Technical white paper

Using BAP for HP OpenVMS device drivers

Table of contents

Introduction 2
Performing DMA in HP OpenVMS device drivers 2
Perform DMA using BAP 3
  Overview 3
  Verify mapped register support 5
  Steps to convert the existing HP OpenVMS driver to use BAP 5
BAP operations 5
  BAP registration 5
BAP memory allocation and de-allocation 7
  BAP memory allocation 7
BAP rules 8
  BAP SYSGEN parameters and system data cells 8
Limitations 9
Conclusion 9
References 10
Annexure 10
  HP OpenVMS skeleton driver to perform I/O using BAP 10
For more information 14

Introduction

Newer Itanium servers (Intel® Itanium® processor 9300 series onwards) do not have support for mapped registers. Device drivers that currently use mapped registers to perform direct memory access (DMA) will fail to work on such systems. These drivers need to be modified to use bus-addressable pool (BAP) instead of mapped registers. This paper details the use of BAP to perform DMA operations in HP OpenVMS.

Performing DMA in HP OpenVMS device drivers

HP OpenVMS traditionally allows a device driver developer two mechanisms to perform DMA. These two mechanisms are:

- Scatter-gather DMA
- Physical or direct DMA

PCI bus implementation allows two ways for PCI devices to access main memory namely scatter-gather memory access and physical memory access. With scatter-gather memory access, the PCI addresses are translated to a main memory address using a scatter-gather table. With physical memory access the PCI addresses are translated to a main memory address by the addition of a constant. Scatter-gather memory access is called scatter-gather DMA. Physical memory access is called physical (or direct) DMA.

When you perform scatter-gather DMA, a driver writer would use the map register mechanism to get the device mapped into the systems address space. This is achieved by using a set of tables that help to translate the device address to system address.

With direct DMA, direct/physical memory access is used. The device address is translated to system address by addition of a constant to the device address. In other words, there is a single mapping for all of PCI address space.

Figure 1. PCI memory address space mapped to main memory

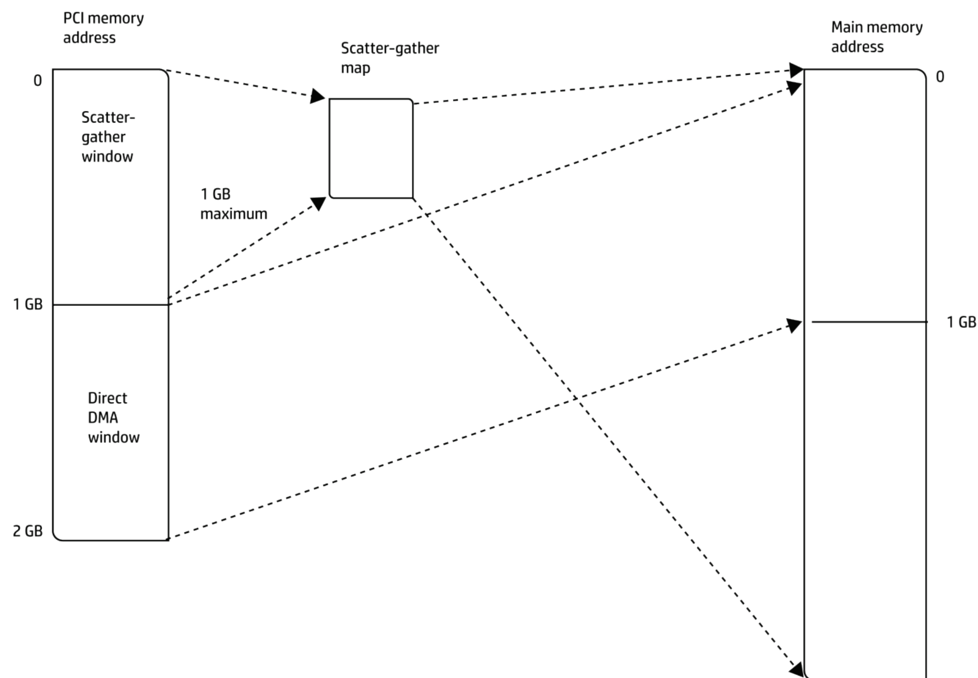


Figure 1 shows a typical platform, in which a maximum of 2 GB of the PCI memory address space maps to main memory. A PCI address in the range 1 GB to 2 GB, which is the direct DMA window, is combined with a base register to produce a main memory address between 0 and 1 GB. Addresses in the scatter-gather window are translated to main memory addresses using the scatter-gather map.

