
# VSI OpenVMS x86-64 Driver Developer Guide for the I/O Buffer Descriptor

**Publication Date:** October 2024

**Operating System:** VSI OpenVMS x86-64

**VMS Software, Inc. (VSI)**
**Boston, Massachusetts, USA**

# Table of Contents

# 1. Mapping User Buffers for DMA

This document describes a transitional approach to using an I/O Buffer Descriptor (IOBD), which is the mechanism for OpenVMS x86-64 that will replace the SVAPTE/BOFF/BCNT triplet and the Direct I/O Buffer Map (DIOBM).

Device drivers that perform Direct Memory Access (DMA) need to provide mapping information to the controller's DMA engine in order to transfer data to or from the user buffer that has been locked in memory.

However, the user buffer may not reside in physically contiguous memory. Because of this, a scatter/gather list is used to describe each non-contiguous segment of a buffer consisting of pairs of the following:

- Physical address of a buffer segment
- Byte count of physically contiguous bytes for this segment

The handoff method to the controller's DMA engine is specific to each type or family of controller. What is common to all of them, however, is that each entry in the scatter/gather list must reference a single range of physically contiguous memory.

The driver is responsible for creating this scatter/gather list and providing it to the device-specific Host Bus Adapter (HBA).

## 1.1. Mapping User Buffers for DMA Prior to x86-64

The user buffer that is passed to a device driver for I/O on platforms prior to x86-64 was typically described by a three-element triplet within the I/O Request Packet (IRP) consisting of the following:

- IRP$L_SVAPTE – System Virtual Address of a Page Table Entry
- IRP$L_BOFF – Byte offset into the first page
- IRP$L_BCNT – Total byte count of the transfer

IRP$L_SVAPTE is a 32-bit System Virtual Address (SVA) that points to the first element in a virtually contiguous list of Page Table Entries (PTEs).

If a buffer is in system space, IRP$L_SVAPTE will normally point to the first element of a list of real PTEs in a page table. PTEs (and the PT pages that contain them) are as virtually contiguous as the buffers that they map, so incrementing a SVAPTE by the size of a PTE simply crosses PT boundaries as needed.

If a buffer is in process space, which cannot be accessed from kernel mode, IRP$L_SVAPTE will normally point to the first element of a list of PTE copies in a DIOBM structure that is also located in the IRP. However, a driver may not modify these PTEs to point to a different page of memory or modify the protection bits contained within these PTEs.

Drivers on these platforms also have to handle the byte offset into the first page of the buffer.

Prior to x86-64, every driver needing to perform DMA had to include this user buffer mapping code within the driver itself. This meant having copies of this mapping code in every driver. If one driver needed changing, then other drivers would likely also need changing.

## 1.2. Mapping User Buffers for DMA for x86-64

OpenVMS on x86-64 does not implement the 32-bit Page Table space. As PTEs now all have 64-bit SVAs, and since x86-64 supports multiple page sizes and PTE formats, direct access to PTEs outside of the memory management code is not supported. This prohibits device drivers from accessing SVAPTEs. The method of mapping user buffers using the SVAPTE/BOFF/BCNT triplet has been replaced with a different buffer mapping mechanism on x86-64.

OpenVMS on x86-64 provides a new data structure called an IOBD. This new structure, created in the I/O Exec, contains all of the required user buffer mapping information necessary for performing DMA. Device drivers need only copy the mapping information into their respective scatter/gather structures and hand it off to the device.

This document describes the device driver interface for using an IOBD to map a user buffer for DMA on the x86-64 platform.

# 2. IOBD Overview

An IOBD is used to map a virtual address range to a sequence of Extents (physical address and byte count pairs), each of which describes a range of physically contiguous memory. An IOBD contains a list of Extents that describe the non-contiguous buffer segments. The purpose of an IOBD is to support drivers which need physical addresses in order to set up DMA transactions. The IOBD replaces the DIOBM because direct access to PTEs is restricted to the memory management code.

The two types of IOBDs are described below.

## 2.1. Base IOBD

A base IOBD has space for up to four Extents. If the user buffer may be mapped in four Extents or less, only a base IODB is required.

In a base IOBD, the Extent vectors for the first Extent are:

- IOBD$IQ_BASE_PA_VECTOR
- IOBD$IL_BASE_LENGTH_VECTOR

---

**Note**

An IRP always contains an imbedded base IOBD. The Extents in the IOBD are referenced through the IRP vector IRP$PQ_EXTENT.

