Linux Sparc Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

STEPS FOR SENDING ‘BREAK’ ON SUNHV CONSOLE

On Baremetal:

1. press Esc + ‘B’

On LDOM:

1. press Ctrl + ‘]’
2. telnet> send break

CHAPTER TWO

APPLICATION DATA INTEGRITY (ADI)

SPARC M7 processor adds the Application Data Integrity (ADI) feature. ADI allows a task to set version tags on any subset of its address space. Once ADI is enabled and version tags are set for ranges of address space of a task, the processor will compare the tag in pointers to memory in these ranges to the version set by the application previously. Access to memory is granted only if the tag in given pointer matches the tag set by the application. In case of mismatch, processor raises an exception.

Following steps must be taken by a task to enable ADI fully:

1. Set the user mode PSTATE.mcde bit. This acts as master switch for the task' s entire address space to enable/disable ADI for the task.
2. Set TTE.mcd bit on any TLB entries that correspond to the range of addresses ADI is being enabled on. MMU checks the version tag only on the pages that have TTE.mcd bit set.
3. Set the version tag for virtual addresses using stxa instruction and one of the MCD specific ASIs. Each stxa instruction sets the given tag for one ADI block size number of bytes. This step must be repeated for entire page to set tags for entire page.

ADI block size for the platform is provided by the hypervisor to kernel in machine description tables. Hypervisor also provides the number of top bits in the virtual address that specify the version tag. Once version tag has been set for a memory location, the tag is stored in the physical memory and the same tag must be present in the ADI version tag bits of the virtual address being presented to the MMU. For example on SPARC M7 processor, MMU uses bits 63-60 for version tags and ADI block size is same as cacheline size which is 64 bytes. A task that sets ADI version to, say 10, on a range of memory, must access that memory using virtual addresses that contain 0xa in bits 63-60.

ADI is enabled on a set of pages using mprotect() with PROT_ADI flag. When ADI is enabled on a set of pages by a task for the first time, kernel sets the PSTATE.mcde bit fot the task. Version tags for memory addresses are set with an stxa instruction on the addresses using ASI_MCD_PRIMARY or ASI_MCD_ST_BLKINIT_PRIMARY. ADI block size is provided by the hypervisor to the kernel. Kernel returns the value of ADI block size to userspace using auxiliary vector along with other ADI info. Following auxiliary vectors are provided by the kernel:

AT_ADI_BLI	ADI block size. This is the granularity and alignment, in bytes, of ADI versioning.
AT_ADI_NB	Number of ADI version bits in the VA

2.1 IMPORTANT NOTES

- Version tag values of 0x0 and 0xf are reserved. These values match any tag in virtual address and never generate a mismatch exception.
- Version tags are set on virtual addresses from userspace even though tags are stored in physical memory. Tags are set on a physical page after it has been allocated to a task and a pte has been created for it.
- When a task frees a memory page it had set version tags on, the page goes back to free page pool. When this page is re-allocated to a task, kernel clears the page using block initialization ASI which clears the version tags as well for the page. If a page allocated to a task is freed and allocated back to the same task, old version tags set by the task on that page will no longer be present.
- ADI tag mismatches are not detected for non-faulting loads.
- Kernel does not set any tags for user pages and it is entirely a task's responsibility to set any version tags. Kernel does ensure the version tags are preserved if a page is swapped out to the disk and swapped back in. It also preserves that version tags if a page is migrated.
- ADI works for any size pages. A userspace task need not be aware of page size when using ADI. It can simply select a virtual address range, enable ADI on the range using mprotect() and set version tags for the entire range. mprotect() ensures range is aligned to page size and is a multiple of page size.
- ADI tags can only be set on writable memory. For example, ADI tags can not be set on read-only mappings.

2.2 ADI related traps

With ADI enabled, following new traps may occur:

2.2.1 Disrupting memory corruption

When a store accesses a memory localtion that has TTE.mcd=1, the task is running with ADI enabled (PSTATE.mcde=1), and the ADI tag in the address used (bits 63:60) does not match the tag set on the corresponding cacheline, a memory corruption trap occurs. By default, it is a disrupting trap and is sent to the hypervisor first. Hypervisor creates a sun4v error report and sends a resumable error (TT=0x7e) trap to the kernel. The kernel sends a SIGSEGV to the task that resulted in this trap with the following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ADIDERR;
siginfo.si_addr = addr; /* PC where first mismatch occurred */
siginfo.si_trapno = 0;
```

2.2.2 Precise memory corruption

When a store accesses a memory location that has TTE.mcd=1, the task is running with ADI enabled (PSTATE.mcde=1), and the ADI tag in the address used (bits 63:60) does not match the tag set on the corresponding cacheline, a memory corruption trap occurs. If MCD precise exception is enabled (MCDPERR=1), a precise exception is sent to the kernel with TT=0x1a. The kernel sends a SIGSEGV to the task that resulted in this trap with the following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ADIPERR;
siginfo.si_addr = addr; /* address that caused trap */
siginfo.si_trapno = 0;
```

NOTE:
ADI tag mismatch on a load always results in precise trap.

2.2.3 MCD disabled

When a task has not enabled ADI and attempts to set ADI version on a memory address, processor sends an MCD disabled trap. This trap is handled by hypervisor first and the hypervisor vectors this trap through to the kernel as Data Access Exception trap with fault type set to 0xa (invalid ASI). When this occurs, the kernel sends the task SIGSEGV signal with following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ACCADI;
siginfo.si_addr = addr; /* address that caused trap */
siginfo.si_trapno = 0;
```

2.2.4 Sample program to use ADI

Following sample program is meant to illustrate how to use the ADI functionality:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <asm/asi.h>

#ifndef AT_ADI_BLKSZ
#define AT_ADI_BLKSZ  48
#endif
#ifndef AT_ADI_NBITS
#define AT_ADI_NBITS  49
#endif

#ifndef PROT_ADI
#define PROT_ADI      0x10
#endif

#define BUFFER_SIZE     32*1024*1024UL

main(int argc, char* argv[], char* envp[])
{
        unsigned long i, mcde, adi_blksz, adi_nbits;
        char *shmaddr, *tmp_addr, *end, *veraddr, *clraddr;
        int shmid, version;
      Elf64_auxv_t *auxv;

      adi_blksz = 0;
```

(continues on next page)

(continued from previous page)

```
        while(*envp++ != NULL);
        for (auxv = (Elf64_auxv_t *)envp; auxv->a_type != AT_NULL;␣
↪auxv++) {
                switch (auxv->a_type) {
                case AT_ADI_BLKSZ:
                        adi_blksz = auxv->a_un.a_val;
                        break;
                case AT_ADI_NBITS:
                        adi_nbits = auxv->a_un.a_val;
                        break;
                }
        }
        if (adi_blksz == 0) {
                fprintf(stderr, "Oops! ADI is not supported\n");
                exit(1);
        }

        printf("ADI capabilities:\n");
        printf("\tBlock size = %ld\n", adi_blksz);
        printf("\tNumber of bits = %ld\n", adi_nbits);

          if ((shmid = shmget(2, BUFFER_SIZE,
                                  IPC_CREAT | SHM_R | SHM_W)) < 0) {
                  perror("shmget failed");
                  exit(1);
          }

          shmaddr = shmat(shmid, NULL, 0);
          if (shmaddr == (char *)-1) {
                  perror("shm attach failed");
                  shmctl(shmid, IPC_RMID, NULL);
                  exit(1);
          }

        if (mprotect(shmaddr, BUFFER_SIZE, PROT_READ|PROT_WRITE|PROT_
↪ADI)) {
                perror("mprotect failed");
                goto err_out;
        }

          /* Set the ADI version tag on the shm segment
           */
          version = 10;
          tmp_addr = shmaddr;
          end = shmaddr + BUFFER_SIZE;
          while (tmp_addr < end) {
                  asm volatile(
                          "stxa %1, [%0]0x90\n\t"
                          :
```

(continues on next page)

(continued from previous page)

```
                        : "r" (tmp_addr), "r" (version));
                tmp_addr += adi_blksz;
        }
      asm volatile("membar #Sync\n\t");

        /* Create a versioned address from the normal address by␣
↪placing
         * version tag in the upper adi_nbits bits
          */
        tmp_addr = (void *) ((unsigned long)shmaddr << adi_nbits);
        tmp_addr = (void *) ((unsigned long)tmp_addr >> adi_nbits);
        veraddr = (void *) (((unsigned long)version << (64-adi_
↪nbits))
                        | (unsigned long)tmp_addr);

        printf("Starting the writes:\n");
        for (i = 0; i < BUFFER_SIZE; i++) {
                veraddr[i] = (char)(i);
                if (!(i % (1024 * 1024)))
                        printf(".");
        }
        printf("\n");

        printf("Verifying data...");
      fflush(stdout);
        for (i = 0; i < BUFFER_SIZE; i++)
                if (veraddr[i] != (char)i)
                        printf("\nIndex %lu mismatched\n", i);
        printf("Done.\n");

        /* Disable ADI and clean up
          */
      if (mprotect(shmaddr, BUFFER_SIZE, PROT_READ|PROT_WRITE)) {
              perror("mprotect failed");
              goto err_out;
      }

        if (shmdt((const void *)shmaddr) != 0)
                perror("Detach failure");
        shmctl(shmid, IPC_RMID, NULL);

        exit(0);

err_out:
        if (shmdt((const void *)shmaddr) != 0)
                perror("Detach failure");
        shmctl(shmid, IPC_RMID, NULL);
        exit(1);
}
```

CHAPTER

THREE

ORACLE DATA ANALYTICS ACCELERATOR (DAX)

DAX is a coprocessor which resides on the SPARC M7 (DAX1) and M8 (DAX2) processor chips, and has direct access to the CPU's L3 caches as well as physical memory. It can perform several operations on data streams with various input and output formats. A driver provides a transport mechanism and has limited knowledge of the various opcodes and data formats. A user space library provides high level services and translates these into low level commands which are then passed into the driver and subsequently the Hypervisor and the coprocessor. The library is the recommended way for applications to use the coprocessor, and the driver interface is not intended for general use. This document describes the general flow of the driver, its structures, and its programmatic interface. It also provides example code sufficient to write user or kernel applications that use DAX functionality.

The user library is open source and available at:

> https://oss.oracle.com/git/gitweb.cgi?p=libdax.git

The Hypervisor interface to the coprocessor is described in detail in the accompanying document, dax-hv-api.txt, which is a plain text excerpt of the (Oracle internal) "UltraSPARC Virtual Machine Specification" version 3.0.20+15, dated 2017-09-25.

3.1 High Level Overview

A coprocessor request is described by a Command Control Block (CCB). The CCB contains an opcode and various parameters. The opcode specifies what operation is to be done, and the parameters specify options, flags, sizes, and addresses. The CCB (or an array of CCBs) is passed to the Hypervisor, which handles queueing and scheduling of requests to the available coprocessor execution units. A status code returned indicates if the request was submitted successfully or if there was an error. One of the addresses given in each CCB is a pointer to a "completion area", which is a 128 byte memory block that is written by the coprocessor to provide execution status. No interrupt is generated upon completion; the completion area must be polled by software to find out when a transaction has finished, but the M7 and later processors provide a mechanism to pause the virtual processor until the completion status has been updated by the coprocessor. This is done using the monitored load and mwait instructions, which are described in more detail later. The DAX coprocessor was designed so that after a request is submitted, the kernel is no longer involved in the processing of it. The polling is done at the user level, which results in almost zero latency between completion of a request and resumption of execution of the requesting thread.

