TASKING
Quality Development Tools Worldwide

APPLICATION NOTE

PROGRAMMING AND TESTING EXCEPTION HANDLERS ON THE MIPS ARCHITECTURE

INTRODUCTION

Writing exception handlers on the MIPS architecture and especially running them in a development environment requires that you take some precautions into account. If you test and debug your program on an evaluation board with a target ROM monitor, it is important that you realize that your exception handler has to coexist with the handler that has been installed by the debug monitor. Improper installation of your handler may cause the target system to stop responding, a situation that can only be changed by a hardware reset.

This application note describes the exception mechanism of the MIPS architecture in general, followed by some guidelines how to install your handler and tips and tricks for debugging.

The MIPS R3000 architecture supports 5 exception vectors:

- reset exception vector 0xbfc00000
- bootstrap general exception vector 0xbfc00180
- bootstrap UTLB exception vector 0xbfc00100
- general exception vector 0x80000080
- UTLB exception vector 0x80000000

The processor jumps to the reset exception vector on a hardware reset. On evaluation boards the first 128K/256K memory space, starting at 0xbfc00000, is the typical place for a ROM with a debug monitor. The code located at the first words in the ROM usually jumps directly to another part of the ROM to avoid overlap with the other vectors.
The R3000 supports two sets of vectors, controlled by bit 22 (BEV) in the *Status Register*. If bit 22 is set, the processor uses the bootstrap vectors. If BEV is cleared, the normal vectors are used.

Thus a target system can support different exception handling strategies during system startup and normal operation. The BEV-bit is set automatically to 1 by the reset exception and can be cleared when the system is ready for normal operation.

So, on an evaluation board it is not possible to test your own bootstrap exception code

© 1996 Tasking Software BV

without replacing the ROM with the debug monitor. Therefore boot code will be disabled (or writing will be postponed) until the final stage of the project when hardware/software integration takes place.

The MIPS architecture supports 17 different exception causes. One exception, UTLB miss (the memory management unit did not contain a valid translation entry for a memory reference into KUSEG),causes the processor jumps to a dedicated vector: the UTLB exception vector.
All other exceptions cause the processor to transfer control to the code located at the general exception vector. Since this vector is the common entry for many different exceptions, the handler at this vector and its substructures will be discussed in the remainder of this application note. The UTLB exception handler and strategies to setup an efficient memory translation table will not be discussed since the majority of all embedded MIPS applications run from KSEG0/KSEG1 without using the TLB at all.

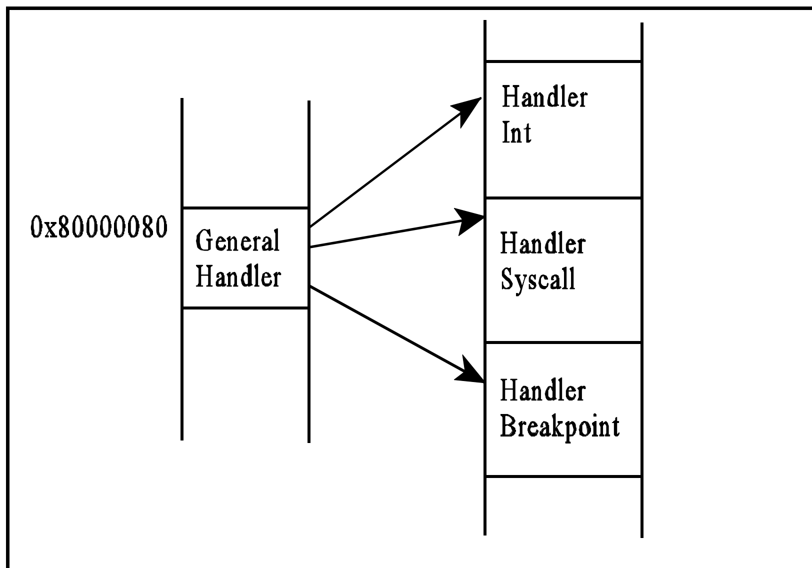
General Exception Handler

The following table lists the causes of various exceptions:

ExcCode field encoding in Cause register		
Number	**Mnemonic**	**Description**
0	Int	External Interrupt
1	MOD	TLB modification exception
2	TLBL	TLB miss exception (load/instruction fetch)
3	TLBS	TLB miss exception (store)
4	AdEL	Address error exception (load/instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Buss error exception (instruction fetch)
7	DBE	Buss error exception (data load/store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved Instruction exception
11	CPU	Coprocessor Unusable exception
12	Ovf	Arithmetic Overflow
13-15	-	reserved

The general exception handler at location 0x80000080 reads the cause register (CP0, register 13) to determine the exception cause. Since ExcCode is located at bits 5-2 (0 and 1 unused, value 0), the lower six bits of the cause register can be used as an index in a jump table, where addresses can be found of the handlers for the various exception causes.
The **exception library** in the TASKING R3000 product distribution contains a general handler, written in assembly providing the **most efficient implementation** of this general handler. The diagram on the next page shows the global structure of the exception handlers.

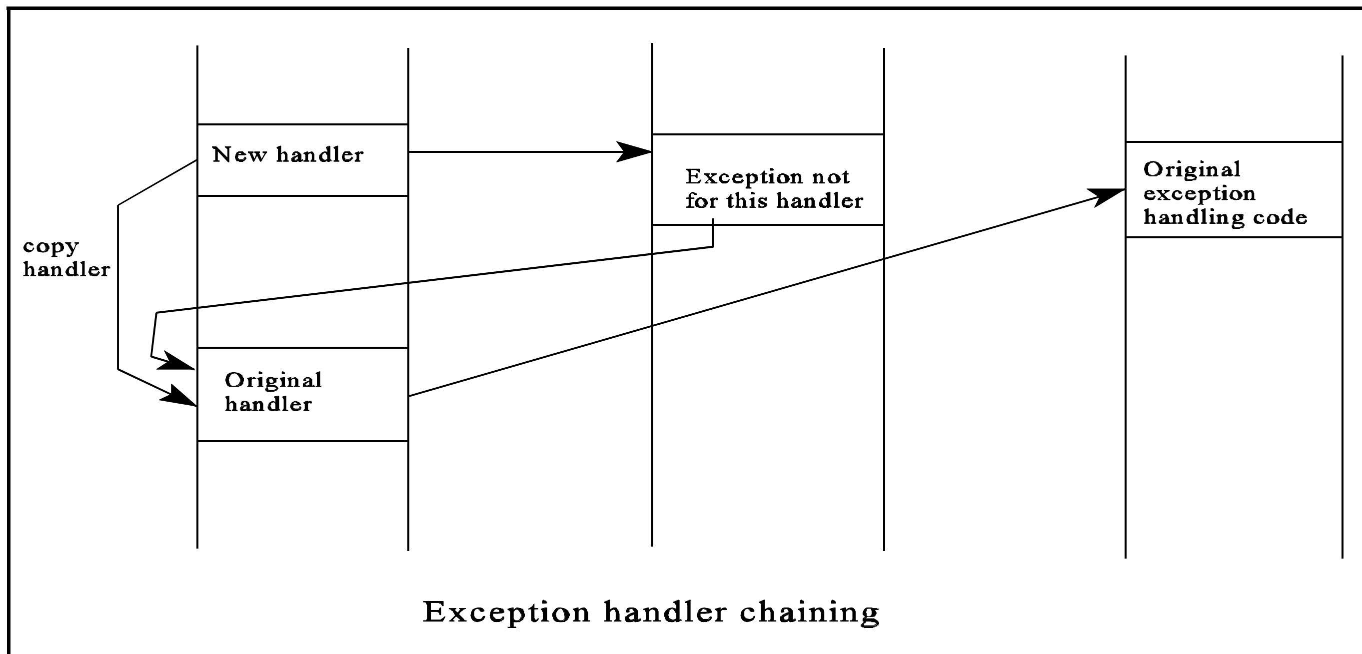
© 1996 Tasking Software BV



The debug monitor that runs on the evaluation board installs an exception handler to service the break and syscall exceptions. Also the serial interface that may have been programmed to generate hardware interrupts needs to be serviced. It is obvious that programs that are downloaded into the target at address 0x80000080 and other addresses where exception handler code of the debug monitor resides, overwrite code that is vital for proper operation of the monitor. So, application code that deals with exceptions and runs in an environment where other handler already exist (and stay alive), have to use a special vector installation procedure. In the program startup code, the vector installation procedure has to ***steal the original vector***: the original vector is copied to a safe memory location and a new handler is installed at the vector location. This new handler should be written such, that when it detects that it got invoked for a exception it cannot or may not handle, it transfers control to the original handler which is now copied to a different location. This is called chaining of exception handlers.