---

## 2.2. Auxiliary IOBD

If the user buffer is fragmented such that it requires more than four Extents to fully map, an auxiliary IOBD is required. An auxiliary IOBD is sized dynamically to hold the exact number of Extents required to map this buffer and is allocated from non-paged pool.

In an auxiliary IOBD, the Extent vectors for the first Extent are:

- IOBD$PQ_AUX_PA_VECTOR

- IOBD$PQ_AUX_LENGTH_VECTOR

All IOBDs contain the IOBD$IL_FLAGS longword. The IOBD$IL_FLAGS longword contains bits that are used for IOBD management. The following table describes the bit definitions in the IOBD $IL_FLAGS longword:

| Flag Name | Flag Description |
|---|---|
| IOBD$V_INUSE | This IOBD is in use. |
| IOBD$V_AUX_EXTENTS | IOBD$PQ_EXTENTS points to an auxiliary Extents list. |
| IOBD$V_AUX_INUSE | IOBD$PQ_AUX_IOBD points to an auxiliary IOBD. |
| IOBD$V_REL_DEALLOC | Deallocate this IOBD on release. |
| IODB$V_AUX_IODB | This is an auxiliary IOBD. |

An auxiliary IOBD can be recognized by the IOBD$V_AUX_IOBD bit being set in IOBD$IL_FLAGS. Because auxiliary IOBDs always come from non-paged pool, each will have IOBD$V_REL_DEALLOC set, indicating that when the IOBD is released it must be deallocated. An auxiliary IOBD is never reused.

### Note

All of the Extents that map a buffer must be contained in one IOBD. This means they will all be contained in a base IOBD or all contained in an auxiliary IOBD. Extents do not start in a base IOBD and continue in an auxiliary IOBD.

## 2.3. Extents

The following table shows four ways of looking at a single buffer that spans six pages, assuming an 8KB page size. All values are hexadecimal values.

| Page Num | Virtual Extent | | SVAPTE/BOFF/ BCNT Triplets | | | Physical Pages | | | Physical Extents | |
|---|---|---|---|---|---|---|---|---|---|---|
| | VA | BCNT | SVAPTE | BOFF | BCNT | PA | BOFF | BCNT | PA | BCNT |
| 0 | 83527600 | A980 | 83526000 | 1600 | 0A00 | 40208000 | 1600 | 0A00 | 40209600 | 2A00 |
| 1 | | | 83528000 | 0000 | 2000 | 4020A000 | 0000 | 2000 | | |
| 2 | | | 8352A000 | 0000 | 2000 | 29304000 | 0000 | 2000 | 29304000 | 2000 |
| 3 | | | 8352C000 | 0000 | 2000 | 49AB6000 | 0000 | 2000 | 49AB6000 | 5F80 |
| 4 | | | 8352E000 | 0000 | 2000 | 49AB8000 | 0000 | 2000 | | |
| 5 | | | 83530000 | 0000 | 1F80 | 49ABA000 | 0000 | 1F80 | | |

**Virtual Extent**

All pages in the buffer are virtually contiguous. It can be described by a single Virtual Address (VA) and a single byte count.

**SVAPTE/BOFF/BCNT Triplet**

The buffer's use of any virtual page is described by a SVAPTE/BOFF/BCNT triplet. This information was used by drivers prior to x86-64.

**Physical Pages**

Each SVAPTE maps a single physical page. Even though the buffer is virtually contiguous, it can reference physical pages that are not only discontiguous but can also appear in any order. Note that physical page 2 is at a lower address than physical page 1, and that there are physical address gaps between pages 1 and 2 and between pages 2 and 3.

**Physical Extents**

These are the Extents (physical address and byte count pairs) that will appear in an IOBD. Each one is an aggregate of a set of physically contiguous pages and has a byte count that only includes bytes mapped by the virtual Extent. Using physical Extents allows a developer to not concern themselves with page size and byte offset into the first page of the buffer. This makes drivers less complex on x86-64.

The physical address contained in the first Extent always points to the first byte of the buffer. This means that it does not point to the first byte of the page (as was in the prior SVAPTE case) unless the buffer happens to start on a page boundary. This eliminates the need for a byte offset field. If the byte address of the first byte of the page is necessary, it can be calculated by masking off the least significant 13 bits of the physical address of the buffer.