3.2 Addressing Memory

The kernel does not have access to physical memory in the Sun4v architecture, as there is an additional level of memory virtualization present. This intermediate level is called "real" memory, and the kernel treats this as if it were physical. The Hypervisor handles the translations between real memory and physical so that each logical domain (LDOM) can have a partition of physical memory that is isolated from that of other LDOMs. When the kernel sets up a virtual mapping, it specifies a virtual address and the real address to which it should be mapped.

The DAX coprocessor can only operate on physical memory, so before a request can be fed to the coprocessor, all the addresses in a CCB must be converted into physical addresses. The kernel cannot do this since it has no visibility into physical addresses. So a CCB may contain either the virtual or real addresses of the buffers or a combination of them. An "address type" field is available for each address that may be given in the CCB. In all cases, the Hypervisor will translate all the addresses to physical before dispatching to hardware. Address translations are performed using the context of the process initiating the request.

3.3 The Driver API

An application makes requests to the driver via the write() system call, and gets results (if any) via read(). The completion areas are made accessible via mmap(), and are read-only for the application.

The request may either be an immediate command or an array of CCBs to be submitted to the hardware.

Each open instance of the device is exclusive to the thread that opened it, and must be used by that thread for all subsequent operations. The driver open function creates a new context for the thread and initializes it for use. This context contains pointers and values used internally by the driver to keep track of submitted requests. The completion area buffer is also allocated, and this is large enough to contain the completion areas for many concurrent requests. When the device is closed, any outstanding transactions are flushed and the context is cleaned up.

On a DAX1 system (M7), the device will be called "oradax1", while on a DAX2 system (M8) it will be "oradax2". If an application requires one or the other, it should simply attempt to open the appropriate device. Only one of the devices will exist on any given system, so the name can be used to determine what the platform supports.

The immediate commands are CCB_DEQUEUE, CCB_KILL, and CCB_INFO. For all of these, success is indicated by a return value from write() equal to the number of bytes given in the call. Otherwise -1 is returned and errno is set.

3.3.1 CCB_DEQUEUE

Tells the driver to clean up resources associated with past requests. Since no interrupt is generated upon the completion of a request, the driver must be told when it may reclaim resources. No further status information is returned, so the user should not subsequently call read().

3.3.2 CCB_KILL

Kills a CCB during execution. The CCB is guaranteed to not continue executing once this call returns successfully. On success, read() must be called to retrieve the result of the action.

3.3.3 CCB_INFO

Retrieves information about a currently executing CCB. Note that some Hypervisors might return 'notfound' when the CCB is in 'inprogress' state. To ensure a CCB in the 'notfound' state will never be executed, CCB_KILL must be invoked on that CCB. Upon success, read() must be called to retrieve the details of the action.

3.3.4 Submission of an array of CCBs for execution

A write() whose length is a multiple of the CCB size is treated as a submit operation. The file offset is treated as the index of the completion area to use, and may be set via lseek() or using the pwrite() system call. If -1 is returned then errno is set to indicate the error. Otherwise, the return value is the length of the array that was actually accepted by the coprocessor. If the accepted length is equal to the requested length, then the submission was completely successful and there is no further status needed; hence, the user should not subsequently call read(). Partial acceptance of the CCB array is indicated by a return value less than the requested length, and read() must be called to retrieve further status information. The status will reflect the error caused by the first CCB that was not accepted, and status_data will provide additional data in some cases.

3.3.5 MMAP

The mmap() function provides access to the completion area allocated in the driver. Note that the completion area is not writeable by the user process, and the mmap call must not specify PROT_WRITE.

3.4 Completion of a Request

The first byte in each completion area is the command status which is updated by the coprocessor hardware. Software may take advantage of new M7/M8 processor capabilities to efficiently poll this status byte. First, a "monitored load" is achieved via a Load from Alternate Space (ldxa, lduba, etc.) with ASI 0x84 (ASI_MONITOR_PRIMARY). Second, a "monitored wait" is achieved via the mwait instruction (a write to %asr28). This instruction is like pause in that it suspends execution of the virtual processor for the given number of nanoseconds, but in addition will terminate early when one of several events occur. If the block of data containing the monitored location is modified, then the mwait terminates. This causes software to resume execution immediately (without a context switch or kernel to user transition) after a transaction completes. Thus the latency between transaction completion and resumption of execution may be just a few nanoseconds.

3.5 Application Life Cycle of a DAX Submission

- open dax device
- call mmap() to get the completion area address
- allocate a CCB and fill in the opcode, flags, parameters, addresses, etc.
- submit CCB via write() or pwrite()
- go into a loop executing monitored load + monitored wait and terminate when the command status indicates the request is complete (CCB_KILL or CCB_INFO may be used any time as necessary)
- perform a CCB_DEQUEUE
- call munmap() for completion area
- close the dax device

3.6 Memory Constraints

The DAX hardware operates only on physical addresses. Therefore, it is not aware of virtual memory mappings and the discontiguities that may exist in the physical memory that a virtual buffer maps to. There is no I/O TLB or any scatter/gather mechanism. All buffers, whether input or output, must reside in a physically contiguous region of memory.

The Hypervisor translates all addresses within a CCB to physical before handing off the CCB to DAX. The Hypervisor determines the virtual page size for each virtual address given, and uses this to program a size limit for each address. This prevents the coprocessor from reading or writing beyond the bound of the virtual page, even though it is accessing physical memory directly. A simpler way of saying this is that a DAX operation will never "cross" a virtual page boundary. If an 8k virtual page is used, then the data is strictly limited to 8k. If a user's buffer is larger than 8k, then a larger page size must be used, or the transaction size will be truncated to 8k.

Huge pages. A user may allocate huge pages using standard interfaces. Memory buffers residing on huge pages may be used to achieve much larger DAX transaction sizes, but the rules must still be followed, and no transaction will cross a page boundary, even a huge page. A major caveat is that Linux on Sparc presents 8Mb as one of the huge page sizes. Sparc does not actually provide a 8Mb hardware page size, and this size is synthesized by pasting together two 4Mb pages. The reasons for this are historical, and it creates an issue because only half of this 8Mb page can actually be used for any given buffer in a DAX request, and it must be either the first half or the second half; it cannot be a 4Mb chunk in the middle, since that crosses a (hardware) page boundary. Note that this entire issue may be hidden by higher level libraries.

3.6.1 CCB Structure

A CCB is an array of 8 64-bit words. Several of these words provide command opcodes, parameters, flags, etc., and the rest are addresses for the completion area, output buffer, and various inputs:

```
struct ccb {
    u64   control;
    u64   completion;
    u64   input0;
    u64   access;
    u64   input1;
    u64   op_data;
    u64   output;
    u64   table;
};
```

See libdax/common/sys/dax1/dax1_ccb.h for a detailed description of each of these fields, and see dax-hv-api.txt for a complete description of the Hypervisor API available to the guest OS (ie, Linux kernel).

The first word (control) is examined by the driver for the following:

- CCB version, which must be consistent with hardware version
- Opcode, which must be one of the documented allowable commands
- Address types, which must be set to "virtual"for all the addresses given by the user, thereby ensuring that the application can only access memory that it owns

3.7 Example Code

The DAX is accessible to both user and kernel code. The kernel code can make hypercalls directly while the user code must use wrappers provided by the driver. The setup of the CCB is nearly identical for both; the only difference is in preparation of the completion area. An example of user code is given now, with kernel code afterwards.

In order to program using the driver API, the file arch/sparc/include/uapi/asm/oradax.h must be included.

First, the proper device must be opened. For M7 it will be /dev/oradax1 and for M8 it will be /dev/oradax2. The simplest procedure is to attempt to open both, as only one will succeed:

```
fd = open("/dev/oradax1", O_RDWR);
if (fd < 0)
        fd = open("/dev/oradax2", O_RDWR);
if (fd < 0)
       /* No DAX found */
```

Next, the completion area must be mapped:

```
completion_area = mmap(NULL, DAX_MMAP_LEN, PROT_READ, MAP_SHARED,␣
↪fd, 0);
```

All input and output buffers must be fully contained in one hardware page, since as explained above, the DAX is strictly constrained by virtual page boundaries. In addition, the output buffer must be 64-byte aligned and its size must be a multiple of 64 bytes because the coprocessor writes in units of cache lines.

This example demonstrates the DAX Scan command, which takes as input a vector and a match value, and produces a bitmap as the output. For each input element that matches the value, the corresponding bit is set in the output.

In this example, the input vector consists of a series of single bits, and the match value is 0. So each 0 bit in the input will produce a 1 in the output, and vice versa, which produces an output bitmap which is the input bitmap inverted.

For details of all the parameters and bits used in this CCB, please refer to section 36.2.1.3 of the DAX Hypervisor API document, which describes the Scan command in detail:

```
ccb->control =       /* Table 36.1, CCB Header Format */
          (2L << 48)     /* command = Scan Value */
        | (3L << 40)     /* output address type = primary virtual */
        | (3L << 34)     /* primary input address type = primary␣
↪virtual */
                     /* Section 36.2.1, Query CCB Command Formats */
        | (1 << 28)     /* 36.2.1.1.1 primary input format = fixed␣
↪width bit packed */
        | (0 << 23)     /* 36.2.1.1.2 primary input element size =␣
↪0 (1 bit) */
```

(continues on next page)

(continued from previous page)

```
        | (8 << 10)     /* 36.2.1.1.6 output format = bit vector */
        | (0 <<  5)     /* 36.2.1.3 First scan criteria size = 0 (1␣
↪byte) */
        | (31 << 0);    /* 36.2.1.3 Disable second scan criteria */

ccb->completion = 0;    /* Completion area address, to be filled in␣
↪by driver */

ccb->input0 = (unsigned long) input; /* primary input address */

ccb->access =       /* Section 36.2.1.2, Data Access Control */
          (2 << 24)    /* Primary input length format = bits */
        | (nbits - 1); /* number of bits in primary input stream,␣
↪minus 1 */

ccb->input1 = 0;       /* secondary input address, unused */

ccb->op_data = 0;      /* scan criteria (value to be matched) */

ccb->output = (unsigned long) output;   /* output address */

ccb->table = 0;        /* table address, unused */
```

The CCB submission is a write() or pwrite() system call to the driver. If the call fails, then a read() must be used to retrieve the status:

```
if (pwrite(fd, ccb, 64, 0) != 64) {
        struct ccb_exec_result status;
        read(fd, &status, sizeof(status));
        /* bail out */
}
```

After a successful submission of the CCB, the completion area may be polled to determine when the DAX is finished. Detailed information on the contents of the completion area can be found in section 36.2.2 of the DAX HV API document:

```
while (1) {
        /* Monitored Load */
        __asm__ __volatile__("lduba [%1] 0x84, %0\n"
                             : "=r" (status)
                             : "r"  (completion_area));

        if (status)          /* 0 indicates command in progress */
                break;

        /* MWAIT */
        __asm__ __volatile__("wr %%g0, 1000, %%asr28\n" ::);    /*␣
↪1000 ns */
}
```

A completion area status of 1 indicates successful completion of the CCB and validity of the output bitmap, which may be used immediately. All other non-zero values indicate error conditions which are described in section 36.2.2:

```
if (completion_area[0] != 1) {  /* section 36.2.2, 1 = command ran␣
↪and succeeded */
        /* completion_area[0] contains the completion status */
        /* completion_area[1] contains an error code, see 36.2.2 */
}
```

After the completion area has been processed, the driver must be notified that it can release any resources associated with the request. This is done via the dequeue operation:

```
struct dax_command cmd;
cmd.command = CCB_DEQUEUE;
if (write(fd, &cmd, sizeof(cmd)) != sizeof(cmd)) {
        /* bail out */
}
```

Finally, normal program cleanup should be done, i.e., unmapping completion area, closing the dax device, freeing memory etc.