The module e_ihandl.s in the exception library shows how to properly install a vector. Notice that it is important to disable interrupts while installing a new vector. A hardware interrupt (exception cause 0) has a devastating effect when it occurs in the middle of the modification of the general handler. Also it is important, after installing the new handler, to flush the caches: since the new code is installed by writing the appropriate memory locations in data cache (0x80000080 is cached memory). To get the handler out of data cache into memory, the data cache needs to be flushed. The old exception handler may still be in the instruction cache, so even after your new handler has been written in memory the old one may still become active via the instruction cache. Flushing (disabling) the instruction cache forces a reload of the new handler when the next exception occurs. The diagram on the following page shows the control flow after chaining in a new handler:

© 1996 Tasking Software BV



Exception handler chaining

Writing code that transfers control from one handler to another isn't as simple as calling subroutines in regular program. Exception handlers on the MIPS architecture have to save the context of the code that was interrupted explicitly (only the return address and status register are saved by hardware). So the first exception handler that gets control starts saving all register values into a safe area. A straight jump from one exception handler into the next one would cause a 'nested register context save', where the second context is the context of the first exception handler. However, the return address is not 'nested'. Upon a return from exception (rfe), the processor returns to the user program. Since the register context that has been saved is the register context of the first exception handler, **the user program finds its registers corrupted**!

To overcome this problem, transfer of control from one handler to the other needs special preparation: before jumping, the first handler has to restore all register as if it were to return to the user program. Instead of doing this, it jumps to the entry point of the next handler where the registers are saved again, only this time the values of the user program are saved and not the register status of the first exception handler.

Coding example:

```
general_handler:
        .set    noreorder
        .set    noat
        mfc0    $kt1, $cause
        la      $kt0, _vector_table
        andi    $kt1, $kt1, 0x3c
        addu    $kt0,$kt0,$kt1
```

```
        lw      $kt0, ($kt0)
        nop
        j       $kt0
        nop
```

Note that this general handler only uses $kt0 and$ kt1. These two registers are reserved in the MIPS register protocol for exception processing. They can be used without being saved in advance. No regular application will use $kt0,$kt1 outside exception handlers. _vector_table is a table with the addresses of the various handlers for individual exceptions.

The following code 'steals' the exception handler at 0x80000080 by copying 64 memory words to a free location (exclude this space in the locator description file, so that it won't be used by other application code!) Address 0x800000c0 is chosen to save the vector.

```
vector_copy:
   la      t0, 0x80000080
   la      t1, 0x800000c0
   li      t4, 64
mcp:       lw      t2, (t0)
   sw      t2, (t1)
   add     t0, t0, 4
   add     t1, t1, 4
   sub     t4, t4, 1
   bgt     t4, $0, mcp
#
# at this point the instruction and data cache have to be flushed to make
# the installation of the new code at the exception vector address effective.
#
```

Note that the most likely part of your application where this code will be included is the startup code, executed at the begin of your program. Before using this code samples in this application note, make sure that it applies to the handler you want to save. Copying only a part of a handler will cause your target to crash on an exception by executing random code.

An exception handler typically starts saving registers to a reserved memory space. At this point various options are open, where to save register contents for exception processing:

- use a static memory area
- use the user stack
- use a special exception stack

If you want to handle nested exceptions (or nested hardware interrupts that cause exceptions), a stack approach is required. If you use the user stack, you reserve as many space on that stack as you need to save registers. If you don't want your exception handler to 'pollute' the user stack you have to setup a special 'stack area' in memory for exception processing, of which the first location can be used to contain the stack pointer value itself. Loading this pointer must be done using $kt0 or $kt1 since none of the other registers is available before they are

