Debugger Support Under Virtual Memory

> **Note:** You need the information in this section only if you are writing a debugger that is to operate under virtual memory.

Debuggers running under virtual memory can use any of the ordinary virtual memory routines. For example, if a debugger is in a situation where page faulting would be fatal, it can use **DeferUserFn** to defer the debugging until paging is safe. However, debuggers running under virtual memory might require a few routines that differ from those available to all applications. In addition, debuggers might depend on some specific features of virtual memory that other applications should not.

For example, because debugger code might be entered at a time when paging would be unsafe, you should lock (and not just hold) the debugger and all of its data and buffer space in memory. Normally, the locking operation is used to allow NuBus masters or other DMA devices to transfer data directly into physical memory. This requires that data caching be disabled on the locked page. You might, however, want your debugger to benefit from the performance of the data cache on pages belonging solely to the debugger. The **DebuggerLockMemory** function does exactly what **LockMemory** does, except that it leaves data caching enabled on the affected pages. The **DebuggerUnlockMemory** function reverses the effects of **DebuggerLockMemory**.

Other special debugger support functions determine whether paging is safe, allow the debugger to enter supervisor mode, enter and exit the debugging state, obtain keyboard input while in the debugging state, and determine the state of a page of logical memory. All of these functions are implemented as extensions of the **DebugUtil** trap, a new trap intended for use by debuggers to allow greater machine independence. This trap is not present in the Macintosh II, Macintosh IIx, Macintosh IIcx, or Macintosh SE/30, but it is present in all later ROMs. Virtual memory implements this trap for all machines that it supports, so a debugger can use **DebugUtil** (and functions defined in terms of **DebugUtil**) if **Gestalt** reports that virtual memory is present.

When the virtual memory extensions to **DebugUtil** are not present (that is, the machine supports virtual memory but is *not* a Macintosh II, Macintosh IIx, Macintosh IIcx, or Macintosh SE/30), **DebugUtil** provides functions that can determine the highest **DebugUtil** function supported, enter the debugging state, poll the keyboard for input, and exit the debugging state.

You can call **DebugUtil** to determine how many of the debugger functions or extensions are present. The **DebuggerGetMax** function returns the highest function number supported by the **DebugUtil** trap. The numbers correspond to the following functions:

Selector	Routine
0x0000	**DebuggerGetMax**
0x0001	**DebuggerEnter**
0x0002	**DebuggerExit**
0x0003	**DebuggerPoll**

0x0004 **GetPageState**

0x0005 **PageFaultFatal**

0x0006 **DebuggerLockMemory**

0x0007 **DebuggerUnlockMemory**

0x0008 **EnterSupervisorMode**

Bus Error Vectors

When a page of memory is read in from disk, it is triggered by a bus error (this kind of bus error is called a *page fault*). The Operating System needs to intercept these page faults and do the necessary paging. In addition, various applications and pieces of system software need to handle other kinds of bus errors. Virtual memory takes care of the complications of bus error handling by having two bus error vectors. The vector that applications and other system software see is the one in low memory (at address 0x8). The vector that virtual memory uses (the one actually used by the processor) is in virtual memory's private storage and is pointed to by the vector base register (VBR). Virtual memory's bus error handler handles page faults and passes other bus errors to the vector in low memory at address 0x8.

When a debugger wants the contents of a page to be loaded into memory, it can read a byte from that page. The Operating System detects the page fault and loads the appropriate page (perhaps swapping another page to disk).

Note that a debugger will probably temporarily replace one or both of the bus error vectors while it is executing. A debugger that wants virtual memory to continue paging while the debugger runs can put a handler only in the low-memory bus error vector. A debugger that displays memory without allowing virtual memory to continue paging can put a handler in the virtual memory's bus error vector (at VBR+0x8).

Because the current version of virtual memory is not reentrant, there are times when trying to load a page into memory would be fatal. To allow for this, you can use the **PageFaultFatal** function to determine whether a page fault would be fatal at that time. If this function returns TRUE, the debugger should not allow the virtual memory's bus error handler to detect any page faults. This means that the virtual memory's bus error vector should always be replaced if the **PageFaultFatal** function returns TRUE.

Special Nonmaskable Interrupt Needs

Since a debugger can be triggered with a nonmaskable interrupt (level 7, triggered by the interrupt switch), it has special needs that other code in the system does not. For example, because a nonmaskable interrupt might occur while virtual memory is moving pages around (to make them contiguous, for example), debugger code must be locked down (instead of held, like most other code that must run at a time when page faults would be fatal). Unfortunately, the **LockMemory** function is intended for use by device drivers and automatically disables data caching for the locked pages. Because this is not desirable for the debugger, the debugger support functions **DebuggerLockMemory** and **DebuggerUnlockMemory** lock pages without inhibiting the caching of those pages. Note that the stack, code, and other

storage used by the debugger might need to be locked in this way.

Supervisor Mode

Because a debugger is typically activated through one of the processor vectors, it normally executes in supervisor mode, allowing it access to all of memory and all processor registers. When the debugger is entered in another way-for example, through the **Debugger** or **DebugStr** trap or when it is first loaded-it is necessary to enter supervisor mode. You can accomplish this with the following assembly-language instructions:

```
#define _DebugUtil  0xA08D

asm {
    MOVEQ EnterSupervisorMode,D0
    _DebugUtil     ;OS trap to DebugUtils
    ;on exit, D0 holds old SR
}
```

This code switches the caller into supervisor mode. The previous status register is returned in register D0 so that when the debugger returns to the interrupted code, the previous interrupt level, condition codes, and so forth, can be restored. When the debugger is ready to return to user mode, it simply loads the status register with the result returned in D0.

There is also a high level language interface for **EnterSupervisorMode**.

The Debugging State

When activated by an exception, **Debugger** or **DebugStr** trap, or any other means, the debugger should call **DebuggerEnter** to notify **DebugUtil** that the debugger is entering the debugging state. Then **DebugUtil** can place hardware in a quiescent state and prepare for subsequent **DebugUtil** calls.

Before returning to the interrupted application code, the debugger must call **DebuggerExit** to allow **DebugUtil** to return hardware affected by **DebuggerEnter** to its previous state.

Keyboard Input

A debugger can obtain the user's keyboard input by using the **DebuggerPoll** procedure. This routine can obtain keyboard input even with interrupts disabled. After you call this service, you must then obtain keyboard events through the normal event queue mechanism.

Page States

Debuggers need a way to display the contents of memory without paging or to display the contents of pages that are currently on disk. The **GetPageState** function returns the state of a page containing a virtual address. **GetPageState** returns one of these values:

```
typedef short PageState;

kPageInMemory          //page is in RAM
```

kPageOnDisk	//page is on disk
kNotPaged	//address is not paged

A debugger can use this information to determine whether certain memory addresses should be referenced. Note that ROM and I/O space are not pageable and therefore are considered not paged.