sun4m Architecture Porting Guide

Hal L. Stern
Sun Microsystems Computer Company
Northeast Area Systems Engineering Group
Lincoln, MA

Introduction

The sun4m architecture is Sun’s first multiprocessing system, incorporating MBus-based SPARC CPUs and the SPARC Reference MMU (SRMMU). The new architecture is an extension of the sun4 product family:

- sun4: SPARCsystem 3xx and 4xx
- sun4c: SPARCstation 1, 1+, 2, IPC, SLC, ELC, IPX
- sun4m: SPARCsystem 600MP, SPARCstation 10

The sun4m family is binary compatible with other sun4 machines, but the new kernel (system) architecture may require changes to existing drivers for SBus and VME bus devices.

This document summarizes issues concerning device driver porting. It is intended as a companion guide to the following documents:
- Writing Device Drivers
- Addendum to Writing Device Drivers
- SunOS 4.1 Performance Tuning

There are three chapters in this guide:

Section 1, System Architecture, discusses aspects of the sun4m kernel architecture that impact device drivers.

Section 2, I/O MMU and I/O Cache, gives a brief explanation of the I/O MMU in the sun4m and the I/O cache on the 600MP, and provides details on using the new DVMA spaces.

Section 3, Kernel Memory, describes differences between the SunOS 4.1.1 and 4.1.2 kernel memory management, mbuf and buffer cache systems.

1. System Architecture

A sun4m machine is a tightly-coupled, shared memory multiprocessor that runs a single, shared kernel image. Any processor may trap into the kernel, avoiding the high context switching overhead normally associated with asymmetric multiprocessors. Access to the kernel code and data is governed by a single kernel lock: upon entry into the kernel, a processor requests the lock and will spin waiting for it if the lock is held by another processor. The kernel lock is released when the processor holding it traps back to user space.
In addition to the kernel locking code, the SunOS 4.1.2 kernel contains a modified scheduler and interrupt steering logic. However, these MP-specific changes are primarily invisible to device drivers (and certainly transparent to user-level applications). Kernel facilities maintain their same interfaces and calling conventions. For example, a driver that must send a signal to process does not need to know the processor on which the process is running; the `psig()` routine locates the process and performs cross-CPU communication when required. Kernel locks are handled above the device driver level by the generic trap and operating system routines that dispatch to the device drivers; lower-level interrupt distribution and cache management are handled below the device driver.

1.1. **Bus Relationships**

Both SBus and VME bus devices are supported on the SPARCsystem 600MP. During autoconfiguration, the SBus devices will be configured first, followed by the usual probe of VME devices. Drivers for VME devices do not have to be modified to use the `devinfo` structures created and managed by the OpenBoot PROM; they will continue to be probed and register themselves as in other `sun4` systems.

The following table summarizes the bus master-slave relationships for DVMA transfers on `sun4m` systems:

<table>
<thead>
<tr>
<th>Bus Slave</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SBus</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>VME bus</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

DVMA is not possible between an SBus master and a VME slave device. To move data between an SBus master (such as a disk on the SCSI host adaptor) to a VME bus slave (such as a VME memory card), data must first be transferred into CPU memory (a kernel or user buffer) and then copied out to the VME slave device.

1.2. **Interrupt Levels**

The interrupt level mappings on a `sun4m` machine are different than those of a `sun4` or `sun4c`, due to the presence of both SBus and VME bus interfaces on the 600MP. SBus and VME bus interrupts map into the following SPARC CPU interrupt levels:

<table>
<thead>
<tr>
<th>SPARC IRL</th>
<th>VME</th>
<th>SBus</th>
<th>On-board</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>SCSI</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>Ethernet</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SPARC IRL    | VME | SBus | On-board
----------|-----|------|---------
    7    |  4  |  4  | Video   
    8    |     |     |         
    9    |  5  |  5  | Counter/Timer 
   10    |     |     | Floppy   
   11    |  6  |  6  | Keyboard, mouse, serial ports 
   12    |     |     | Audio    
   13    |  7  |  7  |         

SPARC interrupt request levels above IRL 13 are reserved for cross-CPU or kernel-specific functions and are not usable by SBus or VME devices.