© 1996 Tasking Software BV

saved on the exception stack. Also it is important to have interrupts disabled while setting up the exception stack. (Note that interrupt are disabled by hardware until you enable them explicitly in your handler)

Suppose your exception handler reserves a private stack space of adequate size. The actual stack pointer value in this area will be maintained in the memory word exc_stack (the symbolic start of the exception stack area). The following code sample sets up an exception stack, using $kt0, $kt1:

```
#
# start of atomic initialization of the exception stack pointer
#
la      $kt1, exc_stack         # get exception stack pointer address
lw      $kt0, ($kt1)            # get exception stack pointer
sub     $kt0, 36                # reserve register save area
sw      $kt0, ($kt1)            # update stack pointer memory location
sw      $sp, ($kt0)             # preserve user stack pointer value
move    $sp, $kt0               # now use regular stack pointer register
#
# end of atomic exception stack pointer initialization
# save registers used in exception handler
#
sw      $1, 4($sp)              #
sw      $2, 8($sp)              # need one function return register
sw      $4, 12($sp)             # need two argument registers
sw      $5, 16($sp)
sw      $31, 20($sp)            # and return address register
.
other exception handling code
.
```

If you want to debug your exception handler using a debug monitor, or allow nested exceptions for other reasons, the first part of your exception handler has to preserve the EPC (exception program counter) and the interrupt bits in the cause register also. When you debug an exception handler with a debug monitor you actually create nested exceptions: when you set a software breakpoints the debug monitor patches a break instructions in the code, which on its turn raises an exception (break exception). This critical sharing of the same resource has two important consequences:

1. If your exception handler has the initial control when handling exceptions it also catches the break exceptions which are meant for the debug monitor. So, you cannot set software breakpoints in the part of your exception handler that catches the exception (also the break) initially, before taking the decision to transfer control to the original exception handler of the debug monitor. If you would add a software breakpoint in that part of your handler, the debug monitor never gets in control anymore: before your handler has transferred control to the exception handler of the debug monitor it runs into the break again. The result of this will be that an endless series of nested exceptions that locks up the system.

© 1996 Tasking Software BV

2. Your handler has to preserve the resources that are also used by the exception handler of the debug monitor. So you cannot debug code of your handler that uses $kt0, $kt1. The exception handler of the debug monitor uses these registers also and stepping through this code actually causes corrupt register values! So, for this reason it is important to use $kt0 and $kt1 only at those places in your handler where you have to and use general registers where possible.

Also if the debug monitor uses interrupted I/O (the TASKING debug monitor does this) you have to safe the interrupt bits in the cause register and possibly some values of the control registers in the UART. E.g. if the board has a dual UART, the half of which is used by the debug monitor, and control bits are cleared automatically by reading the UART registers, you have to safe the status of those registers before you can step through your code that deals with interrupts generated by this UART. If you fail to do so, the debug monitor I/O instantly clears the control registers on the first break in the code, because it uses the other half of the UART for the debugging process. Also by acknowledging the interrupts in the debug monitor for debug I/O, the interrupt for the other half may get acknowledged implicitly too, when there only one interrupt line from the UART to the CPU. If your handler which is being debugged has to service this interrupt, you see the interrupt disappear because of the debugging process, which can be very confusing.

However, if there is no such critical sharing of hardware resources, preparing your exception handler for debugging is much easier.
The following code fragment shows initial saving of the EPC register (the register in CP0 that contains the return address into application code), the cause register and the status register containing the interrupt bits.

```
    mfc0   $kt0, $epc          # get epc from coprocessor 0
    sw     $kt0, 24($sp)       # store on exception stack
    mfc0   $kt0, $cause
    sw     $kt0, 28($sp)
    mfc0   $kt0, $sr
    sw     $kt0, 32($sp)
```