Extents are never split across IOBDs. Either the base IOBD has all Extents, or the auxiliary IOBD has all Extents. If there is an auxiliary IOBD, then it will always have Extents.

The IOBD$IL_EXTENT_COUNT offset contains the number of Extents that map the buffer. It is only valid in the IOBD with Extents. In an auxiliary IOBD, this is the exact number of Extents in the Extent list. However, in a base IOBD this may be fewer than the maximum number of Extents (4) for which there is room.

An Extent is used to describe all of (or a segment of) a buffer by a 64-bit address and a 32-bit length, each of which is quadword aligned.

The following is the structure of an Extent from extdef.h. The fields with a reference number are the fields that a driver uses when scanning the Extent list to build the scatter/gather list for the device.

```
$ LIB/TEXT/EXTRACT=EXTDEF/OUT=TT: SYS$SYSDEVICE:[VMS$COMMON.SYSLIB]SYS
$LIB_C.TLB
typedef struct _ext {
    __union {
              unsigned __int64 ext$q_address; ❶
              __struct {
                        unsigned int ext$l_address_low; ❷
                        unsigned int ext$l_address_high; ❸
              } ext$r_addr_longwords;
   } ext$r_address_union;
    __union {
              __struct {
                        int ext$l_length; ❹
                        int ext$l_mbz; ❺
              } ext$r_length_struct;
              unsigned __int64 ext$q_length; ❻
   } ext$r_length_union
   PTE ext$r_common_pte;      /* PTE bits common to all pages in EXT */
} EXT;
```

❶ Quadword-aligned 64-bit physical address of starting byte of Extent.

❷ Quadword-aligned lower 32-bits or the physical address.
❸ Longword-aligned upper 32-bits of the physical address.
❹ Quadword-aligned 32-bit length (in bytes) of this Extent.
❺ Flag for the end of Extent list. If this field is zero, then this Extent is valid. If this field is non-zero, then this Extent is not valid. Useful to test for "end of Extent list" when drivers are scanning the Extent list.
❻ Quadword-aligned 64-bit length field. Note that this field includes the MBZ field above.

# 3. IOBD Management Routines

The following IOBD routines are provided to drivers for IOBD management.

## IOC$CREATE_IOBD

IOC$CREATE_IOBD — Creates an auxiliary IOBD to be filled. Optionally, fills in the Extents if a virtual address was supplied.

### Prototype

```
int ioc$create_iobd (VOID_PQ va, int byte_count, uint32 flags, IOBD_PPQ iobd_pointer)
```

### Parameters

| Name | Access | Description |
|---|---|---|
| va | Input | Virtual address of the buffer to be mapped |
| byte_count | Input | Size (in bytes) of the buffer to be mapped |
| flags | Input | Flags (see below) |
| iobd_pointer | Output | Pointer to the pointer to the allocated IOBD |

### Returns

Status indicating the success or failure of the operation.

### Return Values

| | |
|---|---|
| SS$_NORMAL | The routine completed successfully. |
| SS$_BADPARAM | Invalid buffer size. |
| SS$_NOWAIT | No memory available for IOBD, but the caller indicated not to wait for it. |

### Description

IOC$CREATE_IOBD creates an auxiliary IOBD to be filled. In this case, there is no base IOBD involved. The caller may specify whether or not to wait if there is no memory available for an IOBD using the flags argument.

The flags argument is as follows:

| Flag bit | Action |
|---|---|
| IOBD$M_NORESWAIT | Do not allow a resource wait. Return SS$_NOWAIT instead. |

If a virtual address was specified in the VA argument, it fills the Extents and sets the iobd$v_inuse bit.

The caller must eventually call the `IOC$RELEASE_IOBD` routine to deallocate the auxiliary IOBD.

# IOC$FILL_IOBD

IOC$FILL_IOBD — Fills the passed-in IOBD with Extents which map the buffer described by VA and BYTE_COUNT.

## Prototype

```
int ioc$fill_iobd (IOBD_PQ iobd_pointer, VOID_PQ va, int byte_count, uint32 flags)
```

## Parameters

| Name | Access | Description |
|---|---|---|
| `iobd_pointer` | Input | Pointer to the allocated IOBD |
| `va` | Input | Virtual address of the buffer to be mapped |
| `byte_count` | Input | Size (in bytes) of the buffer to be mapped |
| `flags` | Input | Flags (see below) |

## Returns

Status indicating the success or failure of the operation.