3.7.1 Kernel example

The only difference in using the DAX in kernel code is the treatment of the completion area. Unlike user applications which mmap the completion area allocated by the driver, kernel code must allocate its own memory to use for the completion area, and this address and its type must be given in the CCB:

```
ccb->control |=      /* Table 36.1, CCB Header Format */
        (3L << 32);     /* completion area address type = primary␣
↪virtual */

ccb->completion = (unsigned long) completion_area;   /* Completion␣
↪area address */
```

The dax submit hypercall is made directly. The flags used in the ccb_submit call are documented in the DAX HV API in section 36.3.1/

```
#include <asm/hypervisor.h>

      hv_rv = sun4v_ccb_submit((unsigned long)ccb, 64,
                               HV_CCB_QUERY_CMD |
                               HV_CCB_ARG0_PRIVILEGED | HV_CCB_ARG0_
↪TYPE_PRIMARY |
                               HV_CCB_VA_PRIVILEGED,
                               0, &bytes_accepted, &status_data);

      if (hv_rv != HV_EOK) {
              /* hv_rv is an error code, status_data contains */
```

(continues on next page)

(continued from previous page)

```
                /* potential additional status, see 36.3.1.1 */
        }
```

After the submission, the completion area polling code is identical to that in user land:

```
while (1) {
        /* Monitored Load */
        __asm__ __volatile__("lduba [%1] 0x84, %0\n"
                             : "=r" (status)
                             : "r"  (completion_area));

        if (status)          /* 0 indicates command in progress */
                break;

        /* MWAIT */
        __asm__ __volatile__("wr %%g0, 1000, %%asr28\n" ::);    /*␣
↪1000 ns */
}

if (completion_area[0] != 1) {  /* section 36.2.2, 1 = command ran␣
↪and succeeded */
        /* completion_area[0] contains the completion status */
        /* completion_area[1] contains an error code, see 36.2.2 */
}
```

The output bitmap is ready for consumption immediately after the completion status indicates success.