The Intel 9300-based servers do not have map registers. Physical DMA has limitation on amount of main memory that can be addressed, hence drivers written to use map registers need to be modified.

An important criterion in deciding the kind of modifications necessary is whether the device is capable of 64-bit addressing or is limited by 32-bit addressing capability.

Devices that are capable of only 32-bit addresses typically use the scatter-gather mapping and use map registers to accomplish the PCI-system address mapping.

This paper speaks about how drivers for devices that are constrained by 32-bit addressing can be modified to deal with the limitation of not having the map register (hence scatter-gather capability). The modification requires using BAP in place of mapped registers.

BAP is a separate pool similar to nonpaged pool. If a component allocates memory from BAP, that memory is guaranteed to have physical addresses within specific limits and is guaranteed to be physically contiguous. Drivers can allocate memory from it to use in driver-adapter control structures and not have to validate the memory's physical address (PA) range. BAP memory is guaranteed to be accessible through the PCI direct DMA (DDMA) window.

BAP is allocated, de-allocated, and has pool checking just like nonpaged pool.

BAP differs from nonpaged pool in that the user gets to define the BAP properties and the amount that is needed.

The rest of this document illustrates how a device driver developer can use BAP in writing OpenVMS device drivers.

Perform DMA using BAP

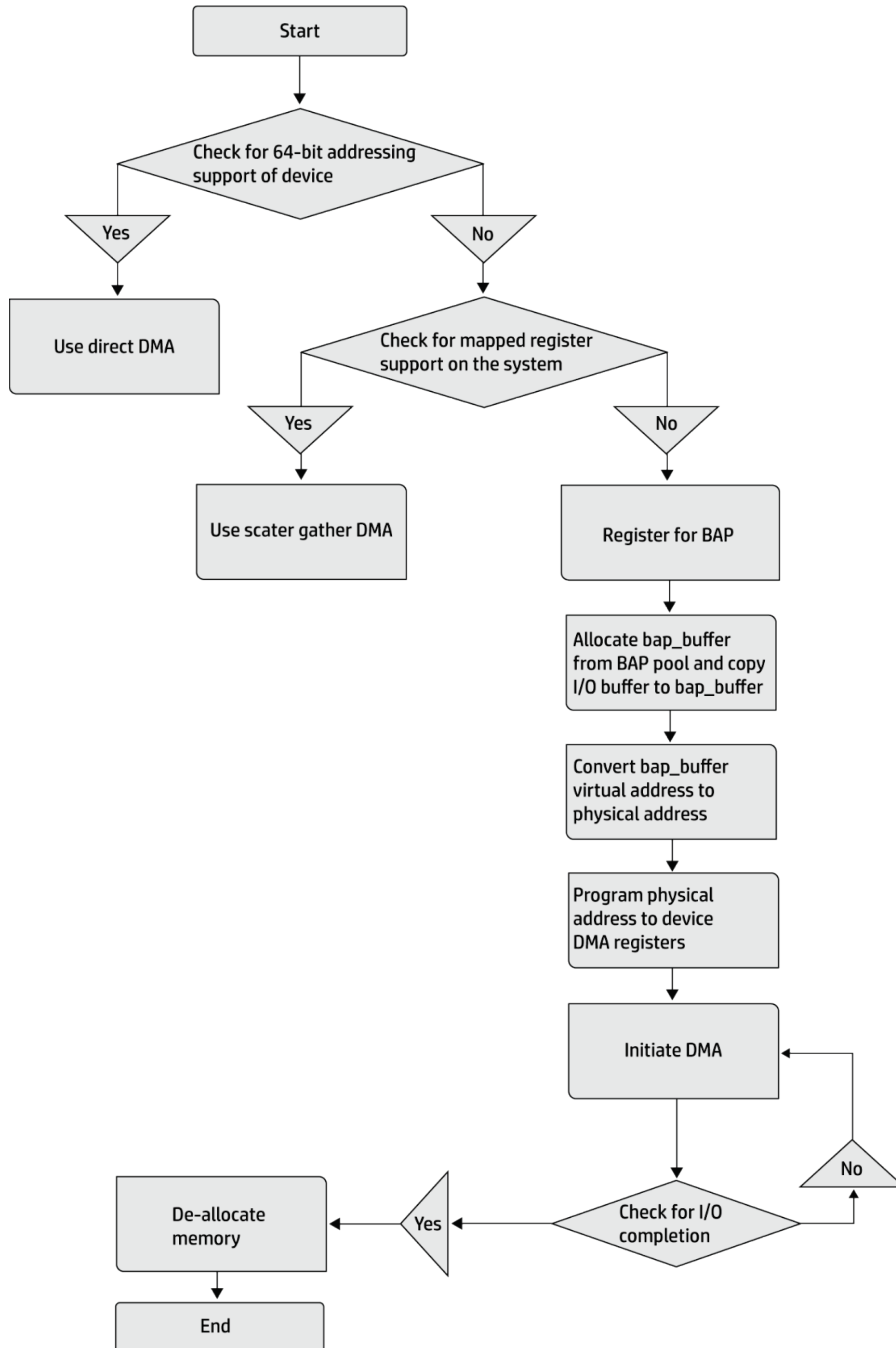
OpenVMS device driver writers need to perform the following tasks when setting up and completing a DMA transfer for devices that do not support 64-bit addressing capability.