Devices interrupting on SBus level 5 formerly came in below the VM system, but now they are mapped above the IRL used by the VM system. Drivers that called directly into the VM system may have worked without calling adddma(), but these drivers should now be certain to use adddma(). Calling adddma() for interrupt levels at or above IRL 10 may cause clock ticks to be lost, and is not recommended.

In general, an SBus device that interrupts at SBus level 6 or above and behaves properly on a sun4c may not function properly on a sun4m machine. Level 6 interrupts are mapped above the hardclock interrupt on a sun4m. Network or serial line interfaces that require softclock processing of their queues must interrupt at SBus level 5 or below to ensure that the hardclock will be able to run under high input loads.

1.3. Driver Portability

In general, well-behaved SBus and VME bus device drivers should migrate to a sun4m kernel with no or few modifications. “Well-behaved” is minimally defined as:

- The driver does not peek into a DVMA buffer without properly mapping it into the kernel’s address space.
- Only the standard mb_XXX() bus setup and tear-down routines are used to effect mappings between kernel and device address spaces.
- The driver does not modify kernel page table entries or mappings, nor does it call the hat_XXX() or segkmem_XXX() routines.

The primary difference between the sun4m and earlier sun4 systems (from the device driver implementer’s perspective) is the introduction of an I/O MMU (IOMMU). Details on managing IOMMU mappings are provided in the next section. The 600MP has an I/O cache (IOC) which is similar to the same cache present on the SPARCserver 490. There are also write buffers between the MBus and the SBus/VME bus interfaces, described in the next section.

I/O cache dependencies should be conditionally compiled using the IOC constant. Code included to support the sun4m architecture, which may include IOC or IOMMU setup, should be surrounded with sun4m conditional compilation directives:

```
#ifdef sun4m
```

- 3 - Rev 2.0 of 10 August 1992
In general, device drivers that use the default SBus and VME I/O mappings should remain portable. Drivers that create mappings larger than 1 Mbyte will be accelerated by the larger SBus and VME I/O maps available on the \texttt{sun4m} platform, but using the larger VME map may make the driver less portable. Mappings, I/O space and IOC interactions are discussed in Chapter 2.

Although the interrupt steering and trap handling is invisible to most device drivers, care should be taken not to call routines that change the CPU priority level as a side effect. A device that interrupts at SBus priority X and was mapped to IRL Y on a sun4 or sun4c machine will now have its interrupt mapped to IRL Z on the \texttt{sun4m} architecture. Z may be greater than Y, since both VME and SBus interrupt levels are “merged” in the \texttt{sun4m} SPARC IRL mappings.

Calling a kernel service routine that changes the CPU priority to a level below the current one may result in corrupted data structures or a variety of data fault panics. The following guidelines should be observed:

- Networking and \texttt{mbuf} utilities may be called while handling a device interrupt at SBus or VME level 4 and below.
- STREAMS utilities may be called while handling a device interrupt at SBus or VME level 5 and below.
- Networking and \texttt{mbuf} utilities should not be called while the CPU priority is above IRL 7 (\texttt{splimp}).
- STREAMS utilities should not be called with the CPU priority above IRL 10 (\texttt{splstr}).

More detail may be found in the \textit{Writing Device Drivers Addendum}.

The \texttt{sun4m} \texttt{user} and \texttt{proc} structures are larger than the equivalent sun4 or sun4c structures. New fields have been added to handle multiple processors and the SRMMU. Drivers that reference \texttt{user} or \texttt{proc} structures must be recompiled on a \texttt{sun4m} machine because offsets into these structures are machine architecture-specific.

2. I/O MMU and I/O Cache

The \texttt{sun4m} IOMMU separates the I/O address space from the kernel’s memory mappings. In previous sun4 and sun4c machines, the DVMA space was carved out of the kernel’s address space, and DVMA operations shared the single MMU with the CPU. Memory, processors and the SBus interface reside on the MBus. Each non-memory module on the MBus requires an MMU to translate virtual to physical addresses: the CPU uses the SRMMU, and the SBus uses the IOMMU. VME bus access is handling through the SBus/VME bus interface, so the same SBus-MBus interconnection rules apply.