If at a certain point in the handler you find out that the original handler you have stolen the vector from needs to get control, the complete context will be restored first:

```
    #
    # restore registers used in exception handler
    # interrupts must be disabled now
context_restore:
    lw     $kt0, 32($sp)
    mtc0   $kt0, $sr           # restore status register
    lw     $kt0, 28($sp)
    mtc0   $kt0, $cause        # restore cause register
    lw     $kt0, 24($sp)
    mtc0   $kt0, $epc
```

```
lw      $1, 4($sp)          # restore registers used in the
lw      $2, 8($sp)          # exception handler
lw      $4, 12($sp)
lw      $5, 16($sp)
lw      $31, 20($sp)
#
# start of atomic restore of exception stack pointer
#
la      $kt1, exc_stack     # get exception stack pointer address
move    $kt0, $sp           # needed to reload user stack pointer
add     $sp, 36             # free register save area
sw      $sp, ($kt1)         # update stack pointer memory location
lw      $sp, ($kt0)         # restore user stack pointer value
#
# end of atomic restore exception stack pointer
#
#
# now control can be transferred to the handler at 0x800000c0
#
li      $kt0, 0x800000c0
jr      $kt0
```

If the exception completes normally, the handler returns to the program (after restoring the registers), using the following code:

```
# start of atomic exception stack pointer restore
la      $kt1, exc_stack     # get exception stack pointer address
move    $kt0, $sp           # needed to reload user stack pointer
add     $sp, 36             # free register save area
sw      $sp, ($kt1)         # update stack pointer memory location
sub     $kt1, $sp, 36       # recalculate stack frame to pick up
lw      $kt1, 24($kt1)      # $epc value
lw      $sp, ($kt0)         # restore user stack pointer value
.set noreorder
jr      $kt1
rfe
```

However, this code is not adequate to return from exceptions in all circumstances. When the exception was raised when the program was executing an instruction in a branch delay slot, special action is required.

Exceptions in Branch Delay Slots

If an exception occurs in a branch delay slot, the exception handler cannot simply return to the program by reading the $epc register and jumping to that address. If the branch delay bit (BD) in the cause register is set, $epc contains the address of the preceding branch. This branch has to be reevaluated to determine the correct return address. If the instruction in the branch delay

© 1996 Tasking Software BV

slot is not the cause of the exception (which is the case with external interrupts), the handler can simply resume execution with the branch. So, in that case the return code as shown above still applies.
However, if the instruction in the branch delay slot is the cause of the exception (e.g. an address error exception cause by an unaligned memory read), resuming execution at the branch would immediately cause the same exception to occur again. The appropriate procedure to follow here is:

1. Determine the cause of the exception raised by the instruction in the branch delay slot.
2. Handle this situation in the exception handler.
3. Read the branch instruction at the address in $epc.
4. Interpret the branch in the handler to determine whether the branch would be taken or not.
5. If branch taken compute branch target address and resume execution at that address.
6. If branch not taken resume execution at $epc + 8

If your handler has to service exceptions when the branch delay bit (BD) has been set, you have to save all registers (including $0 !) at the start of your exception handler and not only the ones you intend to use in the handler. The reason for is that the code that interprets the branch code has to have uniformous access to the register values when interpreting the instructions. E.g. when the branch uses register $4 and $5, the exception handler has to compute the corresponding offsets in the register table on the exception stack from the register numbers in the opcode of the branch instruction.

At the start of the exception handler the registers are saved:

```
#
# save registers used in exception handler
#
sw      $0, 4($sp) ;      sw      $1, 8($sp);       sw      $2, 12($sp)
sw      $3, 16($sp);      sw      $4, 20($sp);      sw      $5, 24($sp)
sw      $6, 28($sp);      sw      $7, 32($sp);      sw      $8, 36($sp)
sw      $9, 40($sp);      sw      $10, 44($sp);     sw      $11, 48($sp)
sw      $12, 52($sp);     sw      $13, 56($sp);     sw      $14, 60($sp)
sw      $15, 64($sp);     sw      $16, 68($sp);     sw      $17, 72($sp)
sw      $18, 76($sp);     sw      $19, 80($sp);     sw      $20, 84($sp)
sw      $21, 88($sp);     sw      $22, 92($sp);     sw      $23, 96($sp)
sw      $24, 100($sp);    sw      $25, 104($sp);    sw      $26, 108($sp)
sw      $27, 112($sp);    sw      $28, 116($sp);
#
# need to get the original value of $sp ($29)
#
lw $2, ($sp)        # the first entry on the stack is used to save $sp itself
sw      $2, 120($sp)
sw      $30, 124($sp);    sw      $31, 128($sp)
mfc0    $kt0, $epc                # get epc from coprocessor 0
```