3.8 Excer[t from UltraSPARC Virtual Machine Specification

```
Excerpt from UltraSPARC Virtual Machine Specification
Compiled from version 3.0.20+15
Publication date 2017-09-25 08:21
Copyright © 2008, 2015 Oracle and/or its affiliates. All rights␣
↪reserved.
Extracted via "pdftotext -f 547 -l 572 -layout sun4v_20170925.
↪pdf"
Authors:
         Charles Kunzman
         Sam Glidden
         Mark Cianchetti


Chapter 36. Coprocessor services
        The following APIs provide access via the Hypervisor to␣
↪hardware assisted data processing functionality.
```

```
        These APIs may only be provided by certain platforms,␣
↪and may not be available to all virtual machines
        even on supported platforms. Restrictions on the use of␣
↪these APIs may be imposed in order to support
        live-migration and other system management activities.

36.1. Data Analytics Accelerator
        The Data Analytics Accelerator (DAX) functionality is a␣
↪collection of hardware coprocessors that provide
        high speed processoring of database-centric operations.␣
↪The coprocessors may support one or more of
        the following data query operations: search, extraction,
↪ compression, decompression, and translation. The
        functionality offered may vary by virtual machine␣
↪implementation.

        The DAX is a virtual device to sun4v guests, with␣
↪supported data operations indicated by the virtual device
        compatibilty property. Functionality is accessed␣
↪through the submission of Command Control Blocks
        (CCBs) via the ccb_submit API function. The operations␣
↪are processed asynchronously, with the status
        of the submitted operations reported through a␣
↪Completion Area linked to each CCB. Each CCB has a
        separate Completion Area and, unless execution order is␣
↪specifically restricted through the use of serial-
        conditional flags, the execution order of submitted␣
↪CCBs is arbitrary. Likewise, the time to completion
        for a given CCB is never guaranteed.

        Guest software may implement a software timeout on CCB␣
↪operations, and if the timeout is exceeded, the
        operation may be cancelled or killed via the ccb_kill␣
↪API function. It is recommended for guest software
        to implement a software timeout to account for certain␣
↪RAS errors which may result in lost CCBs. It is
        recommended such implementation use the ccb_info API␣
↪function to check the status of a CCB prior to
        killing it in order to determine if the CCB is still in␣
↪queue, or may have been lost due to a RAS error.

        There is no fixed limit on the number of outstanding␣
↪CCBs guest software may have queued in the virtual
        machine, however, internal resource limitations within␣
↪the virtual machine can cause CCB submissions
        to be temporarily rejected with EWOULDBLOCK. In such␣
↪cases, guests should continue to attempt
        submissions until they succeed; waiting for an␣
↪outstanding CCB to complete is not necessary, and would
        not be a guarantee that a future submission would␣
↪succeed.
```

```
        The availablility of DAX coprocessor command service is␣
↪indicated by the presence of the DAX virtual
        device node in the guest MD (Section 8.24.17,␣
↪ “Database Analytics Accelerators (DAX) virtual-device
        node” ).

36.1.1. DAX Compatibility Property
        The query functionality may vary based on the␣
↪compatibility property of the virtual device:

36.1.1.1. "ORCL,sun4v-dax" Device Compatibility
        Available CCB commands:

          • No-op/Sync

          • Extract

          • Scan Value

          • Inverted Scan Value

          • Scan Range

                                                      509

                                              Coprocessor␣
↪services


          • Inverted Scan Range

          • Translate

          • Inverted Translate

          • Select

        See Section 36.2.1,  “Query CCB Command Formats”  for␣
↪the corresponding CCB input and output formats.

        Only version 0 CCBs are available.

36.1.1.2. "ORCL,sun4v-dax-fc" Device Compatibility
        "ORCL,sun4v-dax-fc" is compatible with the "ORCL,
↪sun4v-dax" interface, and includes additional CCB
        bit fields and controls.

36.1.1.3. "ORCL,sun4v-dax2" Device Compatibility
        Available CCB commands:
```

```
        • No-op/Sync

        • Extract

        • Scan Value

        • Inverted Scan Value

        • Scan Range

        • Inverted Scan Range

        • Translate

        • Inverted Translate

        • Select

        See Section 36.2.1, “Query CCB Command Formats” for the corresponding CCB input and output formats.

        Version 0 and 1 CCBs are available. Only version 0 CCBs may use Huffman encoded data, whereas only
        version 1 CCBs may use OZIP.

36.1.2. DAX Virtual Device Interrupts
        The DAX virtual device has multiple interrupts associated with it which may be used by the guest if
        desired. The number of device interrupts available to the guest is indicated in the virtual device node of the
        guest MD (Section 8.24.17, “Database Analytics Accelerators (DAX) virtual-device node”). If the device
        node indicates N interrupts available, the guest may use any value from 0 to N - 1 (inclusive) in a CCB
        interrupt number field. Using values outside this range will result in the CCB being rejected for an invalid
        field value.

        The interrupts may be bound and managed using the standard sun4v device interrupts API (Chapter 16,
        Device interrupt services). Sysino interrupts are not available for DAX devices.

36.2. Coprocessor Control Block (CCB)
        CCBs are either 64 or 128 bytes long, depending on the operation type. The exact contents of the CCB
        are command specific, but all CCBs contain at least one memory buffer address. All memory locations
```

```
                                                      510

                                        Coprocessor services


referenced by a CCB must be pinned in memory until the CCB␣
↪either completes execution or is killed
via the ccb_kill API call. Changes in virtual address mappings␣
↪occurring after CCB submission are not
guaranteed to be visible, and as such all virtual address␣
↪updates need to be synchronized with CCB
execution.

All CCBs begin with a common 32-bit header.

Table 36.1. CCB Header Format
Bits          Field Description
[31:28]       CCB version. For API version 2.0: set to 1 if CCB␣
↪uses OZIP encoding; set to 0 if the CCB
              uses Huffman encoding; otherwise either 0 or 1.␣
↪For API version 1.0: always set to 0.
[27]          When API version 2.0 is negotiated, this is the␣
↪Pipeline Flag [512]. It is reserved in
              API version 1.0
[26]          Long CCB flag [512]
[25]          Conditional synchronization flag [512]
[24]          Serial synchronization flag
[23:16]       CCB operation code:
               0x00         No Operation (No-op) or Sync
               0x01         Extract
               0x02         Scan Value
               0x12         Inverted Scan Value
               0x03         Scan Range
               0x13         Inverted Scan Range
               0x04         Translate
               0x14         Inverted Translate
               0x05         Select
[15:13]       Reserved
[12:11]       Table address type
               0b'00        No address
               0b'01        Alternate context virtual address
               0b'10        Real address
               0b'11        Primary context virtual address
[10:8]        Output/Destination address type
               0b'000       No address
               0b'001       Alternate context virtual address
               0b'010       Real address
               0b'011       Primary context virtual address
               0b'100       Reserved
               0b'101       Reserved
               0b'110       Reserved
```

```
              0b'111      Reserved
[7:5]         Secondary source address type


                                             511

                                     Coprocessor services


Bits          Field Description
               0b'000       No address
               0b'001       Alternate context virtual address
               0b'010       Real address
               0b'011       Primary context virtual address
               0b'100       Reserved
               0b'101       Reserved
               0b'110       Reserved
               0b'111       Reserved
[4:2]         Primary source address type
               0b'000       No address
               0b'001       Alternate context virtual address
               0b'010       Real address
               0b'011       Primary context virtual address
               0b'100       Reserved
               0b'101       Reserved
               0b'110       Reserved
               0b'111       Reserved
[1:0]         Completion area address type
               0b'00        No address
               0b'01        Alternate context virtual address
               0b'10        Real address
               0b'11        Primary context virtual address

The Long CCB flag indicates whether the submitted CCB is 64 or␣
↪128 bytes long; value is 0 for 64 bytes
and 1 for 128 bytes.

The Serial and Conditional flags allow simple relative ordering␣
↪between CCBs. Any CCB with the Serial
flag set will execute sequentially relative to any previous CCB␣
↪that is also marked as Serial in the same
CCB submission. CCBs without the Serial flag set execute␣
↪independently, even if they are between CCBs
with the Serial flag set. CCBs marked solely with the Serial␣
↪flag will execute upon the completion of the
previous Serial CCB, regardless of the completion status of␣
↪that CCB. The Conditional flag allows CCBs
to conditionally execute based on the successful execution of␣
↪the closest CCB marked with the Serial flag.
A CCB may only be conditional on exactly one CCB, however, a␣
↪CCB may be marked both Conditional
```

```
and Serial to allow execution chaining. The flags do NOT allow␣
 ↪fan-out chaining, where multiple CCBs
execute in parallel based on the completion of another CCB.

The Pipeline flag is an optimization that directs the output of␣
 ↪one CCB (the "source" CCB) directly to
the input of the next CCB (the "target" CCB). The target CCB␣
 ↪thus does not need to read the input from
memory. The Pipeline flag is advisory and may be dropped.

Both the Pipeline and Serial bits must be set in the source CCB.
 ↪ The Conditional bit must be set in the
target CCB. Exactly one CCB must be made conditional on the␣
 ↪source CCB; either 0 or 2 target CCBs
is invalid. However, Pipelines can be extended beyond two CCBs:␣
 ↪the sequence would start with a CCB
with both the Pipeline and Serial bits set, proceed through␣
 ↪CCBs with the Pipeline, Serial, and Conditional
bits set, and terminate at a CCB that has the Conditional bit␣
 ↪set, but not the Pipeline bit.


                                                     512

                                                       Coprocessor␣
 ↪services


          The input of the target CCB must start within 64␣
 ↪bytes of the output of the source CCB or the pipeline flag
          will be ignored. All CCBs in a pipeline must be␣
 ↪submitted in the same call to ccb_submit.

          The various address type fields indicate how the␣
 ↪various address values used in the CCB should be
          interpreted by the virtual machine. Not all of the␣
 ↪types specified are used by every CCB format. Types
          which are not applicable to the given CCB command␣
 ↪should be indicated as type 0 (No address). Virtual
          addresses used in the CCB must have translation␣
 ↪entries present in either the TLB or a configured TSB
          for the submitting virtual processor. Virtual␣
 ↪addresses which cannot be translated by the virtual machine
          will result in the CCB submission being rejected,␣
 ↪with the causal virtual address indicated. The CCB
          may be resubmitted after inserting the translation,␣
 ↪or the address may be translated by guest software and
          resubmitted using the real address translation.

36.2.1. Query CCB Command Formats
36.2.1.1. Supported Data Formats, Elements Sizes and Offsets
```

```
          Data for query commands may be encoded in multiple␣
↪possible formats. The data query commands use a
          common set of values to indicate the encoding formats␣
↪of the data being processed. Some encoding formats
          require multiple data streams for processing,␣
↪requiring the specification of both primary data formats (the
          encoded data) and secondary data streams (meta-data␣
↪for the encoded data).

36.2.1.1.1. Primary Input Format

          The primary input format code is a 4-bit field when␣
↪it is used. There are 10 primary input formats available.
          The packed formats are not endian neutral. Code␣
↪values not listed below are reserved.

          Code        Format                                   ␣
↪Description
          0x0         Fixed width byte packed             Up to␣
↪16 bytes
          0x1         Fixed width bit packed              Up to␣
↪15 bits (CCB version 0) or 23 bits (CCB version
                                                          1);␣
↪bits are read most significant bit to least significant bit
                                                        ␣
↪within a byte
          0x2         Variable width byte packed          Data␣
↪stream of lengths must be provided as a secondary
                                                          input
          0x4         Fixed width byte packed with run Up to 16␣
↪bytes; data stream of run lengths must be
                      length encoding                  provided␣
↪as a secondary input
          0x5         Fixed width bit packed with run Up to 15␣
↪bits (CCB version 0) or 23 bits (CCB version
                      length encoding                 1); bits␣
↪are read most significant bit to least significant bit
                                                      within a␣
↪byte; data stream of run lengths must be provided
                                                      as a␣
↪secondary input
          0x8         Fixed width byte packed with Up to 16␣
↪bytes before the encoding; compressed stream
                      Huffman (CCB version 0) or bits are read␣
↪most significant bit to least significant bit
                      OZIP (CCB version 1) encoding within a␣
↪byte; pointer to the encoding table must be
                                                    provided
          0x9         Fixed width bit packed with Up to 15 bits␣
↪(CCB version 0) or 23 bits (CCB version
                      Huffman (CCB version 0) or 1); compressed␣
```

```
↪stream bits are read most significant bit to
                    OZIP (CCB version 1) encoding least␣
↪significant bit within a byte; pointer to the encoding
                                                  table must␣
↪be provided
         0xA         Variable width byte packed with Up to 16␣
↪bytes before the encoding; compressed stream
                    Huffman (CCB version 0) or bits are read␣
↪most significant bit to least significant bit
                    OZIP (CCB version 1) encoding within a␣
↪byte; data stream of lengths must be provided as
                                                    a␣
↪secondary input; pointer to the encoding table must be
                                                    provided


                                                      513

                                               Coprocessor␣
↪services



         Code        Format                                ␣
↪Description
         0xC         Fixed width byte packed with        Up to␣
↪16 bytes before the encoding; compressed stream
                    run length encoding, followed by    bits␣
↪are read most significant bit to least significant bit
                    Huffman (CCB version 0) or             ␣
↪within a byte; data stream of run lengths must be provided
                    OZIP (CCB version 1) encoding       as a␣
↪secondary input; pointer to the encoding table must
                                                        be␣
↪provided
         0xD         Fixed width bit packed with         Up to␣
↪15 bits (CCB version 0) or 23 bits(CCB version 1)
                    run length encoding, followed by       ␣
↪before the encoding; compressed stream bits are read most
                    Huffman (CCB version 0) or             ␣
↪significant bit to least significant bit within a byte; data
                    OZIP (CCB version 1) encoding          ␣
↪stream of run lengths must be provided as a secondary
                                                        input;
↪ pointer to the encoding table must be provided

         If OZIP encoding is used, there must be no reserved␣
↪bytes in the table.

36.2.1.1.2. Primary Input Element Size

         For primary input data streams with fixed size␣
```

```
↪elements, the element size must be indicated in the CCB
          command. The size is encoded as the number of bits or␣
↪bytes, minus one. The valid value range for this
          field depends on the input format selected, as listed␣
↪in the table above.

36.2.1.1.3. Secondary Input Format

          For primary input data streams which require a␣
↪secondary input stream, the secondary input stream is
          always encoded in a fixed width, bit-packed format.␣
↪The bits are read from most significant bit to least
          significant bit within a byte. There are two encoding␣
↪options for the secondary input stream data elements,
          depending on whether the value of 0 is needed:

          Secondary           Input Description
          Format Code
          0                          Element is stored as value␣
↪minus 1 (0 evalutes to 1, 1 evalutes
                                     to 2, etc)
          1                          Element is stored as value

36.2.1.1.4. Secondary Input Element Size

          Secondary input element size is encoded as a two bit␣
↪field:

          Secondary Input Size Description
          Code
          0x0                        1 bit
          0x1                        2 bits
          0x2                        4 bits
          0x3                        8 bits

