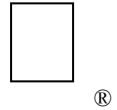


New Technical Notes

Macintosh



Developer Support

Bus Error Handlers

Devices

Revised by: Colleen K. Delgadillo

May 1992

Written by: Wayne Meretsky and Rich Collyer

February 1991

This Technical Note discusses bus errors and how applications and drivers should deal with them.

Changes since February 1991: Discussion of why declaration ROMs are necessary in NuBus™ design. This discussion is important for those who are considering using a workaround instead of declaration ROMs. Also added are some hints that you should be aware of if you are planning to write a bus error handler for the '040.

Introduction

Bus errors occur within Macintosh systems under a variety of circumstances: virtual memory page faults, NuBus transfer acknowledgment other than complete (error, time-out, try-again-later), SCSI blind transfer handshaking, and so on. At present, Apple has not documented a single model for handling bus errors, and as a result, system software, applications, and device drivers use various techniques that do not always work together and therefore compromise system integrity. The following is the second revision of a definitive statement on bus errors and bus error handling.

What Is a Bus Error?

A bus error is an event that forces termination of a bus cycle. In some cases the processor takes no programmer-visible actions, as may happen if a prefetching bus cycle results in a bus error but the prefetched data is not executed, or as may happen during a cache fill burst. Typically, the termination of a bus cycle by a bus error does result in the processor taking programmer-visible actions. These actions, collectively called Bus Error Exception Processing, include termination of the instruction that caused the bus error, creation of an exception stack frame on the appropriate system stack, and transfer of control to the bus error handler designated by the current bus error vector. Bus Error Exception Processing does **not** include the execution of the bus error handler.

The cause of bus errors is different on different members of the M68000 family. On all members up to and including the MC68020, the only cause of bus errors is assertion of the

/BERR signal (note that the assertion of /BERR and /HALT indicates a bus retry operation, not a bus error). On the MC68030, bus errors can be caused by either the assertion of the /BERR signal or by the internal MMU during address translation. The MC68040 is similar to the MC68030, except that the term *bus error* has been replaced with the term *access error*, and

the signal `/BERR` has been replaced with the signal `/TEA` (this Note uses pre-MC68040 terminology).

What Is a Bus Error Handler?

A bus error handler is the exception handler for the bus error exception and is designated by offset `$08` in the Exception Vector Table. The bus error handler is responsible for the recovery from the conditions that led to the bus error. Depending on the cause of the bus error, recovery may be either simple or extremely complex. In the case of a page fault, the recovery process entails loading the desired page into physical memory and updating the MMU Page Tables. In the case of a NuBus try-again-later acknowledgment, the recovery process is to simply retry the bus cycle that caused the bus error.

From the processor's perspective, there is, at any one time, only a single handler for all bus error exceptions. That handler is the one designated by offset `$08` in the Exception Vector Table.

From the system programmer's perspective, there are many pieces of software that work in unison and form the bus error handler. Each of these individual pieces exists in an ad hoc linked list called the Bus Error Handler Chain and must follow certain installation, removal, and invocation rules to ensure proper system behavior.

How Does the Bus Error Handler Chain Work?

The bus error handler chain is rooted in location `$00000008`. This location contains a 32-bit pointer to the first handler. Each handler is responsible for maintaining the links in the chain. Invocation of the bus error handler chain works in different ways depending on the absence or presence of virtual memory (VM).

If VM is present, the processor's VBR (vector base register) points to a special Exception Vector Table which must **never** be modified. When a bus error occurs, the VM bus error handler is invoked and determines whether or not the bus error is a page fault. If the bus error is a page fault, VM takes the appropriate actions. If the bus error is not a page fault, VM invokes the first entry in the bus error handler chain.

In non-VM systems, the VBR points directly to location `$00000000`; therefore, the first entry in the bus error handler chain is invoked directly by the bus error exception processing performed by the processor.

In either case, the techniques for installing and removing an entry in the chain are identical. The only difference is that when VM is running it gets first crack at all bus errors.

What Is the Model for When Bus Errors Occur ... And Who Handles Them?

The only bus errors that are expected during execution of application or Toolbox code are caused by virtual memory page faults (if VM is running). As a general rule applications and the Toolbox should **not** be directly accessing hardware that can cause bus errors. There may be

cases when hardware diagnostic applications need to install a bus error handler, but these should be very rare and they should follow the same guidelines that drivers must follow. The reason for this is that MultiFinder does not switch the bus error vector, so during a minor switch there is no way to know if the correct vector is in place. Applications should not install handlers into the bus error handler chain because the Process Manager does not context switch entries in the chain during its application-level context switches.

Certain parts of the operating system expect bus errors under infrequent and well-controlled circumstances. Each of these managers installs its handler in the bus error handler chain before the instructions that may cause bus errors and removes the handler after these instructions. The managers that currently handle bus errors are as follows:

- The Memory Manager handles some of the bus errors that could arise if it is passed corrupt handles or pointers.
- The SCSI Manager expects and handles bus errors relating to the Blind Transfer handshake on machines that implement that mechanism.
- The Slot Manager expects and handles bus errors during its accesses to configuration ROMs, because accesses to empty or nonexistent NuBus slots generate bus errors.