## Return Values

| | |
|---|---|
| SS$_NORMAL | The routine completed successfully. |
| SS$_BADPARAM | Invalid buffer size. |
| SS$_NOWAIT | No memory available for IOBD, but the caller indicated not to wait for it. |

## Description

`IOC$FILL_IOBD` is most commonly invoked by I/O Exec routines when a request is being issued, so it normally runs at or below IPL$_ASTDEL in the process context. However, it may also be used by a driver to map a local buffer using an IOBD allocated on the stack or from non-paged pool.

This routine is called with the address of a base IOBD to create Extents for a particular buffer. If more Extents are required than will fit into the base IOBD, then an auxiliary IOBD will be allocated and connected to the base IOBD by the IOBD$PQ_AUX_IOBD field and the IOBD$V_AUX_INUSE bit in IOBD$IL_FLAGS.

When an IOBD has valid Extents or references an auxiliary IOBD with valid Extents, the IOBD$V_INUSE bit of the IOBD will be set.

The caller may specify whether or not to wait if there is no memory available for an auxiliary IOBD (if one is required) using the flags argument. The only supported flag is IOBD$M_NORESWAIT. This flag instructs the routine that, if it is necessary to stall for any resources (which is currently just non-paged pool), it should return a failure instead of entering a resource wait state. The default is to wait for resources, but even without this flag the routine will return failure rather than stall if the IPL is above ASTDEL.

The caller must eventually call `IOC$RELEASE_IOBD` to deallocate an auxiliary IOBD if one is connected.

The flags argument is as follows:

| Flag bit | Action |
|---|---|
| IOBD$M_NORESWAIT | Do not allow a resource wait. Return SS$_NOWAIT instead. |

# IOC$RELEASE_IOBD

IOC$RELEASE_IOBD — Release an auxiliary IOBD.

## Prototype

```
int ioc$release_iobd (PQ iobd_pointer)
```

## Parameters

| Name | Access | Description |
|---|---|---|
| `iobd_pointer` | Input | Pointer to the allocated IOBD |

## Returns

Status indicating the success or failure of the operation.

## Return Values

| | |
|---|---|
| SS$_NORMAL | The routine completed successfully and the IODB was deallocated. |
| SS$_NOTHINGDONE | The routine completed successfully and the IODB was not deallocated. |

## Description

`IOC$RELEASE_IOBD` is called to release an auxiliary IOBD, whether it was allocated with the IOC$CREATE_IOBD routine or through the IOC$FILL_IOBD routine.

It is best to call this routine for every IOBD allocated with either routine to keep the counters correct.

# IOC$FIRST_EXTENT

IOC$FIRST_EXTENT — Returns a pointer to the first Extent in the specified IOBD Extent list.

## Prototype

```
int ioc$first_extent (IOBD_PQ iobd_pointer, EXT_PPQ ppq_extent, int *p_extent_boff)
```

## Parameters

| Name | Access | Description |
|---|---|---|
| `iobd_pointer` | Input | Pointer to the allocated IOBD |
| `ppq_extent` | Output | Pointer to the pointer to the first Extent in the Extent list |

| Name | Access | Description |
|---|---|---|
| *p_extent_boff | Output | Pointer to the Extent byte offset |

### Returns

Status indicating the success or failure of the operation.

### Return Values

| | |
|---|---|
| SS$_NORMAL | The routine completed successfully. |
| SS$_NOSUCHEXT | No Extent was found. |

### Description

IOC$FIRST_EXTENT is called to return the address of the first Extent in the specified IOBD Extent list. The address is returned in the ppq_extent parameter.

---

**Note**

The address of the first Extent can also be retrieved using the iobd$pq_extents offset in the IOBD.

---

The *p_extent_boff parameter may be ignored. It is always zero for the first Extent.

# 4. Driver Developer Guidelines

## 4.1. General Guidelines

- All IRP SVAPTE related symbols are undefined on the x86-64 platform.
- An IOBD will always come into a driver in an IRP. The first Extent may be accessed through the IRP pointer IRP$PQ_EXTENT.
- All of the Extents that map a buffer must be contained in one IOBD. This means they will all be contained in a base IOBD or all contained in an auxiliary IOBD. Extents do not start in a base IOBD and continue in an auxiliary IOBD.
- An IOBD address is always a quadword.