36.2.1.1.5. Input Element Offsets

          Bit-wise input data streams may have any alignment␣
↪within the base addressed byte. The offset, specified
          from most significant bit to least significant bit,␣
↪is provided as a fixed 3 bit field for each input type. A
          value of 0 indicates that the first input element␣
↪begins at the most significant bit in the first byte, and a
          value of 7 indicates it begins with the least␣
↪significant bit.

          This field should be zero for any byte-wise primary␣
↪input data streams.


                                                     514
```

```
                                                  Coprocessor␣
↪services


36.2.1.1.6. Output Format

        Query commands support multiple sizes and encodings␣
↪for output data streams. There are four possible
        output encodings, and up to four supported element␣
↪sizes per encoding. Not all output encodings are
        supported for every command. The format is indicated␣
↪by a 4-bit field in the CCB:

         Output Format Code         Description
         0x0                        Byte aligned, 1 byte␣
↪elements
         0x1                        Byte aligned, 2 byte␣
↪elements
         0x2                        Byte aligned, 4 byte␣
↪elements
         0x3                        Byte aligned, 8 byte␣
↪elements
         0x4                        16 byte aligned, 16 byte␣
↪elements
         0x5                        Reserved
         0x6                        Reserved
         0x7                        Reserved
         0x8                        Packed vector of single␣
↪bit elements
         0x9                        Reserved
         0xA                        Reserved
         0xB                        Reserved
         0xC                        Reserved
         0xD                        2 byte elements where each␣
↪element is the index value of a bit,
                                    from an bit vector, which␣
↪was 1.
         0xE                        4 byte elements where each␣
↪element is the index value of a bit,
                                    from an bit vector, which␣
↪was 1.
         0xF                        Reserved

36.2.1.1.7. Application Data Integrity (ADI)

        On platforms which support ADI, the ADI version␣
↪number may be specified for each separate memory
        access type used in the CCB command. ADI checking␣
↪only occurs when reading data. When writing data,
        the specified ADI version number overwrites any␣
```

```
↪existing ADI value in memory.

          An ADI version value of 0 or 0xF indicates the ADI␣
↪checking is disabled for that data access, even if it is
          enabled in memory. By setting the appropriate flag in␣
↪CCB_SUBMIT (Section 36.3.1, “ccb_submit” ) it is
          also an option to disable ADI checking for all inputs␣
↪accessed via virtual address for all CCBs submitted
          during that hypercall invocation.

          The ADI value is only guaranteed to be checked on the␣
↪first 64 bytes of each data access. Mismatches on
          subsequent data chunks may not be detected, so guest␣
↪software should be careful to use page size checking
          to protect against buffer overruns.

36.2.1.1.8. Page size checking

          All data accesses used in CCB commands must be␣
↪bounded within a single memory page. When addresses
          are provided using a virtual address, the page size␣
↪for checking is extracted from the TTE for that virtual
          address. When using real addresses, the guest must␣
↪supply the page size in the same field as the address
          value. The page size must be one of the sizes␣
↪supported by the underlying virtual machine. Using a value
          that is not supported may result in the CCB␣
↪submission being rejected or the generation of a CCB parsing
          error in the completion area.


                                                      515

                                              Coprocessor␣
↪services


36.2.1.2. Extract command

        Converts an input vector in one format to an output␣
↪vector in another format. All input format types are
        supported.

        The only supported output format is a padded,␣
↪byte-aligned output stream, using output codes 0x0 - 0x4.
        When the specified output element size is larger than␣
↪the extracted input element size, zeros are padded to
        the extracted input element. First, if the decompressed␣
↪input size is not a whole number of bytes, 0 bits are
        padded to the most significant bit side till the next␣
↪byte boundary. Next, if the output element size is larger
```

```
       than the byte padded input element, bytes of value 0␣
↪are added based on the Padding Direction bit in the
       CCB. If the output element size is smaller than the␣
↪byte-padded input element size, the input element is
       truncated by dropped from the least significant byte␣
↪side until the selected output size is reached.

       The return value of the CCB completion area is invalid.␣
↪The  “number of elements processed”  field in the
       CCB completion area will be valid.

       The extract CCB is a 64-byte  “short format”  CCB.

       The extract CCB command format can be specified by the␣
↪following packed C structure for a big-endian
       machine:


                 struct extract_ccb {
                        uint32_t header;
                        uint32_t control;
                        uint64_t completion;
                        uint64_t primary_input;
                        uint64_t data_access_control;
                        uint64_t secondary_input;
                        uint64_t reserved;
                        uint64_t output;
                        uint64_t table;
                 };


       The exact field offsets, sizes, and composition are as␣
↪follows:

        Offset         Size            Field Description
        0              4               CCB header (Table 36.1,␣
↪ “CCB Header Format” )
        4              4               Command control
                                       Bits         Field␣
↪Description
                                       [31:28]      Primary␣
↪Input Format (see Section 36.2.1.1.1,  “Primary Input
                                                    Format” )
                                       [27:23]      Primary␣
↪Input Element Size (see Section 36.2.1.1.2,  “Primary
                                                    Input␣
↪Element Size” )
                                       [22:20]      Primary␣
↪Input Starting Offset (see Section 36.2.1.1.5,  “Input
                                                    Element␣
↪Offsets” )
```

```
                                                    [19]        Secondary␣
    ↪Input Format (see Section 36.2.1.1.3,  “Secondary
                                                                Input␣
    ↪Format” )
                                                    [18:16]     Secondary␣
    ↪Input Starting Offset (see Section 36.2.1.1.5,  “Input
                                                                Element␣
    ↪Offsets” )


                                                                   516

                                 Coprocessor services


   Offset   Size   Field Description
                   Bits          Field Description
                   [15:14]       Secondary Input Element Size (see␣
    ↪Section 36.2.1.1.4,
                                  “Secondary Input Element Size”
                   [13:10]       Output Format (see Section 36.2.1.
    ↪1.6,  “Output Format” )
                   [9]           Padding Direction selector: A␣
    ↪value of 1 causes padding bytes
                                 to be added to the left side of␣
    ↪output elements. A value of 0
                                 causes padding bytes to be added␣
    ↪to the right side of output
                                 elements.
                   [8:0]         Reserved
   8        8      Completion
                   Bits          Field Description
                   [63:60]       ADI version (see Section 36.2.1.1.
    ↪7,  “Application Data
                                 Integrity (ADI)” )
                   [59]          If set to 1, a virtual device␣
    ↪interrupt will be generated using
                                 the device interrupt number␣
    ↪specified in the lower bits of this
                                 completion word. If 0, the lower␣
    ↪bits of this completion word
                                 are ignored.
                   [58:6]        Completion area address bits␣
    ↪[58:6]. Address type is
                                 determined by CCB header.
                   [5:0]         Virtual device interrupt number␣
    ↪for completion interrupt, if
                                 enabled.
   16       8      Primary Input
                   Bits          Field Description
                   [63:60]       ADI version (see Section 36.2.1.1.
```

```
↪7, “Application Data
                            Integrity (ADI)” )
               [59:56]      If using real address, these bits␣
↪should be filled in with the
                            page size code for the page␣
↪boundary checking the guest wants
                            the virtual machine to use when␣
↪accessing this data stream
                            (checking is only guaranteed to be␣
↪performed when using API
                            version 1.1 and later). If using a␣
↪virtual address, this field will
                            be used as as primary input␣
↪address bits [59:56].
               [55:0]       Primary input address bits [55:0].␣
↪Address type is determined
                            by CCB header.
24      8      Data Access Control
               Bits         Field Description
               [63:62]      Flow Control
                            Value      Description
                            0b'00      Disable flow control
                            0b'01      Enable flow control␣
↪(only valid with "ORCL,sun4v-
                                       dax-fc" compatible␣
↪virtual device variants)
                            0b'10      Reserved
                            0b'11      Reserved
               [61:60]      Reserved (API 1.0)


                               517

                      Coprocessor services


Offset   Size   Field Description
               Bits        Field Description
                           Pipeline target (API 2.0)
                           Value      Description
                           0b'00      Connect to primary input
                           0b'01      Connect to secondary␣
↪input
                           0b'10      Reserved
                           0b'11      Reserved
               [59:40]     Output buffer size given in units␣
↪of 64 bytes, minus 1. Value of
                           0 means 64 bytes, value of 1 means␣
↪128 bytes, etc. Buffer size is
                           only enforced if flow control is␣
↪enabled in Flow Control field.
```

```
                    [39:32]     Reserved
                    [31:30]     Output Data Cache Allocation
                                Value       Description
                                0b'00       Do not allocate cache␣
↪lines for output data stream.
                                0b'01       Force cache lines for␣
↪output data stream to be
                                            allocated in the cache␣
↪that is local to the submitting
                                            virtual cpu.
                                0b'10       Allocate cache lines for␣
↪output data stream, but allow
                                            existing cache lines␣
↪associated with the data to remain
                                            in their current cache␣
↪instance. Any memory not
                                            already in cache will be␣
↪allocated in the cache local
                                            to the submitting␣
↪virtual cpu.
                                0b'11       Reserved
                    [29:26]     Reserved
                    [25:24]     Primary Input Length Format
                                Value       Description
                                0b'00       Number of primary symbols
                                0b'01       Number of primary bytes
                                0b'10       Number of primary bits
                                0b'11       Reserved
                    [23:0]      Primary Input Length
                                Format                       Field␣
↪Value
                                # of primary symbols         Number␣
↪of input elements to process,
                                                             minus 1.
↪ Command execution stops
                                                             once␣
↪count is reached.
                                # of primary bytes           Number␣
↪of input bytes to process,
                                                             minus 1.
↪ Command execution stops
                                                             once␣
↪count is reached. The count is
                                                             done␣
↪before any decompression or
                                                            ␣
↪decoding.
                                # of primary bits            Number␣
↪of input bits to process,
                                                             minus 1.
↪ Command execution stops
```

```
                                   518

                                                                   Coprocessor␣
↪services


                Offset          Size            Field Description
                                                 Bits         Field␣
↪Description
                                                              Format    ␣
↪                  Field Value
                                                                        ␣
↪                  once count is reached. The count is
                                                                        ␣
↪                  done before any decompression or
                                                                        ␣
↪                  decoding, and does not include any
                                                                        ␣
↪                  bits skipped by the Primary Input
                                                                        ␣
↪                  Offset field value of the command
                                                                        ␣
↪                  control word.
                32              8               Secondary Input, if used␣
↪by Primary Input Format. Same fields as Primary
                                                Input.
                40              8               Reserved
                48              8               Output (same fields as␣
↪Primary Input)
                56              8               Symbol Table (if used by␣
↪Primary Input)
                                                 Bits         Field␣
↪Description
                                                 [63:60]      ADI␣
↪version (see Section 36.2.1.1.7,  “Application Data
                                                              Integrity␣
↪(ADI)” )
                                                 [59:56]      If using␣
↪real address, these bits should be filled in with the
                                                              page size␣
↪code for the page boundary checking the guest wants
                                                              the␣
↪virtual machine to use when accessing this data stream
                                                              (checking␣
↪is only guaranteed to be performed when using API
                                                              version 1.
↪1 and later). If using a virtual address, this field will
                                                              be used as␣
```

```
↪as symbol table address bits [59:56].
                                                [55:4]      Symbol␣
↪table address bits [55:4]. Address type is determined
                                                            by CCB␣
↪header.
                                                [3:0]       Symbol␣
↪table version
                                                            Value   ␣
↪Description
                                                            0       ␣
↪Huffman encoding. Must use 64 byte aligned table
                                                                    ␣
↪address. (Only available when using version 0 CCBs)
                                                            1       ␣
↪OZIP encoding. Must use 16 byte aligned table
                                                                    ␣
↪address. (Only available when using version 1 CCBs)


36.2.1.3. Scan commands

       The scan commands search a stream of input data␣
↪elements for values which match the selection criteria.
       All the input format types are supported. There are␣
↪multiple formats for the scan commands, allowing the
       scan to search for exact matches to one value, exact␣
↪matches to either of two values, or any value within
       a specified range. The specific type of scan is␣
↪indicated by the command code in the CCB header. For the
       scan range commands, the boundary conditions can be␣
↪specified as greater-than-or-equal-to a value, less-
       than-or-equal-to a value, or both by using two boundary␣
↪values.

       There are two supported formats for the output stream:␣
↪the bit vector and index array formats (codes 0x8,
       0xD, and 0xE). For the standard scan command using the␣
↪bit vector output, for each input element there
       exists one bit in the vector that is set if the input␣
↪element matched the scan criteria, or clear if not. The
       inverted scan command inverts the polarity of the bits␣
↪in the output. The most significant bit of the first
       byte of the output stream corresponds to the first␣
↪element in the input stream. The standard index array
       output format contains one array entry for each input␣
↪element that matched the scan criteria. Each array
```