Overview

OpenVMS device driver writers need to perform following tasks to perform DMA by using BAP:

1. Check whether device supports 64-bit addressing; this can be ascertained from the device registers.
2. If device supports only 32-bit addressing, then check for mapped registers support on system.
3. If system doesn't support mapped registers then the driver needs to allocate memory from BAP for DMA.
4. To allocate memory from BAP, device driver should first register for BAP pool with appropriate parameters. The procedure to register BAP is described in section BAP registration.
5. Perform I/O:
   a. Map the I/O buffer to system space.
   That is, allocate memory (bap_buffer) from BAP using exe$allocate_pool as described in section BAP memory allocation and copy the contents of I/O buffer to the allocated bap_buffer.
   b. Convert the bap_buffer virtual address to physical address, physical address conversion can be performed as below:
   pa = (U64) ioc_std$va_to_pa((VOID_PQ)bp_va, NULL);
   bp_va is the bap_buffer virtual address retuned from exe$allocate_pool.
   pa is the physical address of bap_buffer.
   c. Program the physical address to device register for DMA.
6. On I/O completion, deallocate bap_buffer using exe$deallocate_pool as described in section BAP memory de-allocation.

The following flowchart describes steps involved in performing DMA using BAP.



Verify mapped register support

The existence of mapped register support on a system can be verified using the IOC$NODE_DATA routine

```
status = ioc$node_data (crb, IOC$K_SG_MAP_PRESENT, &crab);
```

A return status of SS$_NORMAL or SS$_ILLIOFUNC indicates that the system supports mapped registers.

If the return status is SS$_ITEMNOTFOUND or any other value, the system does not support mapped registers.
For additional details on usage of the ioc$node_data routine refer to “Writing OpenVMS Alpha Device Drivers in C.”

Systems with support for mapped registers

If the system supports mapped registers the driver should use mapped registers to perform DMA. Refer “Writing OpenVMS Alpha Device Drivers in C” for details on how to perform DMA using mapped registers.

Systems with no support for mapped registers

As mentioned earlier on systems that do not have mapped registers, devices with only 32-bit addressing capability need to use BAP for mapping device addresses into system space.

The driver writer needs to specify the BAP properties and amount of needed for the driver. This is accomplished by a registration call described in section BAP registration.