The only nonsystem software that should attempt to handle bus errors are NuBus device drivers. Typically, the only bus errors that happen during driver execution are those related to the device. Because device drivers, DCEs, and heap space allocated by drivers are all supposed to be in the system heap (which cannot be paged), no page fault bus errors should occur. I/O buffers that are passed to drivers through normal Device Manager entry points cannot be paged BEFORE the Device Manager hands the call off to the device driver. Drivers that access other memory in their caller's address space at interrupt level must cooperate with VM to ensure that those pages cannot be paged prior to receiving any interrupt that may access them. Page faults are not allowed during device driver interrupt handlers.

Adding Code to and Removing Code From the System Bus Error Handler

The technique for a NuBus device driver to install code in the bus error handler chain is fairly simple. Location \$00000008 in the logical address space points to the first entry in the bus error handler chain. The installer must save the current contents of location \$00000008 and place the address of its handler into location \$00000008. To remove its code from the bus error handler chain, a NuBus device driver should simply replace the old value that was saved from location \$00000008 during bus error handler installation.

When to Install Code to the System Bus Error Handler

A NuBus device driver should install a bus error handler around certain instructions or

groups of instructions that access the NuBus device and could generate bus errors. The handler should be installed only when executing code that is part of the device driver. It is acceptable to enclose fairly large loops with a single install and remove operation rather than have an install and remove operation within the loop. It is **not** acceptable to install a handler when the driver is **opened** and remove it when the driver is **closed**.

What Should a NuBus Device Driver's Bus Error Handler Do?

First a word of caution regarding bus error handlers in general. The Exception Stack Frames generated by different M68000 family members under a given condition are quite different. Furthermore, the recovery mechanisms implemented by the handler must be fully aware of the limitations of the processor's RTE policy. For example, the MC68000 is not capable of finishing an instruction that was terminated by a bus error; the MC68010, MC68020, and MC68030 finish the instruction by resuming its execution at the point of termination; and the MC68040 can only finish some instructions by restarting their execution. One must therefore be sure that their bus error handler can handle any error that may occur on the '040. Techniques for writing bus error handlers are not contained in this Note, which discusses only how to register your handler with the system and how to pass along bus errors to other handlers in the chain.

The bus error handler must first ensure that the bus error is one that the bus error handler expects. To do this it must inspect the Exception Stack Frame pushed onto the system stack by the processor, for example by examining the PC or the Data Cycle Fault Address in the Exception Stack Frame. Extra caution must be used when examining the PC value because the PC in the Exception Stack Frame is not always the PC of the instruction that caused the bus error. This is true for the '030 and on the '040 the PC in the Exception Stack Frame will almost never be the PC for the instruction that caused the bus error. Due to caching on the '040, the instruction that caused the error may have been sitting in the cache for a long time before it was executed. This makes it hard to properly tell where the error originated. (This type of exception is called a name imprecise exception.)

If the bus error is one that is expected by the bus error handler, it should cure the problem and unwind. This can be done in any number of ways that are appropriate for the given driver and device. On one end of the spectrum, the bus error handler may simply use an RTE instruction to cause the bus cycle to be rerun whereas in other cases it may completely remove the Exception Stack Frame from the stack and jump to some other point in the driver. Exiting a handler by doing an RTE is not a good idea on the '040. You cannot ask the processor to rerun the faulted bus cycle on the '040 as you can on the '030. Due to pipelining, you may end up jumping to a point in the driver where you were not supposed to be.

If the bus error is one that is not expected by the bus error handler, then the course of action depends on whether the bus error happened during execution of an interrupt handler or noninterrupt-level code.

The noninterrupt-level bus error handling scheme requires that each driver's bus error handler pass the bus error exception along to its predecessor if it does not handle the bus error. This is accomplished by restoring the machine to the exact state at the time the driver's handler was invoked and by jumping to the handler address that was in location \$00000008 at the time the handler was installed. The last handler in the chain is the system's handler that generates a system error from an unhandled bus error.

Interrupt-level bus error handling is rather different. These handlers should not chain to their predecessor because noninterrupt-level bus error handlers may be context sensitive and possibly nonreentrant. If a bus error happens at interrupt level in a given NuBus device driver's interrupt handler and that driver cannot handle the bus error, then the driver should call `_SysErr` and cause the machine to crash. If a NuBus device driver's interrupt handler causes a bus error and has not installed a handler in the system chain, the results are unpredictable and that driver is in error.

Why Should I Have Declaration ROMs?

As explained earlier, certain parts of the operating system expect bus errors under infrequent and well-controlled circumstances. One manager that currently handles bus errors is the Slot Manager. The Slot Manager installs its handler in the bus error handler chain before the instructions, which may cause bus errors, and removes the handler after these instructions. It expects and handles bus errors during its accesses to declaration ROMs (also known as configuration ROMs), because accesses to empty or nonexistent NuBus slots generate bus errors. The declaration ROM is an area on a NuBus expansion card that contains firmware that identifies the card and its functions, and allows the card to communicate with the computer through Slot Manager routines. However, communication with the Slot Manager is possible only if you configure your card's declaration ROM firmware properly.