Separating the DVMA and CPU address mappings makes DVMA addresses valid only from
the point of view of a device on the SBus or VME bus. The CPU (and therefore the kernel) cannot interpret DVMA addresses that are mapped through the IOMMU; the CPU only goes through the SRMMU to access memory on the MBus. The converse is also true: valid kernel memory mappings are not necessarily also present in the IOMMU, so kernel addresses are (in general) invalid for DVMA. Some of the default I/O mappings are double-mapped using both the SRMMU and IOMMU, allowing devices and the kernel to access the same underlying physical memory. This section discusses the \textit{sun4m} DVMA memory maps, kernel facilities for their management, and the \textit{sun4m} I/O cache.

### 2.1. IOMMU Mappings

It helps to think of the IOMMU as a gateway between devices (SBus or VME) and the MBus. All DVMA masters (including the CPU) share the single IOMMU, which supports DVMA spaces of 16M - 2 Gbytes. The default DVMA space on the \textit{sun4m} series is 16 Mbytes, divided into 4 DVMA maps:

- \texttt{sbusmap}  
  Default for SBus devices, about 1 Mbyte
- \texttt{bigsbusmap}  
  8 Mbyte map for SBus devices
- \texttt{vme24map}  
  Default map for VME devices, about 1 Mbyte
- \texttt{vme32map}  
  32-bit VME map, about 6 Mbyte

32-bit VME devices may be accessed through the \texttt{vme24map}; the map name is more descriptive of the addressing range of the map rather than the devices on it. Only 32-bit devices may be mapped in the \texttt{vme32map}. Most devices will use the default maps, which retain their names from other platforms. Using the default maps ensures compatibility across \textit{sun4}, \textit{sun4c} and \textit{sun4m} platforms: SBus and VME devices will continue to use \texttt{dvmamap} and \texttt{mb_hd.mh_map} (respectively) as arguments to \texttt{mb_XXX()} routines.

Use of the other two maps may improve performance on the 600MP series at the slight expense of reduced portability. The \texttt{vme32map} allows large VME bus transfers to be performed more efficiently, with fewer map allocations and releases. Similarly, the \texttt{bigsbusmap} may be used by SBus devices with large device address spaces. The larger DVMA space allows the device to be mapped in its entirety instead of forcing the driver to manage segments of the mapping.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff000000</td>
<td>\texttt{bigsbusmap start}</td>
<td>8 Mbyte</td>
</tr>
<tr>
<td>ff7fffff</td>
<td>\texttt{bigsbusmap end}</td>
<td>2 pages</td>
</tr>
<tr>
<td>ff800000</td>
<td>\texttt{IOBP map for VME}</td>
<td>1 Mbyte</td>
</tr>
<tr>
<td>ff802000</td>
<td>\texttt{vme24map start}</td>
<td>6 Mbyte</td>
</tr>
<tr>
<td>ff900000</td>
<td>\texttt{vme24map end}</td>
<td>2 pages</td>
</tr>
<tr>
<td>ffefffff</td>
<td>\texttt{vme32map start}</td>
<td>1 Mbyte</td>
</tr>
<tr>
<td>fff00000</td>
<td>\texttt{vme32map end}</td>
<td>2 pages</td>
</tr>
<tr>
<td>fff02000</td>
<td>\texttt{IOBP map for SBus}</td>
<td>1 Mbyte</td>
</tr>
<tr>
<td>ffffffff</td>
<td>\texttt{end of sbusmap}</td>
<td></td>
</tr>
</tbody>
</table>
The vme24map is visible from the kernel: it is the only map that can be accessed through the SRMMU. The SBus maps and vme32map are only accessible through the IOMMU. The figure above shows the complete IOMMU memory map for the 600MP; the SPARCStation 10 has no VME address space. Note that the VME A24 device space starts at offset 0xff800000, which is not the same offset used by other sun4 platforms.

The IOBP areas are double-mapped (by default) into CPU and device address spaces, since the IOBP buffers are frequently used by both driver code and the devices themselves. Regions of the vme24map may be double-mapped into CPU address space as well when the DVMA mapping is created via the mb_XXX() routines.

2.2. Managing IOMMU Translations

IOMMU mappings are not synchronized with the SRMMU. Therefore, it is imperative that drivers to not interleave CPU and device accesses to memory without first calling one of the mb_XXX() setup routines. IOMMU translations may be created once at device initialization, or managed dynamically using mbsetup() and mbrelease().

