Quick Interrupts

by Michael Sinz

[Editor's note: This article was written from the programmer's perspective, and doesn't discuss any of the hardware issues. See the ''Appendix K: Zorro Expansion Bus'' section of the third edition of the Amiga Hardware Reference Manual *for more information.]*

One of the features of the Zorro III bus is the Quick Interrupt, also known as the vectored interrupt. This feature allows Zorro III hardware to supply a vector number to the system when an interrupt occurs. The system uses this vector number to go directly to an interrupt routine.

Conventional Amiga Interrupts

The Amiga handles normal interrupts from Zorro II cards using an interrupt server chain. There are two interrupts available from the Zorro II bus, the PORTS and EXTER interrupt server chains. If a driver for a Zorro II card needs to use an interrupt, it adds an interrupt routine to the appropriate chain. When the interrupt occurs, Exec calls each routine in the interrupt chain, which are sorted in priority order. Exec continues until it finds the routine that corresponds to the device that triggered the interrupt.

The server chain allows several routines to share a single interrupt. This means that several devices trigger the same interrupt, so each interrupt routine must do some processing to determine if its card triggered the interrupt or if some other source caused the interrupt. For example, an interrupt routine might examine a register on its card to determine that the card triggered the interrupt.

Although this scheme allows unrelated pieces of software to easily share an interrupt, it can make the interrupt overhead rather high. These two interrupt server chains also handle interrupts from the CIA chips, which are used to trigger a variety of events. As a result, these server chains can contain a multitude of interrupt routines.

Consider what happens when a Zorro II card generates a PORTS interrupt. Exec has to perform some set up and then step through the PORTS server chain. Exec calls each interrupt routine in priority order looking for the routine that services this interrupt. If there are 20 interrupt routines of higher priority than the card's interrupt routine in the server chain, Exec has to call 20 other routines before it gets to the correct routine.

Zorro III Quick Interrupts

Quick interrupts avoid the overhead involved in Exec's interrupt server chains. Exec only helps set up the quick interrupt, which it does via the *exec.library* function ObtainQuickVector() (see the Autodoc at the end of this article). Once Exec has set up the quick interrupt routine, it does not intervene. Unlike conventional Amiga interrupt routines, which are called as subroutines from Exec's main interrupt code, the Amiga jumps directly to the quick interrupt routine using a private vector. This behavior requires quick interrupt routines to take some special precautions.

There are two important differences between a conventional Amiga interrupt routine and a quick quick interrupt routine. *A quick interrupt routine must save and restore all of the registers it changes, including D0, D1, A0, and A1.* It must do this because, unlike regular interrupt routines, Exec doesn't do it for you. Also, a quick interrupt routine ends with a RTE (return from exception) instruction.

If your quick interrupt routine is 100% self-contained and does not access any operating system structures or routines, then the work is rather simple. Just save the registers you use, perform your interrupt processing, restore the registers, and end with an RTE. If, however, the routine needs to call the OS or use an OS structure, it must check if the interrupt has been delayed. This is necessary in case the interrupt hit the CPU just after the CPU had told the hardware to hold off interrupts (see the Autodoc for ObtainQuickVector() to find out how to perform this test).

As the Amiga OS is a dynamic operating system, quick interrupts are allocated by the OS. If your hardware/software wants to use a quick interrupt, it must allocate a vector with ObtainQuickVector(). This routine accepts a pointer to the quick interrupt code (*not* a pointer to an Interrupt structure). If Exec installed the vector, ObtainQuickVector() returns the vector number. When the quick interrupt occurs, the Zorro III card sends this vector number to the CPU, which tells the CPU where the interrupt code is.

ObtainQuickVector() returns 0 if there are no more vectors. Since the number of vectors is limited, any Zorro III device that uses quick interrupts must be able to fall back to the Amiga's conventional interrupt scheme.

The LVO for ObtainQuickVector() was added for V39, but it was not fully implemented until after the initial release. This means the OS that currently ships with the Amiga 4000 and Amiga 1200, Release 3.00, will always return 0 (no SetPatch currently exists to correct this, but a future SetPatch may do so). ObtainQuickVector() only works in the developer releases of the OS that follwed the initial release. Since a Zorro III device driver must handle the case where it cannot obtain a vector, this function should never cause a hardware product to fail. There is no reliable way to obtain a vector before V39.

```
exec.library/ObtainQuickVector                 exec.library/ObtainQuickVector

   NAME
        Function to obtain an install a Quick Interrupt vector           (V39)

   SYNOPSIS
        vector=ObtainQuickVector(interruptCode)
        d0                       a0

        ULONG ObtainQuickVector(APTR);

   FUNCTION
        This function will install the code pointer into the quick interrupt
        vector it allocates and returns to you the interrupt vector that
        your Quick Interrupt system needs to use.

        This function may also return 0 if no vectors are available.  Your
        hardware should be able to then fall back to using the shared
        interrupt server chain should this happen.

        The interrupt code is a direct connect to the physical interrupt.
        This means that it is the responsibility of your code to do all
        of the context saving/restoring required by interrupt code.

        Also, due to the performance of the interrupt controller, you may
        need to also watch for "false" interrupts.  These are interrupts
        that come in just after a DISABLE.  The reason this happens is
        because the interrupt may have been posted before the DISABLE
        hardware access is completed.  For example:

        myInt:          move.l  d0,-(sp)        ; Save d0...
                        move.w  _intenar,d0     ; Get interrupt enable state
                        btst.l  #INTB_INTEN,d0  ; Check if pending disable
                        bne.s   realInt         ; If not, do real one...
        exitInt:        move.l  (sp)+,d0        ; Restore d0
                        rte                     ; Return from int...
        ;
        realInt:        ; Now do your int code...  d0 is already saved
                        ; ALL other registers need to be saved if needed
```

```
                    ; This includes a0/a1/d0/d1 as this is an interrupt
                    ; and not a function call...
                    ;
                    bra.s   exitInt         ; Exit interrupt...
```

```
    If your interrupt will not play with system (OS) structures and your
    own structures are safe to play with you do not need to check for
    the disable.  It is only needed for when the system is in disable but
    that "one last interrupt" still got through.

NOTE
    This function was not implemented fully until V39.  Due to a miscue
    it is not safe to call in V37 EXEC.  (Sorry)

INPUTS
    A pointer to your interrupt code.  This code is not an EXEC interrupt
    but is directly connected to the hardware interrupt.  Thus, the
    interrupt code must not modify any registers and must return via
    an RTE.

RESULTS
    The 8-bit vector number used for Zorro-III Quick Interrupts
    If it returns 0, no quick interrupt was allocatable.  The device
    should at this point switch to using the shared interrupt server
    method.

SEE ALSO
```