519

```
                                      Coprocessor services


entry is the index of an input element that matched the scan␣
↪criteria. An inverted scan command produces
a similar array, but of all the input elements which did NOT␣
↪match the scan criteria.

The return value of the CCB completion area contains the number␣
↪of input elements found which match
the scan criteria (or number that did not match for the␣
↪inverted scans). The “number of elements processed”
field in the CCB completion area will be valid, indicating the␣
↪number of input elements processed.

These commands are 128-byte “long format” CCBs.

The scan CCB command format can be specified by the following␣
↪packed C structure for a big-endian
machine:


         struct scan_ccb        {
                uint32_t        header;
                uint32_t        control;
                uint64_t        completion;
                uint64_t        primary_input;
                uint64_t        data_access_control;
                uint64_t        secondary_input;
                uint64_t        match_criteria0;
                uint64_t        output;
                uint64_t        table;
                uint64_t        match_criteria1;
                uint64_t        match_criteria2;
                uint64_t        match_criteria3;
                uint64_t        reserved[5];
         };


The exact field offsets, sizes, and composition are as follows:

Offset          Size            Field Description
0               4               CCB header (Table 36.1, “CCB␣
↪Header Format”)
4               4               Command control
                                Bits          Field Description
                                [31:28]       Primary Input␣
↪Format (see Section 36.2.1.1.1, “Primary Input
                                              Format”)
                                [27:23]       Primary Input␣
↪Element Size (see Section 36.2.1.1.2, “Primary
```

```
                                                Input Element␣
↪Size” )
                                   [22:20]      Primary Input␣
↪Starting Offset (see Section 36.2.1.1.5,  “Input
                                                Element Offsets” )
                                   [19]         Secondary Input␣
↪Format (see Section 36.2.1.1.3,  “Secondary
                                                Input Format” )
                                   [18:16]      Secondary Input␣
↪Starting Offset (see Section 36.2.1.1.5,  “Input
                                                Element Offsets” )
                                   [15:14]      Secondary Input␣
↪Element Size (see Section 36.2.1.1.4,
                                                 “Secondary Input␣
↪Element Size”
                                   [13:10]      Output Format (see␣
↪Section 36.2.1.1.6,  “Output Format” )
                                   [9:5]        Operand size for␣
↪first scan criteria value. In a scan value
                                                operation, this is␣
↪one of two potential extact match values.
                                                In a scan range␣
↪operation, this is the size of the upper range


                                                   520

                            Coprocessor services


Offset   Size   Field Description
                Bits         Field Description
                             boundary. The value of this field␣
↪is the number of bytes in the
                             operand, minus 1. Values 0xF-0x1E␣
↪are reserved. A value of
                             0x1F indicates this operand is not␣
↪in use for this scan operation.
                [4:0]        Operand size for second scan␣
↪criteria value. In a scan value
                             operation, this is one of two␣
↪potential extact match values.
                             In a scan range operation, this is␣
↪the size of the lower range
                             boundary. The value of this field␣
↪is the number of bytes in the
                             operand, minus 1. Values 0xF-0x1E␣
↪are reserved. A value of
                             0x1F indicates this operand is not␣
↪in use for this scan operation.
8        8      Completion (same fields as Section 36.2.1.2,␣
```

```
↪ “Extract command” )
16       8      Primary Input (same fields as Section 36.2.1.2,␣
↪ “Extract command” )
24       8      Data Access Control (same fields as Section 36.
↪2.1.2,  “Extract command” )
32       8      Secondary Input, if used by Primary Input␣
↪Format. Same fields as Primary
                Input.
40       4      Most significant 4 bytes of first scan criteria␣
↪operand. If first operand is less
                than 4 bytes, the value is left-aligned to the␣
↪lowest address bytes.
44       4      Most significant 4 bytes of second scan␣
↪criteria operand. If second operand
                is less than 4 bytes, the value is left-aligned␣
↪to the lowest address bytes.
48       8      Output (same fields as Primary Input)
56       8      Symbol Table (if used by Primary Input). Same␣
↪fields as Section 36.2.1.2,
                 “Extract command”
64       4      Next 4 most significant bytes of first scan␣
↪criteria operand occuring after the
                bytes specified at offset 40, if needed by the␣
↪operand size. If first operand
                is less than 8 bytes, the valid bytes are␣
↪left-aligned to the lowest address.
68       4      Next 4 most significant bytes of second scan␣
↪criteria operand occuring after
                the bytes specified at offset 44, if needed by␣
↪the operand size. If second
                operand is less than 8 bytes, the valid bytes␣
↪are left-aligned to the lowest
                address.
72       4      Next 4 most significant bytes of first scan␣
↪criteria operand occuring after the
                bytes specified at offset 64, if needed by the␣
↪operand size. If first operand
                is less than 12 bytes, the valid bytes are␣
↪left-aligned to the lowest address.
76       4      Next 4 most significant bytes of second scan␣
↪criteria operand occuring after
                the bytes specified at offset 68, if needed by␣
↪the operand size. If second
                operand is less than 12 bytes, the valid bytes␣
↪are left-aligned to the lowest
                address.
80       4      Next 4 most significant bytes of first scan␣
↪criteria operand occuring after the
                bytes specified at offset 72, if needed by the␣
↪operand size. If first operand
                is less than 16 bytes, the valid bytes are␣
```

```
↪left-aligned to the lowest address.
84       4      Next 4 most significant bytes of second scan␣
↪criteria operand occuring after
                the bytes specified at offset 76, if needed by␣
↪the operand size. If second
                operand is less than 16 bytes, the valid bytes␣
↪are left-aligned to the lowest
                address.



                                521

                                                   Coprocessor␣
↪services


36.2.1.4. Translate commands

        The translate commands takes an input array of indicies,
↪ and a table of single bit values indexed by those
        indicies, and outputs a bit vector or index array␣
↪created by reading the tables bit value at each index in
        the input array. The output should therefore contain␣
↪exactly one bit per index in the input data stream,
        when outputing as a bit vector. When outputing as an␣
↪index array, the number of elements depends on the
        values read in the bit table, but will always be less␣
↪than, or equal to, the number of input elements. Only
        a restricted subset of the possible input format types␣
↪are supported. No variable width or Huffman/OZIP
        encoded input streams are allowed. The primary input␣
↪data element size must be 3 bytes or less.

        The maximum table index size allowed is 15 bits,␣
↪however, larger input elements may be used to provide
        additional processing of the output values. If 2 or 3␣
↪byte values are used, the least significant 15 bits are
        used as an index into the bit table. The most␣
↪significant 9 bits (when using 3-byte input elements) or␣
↪single
        bit (when using 2-byte input elements) are compared␣
↪against a fixed 9-bit test value provided in the CCB.
        If the values match, the value from the bit table is␣
↪used as the output element value. If the values do not
        match, the output data element value is forced to 0.

        In the inverted translate operation, the bit value read␣
↪from bit table is inverted prior to its use. The additional
        additional processing based on any additional non-index␣
```

```
↪bits remains unchanged, and still forces the output
        element value to 0 on a mismatch. The specific type of␣
↪translate command is indicated by the command
        code in the CCB header.

        There are two supported formats for the output stream:␣
↪the bit vector and index array formats (codes 0x8,
        0xD, and 0xE). The index array format is an array of␣
↪indicies of bits which would have been set if the
        output format was a bit array.

        The return value of the CCB completion area contains␣
↪the number of bits set in the output bit vector,
        or number of elements in the output index array. The␣
↪ “number of elements processed”  field in the CCB
        completion area will be valid, indicating the number of␣
↪input elements processed.

        These commands are 64-byte  “short format”  CCBs.

        The translate CCB command format can be specified by␣
↪the following packed C structure for a big-endian
        machine:


                 struct translate_ccb {
                        uint32_t header;
                        uint32_t control;
                        uint64_t completion;
                        uint64_t primary_input;
                        uint64_t data_access_control;
                        uint64_t secondary_input;
                        uint64_t reserved;
                        uint64_t output;
                        uint64_t table;
                 };


        The exact field offsets, sizes, and composition are as␣
↪follows:


        Offset          Size            Field Description
        0               4               CCB header (Table 36.1,
↪  “CCB Header Format” )


                                                        522

                                Coprocessor services
```

```
Offset   Size   Field Description
4        4      Command control
                Bits          Field Description
                [31:28]       Primary Input Format (see Section␣
↪36.2.1.1.1, “Primary Input
                              Format” )
                [27:23]       Primary Input Element Size (see␣
↪Section 36.2.1.1.2, “Primary
                              Input Element Size” )
                [22:20]       Primary Input Starting Offset (see␣
↪Section 36.2.1.1.5, “Input
                              Element Offsets” )
                [19]          Secondary Input Format (see␣
↪Section 36.2.1.1.3, “Secondary
                              Input Format” )
                [18:16]       Secondary Input Starting Offset␣
↪(see Section 36.2.1.1.5, “Input
                              Element Offsets” )
                [15:14]       Secondary Input Element Size (see␣
↪Section 36.2.1.1.4,
                               “Secondary Input Element Size”
                [13:10]       Output Format (see Section 36.2.1.
↪1.6, “Output Format” )
                [9]           Reserved
                [8:0]         Test value used for comparison␣
↪against the most significant bits
                              in the input values, when using 2␣
↪or 3 byte input elements.
8        8      Completion (same fields as Section 36.2.1.2,␣
↪ “Extract command”
16       8      Primary Input (same fields as Section 36.2.1.2,␣
↪ “Extract command”
24       8      Data Access Control (same fields as Section 36.
↪2.1.2, “Extract command” ,
                except Primary Input Length Format may not use␣
↪the 0x0 value)
32       8      Secondary Input, if used by Primary Input␣
↪Format. Same fields as Primary
                Input.
40       8      Reserved
48       8      Output (same fields as Primary Input)
56       8      Bit Table
                Bits          Field Description
                [63:60]       ADI version (see Section 36.2.1.1.
↪7, “Application Data
                              Integrity (ADI)” )
                [59:56]       If using real address, these bits␣
↪should be filled in with the
                              page size code for the page␣
↪boundary checking the guest wants
```

```
                                    the virtual machine to use when␣
↪accessing this data stream
                                    (checking is only guaranteed to be␣
↪performed when using API
                                    version 1.1 and later). If using a␣
↪virtual address, this field will
                                    be used as as bit table address␣
↪bits [59:56]
                   [55:4]           Bit table address bits [55:4].␣
↪Address type is determined by
                                    CCB header. Address must be␣
↪64-byte aligned (CCB version
                                    0) or 16-byte aligned (CCB version␣
↪1).
                   [3:0]            Bit table version
                                    Value       Description
                                    0           4KB table size
                                    1           8KB table size



                                          523

                                                         Coprocessor␣
↪services


36.2.1.5. Select command
        The select command filters the primary input data␣
↪stream by using a secondary input bit vector to determine
        which input elements to include in the output. For each␣
↪bit set at a given index N within the bit vector,
        the Nth input element is included in the output. If the␣
↪bit is not set, the element is not included. Only a
        restricted subset of the possible input format types␣
↪are supported. No variable width or run length encoded
        input streams are allowed, since the secondary input␣
↪stream is used for the filtering bit vector.

        The only supported output format is a padded,␣
↪byte-aligned output stream. The stream follows the same
        rules and restrictions as padded output stream␣
↪described in Section 36.2.1.2, “Extract command”.

        The return value of the CCB completion area contains␣
↪the number of bits set in the input bit vector. The
        "number of elements processed" field in the CCB␣
↪completion area will be valid, indicating the number
        of input elements processed.

        The select CCB is a 64-byte “short format” CCB.
```

```
      The select CCB command format can be specified by the␣
↪following packed C structure for a big-endian
      machine:


                struct select_ccb {
                       uint32_t header;
                       uint32_t control;
                       uint64_t completion;
                       uint64_t primary_input;
                       uint64_t data_access_control;
                       uint64_t secondary_input;
                       uint64_t reserved;
                       uint64_t output;
                       uint64_t table;
                };


      The exact field offsets, sizes, and composition are as␣
↪follows:

       Offset        Size            Field Description
       0             4               CCB header (Table 36.1,␣
↪ “CCB Header Format” )
       4             4               Command control
                                     Bits         Field␣
↪Description
                                     [31:28]      Primary␣
↪Input Format (see Section 36.2.1.1.1,  “Primary Input
                                                  Format” )
                                     [27:23]      Primary␣
↪Input Element Size (see Section 36.2.1.1.2,  “Primary
                                                  Input␣
↪Element Size” )
                                     [22:20]      Primary␣
↪Input Starting Offset (see Section 36.2.1.1.5,  “Input
                                                  Element␣
↪Offsets” )
                                     [19]         Secondary␣
↪Input Format (see Section 36.2.1.1.3,  “Secondary
                                                  Input␣
↪Format” )
                                     [18:16]      Secondary␣
↪Input Starting Offset (see Section 36.2.1.1.5,  “Input
                                                  Element␣
↪Offsets” )
                                     [15:14]      Secondary␣
↪Input Element Size (see Section 36.2.1.1.4,
                                                   “Secondary␣
↪Input Element Size”
```

```
                                                        524
