NAME kstat – kernel statistics facility

DESCRIPTION The **kstat** facility is a general-purpose mechanism for providing kernel statistics to users.

The kstat model The kernel maintains a linked list of statistics structures, or kstats. Each kstat has a common header section and a type-specific data section. The header section is defined by the **kstat_t** structure:

kstat header

```
typedef long        kid_t;                     /* unique kstat id */

typedef struct kstat {
    /*
     * Fields relevant to both kernel and user
     */
    hrtime_t        ks_crtime;                 /* creation time */
    struct kstat    *ks_next;                  /* kstat chain linkage */
    kid_t           ks_kid;                    /* unique kstat ID */
    char            ks_module[KSTAT_STRLEN];   /* module name */
    uchar_t         ks_resv;                   /* reserved */
    int             ks_instance;               /* module's instance */
    char            ks_name[KSTAT_STRLEN];     /* kstat name */
    uchar_t         ks_type;                   /* kstat data type */
    char            ks_class[KSTAT_STRLEN];    /* kstat class */
    uchar_t         ks_flags;                  /* kstat flags */
    void            *ks_data;                  /* kstat type-specific data */
    ulong_t         ks_ndata;                  /* # of data records */
    ulong_t         ks_data_size;              /* size of kstat data section */
    hrtime_t        ks_snaptime;               /* time of last data snapshot */

    /*
     * Fields relevant to kernel only
     */
    int             (*ks_update)(struct kstat *, int);
    void            *ks_private;
    int             (*ks_snapshot)(struct kstat *, void *, int);
    void            *ks_lock;
} kstat_t;
```

The fields that are of significance to the user are:

ks_crtime The time the kstat was created. This allows you to compute the rates of various counters since the kstat was created; "rate since boot" is replaced by the more general concept of "rate since kstat creation".

All times associated with kstats (e.g. creation time, last snapshot time, **kstat_timer_t** and **kstat_io_t** timestamps, etc.) are 64-bit nanosecond values. The accuracy of kstat timestamps is machine dependent, but the precision (units) is the same across all platforms. See **gethrtime**(3C) for general information about high-resolution timestamps.

ks_next kstats are stored as a linked list, or chain. **ks_next** points to the next kstat in the chain.

ks_kid A unique identifier for the kstat.

ks_module, **ks_instance**
contain the name and instance of the the module that created the kstat. In cases where there can only be one instance, **ks_instance** is 0.

ks_name gives a meaningful name to a kstat. The full kstat namespace is **<ks_module**, **ks_instance**, **ks_name** so the name only need be unique within a module.

ks_type The type of data in this kstat. kstat data types are discussed below.

ks_class Each kstat can be characterized as belonging to some broad class of statistics, e.g. disk, tape, net, vm, streams, etc. This field can be used as a filter to extract related kstats. The following values are currently in use: **disk**, **tape**, **controller**, **net**, **rpc**, **vm**, **kvm**, **hat**, **kmem**, **kmem_cache**, **kstat**, and **misc**. (The kstat class encompasses things like *kstat_types*.)

ks_data, **ks_ndata**, **ks_data_size**
ks_data is a pointer to the kstat's data section. The type of data stored there depends on **ks_type**.

ks_ndata indicates the number of data records. Only some kstat types support multiple data records. Currently, **KSTAT_TYPE_RAW, KSTAT_TYPE_NAMED** and **KSTAT_TYPE_TIMER** kstats support multiple data records. **KSTAT_TYPE_INTR** and **KSTAT_TYPE_IO** kstats support only one data record.

ks_data_size is the total size of the data section, in bytes.

ks_snaptime The timestamp for the last data snapshot. This allows you to compute activity rates:

rate = (new_count - old_count) / (new_snaptime - old_snaptime);

kstat data types

The following types of kstats are currently available:

```
#define KSTAT_TYPE_RAW      0    /* can be anything */
#define KSTAT_TYPE_NAMED    1    /* name/value pairs */
#define KSTAT_TYPE_INTR     2    /* interrupt statistics */
#define KSTAT_TYPE_IO       3    /* I/O statistics */
#define KSTAT_TYPE_TIMER    4    /* event timers */
```

To get a list of all kstat types currently supported in the system, tools can read out the standard system kstat *kstat_types* (full name spec is <*''unix'', 0, ''kstat_types''*>). This is a **KSTAT_TYPE_NAMED** kstat in which the *name* field describes the type of kstat, and the *value* field is the kstat type number (e.g., **KSTAT_TYPE_IO** is type 3 -- see above).

Raw kstat

KSTAT_TYPE_RAW: raw data

The "raw" kstat type is just treated as an array of bytes. This is generally used to export well-known structures, like *sysinfo*.

Name=value kstat

KSTAT_TYPE_NAMED: A list of arbitrary *name=value* statistics.

```
typedef struct kstat_named {
    char        name[KSTAT_STRLEN];   /* name of counter */
    uchar_t     data_type;            /* data type */
    union {
            char            c[16];    /* enough for 128-bit ints */
            long            l;
            ulong_t         ul;
            longlong_t      ll;
            u_longlong_t    ull;
            float           f;
            double          d;
            } value;                  /* value of counter */
} kstat_named_t;

#define KSTAT_DATA_CHAR         0
#define KSTAT_DATA_LONG         1
#define KSTAT_DATA_ULONG        2
#define KSTAT_DATA_LONGLONG     3
#define KSTAT_DATA_ULONGLONG    4
#define KSTAT_DATA_FLOAT        5
#define KSTAT_DATA_DOUBLE       6
```

Interrupt kstat

KSTAT_TYPE_INTR: Interrupt statistics.