## 4.2. Header Files

The following header files contain the symbols and offsets necessary for IOBD use for drivers written in C:

```
#include <iobddef.h>                    /* I/O Buffer Descriptor structure */
#include <extdef.h>                     /* Extent (address + length) structure */
```

The following header files contain the symbols and offsets necessary for IOBD use for drivers written in MACRO:

```
$IOBDDEF                                ; I/O Buffer Descriptor structure
```

```
$EXTDEF                                 ; Extent (address + length) structure
```

# 4.3. Driver IOBD Handling of I/O Requests Originated by $QIO

For most I/O requests (e.g. $QIO), IOBDs are used by drivers as follows:

1. The I/O Exec fills an IOBD embedded in the IRP with the Extents of the user buffer. IRP$PQ_EXTENT points to the first Extent.

2. When the driver is entered, the driver uses IRP$PQ_EXTENT to access the first Extent and scan through the Extent list, copying the Extent info to the scatter/gather entries for the HBA. Here are three methods a driver can use to scan through the Extent list using a loop:

   **Method One**

   Use IOBD$IL_EXTENT_COUNT as a loop counter. However, IOBD$IL_EXTENT_COUNT is only valid in the IOBD that contains the Extents. To use this method, the driver must do the following:

   a. Get the IOBD address using IRP$R_IOBD.

   b. Use IOBD$IL_EXTENT_COUNT for the loop index.

   ---

   **Note**

   IOBD$IL_EXTENT_COUNT is only valid in the IOBD that has the Extents (e.g. Base or Auxiliary) . Use the bits in the IOBD$IL_FLAGS longword to determine the correct IOBD to use.

   ---

   **Method Two**

   For each loop iteration, check EXT$L_MBZ in each Extent. If EXT$L_MBZ is zero, then the Extent is valid. If EXT$L_MBZ is non-zero, then you are at the end of the Extent list. Break out of the loop.

   **Method Three**

   For each loop iteration, subtract the current Extent byte count from the total byte count until a zero result is reached. Break out of loop.

3. I/O post processing releases the IRP/IOBD.

## 4.3.1. Code Example

The following example uses Method Three from above:

```
#include <extdef.h> /* Extent (address + length) structure */
EXT_PQ extent = 0L;
int byte_count = <total_byte_count>;
extent = irp->irp$pq_extent;
index = 0;
/* Break inside the loop */
for (;;)
```

```
{
        (*sge)->Addr[index] = extent->ext$q_address;
        (*sge)->Len[index] = extent->ext$l_length;

        /* Exit loop if no more bytes to map */
        if ((bcnt -= ext$l_length) == 0)
            break;        /* Done. Mapped all data */

        /* More to go */
        extent++;         /* Step to next Extent */
        (*sge)++;         /* Advance controller's SGE pointer */
        index++;          /* Index to next Extent */
}
/* All done */
return SS$_NORMAL;
```

# 4.4. Driver IOBD Handling of I/O Requests Originated Within the Driver

In certain scenarios, a driver needs to allocate and map an internal buffer for driver use. In this case, the driver will have to manage (allocate, fill, release) an IOBD as follows:

1. Driver allocates a buffer from non-paged pool.
2. Driver calls the IOC$CREATE_IOBD routine, passing the SVA of the buffer to allocate an IOBD and fill the Extent list for the buffer.
3. Driver calls the IOC$FIRST_EXTENT routine to get the pointer to the first Extent in the Extent list.
4. Driver scans the Extent list and copies the Extent information into the scatter/gather list for the device.
5. Driver calls the IOC$RELEASE_IOBD routine.

## 4.4.1. Code Example

```
#include <iobddef.h>             /* I/O Buffer Descriptor structure */
#include <extdef.h>              /* Extent (address + length) structure */
IOBD_PQ iobd = NULL;
EXT_PQ extent = 0L;
int byte_count = <total_byte_count>;
int index;

/* Get a buffer out of non-paged pool */
exe_std$alononpaged(BUFFER_SIZE, &size, (void**)&va_buffer);

/* Allocate and fill an IOBD with Extents that map allocated buffer */
status = ioc$create_iobd(va_buffer, size, IOBD$M_NORESWAIT, iobd);
if ($VMS_STATUS_SUCCESS(status)) {
      status = ioc$first_extent(*iobd, ext, &extent_boff);
      if (!$VMS_STATUS_SUCCESS(status)) {
          ioc$release_iobd(*iobd);  /* Error status */
          return (status);
      }
}
/* Success status */
```

```
extent = irp->irp$pq_extent;
index = 0;
/* Break inside the loop */
for (;;)
{
        (*sge)->Addr[index] = extent->ext$q_address;
        (*sge)->Len[index] = extent->ext$l_length;

         /* Exit loop if no more bytes to map */
         if ((bcnt -= ext$l_length) == 0)
                  break;     /* Done. Mapped all data */

         /* More to go */
         extent++;           /* Step to next Extent */
         (*sge)++;           /* Advance controller's SGE pointer */
         index++;            /* Index to next Extent */
}

ioc$release_iobd(*iobd);            /* Done with IOBD */

/* All done */
return SS$_NORMAL;
```