                                                   Coprocessor␣
↪services


        Offset          Size            Field Description
                                        Bits            Field␣
↪Description
                                        [13:10]         Output␣
↪Format (see Section 36.2.1.1.6,  “Output Format” )
                                        [9]             Padding␣
↪Direction selector: A value of 1 causes padding bytes
                                                        to be added␣
↪to the left side of output elements. A value of 0
                                                        causes␣
↪padding bytes to be added to the right side of output
                                                        elements.
                                        [8:0]           Reserved
        8               8               Completion (same fields␣
↪as Section 36.2.1.2,  “Extract command”
        16              8               Primary Input (same␣
↪fields as Section 36.2.1.2,  “Extract command”
        24              8               Data Access Control␣
↪(same fields as Section 36.2.1.2,  “Extract command” )
        32              8               Secondary Bit Vector␣
↪Input. Same fields as Primary Input.
        40              8               Reserved
        48              8               Output (same fields as␣
↪Primary Input)
        56              8               Symbol Table (if used by␣
↪Primary Input). Same fields as Section 36.2.1.2,
                                         “Extract command”


36.2.1.6. No-op and Sync commands
        The no-op (no operation) command is a CCB which has no␣
↪processing effect. The CCB, when processed
        by the virtual machine, simply updates the completion␣
↪area with its execution status. The CCB may have
        the serial-conditional flags set in order to restrict␣
↪when it executes.

        The sync command is a variant of the no-op command␣
↪which with restricted execution timing. A sync
        command CCB will only execute when all previous␣
↪commands submitted in the same request have
        completed. This is stronger than the conditional flag␣
↪sequencing, which is only dependent on a single
        previous serial CCB. While the relative ordering is␣
```

```
↪guaranteed, virtual machine implementations with
        shared hardware resources may cause the sync command to␣
↪wait for longer than the minimum required
        time.

        The return value of the CCB completion area is invalid␣
↪for these CCBs. The "number of elements
        processed" field is also invalid for these CCBs.

        These commands are 64-byte "short format" CCBs.

        The no-op CCB command format can be specified by the␣
↪following packed C structure for a big-endian
        machine:


                struct nop_ccb {
                        uint32_t header;
                        uint32_t control;
                        uint64_t completion;
                        uint64_t reserved[6];
                };


        The exact field offsets, sizes, and composition are as␣
↪follows:

        Offset          Size            Field Description
        0               4               CCB header (Table 36.1,␣
↪ "CCB Header Format" )


                                                        525

                                                Coprocessor services



       Offset          Size            Field Description
       4               4               Command control
                                       Bits         Field Description
                                       [31]         If set, this CCB␣
↪functions as a Sync command. If clear, this
                                                    CCB functions as␣
↪a No-op command.
                                       [30:0]       Reserved
       8               8               Completion (same fields as␣
↪Section 36.2.1.2, "Extract command"
       16              46              Reserved

36.2.2. CCB Completion Area
       All CCB commands use a common 128-byte Completion Area␣
```

```
↪format, which can be specified by the
        following packed C structure for a big-endian machine:


                struct completion_area {
                        uint8_t status_flag;
                        uint8_t error_note;
                        uint8_t rsvd0[2];
                        uint32_t error_values;
                        uint32_t output_size;
                        uint32_t rsvd1;
                        uint64_t run_time;
                        uint64_t run_stats;
                        uint32_t elements;
                        uint8_t rsvd2[20];
                        uint64_t return_value;
                        uint64_t extra_return_value[8];
                };


        The Completion Area must be a 128-byte aligned memory␣
↪location. The exact layout can be described
        using byte offsets and sizes relative to the memory base:

        Offset          Size            Field Description
        0               1               CCB execution status
                                        0x0                     Command␣
↪not yet completed
                                        0x1                     Command␣
↪ran and succeeded
                                        0x2                     Command␣
↪ran and failed (partial results may be been
                                                                ␣
↪produced)
                                        0x3                     Command␣
↪ran and was killed (partial execution may
                                                                have␣
↪occurred)
                                        0x4                     Command␣
↪was not run
                                        0x5-0xF                 Reserved
        1               1               Error reason code
                                        0x0                     Reserved
                                        0x1                     Buffer␣
↪overflow


                                                          526

                                             Coprocessor services
```

```
Offset          Size            Field Description
                                 0x2                 CCB␣
 ↪decoding error
                                 0x3                 Page␣
 ↪overflow
                                 0x4-0x6             Reserved
                                 0x7                 Command was␣
 ↪killed
                                 0x8                 Command␣
 ↪execution timeout
                                 0x9                 ADI␣
 ↪miscompare error
                                 0xA                 Data format␣
 ↪error
                                 0xB-0xD             Reserved
                                 0xE                 Unexpected␣
 ↪hardware error (Do not retry)
                                 0xF                 Unexpected␣
 ↪hardware error (Retry is ok)
                                 0x10-0x7F           Reserved
                                 0x80                Partial␣
 ↪Symbol Warning
                                 0x81-0xFF           Reserved
2               2               Reserved
4               4               If a partial symbol warning was␣
 ↪generated, this field contains the number
                                of remaining bits which were not␣
 ↪decoded.
8               4               Number of bytes of output␣
 ↪produced
12              4               Reserved
16              8               Runtime of command (unspecified␣
 ↪time units)
24              8               Reserved
32              4               Number of elements processed
36              20              Reserved
56              8               Return value
64              64              Extended return value

The CCB completion area should be treated as read-only by guest␣
 ↪software. The CCB execution status
byte will be cleared by the Hypervisor to reflect the pending␣
 ↪execution status when the CCB is submitted
successfully. All other fields are considered invalid upon CCB␣
 ↪submission until the CCB execution status
byte becomes non-zero.

CCBs which complete with status 0x2 or 0x3 may produce partial␣
 ↪results and/or side effects due to partial
execution of the CCB command. Some valid data may be accessible␣
```

```
↪depending on the fault type, however,
it is recommended that guest software treat the destination␣
↪buffer as being in an unknown state. If a CCB
completes with a status byte of 0x2, the error reason code byte␣
↪can be read to determine what corrective
action should be taken.

A buffer overflow indicates that the results of the operation␣
↪exceeded the size of the output buffer indicated
in the CCB. The operation can be retried by resubmitting the␣
↪CCB with a larger output buffer.

A CCB decoding error indicates that the CCB contained some␣
↪invalid field values. It may be also be
triggered if the CCB output is directed at a non-existent␣
↪secondary input and the pipelining hint is followed.

A page overflow error indicates that the operation required␣
↪accessing a memory location beyond the page
size associated with a given address. No data will have been␣
↪read or written past the page boundary, but
partial results may have been written to the destination buffer.
↪ The CCB can be resubmitted with a larger
page size memory allocation to complete the operation.


                                                   527

                                                Coprocessor services


       In the case of pipelined CCBs, a page overflow error␣
↪will be triggered if the output from the pipeline source
       CCB ends before the input of the pipeline target CCB.␣
↪Page boundaries are ignored when the pipeline
       hint is followed.

       Command kill indicates that the CCB execution was halted␣
↪or prevented by use of the ccb_kill API call.

       Command timeout indicates that the CCB execution began,␣
↪but did not complete within a pre-determined
       limit set by the virtual machine. The command may have␣
↪produced some or no output. The CCB may be
       resubmitted with no alterations.

       ADI miscompare indicates that the memory buffer version␣
↪specified in the CCB did not match the value
       in memory when accessed by the virtual machine. Guest␣
↪software should not attempt to resubmit the CCB
       without determining the cause of the version mismatch.
```

```
        A data format error indicates that the input data stream did not follow the specified data input formatting
        selected in the CCB.

        Some CCBs which encounter hardware errors may be resubmitted without change. Persistent hardware
        errors may result in multiple failures until RAS software can identify and isolate the faulty component.

        The output size field indicates the number of bytes of valid output in the destination buffer. This field is
        not valid for all possible CCB commands.

        The runtime field indicates the execution time of the CCB command once it leaves the internal virtual
        machine queue. The time units are fixed, but unspecified, allowing only relative timing comparisons
        by guest software. The time units may also vary by hardware platform, and should not be construed to
        represent any absolute time value.

        Some data query commands process data in units of elements. If applicable to the command, the number of
        elements processed is indicated in the listed field. This field is not valid for all possible CCB commands.

        The return value and extended return value fields are output locations for commands which do not use
        a destination output buffer, or have secondary return results. The field is not valid for all possible CCB
        commands.

36.3. Hypervisor API Functions
36.3.1. ccb_submit
        trap#                   FAST_TRAP
        function#               CCB_SUBMIT
        arg0                    address
        arg1                    length
        arg2                    flags
        arg3                    reserved
        ret0                    status
        ret1                    length
        ret2                    status data
        ret3                    reserved

        Submit one or more coprocessor control blocks (CCBs) for evaluation and processing by the virtual
        machine. The CCBs are passed in a linear array indicated by address. length indicates the size of
        the array in bytes.
```

```
                                                528

                                      Coprocessor services


The address should be aligned to the size indicated by length,␣
↪rounded up to the nearest power of
two. Virtual machines implementations may reject submissions␣
↪which do not adhere to that alignment.
length must be a multiple of 64 bytes. If length is zero, the␣
↪maximum supported array length will be
returned as length in ret1. In all other cases, the length␣
↪value in ret1 will reflect the number of bytes
successfully consumed from the input CCB array.

      Implementation note
      Virtual machines should never reject submissions based on␣
↪the alignment of address if the
      entire array is contained within a single memory page of␣
↪the smallest page size supported by the
      virtual machine.

A guest may choose to submit addresses used in this API␣
↪function, including the CCB array address,
as either a real or virtual addresses, with the type of each␣
↪address indicated in flags. Virtual addresses
must be present in either the TLB or an active TSB to be␣
↪processed. The translation context for virtual
addresses is determined by a combination of CCB contents and␣
↪the flags argument.

The flags argument is divided into multiple fields defined as␣
↪follows:


Bits            Field Description
[63:16]         Reserved
[15]            Disable ADI for VA reads (in API 2.0)
                Reserved (in API 1.0)
[14]            Virtual addresses within CCBs are translated in␣
↪privileged context
[13:12]         Alternate translation context for virtual␣
↪addresses within CCBs:
                 0b'00       CCBs requesting alternate context␣
↪are rejected
                 0b'01       Reserved
                 0b'10       CCBs requesting alternate context␣
↪use secondary context
                 0b'11       CCBs requesting alternate context␣
```

```
↪use nucleus context
[11:9]          Reserved
[8]             Queue info flag
[7]             All-or-nothing flag
[6]             If address is a virtual address, treat its␣
↪translation context as privileged
[5:4]           Address type of address:
                 0b'00        Real address
                 0b'01        Virtual address in primary context
                 0b'10        Virtual address in secondary␣
↪context
                 0b'11        Virtual address in nucleus context
[3:2]           Reserved
[1:0]           CCB command type:
                 0b'00        Reserved
                 0b'01        Reserved
                 0b'10        Query command
                 0b'11        Reserved



                                                  529

                                                 Coprocessor␣
↪services


        The CCB submission type and address type for the CCB␣
↪array must be provided in the flags argument.
        All other fields are optional values which change the␣
↪default behavior of the CCB processing.

        When set to one, the "Disable ADI for VA reads" bit␣
↪will turn off ADI checking when using a virtual
        address to load data. ADI checking will still be done␣
↪when loading real-addressed memory. This bit is only
        available when using major version 2 of the␣
↪coprocessor API group; at major version 1 it is reserved. For
        more information about using ADI and DAX, see Section␣
↪36.2.1.1.7, “Application Data Integrity (ADI)” .

        By default, all virtual addresses are treated as user␣
↪addresses. If the virtual address translations are
        privileged, they must be marked as such in the␣
↪appropriate flags field. The virtual addresses used within
        the submitted CCBs must all be translated with the␣
↪same privilege level.

        By default, all virtual addresses used within the␣
↪submitted CCBs are translated using the primary context
        active at the time of the submission. The address type␣
```

```
↪field within a CCB allows each address to request
        translation in an alternate address context. The␣
↪address context used when the alternate address context is
        requested is selected in the flags argument.

        The all-or-nothing flag specifies whether the virtual␣
↪machine should allow partial submissions of the
        input CCB array. When using CCBs with␣
↪serial-conditional flags, it is strongly recommended to use
        the all-or-nothing flag to avoid broken conditional␣
↪chains. Using long CCB chains on a machine under
        high coprocessor load may make this impractical,␣