Steps to convert the existing HP OpenVMS driver to use BAP

1. Determine whether BAP is needed.
2. Determine Min, Max BAP memory requirements for the device.
3. Get DDMA_BASE_PA, and DDMA_WIN_SIZE values using the IOC$NODE_DATA function call.
4. BAP SPIs use quadwords, so convert all locations/values that will be used in BAP calls to quadwords, or figure out how to safely typecast them as quadwords in the BAP calls. Similarly pointers, values returned by BAP calls may have to be typecasted to longwords.
5. Use as few BAP registration call(s) as possible.
   – Only 1st registration call should supply DDMA_BASE_PA and DDMA_WIN_SIZE.
6. Convert memory allocation and de-allocation calls to BAP equivalents.
7. Same header files that are used for nonpaged pool programming hold good for BAP as well
   – exe_routines.h
   – mmgdef.h

BAP operations

BAP registration

BAP pool registration is accomplished using the EXE$REGISTER_POOL routine

```
int exe$register_pool_info(
       int (*need_memory_callback)( MMG$K_POOLTYPE_BAP,
       uint64 user_info, CallbackInfo *info),
       MMG$K_POOLTYPE_BAP,
       uint64 user_info,
       uint64 max_size,
       uint64 min_size,
       uint64 minPA,
       uint64 maxPA);
```

- First parameter is usually NULL. The driver writer can optionally supply pointer to a callback routine that the memory management code calls as a last resort when BAP allocation requests from any consumer cannot be fulfilled. When this callback routine is invoked by the memory management code the driver is expected to release BAP pool to the MIN size that it registered for. The intent here is callback routine should return some of your own BAP if another user needed it.
- pool_type: POOL type should be MMG$K_POOLTYPE_BAP.
- user_info: This is a parameter to the callback routine.
  user_info and callback must be registered together. If callback is specified as zero, neither parameter will be registered.
- max_size: Max BAP size (in bytes) that device can use.
- min_size: Min BAP size (in bytes) absolutely required by the device, this is the amount that is made available on first boot, typically. If you get called back by the need-memory callback, you should reduce your memory usage to this amount.

For example if a device absolutely needs 1K to function, the min_size would be 1K. The device however can use 64K memory if it was provided the same resulting in better performance. 64K is the max_size.

- MinPA: Minimum acceptable PA, which can be obtained from ioc$node_data as shown below.

```
ioc$node_data(crb, IOC$K_DDMA_BASE_PA, &ddma_base_pa)
```

- MaxPA: This can be obtained from ioc$node_data as shown below

```
ioc$node_data(crb, IOC$K_DIRECT_DMA_SIZE, &ddma_win_size)
```

Convert ddma_base_pa and ddma_win_size to bytes before passing them to the registration call. After converting to bytes MinPA and MaxPA can be obtained as below.

MinPA = ddma_base_pa
MaxPA = ddma_base_pa + ddma_win_size

Note: All device drivers should use the same values for MinPA and MaxPA. This can be ensured by following above guidelines using ioc$node_data to get the values.

Below example illustrates how BAP poll is allocated in response to requests from two drivers when different values of MinPA and MaxPA are requested:

The driver for device A registers for BAP with the following values:

MinPA: 0x00000010

MaxPA: 0x7FFFFFFF

Min size: 0x0000A000

Max size: 0x00020000

The driver for device B registers for BAP with following values:

MinPA: 0x0

MaxPA: 0x00001000

Min size: 0x00032000

Max size: 0x00064000

The above example illustrated registrations made with different values of MinPA and MaxPA. After AUTOGEN is run and machine is rebooted, BAP pool is allocated in the range of 0x00000010(max of MinPA) to 0x00001000(Min of MaxPA).

The min size would be 0x0003C000 (min size of device A + min size of device B) and max size would be 0x00084000(max size of device A + max size of device B).