© 1996 Tasking Software BV

```
        sw      $kt0, 132($sp)          # store on exception stack
        mfc0    $kt0, $cause
        sw      $kt0, 136($sp)
        mfc0    $kt0, $sr
        sw      $kt0, 140($sp)
        .
        Exception handling code
        .
        .
```

At the end of the handler when the BD bit has been detected, the following code computes the return address:

```
rfe_branch_delay:
        lw      $5, 136($sp)            # get $cause
        lw      $6, 132($sp)            # get $epc
        and     $2, $5, 0x80000000      #test if BD bit is set
        beq     $2, $0, no_bd
        #
        # branch/jump encoding in instruction word, bits 31..26
        # special (0x00) :  jr, jalr (5..0)
        # bcond  (0x01) :  bltz, bgez, bltzal, bgezal (20..16)
        # 0x04-0x07     :  beq, bne, blez, bgtz
        # COPz (0x16-0x19):     bctz, bcz
        #

        lw      $2, ($6)  # get branch/jump at $epc address
        srl     $3, $2, 26      # get bits 31..26
        bne     $3, $0, no_special
        #
        # special instructions jr, jalr
        #
        and     $3, $2, 1       # mask for jalr (1) or jr (0)
        #
        # jalr rd, rs   rd in 15..11, rs in 25..21
        # jr rs         rs in 25..21
        #
        and     $7, $2, 0x03e00000      # get rs
        srl     $7, 19                  # get regnum rs * 4
        and     $8, $2, 0x0000f800      # get rd
        addu    $7, $7, $sp             # $sp + 4 * regnum + 4 is stack offset
        lw      $2, 4($7)               # get value rs in $2
        beq     $3, $0, end_adj         # if instr == jr -> ready
        srl     $8, 9                   # get regnum rd * 4
        addu    $7, $6, 8               # adjust return address value
        addu    $8, $8, $sp             # stack offset rd = regnum *4 + $sp + 4
        sw      $7, 4($8)               # update return register contents
        b       end_adj                 # complete adjustment
```

© 1996 Tasking Software BV

```
no_special:
        bne     $3, 1, no_bcond
        #
        # bcond bits (20..16) in opcode are set
        # instructions:
        #               bltz, bgez, bltzal, bgezal
        #
        and     $3, $2, 0x001f0000      # get instruction code
        srl     $3, 16
        and     $7, $2, 0xffff          # get branch offset
        sll     $7, 16                  # sign extend
        sra     $7, 14                  # multiply offset in opcode by four for real address
        and     $8, $2, 0x03e00000      # get rs register
        srl     $8, 19
        addu    $8, $8, $sp             # get stack address of register in instruction
        lw      $8, 4($sp)
        addu    $2, $6, 8               # return addr ($2) = epc ($6) + 8

        bne     $3, $0, no_bltz         # bit 21..16 = 0 -> bltz
        #
        # bltz
        #
        bgez    $8, end_adj             # branch not taken $2 is target
        b       bra_adj                 # adjust new pc to epc + bra offset
no_bltz:
        bne     $3, 1, no_bgez          # test opcode for bgez
        #
        # bgez
        #
        bltz    $8, end_adj             # if branch not taken $2 is $epc +8
        b       bra_adj                 # adjust pc to epc + bra offset
no_bgez:
        sw      $2, 128($sp)            # store in $31 (link instruction)
        bne     $3, 16, no_bltzal   # test opcode for bltzal
        #
        # bltzal
        #
        bgez    $8, end_adj             # pc = epc +8
        b       bra_adj                 # pc = epc + branch offset
no_bltzal:
        bne     $3, 17, no_bgezal       # test opcode bgezal
        #
        # bgezal
        #
        bltz    $8, end_adj             # pc = epc + 8
        b       bra_adj                 # pc = epc + branch offset
no_bgezal:
        beq     $3, 2, opc_jal          # test on opcode j and jal
        bne     $3, 3, other_bra
opc_jal:
        #
        # j    : return address after exception is jump target
```