The IOBP maps occupy the first two pages of the default SBus and VME bus DVMA maps, leaving slightly less than 1 Mbyte for driver-created mappings. IOBP areas are double mapped into the IOMMU and kernel’s address space, since they are frequently used for command blocks and must be accessed by both drivers and devices.

Three new flags have been added to sundev/mbvar.h to indicate that a non-default DVMA map should be used, or that the mapping should be made in both the IOMMU and the SRMMU. The following table summarizes the map names, flags to pass to the mb_XXX() routines, and the addresses used by the kernel and device:

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Map Name</th>
<th>Flags</th>
<th>Map Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBus IOBP</td>
<td>iopbmap</td>
<td></td>
<td>iopbmap</td>
</tr>
<tr>
<td>VME IOBP</td>
<td>iopbmap</td>
<td></td>
<td>iopbmap</td>
</tr>
<tr>
<td>VME A24, &lt; 1M</td>
<td>md_mh.mh_map</td>
<td></td>
<td>vme24map</td>
</tr>
<tr>
<td>VME A32, &lt; 1M</td>
<td>md_mh.mh_map</td>
<td>MDR_DVMA_PEEK</td>
<td>vme24map</td>
</tr>
<tr>
<td>VME A32, &gt; 1M</td>
<td>md_mh.mh_map</td>
<td>MDR_VME32</td>
<td>vme32map</td>
</tr>
<tr>
<td>SBus, &lt; 1M</td>
<td>dvmamap</td>
<td></td>
<td>sbusmap</td>
</tr>
<tr>
<td>SBus, &gt; 1M</td>
<td>dvmamap</td>
<td>MDR_BIGSBUS</td>
<td>bigsbusmap</td>
</tr>
</tbody>
</table>

Some additional notes on IOMMU mappings:

- The IOBP address passed to a VME device should have the offset of the DVMA area subtracted from it, but SBus IOBPs do not require this translation:
  ```c
  vme_dev_address = iopbmem - DVMA;
  sbus_dev_address = iopbmem;
  ```
- DVMA space for VME A24 devices as well as VME A32 devices that use the MDR_DVMA_PEEK flag will be double-mapped. No mapping created in the vme32map
can be seen by the kernel: these addresses are only for use by the VME devices.

- Device drivers can access the DVMA area using `DVMA[]`. If a drive allocates a mapping with both the `MDR_DVMA_PEEK` and `MDR_VME32` flags, the `vme24map` will be used instead of the `vme32map`.

- A VME A32 device can use the `vme32map` and access the DVMA area after mapping it into the kernel’s address space using `bp_mapin()` and `bp_mapout()`. In general, any driver that peeks into the DVMA area should be modified to use `bp_mapin()` and `bp_mapout()` instead.

- None of the SBus maps can be seen from the kernel. When the regular SBus DVMA map is used, the address passed to the SBus device should have the DVMA offset added to it; when the `bigsbusmap` is used, no DVMA offset is required:
  
  ```c
  dev_address = mb_mapalloc(dvmamap, MDR_BIGSBUS...);
  dev_address = mb_mapalloc(dvmamap,...) + DVMA;
  ```

Using the larger SBus and VME DVMA maps may make drivers more efficient, since they will not be competing with other devices for the smaller 1 Mbyte DVMA spaces. Disk transfers using the on-board SCSI host adaptor, for example, create mappings in the 1 Mbyte `sbusmap`, so using the `bigsbusmap` for other SBus device will reduce contention for the small DVMA space. However, using the larger SBus map makes the driver non-portable.

### 2.3. I/O Cache

The IOC on the 600MP series is a write-back cache that is coherent with main memory on a 32-byte burst basis. Only transfers with a VME bus master and either MBus (memory) or SBus slave are cached. The IOC does not affect SBus drivers. The `mb_XXX()` routines generally manage the IOC such that it is transparent to the driver. However, some knowledge of the underlying cache structure and management policies is useful for optimizing VME device drivers.