USB drivers register for BAP with below values on Intel Itanium 9300-based servers and later

Drivers name: SYS$UHCIDRIVER

MaxPA : 0x00001000

MinPA : 0x00000000

Min size : 0x00032000

Max size : 0x00064000

Driver name: SYS$EHCIDRIVER

MaxPA : 00001000x

MinPA : 00000000x

Min size : 0000A000x

Max size : 00032000x

Consider an HP Integrity BL870c i2 Server where the system has EHA0, UHA0, UHB0, UHC0 devices, after AUTOGEN is run and machine is rebooted BAP pool will be allocated as follows:

MinPA : 0x00000000 (max of all MinPAs of all four devices)

MaxPA : 0x00001000 (min of all MaxPAs of all four devices)

Min size : 0xA0000 (sum of all min sizes of all four devices)

Max size : 0X15E000 (sum of all max sizes of all four devices)

BAP memory allocation and de-allocation

BAP memory allocation

Memory allocation from BAP pool is accomplished using the exe$allocate_pool routine.

```
status = exe$allocate_pool(Req_BufferLength, MMG$K_POOLTYPE_BAP,
                           alignment, Returned_buffer_size,
                           (VOID_PPQ) & buffer);
```

Arguments:

Req_BufferLength	:	Requested buffer size in bytes
MMG$K_POOLTYPE_BAP	:	Pool type BAP
Alignment	:	The number of lsbits that must be 0 in the returned address (this value is 5 for 64-byte alignment)
Returned_buffer_size	:	Allocated buffer size
Buffer	:	Allocated buffer

Return Values:

SS$_NORMAL	:	Block is allocated. Pool may be expanded.
SS$_INSFMEM	:	Allocation failed: Virtual address space exhaustion or expansion failure.

BAP memory de-allocation

Memory de-allocation from BAP poll is accomplished using the exe$deallocate_pool routine

```
exe$deallocate_pool ((VOID_PPQ) buffer,
      MMG$K_POOLTYPE_BAP, buffer_size);
```

Arguments:

Buffer	:	Buffer to be deallocated
MMG$K_POOLTYPE_BAP	:	Pool type BAP
buffer_size	:	Buffer size(Returned_buffer_size from exe$allocate_pool)

BAP rules

- **Supply MinPA and length of acceptable PAs**
  You must supply your minimum acceptable PA and length of acceptable PAs values in your first registration call (Once is enough). Failing this, BAP won't know what PA values are acceptable, at best you'll get what someone else has registered (which better be OK)
- **Your values for minimum acceptable PA and length of acceptable PAs must match the values every other driver is registering.**
  All drivers should use ioc$node_data to get these values so that all driver registrations match.
- **Your registered minimum and maximum size values must work when your driver is the only BAP user.**
  Otherwise, you'll get allocation failures.
- **Assume the memory BAP allocates to meet your registration is only 64-byte aligned.**
  Remember that if you allocate BAP with many different alignments and sizes, fragmentation becomes more likely. Be sure to register for enough memory to account for fragmentation.
- **Never allocate more memory than the sum of your registered max_size.**
  You'll be taking someone else's memory. Never exceeding your min_size registration values is sometimes very hard to do. Especially as there is no way to tell that BAP has only satisfied your minimum request. Sometimes you just have to register the same value for both min_size and max_size, or take a calculated risk on min_size.

BAP SYSGEN parameters and system data cells

There are six BAP System Data Cells for BAP:

EXE$GQ_BAP_MAX_REQUEST_SIZE—Sum of maximum BAP sizes requested via exe$register_pool_info from all drivers

EXE$GQ_BAP_MIN_REQUEST_SIZE—Sum of minimum BAP sizes requested via exe$register_pool_info from all drivers

EXE$GQ_BAP_MAX_PA_REGISTERED—The smallest of maximum BAP physical address requested via exe$register_pool_info.

EXE$GQ_BAP_MIN_PA_REGISTERED—Largest minimum BAP physical address requested via exe$register_pool_info.

EXE$GQ_BAP_NUM_REGISTRATIONS—Number of BAP registrations via exe$register_pool_info.

Below is a view of these system data cells from a typical machine.

EXE$GQ_BAP_MAX_PA_REGISTERED	FFFFFFFF A5017818: 00000000.00001000
EXE$GQ_BAP_MAX_REQUEST_SIZE	FFFFFFFF.A5017808: 00000000.0015E000
EXE$GQ_BAP_MIN_PA_REGISTERED	FFFFFFFF.A5017820: 00000000.00000000
EXE$GQ_BAP_MIN_REQUEST_SIZE	FFFFFFFF.A5017810: 00000000.000A0000
EXE$GQ_BAP_NUM_REGISTRATIONS	FFFFFFFF.A5017830: 00000000.00000004

There are four SYSGEN parameters associated with BAP:

NPAG_BAP_MIN—SYSGEN parameter indicating the minimum BAP size (bytes) that the system can tolerate.