To individuals who are contemplating using a workaround instead of declaration ROMs in their NuBus design: Don't do this! First of all, your design will not conform to the Macintosh implementation of the NuBus specifications. Second, it is very difficult to create a workaround for handling bus errors since you will need to be able to handle any error that may be returned to you. Lastly, bus error handlers will change in future versions of the Macintosh Operating System. When this happens, a design without declaration ROMs will very likely become incompatible with all Macintosh systems running the new software.

Creating a Workaround for Dealing With Bus Error Handlers

It can be very difficult to create a workaround for dealing with the bus error handler for all machines, especially for systems that contain a 68040. One particular point about the '040 is that when you receive a bus error, any writebacks that you may receive will now be pending. Due to the '040 pipeline and caches, when you receive a bus error there may be other writes waiting to be completed that are unrelated to the faulted one. These writes are called writebacks since they typically are cached in the data cache and may correspond logically to various instructions prior to the faulted instruction. It is very important that the bus error handler be able to take care of this. For an example of how MacsBug handles bus errors for the '030, there is a code snippet available on the *Developer Essentials CD Series* disc. There is no code available to describe how MacsBug handles '040 bus errors. The '040 is a much more complex machine. MacsBug handles bus errors in a completely cursory fashion, so it never gets into the complexities on the '040. Its only job is to display the first error that occurred, based on the stack frame's PC. Subsequent bus errors in the frame will likely cause a crash (that is, a hung machine).

Because we are unable to provide sample code to demonstrate how MacsBug handles bus errors on the '040, we have decided to go through all the complexities involved in creating a bus error handler for an '040 machine. You will see why it is a sane idea to include declaration ROMs in a NuBus design.

Getting a bus error handler installed is no real problem, but writing one to handle all CPUs is difficult. The '040 is particularly difficult since the handler must be able to resolve up to three pending writes that may be in progress at the time the bus error is acknowledged. This imposes many constraints on writing a bus error handler. If you have an init or a cdev, you can install a bus error handler when you call the board to do whatever it does. (You should always remove the handler, and not leave it installed, since that is likely to conflict with VM and the rest of the

system.) You cannot ask the processor to rerun the faulted bus cycle, as you can on the '030. Your handler must repair the fault, emulate the access, and perform all writebacks. Performing the writebacks can be quite complex depending on the type. Firstly, the access address may be either physical or logical. This is dependent on VM. If you are running VM you will need to translate the physical address to its logical addresses. Secondly, the access data may be either memory aligned or register aligned. And lastly, performing a writeback may itself cause another bus error which will require that your bus error handler be reentrant so that a fault from a writeback can be handled. After handling the writebacks you will need to alter the PC so that you can properly return and go on to the next instruction. The only problem here is how to find where the PC should be reset to. The PC in the exception frame is for the instruction that was in progress at the time of the fault, that is typically several instructions past the actual faulting instruction. Therefore there is no way to restart the PC in a reasonable spot, since you cannot tell from the stack frame where the instruction starts (in every case). Remember, you get bus errors in the middle of instructions that can be many words long, and in the '040 case, you can actually get bus errors for things for which you do not have a PC. Such a case is presumably rare, but if you do not handle it you will get a further crash. It is possible to put in some heuristic method, but that will fail occasionally.

As you can see, it would be very difficult to create a workaround for dealing with the bus error handler for the '040. The best thing to do is avoid the need for a handler by including declaration ROMs in your NuBus design.

Debugging Hints for Writing a Bus Error Handler for the '040

If you have read the above information and still insist on writing your own bus error handler, the following are some final debugging hints that will help you out.

If you want to have first crack at bus errors, be sure to turn VM off. A '020, '030, and '040 machine has a VBR that points to the exception vectors and it is always active. On processor reset, the VBR is initialized to \$0000 0000. The Macintosh Operating System will leave the VBR at \$0. It may be changed by VM, debuggers, or any system code that wants to set up its own exception vector table at some other address. If VM is present, the VBR points to a special Exception Vector Table. When a bus error occurs, the VM bus error handler is invoked. (This is described in greater detail earlier in this Note.) The 68000, on the other hand, has no VBR. All of its exception vectors are always located at address \$0.

Second, most of the problems we have seen have been cache related. The fundamental difference between the '040 and the '030 is the '040 caches. An instruction cache exists that may read in your code and cause it to run much faster than you intended it to. Therefore, if you are going to be using time-dependent code, be sure to turn the instruction cache off (using the Memory cdev or the `_HwPriv` trap).

Lastly, when you're having a hard time finding what is going wrong, you may want to use a

RAM stuffing routine that takes pertinent numbers and puts them away for you to view later. This would be the best way to see whether your bus error handler is working correctly (for example, to see whether the VBR is pointing to the correct spot in the exception vector table, and so on).

Further Reference:

- MC68030 User's Manual
- MC68040 User's Manual

NuBus™ is a trademark of Texas Instruments.