The IOC is organized as 1024 32-byte lines that act more like write buffers than a direct mapped cache. Each 32-byte line corresponds to a single 8k page in the DVMA space. Unlike the CPU’s virtual address cache, consecutive 32-byte chunks of data do not fill consecutive lines in the cache. Consecutive 32-byte blocks of data re-use the same line in the IOC; consecutive 8k pages use consecutive lines. The tag and index bits simply come from different parts of the address:

```
<table>
<thead>
<tr>
<th>CPU VAC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>tag</td>
</tr>
<tr>
<td>line index</td>
</tr>
<tr>
<td>byte index</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>line index</td>
</tr>
<tr>
<td>byte index</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>tag (2 parts)</td>
</tr>
</tbody>
</table>
```

TheIOC cache model allows an entire 8k I/O page to be cached in a single line, with repeated line fills and flushes or write backs to memory or the VME device. This is known as the “streaming I/O model.”

The 600MP I/O cache is always enabled; that is, all transfers will use the cache if they can.
The general rule is to always use the cache for writes, and to cache reads if they are aligned on a cache line size boundary. On the 600MP system, all writes are cached, since the IOC can load padding bytes to round the transfer up to a 32-byte boundary. For reads, however, the operation will only be cached if the buffer is aligned on a 32-byte boundary and the transfer size is a multiple of the line size as well. Again, the mb_XXX() setup routines will determine if an operation can be cached, and enable or disable the IOC appropriately for the mapping created. Drivers that need to check buffer alignment or transfer sizes should include machine/iocache.h and use the IOC_LINESIZE and IOC_LINEMASK constants.

Because of the asymmetry in read and write caching, it is critical that the same 8k I/O page not be used for both reading and writing without a corresponding IOMMU setup and teardown or an explicit IOC flush. Attempting to read and write from the same page may cause cache consistency problems because writes are always cached, and the cache line may include padding bytes used to align the write buffer to a 32-byte boundary.

 Normally, the IOC flush is done via a call to ioc_flush() by the mb_XXX() tear-down routines. However, a driver can explicitly call ioc_flush() with the IOC line number to be flushed as an argument. If a transfer spans more than one 8k page, ioc_flush() must be called for each page: this corresponds to flushing each cache line used. Drivers ported from other Sun machines with I/O caches should be recompiled if they call ioc_flush(), since this is a macro that is kernel architecture-specific.

2.4. Store Buffering, Interrupts and Faults

Store buffers sit between the MBus and the SBus, between the SBus and the VMEbus (in the form of the I/O cache), and between the CPU and the Mbus itself. The SunOS kernel flushes these buffers before calling a device driver interrupt routine, and any attempt to read from a cached address will cause the write buffer to be flushed as well. Consider this simple sequence of events:

   Write to device
   Delay 100 usec
   Write to device

Relative time-ordering of write operations is not preserved with the write buffers: the execution order is maintained, but the write operations may be performed back-to-back rather than with a delay between them. The actual delay may be less than 100 usec if no cache flushes occur within the delay window. If your driver relies on certain minimum inter-write time delays, you can perform a read-back from the device to force the write buffers to be flushed.

The SuperSPARC CPU has separate instruction and data caches requiring explicit cache flushes for any process that modifies a text segment. LISP interpreters, debuggers or any other self-modifying process should use explicit iflush instructions after modifying SPARC instruction sequences. Failure to flush the SuperSPARC instruction cache may result in inconsistencies between cached instructions and the modified text in the memory image of the process.
Write buffers affect VME and SBus interrupt handling and the generation of VME bus errors. Most device drivers modify a bit in the device’s CSR to clear an interrupt pending condition. Buffering between the CPU and the MBus or between SBus and VME bus will prevent these write operations from clearing the interrupt condition immediately. As a result, it’s possible for an interrupt routine to handle an interrupt, clear the bit in the CSR, and return, only to have the device continue to assert an interrupt condition. This produces the message:

\texttt{iobus interrupt at level XXX not serviced}

when the handler returns. To eliminate the buffered write problem, a driver should force a flush of all write buffers by reading from the CSR location after writing it. The following code fragment is an example of an explicit flush:

```c
xxintr()
{
    csr = 1;
    newcsr = csr;
    return(1);
}
```

After re-reading the CSR, check for another interrupt that may have occurred. The entire write-flush-readback sequence may overlap the generation of another interrupt from the device. Simply reading the CSR without checking for an interrupt will cause interrupts to be lost.