NPAG_BAP_MAX—SYSGEN parameter indicating the maximum BAP size (bytes) that the system might ever need.

NPAG_BAP_MIN_PA—SYSGEN parameter indicating the MinPA for BAP that the system can tolerate.

NPAG_BAP_MAX_PA—SYSGEN parameter indicating the MinPA for BAP that the system can tolerate.

AUTOGEN FEEDBACK derives BAP SYSGEN parameter values from system data cells as shown below:

NPAG_BAP_MIN	=	EXE$GQ_BAP_MIN_REQUEST_SIZE
NPAG_BAP_MAX	=	EXE$GQ_BAP_MAX_REQUEST_SIZE
NPAG_BAP_MIN_PA	=	EXE$GQ_BAP_MIN_PA_REGISTERED
NPAG_BAP_MAX_PA	=	EXE$GQ_BAP_MAX_PA_REGISTERED

Table. The output of parameters from a typical system.

Parameter name	Current	Default	Min	Max	Unit
NPAG_BAP_MIN	655360	0	0	-1	Bytes
NPAG_BAP_MAX	1433600	0	0	-1	Bytes
NPAG_BAP_MIN_PA	0	0	0	-1	Megabytes
NPAG_BAP_MAX_PA	4096	-1	0	-1	Megabytes

System considers SYSGEN parameters for initial allocation and expansion of BAP pool.

Limitations

Drivers that use BAP may face performance issues. The driver needs to copy the data from I/O buffer to the buffer allocated from BAP before triggering DMA.

It should be kept in mind that we do not use BAP memory indiscriminately, BAP memory comes from the S0/S1 region. Reserving/Using excessive BAP may cause shortage of memory in S0/S1 space for other drivers or execlets that need to be loaded in this region.

Conclusion

As the HP Integrity i2 Servers do not support mapped registers, HP OpenVMS drivers written to use map registers need to be modified to use BAP. BAP is a separate pool like nonpaged pool. Device driver writers can use the BAP as described in this paper and modify their existing drivers or write new drivers for IA64 servers.

References

"Writing OpenVMS Alpha Device Drivers in C," by Margie Sherlock and S. Szubowicz, April 1996.

Annexure

HP OpenVMS skeleton driver to perform I/O using BAP

Decide if we need BAP

```
Void check_map_register_support()
{
/*Get flag that indicates if platform supports map registers.
*/
status = ioc$node_data(crb, IOC$K_SG_MAP_PRESENT, &crab);
switch (status)
{
   case SS$_NORMAL :
      flags |= MAP_REGISTERS_SUPPORTED;
      break;
   case SS$_ILLIOFUNC :
/* this function is not supported on Alpha system. So Alpha systems or all system
prior to V8.4 implements map registers.*/
       flags |= MAP_REGISTERS_SUPPORTED;
       break;
   case SS$_ITEMNOTFOUND :
       flags &= ~MAP_REGISTERS_SUPPORTED;
       break;
   default:
flags &= ~MAP_REGISTERS_SUPPORTED;
       break;
}
```

Registering for BAP

```
/* Register for BAP with the appropriate values */
boolean register_bap()
{
  /* Register for BAP pool, define MAX_BAP, MIN_BAP */
  status = exe$register_pool_info(NULL, MMG$K_POOLTYPE_BAP, 0,
                                  MAX_BAP * MMG$GL_PAGE_SIZE,
                                  MIN_BAP * MMG$GL_PAGE_SIZE,
                                  ucb->direct_dma_base,
                                  ucb->direct_dma_size );

if (!(status & SS$_NORMAL))
       {
         return FAIL;
       }
else
    return TRUE;
}
```