An interrupt is a hard interrupt (sourced from the hardware device itself), a soft interrupt (induced by the system via the use of some system interrupt source), a watchdog interrupt (induced by a periodic timer call), spurious (an interrupt entry point was entered but there was no interrupt to service), or multiple service (an interrupt was detected and serviced just prior to returning from any of the other types).

```
#define KSTAT_INTR_HARD         0
#define KSTAT_INTR_SOFT         1
#define KSTAT_INTR_WATCHDOG     2
#define KSTAT_INTR_SPURIOUS     3
#define KSTAT_INTR_MULTSVC      4
#define KSTAT_NUM_INTRS         5

typedef struct kstat_intr {
      ulong_t intrs[KSTAT_NUM_INTRS]; /* interrupt counters */
} kstat_intr_t;
```

Event timer kstat

KSTAT_TYPE_TIMER: Event timer statistics.

These provide basic counting and timing information for any type of event.

```
typedef struct kstat_timer {
    char            name[KSTAT_STRLEN];  /* event name */
    uchar_t         resv;                /* reserved */
    u_longlong_t    num_events;          /* number of events */
    hrtime_t        elapsed_time;        /* cumulative elapsed time */
    hrtime_t        min_time;            /* shortest event duration */
    hrtime_t        max_time;            /* longest event duration */
    hrtime_t        start_time;          /* previous event start time */
    hrtime_t        stop_time;           /* previous event stop time */
} kstat_timer_t;
```

I/O kstat

KSTAT_TYPE_IO: I/O statistics.

```
typedef struct kstat_io {
    /*
     * Basic counters.
     */
    u_longlong_t    nread;               /* number of bytes read */
    u_longlong_t    nwritten;            /* number of bytes written */
    ulong_t         reads;               /* number of read operations */
    ulong_t         writes;              /* number of write operations */
    /*
    * Accumulated time and queue length statistics.
    *
    * Time statistics are kept as a running sum of "active" time.
    * Queue length statistics are kept as a running sum of the
    * product of queue length and elapsed time at that length --
    * i.e., a Riemann sum for queue length integrated against time.
    *
    *                 ^
    *                 |                           _________
    *                 8                           | i4      |
    *                 |                           |         |
    *         Queue   6                           |         |
    *         Length  |         _________         |         |
    *                 4         | i2      |_______|         |
    *                 |         |         i3                |
    *                 2_______|                             |
    *                 |   i1                                |
    *                 |_____________________________________|
    *                 Time->  t1          t2       t3        t4
    *
    * At each change of state (entry or exit from the queue),
```

```
     * we add the elapsed time (since the previous state change)
     * to the active time if the queue length was non-zero during
     * that interval; and we add the product of the elapsed time
     * times the queue length to the running length*time sum.
     *
     * This method is generalizable to measuring residency
     * in any defined system: instead of queue lengths, think
     * of "outstanding RPC calls to server X".
     *
     * A large number of I/O subsystems have at least two basic
     * "lists" of transactions they manage: one for transactions
     * that have been accepted for processing but for which processing
     * has yet to begin, and one for transactions which are actively
     * being processed (but not done). For this reason, two cumulative
     * time statistics are defined here: pre-service (wait) time,
     * and service (run) time.
     *
     * The units of cumulative busy time are accumulated nanoseconds.
     * The units of cumulative length*time products are elapsed time
     * times queue length.
     */
    hrtime_t    wtime;          /* cumulative wait (pre-service) time */
    hrtime_t    wlentime;       /* cumulative wait length*time product */
    hrtime_t    wlastupdate;    /* last time wait queue changed */
    hrtime_t    rtime;          /* cumulative run (service) time */
    hrtime_t    rlentime;       /* cumulative run length*time product */
    hrtime_t    rlastupdate;    /* last time run queue changed */
    ulong_t     wcnt;           /* count of elements in wait state */
    ulong_t     rcnt;           /* count of elements in run state */
} kstat_io_t;
```

Using libkstat

The kstat library, **libkstat**, defines the user interface (API) to the system's kstat facility.

You begin by opening libkstat with **kstat_open**(3K), which returns a pointer to a fully initialized kstat control structure. This is your ticket to subsequent libkstat operations:

```
typedef struct kstat_ctl {
    kid_t       kc_chain_id;    /* current kstat chain ID */
    kstat_t     *kc_chain;      /* pointer to kstat chain */
    int         kc_kd;          /* /dev/kstat descriptor */
} kstat_ctl_t;
```

Only the first two fields, **kc_chain_id** and **kc_chain**, are of interest to **libkstat** clients. (*kc_kd* is the descriptor for **/dev/kstat**, the kernel statistics driver. libkstat functions are built on top of **/dev/kstat ioctl**(2) primitives. Direct interaction with **/dev/kstat** is strongly discouraged, since it is NOT a public interface.)

kc_chain points to your copy of the kstat chain. You typically walk the chain to find and process a certain kind of kstat. For example, to display all I/O kstats:

```
kstat_ctl_t     *kc;
kstat_t         *ksp;
kstat_io_t      kio;

kc = kstat_open();
for (ksp = kc->kc_chain; ksp != NULL; ksp = ksp->ks_next) {
        if (ksp->ks_type == KSTAT_TYPE_IO) {
                kstat_read(kc, ksp, &kio);
                my_io_display(kio);
        }
}
```

kc_chain_id is the kstat chain ID, or KCID, of your copy of the kstat chain. See **kstat_chain_update**(3K) for an explanation of KCIDs.

FILES

/dev/kstat kernel statistics driver

/usr/include/kstat.h

/usr/include/sys/kstat.h

SEE ALSO

ioctl(2), **gethrtime**(3C), **kstat_chain_update**(3K), **kstat_close**(3K), **kstat_data_lookup**(3K), **kstat_lookup**(3K), **kstat_open**(3K), **kstat_read**(3K), **kstat_write**(3K)