Write buffering also impacts VME bus error generation. In previous VME-based systems, VME write errors resulted in a synchronous trap. The 600MP buffering scheme makes it impossible to handle VME write errors synchronously, since the write may have been stalled in one or more buffers (Read faults, however, still result in synchronous traps). Device drivers that rely on being able to detect VME bus errors on writes must be able to restart the failed write operation asynchronously, retraining both the address and data of the write operation. No data is preserved on a VME bus write error.

3. **Kernel Memory**

SunOS 4.1.2 contains three changes to kernel memory management systems: the kernel heap, the buffer cache, and the mbuf map. In previous releases of SunOS, the kernel’s address space looked similar to that of a user-level process: text in the low-order addresses, followed by data and the heap, with the stack growing down from the high-order addresses. In SunOS 4.1.2, various dynamically managed data structure pools have been moved out of the kernel’s heap so that they can be made larger.

3.1. **Buffer Cache and Kernel Heap**

The kernel’s buffer cache is used for writing non-data blocks back to the filesystem. The buffer cache sits in between the in-core inode cache and the filesystem, and it is also used for writing cylinder group information out to disk. SunOS 4.1.2 increases the size of the buffer cache to achieve better performance on servers, and it also clamps the size of the cache at about 200 pages. The entire buffer cache now starts at the symbol \texttt{bufmap}, which is located
below the start of the kernel’s text segment.

The kernel’s heap managers - \texttt{kmem\_alloc()} and \texttt{kmem\_free()} - have also been modified to move the kernel’s dynamically managed memory pool below the kernel’s text segment. The addresses returned by \texttt{kmem\_alloc()} will no longer be above \texttt{Sysmap}, although this should be of no consequence to most drivers. A device using 20- or 24-bit addresses that requires the top byte (or 12 bits) to be 0xff will not be able to use the modified \texttt{kmem\_alloc()} facility. The \texttt{kernelmap\_alloc()} routine can be used instead, although it must be used for statically allocated buffers and not for dynamically managed storage.

\subsection*{3.2. Management of mbufs}

The most far-reaching changes affect the allocation of mbufs. In early versions of SunOS, the mbuf map was statically allocated and fixed in size. In SunOS 4.1, mbufs were dynamically allocated, with a 1 Mbyte upper bound on the size of the mbuf pool. The mbuf pool has been increased to 2 Mbytes in SunOS 4.1.2 and is based at the symbol \texttt{mbutl}.

Regular and cluster type mbufs are now automatically mapped into the IOMMU by the following routines and macros:

\begin{verbatim}
 m\_get()
 MGET()
 mclget()
 MCLGET()
\end{verbatim}

A driver can determine whether the data buffer of an mbuf has been mapped in the IOMMU by comparing the data address to the range of addresses within the mbuf pool:

\begin{verbatim}
caddr_t a;
struct mbuf *mp;
a = mtod(mp);
if ((a > mbutl) && (a < mbutil + 2*1024*1024)) {
   data buffer is already mapped
} else {
   must map address a in IOMMU
}
\end{verbatim}

The I/O address corresponding to a mapped mbuf may be found by adding the offset into the mbuf pool to \texttt{IOMMU\_MBUTL\_BASE}:

\begin{verbatim}
caddr_t a, ioaddr;
a = mtod(mp);
ioaddr = IOMMU\_MBUTL\_BASE + (a - mbutl);
\end{verbatim}

A loaned data type mbuf will not have its data buffer automatically mapped into the IOMMU when it is allocated. The driver allocated or using the mbuf is responsible for creating the mapping, using either \texttt{bp\_mapin()} or \texttt{mb\_mapalloc()}. If the buffer is already wrapped in a buf header, then \texttt{bp\_mapin()} should suffice, otherwise \texttt{mb\_mapalloc()} should be used. In most cases, \texttt{mb\_mapalloc()} is preferred: Loaned data type mbufs should be mapped into the \texttt{bigsbusmap} or \texttt{vme32map}, rather than the default 1 Mbyte maps.
default maps are shared with disk (SCSI or IPI) drivers, and can fill up quickly if large numbers of mbufs are mapped into them as well. For optimal performance of network and protocol drivers, the larger DVMA spaces should be used.