with error number set to one of the following values:

EINVAL
Invalid parameters.

ENOSPC
The **kbuf** parameter has insufficient space to store the key.

ark_next API

Purpose

Returns the next found key in the key-value store.

Syntax

```
int ark_next(iter, kbuflen, klen, kbuf)
ARK * iter
uint64_t buflen;
int64_t *klen;
void *kbuf;
```

Description

The `ark_next` API returns the next key found in the key-value store based on the iterator handle, **`iter`**, in the **`kbuf`** buffer, and the size of the key in the **`klen`** parameter, while the key size (**`klen`**) is less than the **`kbuf`** size (**`kbuflen`**).

If successful, a handle is returned to the calling process that must be used to retrieve the next key in the key-value store by calling the `ark_next` API. If the end of the key-value store is reached, the ENOENT error code is returned.

Note: Because of the dynamic nature of the store, some of the written keys might not be returned.

Parameters

iter
Specifies the iterator handle where to begin the search in the key-value store.

kbuf
Specifies the buffer to hold the key.

kbuflen
Indicates the length of the **`kbuf`** parameter.

klen
Specifies the size of the key returned in the **`kbuf`** parameter.

Return values

On successful completion, the `ark_next` API returns a handle that must be used to iterate through the key-value store on subsequent calls by using the `ark_next` API. If unsuccessful, the `ark_next` API returns one of the following values:

EINVAL
Invalid parameter.

ENOENT
End of the store is reached.

ark_allocated API

Purpose

Returns the number of bytes that are allocated to the store.

Syntax

```
int ark_allocated(ark, size)
ARK * ark;
uint64_t *size;
```

Description

The `ark_allocated` API returns the number of bytes allocated to the key-value store through the **`size`** parameter.

Parameters

ark
Specifies the handle that represents the key-value store.

size
Holds the size of blocks allocated to the key-value store in bytes.

Return values

0
Indicates successful completion.

EINVAL
Indicates failure because of an invalid parameter.

ark_inuse API

Purpose

Returns the number of bytes that are in use in the key-value store.

Syntax

```
int ark_inuse(ark, size)
ARK * ark;
uint64_t *size;
```

Description

The `ark_inuse` API returns the number of bytes in use in the key-value store through the **`size`** parameter.

Parameters

ark
Specifies the handle that represents the key-value store.

size
Holds the size of the store that is in use in bytes.

Return values

0
Indicates successful completion.

EINVAL
Indicates failure because of an invalid parameter.

ark_actual API

Purpose

Returns the number of bytes that are in use in the key-value store.

Syntax

```
int ark_actual(ark, size)
ARK * ark;
uint64 * size;
```

Description

The `ark_actual` API returns the number of bytes that are in use in the key-value store through the **`size`** parameter. This API differs from the `ark_inuse` API such that this API uses the actual size of the individual keys and their values instead of generic block allocations to store these values.

Parameters

ark
Specifies the handle that represents the key-value store.

size
Holds the size of blocks that are in use in bytes.

Return values

0
Indicates successful completion. The **handle** parameter points to the newly created key-value store instance.

EINVAL
Indicates failure because of an invalid parameter.

ark_fork, ark_fork_done API

Purpose

Forks a key-value store for archiving purposes. This service is valid only for the Linux platform.

Syntax

```
int ark_fork(ark)
int ark_fork_done(ark)
ARK * handle;
```

Description

The `ark_fork` and `ark_fork_done` APIs are called by the parent key-value store process to prepare the key-value store to be forked (split into multiple processes), fork the child process, and to clean up the call state after the child process exits. The `ark_fork` API forks a child process, and after completion, the API returns the process ID of the child process to the parent process, and returns 0 to the child process. After the parent process detects that the child process exited, the `ark_fork_done` API is called to clean up any state from the `ark_fork` call.

Note: The `ark_fork` API fails if any outstanding asynchronous commands exist. The `ark_fork` service is valid only for the Linux platform.

Parameters

ark
Specifies the handle that represents the key-value store.

Return values

0
Indicates successful completion.

EINVAL
Indicates failure because of an invalid parameter.

EBUSY
Indicates failure because of outstanding asynchronous operations.

ENOMEM
Indicates failure because of insufficient space to clone the store.

ark_random API

Purpose

Returns a random key from the key-value store.

Syntax

```
int ark_random(ark, kbuflen, klen, kbuf)
ARK * ark;
uint64_t kbuflen
int64_t *klen;
void * kbuf;
```

Description

The `ark_random` API returns a random key from the key-value store based on the **`ark`** handle in the **`kbuf`** buffer, and the size of the key in the **`klen`** parameter, while the key size (**`klen`**) is less than the **`kbuf`** size (**`kbuflen`**).

Parameters

ark
Specifies the handle that represents the key-value store.

kbuflen
Holds the size of the key-value store in bytes.

klen
Specifies the size of the key that is returned in the `kbuf` parameter.

kbuf
Specifies the buffer to hold the key.

Return values

0
Indicates successful completion.

EINVAL
Indicates failure because of an invalid parameter.

ark_count API

Purpose

Returns the count of the number of keys that are found in the key-value store.

Syntax

```
int ark_count(ark, count)
ARK * ark;
int * count;
```

Description

The `ark_count` API returns a the total number of keys in the key-value store based on the **`ark`** handle and stores the result in the **`count`** parameter.

Parameters

ark
Specifies the handle that represents the key-value store.

count
Specifies the number of keys that are found in the key-value store.

Return values

0
Indicates successful completion.

EINVAL
Indicates failure because of an invalid parameter.

ark_stats API

Purpose

Return the number of key-value I/O operations and block I/O operations.

Syntax

```
#include <arkdb.h>

int ark_stats(ARK *ark, uint64_t *ops, uint64_t *ios);
```

Description

The `ark_stats` API returns the total number of key-value I/O operations through the **ops** parameter, and the total number of block I/O operations through the **ios** parameter.

Parameters

ark
Specifies the handle that represents the key-value store.

ops
Indicates the total number of key-value I/O operations.

ios
Indicates the total number of block I/O operations.

Return values

0
Indicates successful completion.

EINVAL
Indicates that an error was encountered.

Related information
Batfile

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM® Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Index

A

C