© 1996 Tasking Software BV

```
        # jal : return address after exception is jump target
        #     Saved value $31 must be epc +8
        #
        and     $7,$2, 0x03ffffff   # get jump offset from instruction
        sll     $7, 2               # target address is offset * 4
        and     $4, $6, 0xf0000000  # get most significant nibble of target from epc
        or      $2, $7, $4          # create jump target address
        bne     $3, 3, end_adj      # ready when j instruction
        addu    $6, 8               # return address = epc +8
        sw      $6, 128($sp)        # update value $31 on stack
        b       end_adj
other_bra:
        and     $7, $2, 0xffff      # get branch offset
        sll     $7, 16              # and sign extend
        srl     $7, 14              # branch target addr is  offset * 4
        and     $4, $2, 0x3e00000   # get register number rs
        srl     $4, 19              # compute stack address
        addu    $4, $4, $sp
        lw      $4, 4($4)           # get register value from exception stack
        and     $8, $2, 0x001f0000  # get register number rt
        srl     $8, 14              # compute address in exception stack
        addu    $8, $8, $sp
        lw      $8, 4($8)           # get value from exception stack
        addu    $2, $5, 8           # bra not taken return to epc +8
        bne     $3, 4, no_beq
        #
        # beq
        #
        bne     $4, $8, end_adj     # pc = epc + 8
        b       bra_adj             # pc = epc + branch offset
no_beq:
        bne     $3, 5, no_bne
        #
        # bne
        #
        beq     4, $8, end_adj      # pc = epc +8
        b       bra_adj             # pc = epc + branch offset
no_bne:
        bne     $3, 6, no_blez
        #
        # blez
        #
        bgtz    $4, end_adj
        b       bra_adj
no_blez:
        bne     $3, 7, no_bgtz
        #
        # bgtz
        #
        blez    $4, end_adj
        b       bra_adj
no_bgtz:
```

© 1996 Tasking Software BV

```
        #
        # COPz bczt/bczf
        #
        srl     $4, $2, 16
        and     $8, $4, 0xff00
        bne     $8, 0x4100, no_bc0      # test on cop0 branch opcode
        and     $8, 0x1         # bit 16 is branch true/false selection
        beq     $8, $0, bc0f
        #
        # bc0t
        #
        bc0f    end_adj
        b       bra_adj
bc0f:
        bc0t    end_adj
        b       bra_adj
no_bc0:
        bne     $8, 0x45, no_bc1
        and     $8, 0x1
        beq     $8, $0, bc1f
        #
        # bc1t
        #
        bc1f    end_adj
        b       bra_adj
bc1t:
        bc1f    end_adj
        b       bra_adj
no_bc1:
        #
        # bc2t, bc2f, bc3t, bc3f not implemented
        #

other_bra:
bra_adj:
        #       $6 = epc
        #       $7 = bra offset * 4
        #
        add     $2, $6, $7
        b       end_adj
        #
        # for instructions other than branches and exceptions not caused
        # by interrupts, skip the instruction causing the exception
        #
        lw      $2, 132($sp)            # get epc register from stack
        addu    $2, 4
end_adj:
        sw      $2, 132($sp)
        #
        # return address fixed, restore registers and return from exception
        #
```

Note that the encoding of this handler can be further optimized for speed. For the sake of clarity a fairly straight forward approach to decode the various branch/jump instruction has been chosen.

Also, if your application runs on a MIPS-II or MIPS-III architecture, this handler has to be extended with support (decoding) for the branch likely instructions.

Conclusion

Writing exception handlers on the MIPS architecture that coexist with exception handlers of a debug monitor and especially debugging them with that debug monitor is possible when you take the guidelines in this application into account. Since the debug environment uses the same resources as your application additional code, your exception handler has to transfer control to the debug monitor exception handler when debug exceptions are raised.

Information furnished is believed to be accurate and reliable. However, Tasking assumes no responsibility for the consequences of use of such information nor for any infringement of patents or rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Tasking. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. Tasking products are not authorized for use as critical components in life support devices or systems without the express written approval of Tasking.

© 1996 Tasking Software BV - All Rights Reserved

Boston Systems Office, BSO, Boston Systems Office/TASKING, BSO/TASKING and EDE are registered trademarks of Tasking Inc and Tasking Software BV.