Obtaining Information About Virtual Memory

If your application or driver needs to use the virtual memory routines described so far, you might also need to obtain information about the operation of virtual memory. You can get two kinds of information: information about the system memory configuration and information about the page-mapping employed by virtual memory. Use the **Gestalt** function for the first sort of information, and use the **GetPhysical** function for the second sort.

Information About the System Memory Configuration

To obtain information regarding the system memory configuration, use the **Gestalt** function, which is documented in **Compatibility Guidelines**. **Gestalt** can provide information about the amount of physical memory installed in a machine, the amount of logical memory available in a machine, the version of virtual memory installed (if any), and the size of a logical page. By obtaining this information from **Gestalt**, you can help insulate your applications or drivers from possible future changes in the details of the virtual memory implementation.

You should always determine whether virtual memory is installed before attempting to use the services it provides. To do this, pass **Gestalt** the gestaltVMAttr selector. **Gestalt**'s response indicates the version of virtual memory, if any, installed. If bit 0 of the response is set to 1, then the system 7.0 implementation of virtual memory is installed.

Information About Page Mapping

To obtain information about page mapping between logical and physical addresses, use the **GetPhysical** function. **GetPhysical** translates logical addresses into their corresponding physical addresses. It provides drivers with actual physical memory addresses of pages in a specified logical address range. This information is needed to permit non-CPU devices to access memory mapped by the CPU. Mapping information is needed to enable data transfers by non-CPU devices to physically discontiguous memory by means of external software or hardware mapping mechanisms.

The **GetPhysical** routine takes as one of its parameters a parameter block with a table to store pairs of physical addresses and counts.

The translation table is an array of ordered pairs of addresses and counts. **GetPhysical** translates up to the size of the table or until the translation is completed, whichever comes first.

If you call **GetPhysical** with a table size of 0, it returns the number of table entries necessary to translate the entire address range. On exit, the virtual information is updated to indicate the next virtual address and the number of bytes left to translate. If the translation is incomplete, the same translation table can again be passed to **GetPhysical** to continue the translation of the remaining addresses. The return value from the routine indicates the number of pairs of physical addresses and counts actually placed in the translation table.

The translation parameter block consists of two elements, the virtual information and the physical translation table, and is defined in the **LogicalToPhysicalTable** structure.

The information is stored as an ordered pair of address and count. The physical translation table is an array of address and count pairs that define sections of physical memory representing the virtual address range input parameter. On exit, the virtual information is updated to reflect the address range that was not translated. The virtual address field contains the next virtual address to be translated, and the virtual count field has the number of bytes left to be translated. The parameter count is used to indicate the size of the translation table array. The actual count value is the number of physical ordered pairs that can be returned in the translation table. Passing in the size of the table allows the calling software to adjust the table size to fit its application. Calling software can then make the necessary trade-offs between memory and complexity versus the overhead for multiple calls.

When **GetPhysical** returns, the physicalEntryCount parameter contains the number of address and count pairs that were filled into the translation table. In addition, if physicalEntryCount contained 0, the total number of entries required to map the entire logical space is returned (and the contents of the table are unchanged). The code example below provides an example of using **GetPhysical**.

```
//Translating logical to physical addresses

// Assuming inclusion of MacHeaders
#include <Memory.h>

// Prototype your translation routine like this prior to calling it
void LogicalToPhysical(void *,unsigned long);

void LogicalToPhysical(void *bufferAddress,ç bufferSize)
{
    LogicalToPhysicalTable table;    // table to use in translation
    OSErr myErr;      // error checking variable
    unsigned long numPhysicalBlocks;    // number of memory blocks

    // prototypes for user defined routines
    void DoError(OSErr);
    void aRoutine(LogicalToPhysicalTable,unsigned long);

    table.logical.address = bufferAddress;     // virtual address
    table.logical.count  = bufferSize; // bytes in buffer

    while ( table.logical.count ) {
        // calculate number of memory blocks
        numPhysicalBlocks = sizeof(table) / sizeof(MemoryBlock) - 1;
        // translate logical address into physical address
        myErr = GetPhysical(&table, &numPhysicalBlocks);
        // if error occured, handle it
        if ( myErr )
            DoError(myErr);

        // your routine to process results
        aRoutine(table, numPhysicalBlocks);
    }
}
```

Note: The address range passed to **GetPhysical** must be locked (using **LockMemory**). This is necessary to guarantee that the translation data returned are accurate (that is, that paging activity has not invalidated the translation data). An error is returned if you call **GetPhysical** on an address range that is not locked.