Buffer mapping

```
int map_request(UCB *ucb, REQUEST *request)
{
unsigned int  size;
int           status = SS$_NORMAL;
U64           bounce_buffer;

/ * Case 1: Controller support 64-bit addresses
    Based on hardware device registers set 64bit DMA
  */
if (64bit_dma_hw)
{
   return (SS$_NORMAL);
}

// Case 2: Platform does not support map registers
if (!(flags & MAP_REGISTERS_SUPPORTED))
status = map_buffer_bap(ucb, request);
return status;
// Case 3: Platform supports map registers
if (request->BufferLength > 0)
{
```

```
  /* map the IO buffer using mapped registers */
  status = map_buffer(ucb, request);
  return(status);
}
int map_buffer_bap(UCB *ucb, REQUEST *request)
{
   request->bounce_buffer = 0;
   request->bounce_buffer_size = 0;
   status = SS$_NORMAL;
   if (request->BufferLength > 0)
   {
      /* allocate memory from BAP Pool */
#define ALIGN_64BYTES 5
      status = exe$allocate_pool(request->BufferLength,
                    MMG$K_POOLTYPE_BAP, ALIGN_64BYTES,
                    &request->bounce_buffer_size,
                    (VOID_PPQ) &bounce_buffer);
       request->bounce_buffer = bounce_buffer;
      /* copy data to bounce buffer which need to be sent to device*/
      memcpy((void *)request->bounce_buffer, request->Buffer,
                    request->BufferLength);
    }
  return(status);
}
```

Unmap the buffer

```
void   unmap_request(UCB *ucb, REQUEST *request)
{
// Case 1: Controller supports 64-bit addresses

if (64bit_dma_hw)
{
   return;
}
// Case 2: Platform does not support map registers
if (!(flags & MAP_REGISTERS_SUPPORTED))
{
   if (request->BufferLength > 0)
   {
      if (we have data from Device to host)
      {
```

```
       /* Copy data from bounce buffer to request buffer */
       memcpy (request->Buffer, (void *) request->bounce_buffer,
       request->BufferLength);
      }
   }

 if (request->bounce_buffer_size > 0)
 {
      exe$deallocate_pool((VOID_PPQ) request->bounce_buffer,
                                 MMG$K_POOLTYPE_BAP,request->bounce_buffer_size);
 }
   return;
}

/* Case 3: Platform support map registers clear of crctx and counted resources */
……

}
```

Fork routine

```
/* unit_fork will be forked from unit_init of the driver
 */
Void unit_fork (UCB *ucb)
{
Int status;

/* get direct DMA base and size */
status = ioc$node_data(ucb->ucb$l_crb, IOC$K_DIRECT_DMA_BASE,
                          (void *) &ucb->direct_dma_base);
ucb->direct_dma_base = ucb->direct_dma_base << 20;  //minPA
if (status & SS$_NORMAL)
   {
     status = ioc$node_data(ucb->ucb$l_crb,
IOC$K_DIRECT_DMA_SIZE,
                           (void *) &ucb->direct_dma_size);
     ucb->direct_dma_size = ucb->direct_dma_size <<  20;
     ucb->direct_dma_size = ucb->direct_dma_base +
 ucb->direct_dma_size  //maxPA

   }
}
```

FDT routine

```
int fdt_ioctl (IRP *irp_ptr, PCB *pcb_ptr, UCB *ucb_ptr, CCB *ccb_ptr)
{
  int status = SS$_BADPARAM;
  int save_ipl;
  REQUEST *request;
  unsigned int p1=0,p2=0,p3=0;
  request->Buffer = irp_ptr->irp$l_qio_p1;
  request->BufferLength = irp_ptr->irp$l_qio_p2;
  /* map the user buffer */
  map_request(ucb_ptr,request);
  ….
  /* Load the DMA address in device DMA registers and perform IO
   */
  /*After completion of IO un map the request */
  unmap_request();
}
```

For more information

To know more about HP OpenVMS, visit http://h71000.www7.hp.com/.

Sign up for updates
hp.com/go/getupdated

Rate this document

© Copyright 2013 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Intel and Itanium are trademarks of Intel Corporation in the U.S. and other countries.

4AA4-4834ENW, January 2013