↪however, and require submitting without the flag.
        When submitting serial-conditional CCBs without the␣
↪all-or-nothing flag, guest software must manually
        implement the serial-conditional behavior at any point␣
↪where the chain was not submitted in a single API
        call, and resubmission of the remaining CCBs should␣
↪clear any conditional flag that might be set in the
        first remaining CCB. Failure to do so will produce␣
↪indeterminate CCB execution status and ordering.

        When the all-or-nothing flag is not specified, callers␣
↪should check the value of length in ret1 to determine
        how many CCBs from the array were successfully␣
↪submitted. Any remaining CCBs can be resubmitted
        without modifications.

        The value of length in ret1 is also valid when the API␣
↪call returns an error, and callers should always
        check its value to determine which CCBs in the array␣
↪were already processed. This will additionally
        identify which CCB encountered the processing error,␣
↪and was not submitted successfully.

        If the queue info flag is used during submission, and␣
↪at least one CCB was successfully submitted, the
        length value in ret1 will be a multi-field value␣
↪defined as follows:
         Bits          Field Description
         [63:48]       DAX unit instance identifier
         [47:32]       DAX queue instance identifier
         [31:16]       Reserved
         [15:0]        Number of CCB bytes successfully␣
↪submitted

        The value of status data depends on the status value.␣
↪See error status code descriptions for details.
        The value is undefined for status values that do not␣
↪specifically list a value for the status data.
```

```
        The API has a reserved input and output register which␣
↪will be used in subsequent minor versions of this
        API function. Guest software implementations should␣
↪treat that register as voltile across the function call
        in order to maintain forward compatibility.

36.3.1.1. Errors
         EOK                       One or more CCBs have been␣
↪accepted and enqueued in the virtual machine
                                   and no errors were been␣
↪encountered during submission. Some submitted
                                   CCBs may not have been␣
↪enqueued due to internal virtual machine limitations,
                                   and may be resubmitted␣
↪without changes.


                                                  530

                       Coprocessor services


EWOULDBLOCK    An internal resource conflict within the virtual␣
↪machine has prevented it from
               being able to complete the CCB submissions␣
↪sufficiently quickly, requiring
               it to abandon processing before it was complete.␣
↪Some CCBs may have been
               successfully enqueued prior to the block, and␣
↪all remaining CCBs may be
               resubmitted without changes.
EBADALIGN      CCB array is not on a 64-byte boundary, or the␣
↪array length is not a multiple
               of 64 bytes.
ENORADDR       A real address used either for the CCB array, or␣
↪within one of the submitted
               CCBs, is not valid for the guest. Some CCBs may␣
↪have been enqueued prior
               to the error being detected.
ENOMAP         A virtual address used either for the CCB array,␣
↪or within one of the submitted
               CCBs, could not be translated by the virtual␣
↪machine using either the TLB
               or TSB contents. The submission may be retried␣
↪after adding the required
               mapping, or by converting the virtual address␣
↪into a real address. Due to the
               shared nature of address translation resources,␣
↪there is no theoretical limit on
               the number of times the translation may fail,␣
↪and it is recommended all guests
```

```
                  implement some real address based backup. The␣
↪virtual address which failed
                  translation is returned as status data in ret2.␣
↪Some CCBs may have been
                  enqueued prior to the error being detected.
EINVAL            The virtual machine detected an invalid CCB␣
↪during submission, or invalid
                  input arguments, such as bad flag values. Note␣
↪that not all invalid CCB values
                  will be detected during submission, and some may␣
↪be reported as errors in the
                  completion area instead. Some CCBs may have been␣
↪enqueued prior to the
                  error being detected. This error may be returned␣
↪if the CCB version is invalid.
ETOOMANY          The request was submitted with the␣
↪all-or-nothing flag set, and the array size is
                  greater than the virtual machine can support in␣
↪a single request. The maximum
                  supported size for the current virtual machine␣
↪can be queried by submitting a
                  request with a zero length array, as described␣
↪above.
ENOACCESS         The guest does not have permission to submit␣
↪CCBs, or an address used in a
                  CCBs lacks sufficient permissions to perform the␣
↪required operation (no write
                  permission on the destination buffer address,␣
↪for example). A virtual address
                  which fails permission checking is returned as␣
↪status data in ret2. Some
                  CCBs may have been enqueued prior to the error␣
↪being detected.
EUNAVAILABLE      The requested CCB operation could not be␣
↪performed at this time. The
                  restricted operation availability may apply only␣
↪to the first unsuccessfully
                  submitted CCB, or may apply to a larger scope.␣
↪The status should not be
                  interpreted as permanent, and the guest should␣
↪attempt to submit CCBs in
                  the future which had previously been unable to␣
↪be performed. The status
                  data provides additional information about scope␣
↪of the retricted availability
                  as follows:
                  Value        Description
                  0            Processing for the exact CCB␣
↪instance submitted was unavailable,
                               and it is recommended the guest␣
↪emulate the operation. The
```

```
                          guest should continue to submit all␣
↪other CCBs, and assume no
                          restrictions beyond this exact CCB␣
↪instance.
             1            Processing is unavailable for all␣
↪CCBs using the requested opcode,
                          and it is recommended the guest␣
↪emulate the operation. The
                          guest should continue to submit all␣
↪other CCBs that use different
                          opcodes, but can expect continued␣
↪rejections of CCBs using the
                          same opcode in the near future.


                                531

                                                   Coprocessor␣
↪services


                                   Value     Description
                                   2         Processing is␣
↪unavailable for all CCBs using the requested CCB
                                             version, and it␣
↪is recommended the guest emulate the operation.
                                             The guest␣
↪should continue to submit all other CCBs that use
                                             different CCB␣
↪versions, but can expect continued rejections of
                                             CCBs using the␣
↪same CCB version in the near future.
                                   3         Processing is␣
↪unavailable for all CCBs on the submitting vcpu,
                                             and it is␣
↪recommended the guest emulate the operation or resubmit
                                             the CCB on a␣
↪different vcpu. The guest should continue to submit
                                             CCBs on all␣
↪other vcpus but can expect continued rejections of all
                                             CCBs on this␣
↪vcpu in the near future.
                                   4         Processing is␣
↪unavailable for all CCBs, and it is recommended
                                             the guest␣
↪emulate the operation. The guest should expect all CCB
                                             submissions to␣
↪be similarly rejected in the near future.


36.3.2. ccb_info
```

```
        trap#               FAST_TRAP
        function#           CCB_INFO
        arg0                address
        ret0                status
        ret1                CCB state
        ret2                position
        ret3                dax
        ret4                queue

       Requests status information on a previously submitted␣
↪CCB. The previously submitted CCB is identified
       by the 64-byte aligned real address of the CCBs␣
↪completion area.

       A CCB can be in one of 4 states:


        State                     Value       Description
        COMPLETED                 0           The CCB has been␣
↪fetched and executed, and is no longer active in
                                              the virtual␣
↪machine.
        ENQUEUED                  1           The requested CCB␣
↪is current in a queue awaiting execution.
        INPROGRESS                2           The CCB has been␣
↪fetched and is currently being executed. It may still
                                              be possible to␣
↪stop the execution using the ccb_kill hypercall.
        NOTFOUND                  3           The CCB could not␣
↪be located in the virtual machine, and does not
                                              appear to have␣
↪been executed. This may occur if the CCB was lost
                                              due to a hardware␣
↪error, or the CCB may not have been successfully
                                              submitted to the␣
↪virtual machine in the first place.

               Implementation note
               Some platforms may not be able to report CCBs␣
↪that are currently being processed, and therefore
               guest software should invoke the ccb_kill␣
↪hypercall prior to assuming the request CCB will never
               be executed because it was in the NOTFOUND state.


                                                       532

                                             Coprocessor␣
↪services
```

```
        The position return value is only valid when the state␣
↪is ENQUEUED. The value returned is the number
        of other CCBs ahead of the requested CCB, to provide a␣
↪relative estimate of when the CCB may execute.

        The dax return value is only valid when the state is␣
↪ENQUEUED. The value returned is the DAX unit
        instance indentifier for the DAX unit processing the␣
↪queue where the requested CCB is located. The value
        matches the value that would have been, or was,␣
↪returned by ccb_submit using the queue info flag.

        The queue return value is only valid when the state is␣
↪ENQUEUED. The value returned is the DAX
        queue instance indentifier for the DAX unit processing␣
↪the queue where the requested CCB is located. The
        value matches the value that would have been, or was,␣
↪returned by ccb_submit using the queue info flag.

36.3.2.1. Errors

         EOK                       The request was proccessed␣
↪and the CCB state is valid.
         EBADALIGN                 address is not on a 64-byte␣
↪aligned.
         ENORADDR                  The real address provided␣
↪for address is not valid.
         EINVAL                    The CCB completion area␣
↪contents are not valid.
         EWOULDBLOCK               Internal resource␣
↪contraints prevented the CCB state from being queried at this
                                   time. The guest should␣
↪retry the request.
         ENOACCESS                 The guest does not have␣
↪permission to access the coprocessor virtual device
                                   functionality.

36.3.3. ccb_kill

         trap#          FAST_TRAP
         function#      CCB_KILL
         arg0           address
         ret0           status
         ret1           result

        Request to stop execution of a previously submitted␣
↪CCB. The previously submitted CCB is identified by
        the 64-byte aligned real address of the CCBs␣
↪completion area.
```

```
        The kill attempt can produce one of several values in␣
↪the result return value, reflecting the CCB state
        and actions taken by the Hypervisor:

         Result                Value        Description
         COMPLETED             0            The CCB has been␣
↪fetched and executed, and is no longer active in
                                            the virtual machine.
↪ It could not be killed and no action was taken.
         DEQUEUED              1            The requested CCB␣
↪was still enqueued when the kill request was
                                            submitted, and has␣
↪been removed from the queue. Since the CCB
                                            never began␣
↪execution, no memory modifications were produced by
                                            it, and the␣
↪completion area will never be updated. The same CCB may
                                            be submitted again,␣
↪if desired, with no modifications required.
         KILLED                2            The CCB had been␣
↪fetched and was being executed when the kill
                                            request was␣
↪submitted. The CCB execution was stopped, and the CCB
                                            is no longer active␣
↪in the virtual machine. The CCB completion area
                                            will reflect the␣
↪killed status, with the subsequent implications that
                                            partial results may␣
↪have been produced. Partial results may include full


                                                      533

                                                Coprocessor␣
↪services


         Result                 Value        Description
                                             command execution␣
↪if the command was stopped just prior to writing
                                             to the completion␣
↪area.
         NOTFOUND               3            The CCB could not␣
↪be located in the virtual machine, and does not
                                             appear to have␣
↪been executed. This may occur if the CCB was lost
                                             due to a hardware␣
↪error, or the CCB may not have been successfully
                                             submitted to the␣
↪virtual machine in the first place. CCBs in the state
                                             are guaranteed to␣
```

```
↪never execute in the future unless resubmitted.

36.3.3.1. Interactions with Pipelined CCBs

         If the pipeline target CCB is killed but the pipeline␣
↪source CCB was skipped, the completion area of the
         target CCB may contain status (4,0) "Command was␣
↪skipped" instead of (3,7) "Command was killed".

         If the pipeline source CCB is killed, the pipeline␣
↪target CCB's completion status may read (1,0) "Success".
         This does not mean the target CCB was processed; since␣
↪the source CCB was killed, there was no
         meaningful output on which the target CCB could␣
↪operate.

36.3.3.2. Errors

          EOK                          The request was proccessed␣
↪and the result is valid.
          EBADALIGN                    address is not on a␣
↪64-byte aligned.
          ENORADDR                     The real address provided␣
↪for address is not valid.
          EINVAL                       The CCB completion area␣
↪contents are not valid.
          EWOULDBLOCK                  Internal resource␣
↪contraints prevented the CCB from being killed at this time.
                                       The guest should retry the␣
↪request.
          ENOACCESS                    The guest does not have␣
↪permission to access the coprocessor virtual device
                                       functionality.

36.3.4. dax_info
          trap#            FAST_TRAP
          function#        DAX_INFO
          ret0             status
          ret1             Number of enabled DAX units
          ret2             Number of disabled DAX units

         Returns the number of DAX units that are enabled for␣
↪the calling guest to submit CCBs. The number of
         DAX units that are disabled for the calling guest are␣
↪also returned. A disabled DAX unit would have been
         available for CCB submission to the calling guest had␣
↪it not been offlined.

36.3.4.1. Errors

          EOK                          The request was proccessed␣
```

```
↪and the number of enabled/disabled DAX units
                                   are valid.

                                                           534
```