# 5. Debugging Info

## 5.1. IOBD Code Counters

These counters are used to determine the level of IOBD use and to help detect or troubleshoot IOBD related pool leakage.

The counters with a reference number are useful for driver developers, whereas the other counters are for VSI use.

```
$ ANALYZE/SYSTEM
SDA> READ/EXEC
SDA> SHOW SYMBOL IOBD_*

Symbols sorted by name
----------------------
IOBD_CTR$Q_CREATE_CALLS ❶       = FFFFFFFF.804BF0C0 : 00000000.000527A0
IOBD_CTR$Q_EXT_ALLOCS           = FFFFFFFF.804BF0F8 : 00000000.0008BA9A
IOBD_CTR$Q_EXT_COUNT_HISTOGRAM  = FFFFFFFF.804BF148 : 00000000.05E08F5F
IOBD_CTR$Q_EXT_DEALLOCS         = FFFFFFFF.804BF100 : 00000000.0008BA9A
IOBD_CTR$Q_EXT_POOL             = FFFFFFFF.804BF108 : 00000000.00000000
IOBD_CTR$Q_FILL_CALLS ❷         = FFFFFFFF.804BF0B8 : 00000000.05DB7147
IOBD_CTR$Q_FIRST_EXTENT_CALLS ❸ = FFFFFFFF.804BF0D0 : 00000000.00000000
IOBD_CTR$Q_INV_ATT_IOBDS        = FFFFFFFF.804BF140 : 00000000.00000000
IOBD_CTR$Q_INV_IRP_SIZES        = FFFFFFFF.804BF120 : 00000000.00000000
IOBD_CTR$Q_INV_IRP_TYPES        = FFFFFFFF.804BF118 : 00000000.00000000
IOBD_CTR$Q_INV_PTE_TRANS        = FFFFFFFF.804BF110 : 00000000.00000000
IOBD_CTR$Q_INV_UCB_SIZES        = FFFFFFFF.804BF138 : 00000000.00000000
IOBD_CTR$Q_INV_UCB_SVAS         = FFFFFFFF.804BF128 : 00000000.00000000
IOBD_CTR$Q_INV_UCB_TYPES        = FFFFFFFF.804BF130 : 00000000.00000000
IOBD_CTR$Q_IOBD_ALLOCS ❹        = FFFFFFFF.804BF0E0 : 00000000.000527A0
IOBD_CTR$Q_IOBD_DEALLOCS ❺      = FFFFFFFF.804BF0E8 : 00000000.000527A0
```

```
IOBD_CTR$Q_IOBD_POOL ❻          = FFFFFFFF.804BF0F0 : 00000000.00000000
IOBD_CTR$Q_NEXT_SEGMENT_CALLS   = FFFFFFFF.804BF0D8 : 00000000.00000000
IOBD_CTR$Q_PTE_4KB_CHECKS       = FFFFFFFF.804BF0A8 : 00000000.00000000
IOBD_CTR$Q_PTE_4KB_MISMATCHES   = FFFFFFFF.804BF0B0 : 00000000.00000000
IOBD_CTR$Q_RELEASE_CALLS ❼      = FFFFFFFF.804BF0C8 : 00000000.000DE23A
```

❶ Number of calls to IOC$CREATE_IOBD
❷ Number of calls to IOC$FILL_IOBD
❸ Number of calls to IOC$FIRST_EXTENT
❹ Number of IOBD allocations from non-paged pool
❺ Number of IOBD deallocations to non-paged pool
❻ Number of bytes of non-paged pool currently in use for IOBDs
❼ Number of calls to IOC$RELEASE_IOBD