CHAPTER 10

EXCEPTION HANDLING

68000 family exceptions are a class of processor operations that change the flow of control of the processor without losing the current state of the program. Each exception condition has a number from 0 to 255, identifying the particular exception.

The exceptions fall into three groups:

a) Explicit program instructions - trap #n, TRAPV, CHK, CHK2, TRAPcc, and cpTRAPcc.

b) Special events occurring during the execution of an instruction - Bus Error, Address Error, Illegal Instruction, Zero Divide, Privilege Violation, Trace, Line 1010 and Line 1111 Emulator, Coprocessor Protocol Violation, Format Error, and Coprocessor Exceptions.

c) External signals - Reset, Auto-vectored Interrupts, and Normal Vectored Interrupts.

Groups (a) and (b) together are sometimes known as the "hardware exceptions". All exceptions cause the same sequence of operations:

1) The current program counter and status register are saved on the supervisor stack. The 68010/020/030/040 also save a stack format word and a vector offset word. Depending on the exception, other information may also be stacked. For example, internal state information is stacked on bus error, to allow virtual memory support (not the 68000). This operation is omitted for the Reset exception.

2) The status register is updated - the supervisor state bit is set, the trace bit is cleared, and the interrupt mask is set to the appropriate level (Reset and Interrupt exceptions only). The Reset exception also sets the Vector Base Register to zero (not the 68000).

3) The appropriate vector (exception handler routine address) is obtained from the exception vector table (indexed by the exception number), and put in the Program Counter. The Reset exception also sets the supervisor stack pointer from the long word at absolute location zero. For normal vectored interrupts the vector number (64 to 255) is read from the interrupting device. For other exceptions the vector number is generated internally (or provided by the coprocessor, for coprocessor exceptions).

10.1 EXCEPTION HANDLING UNDER OS-9

OS-9 provides default handling for all exceptions. It also provides mechanisms for programs or operating system components to handle any or all exceptions.

OS-9 creates four groups of exceptions:

a) The **trap #0** instruction, used to make operating system calls.

b) The other **trap #n** instructions (1 to 15), used to call trap handler modules.

c) Exceptions as the result of instruction execution - the "hardware" exceptions, other than the **trap #n** exceptions.

d) Interrupts.

OS-9 provides separate mechanisms for handling each of these groups. There are also two mechanisms that apply to all exceptions:

a) Overwriting the exception jump table (always in RAM).

b) Overwriting the exception vector table (may be in ROM, and therefore not writable).

These two mechanisms allow slightly more rapid access to exception handling routines, in particular interrupt service routines. However, because

the exception handling is not through the kernel, operating system calls must be used with extreme care. The **D_IOGlob** area of the System Globals can be used for the storage of variables - for example, it can hold a copy of the old vector, so the interrupt service routine can continue on into the kernel's interrupt handler if desired.

Typical uses are pseudo-DMA and software dynamic RAM refresh.

10.2 USER AND SYSTEM STATE RETURN

The "user state return" routine is called by the kernel after handling any exception if the processor was in user state before the exception, or a task switch has been performed and the current process is about to be executed in user state. The kernel performs the following sequence of operations:

1) The kernel tests the "timed out" flag in the process descriptor (and clears it, ready for the next time slice). If it is set, the kernel performs a task switch, unless the active queue is empty, in which case it allows the current process to continue execution. From OS-9 version 2.3 onwards, it first checks that the priority of the process is not below the "minimum priority" threshold (**D_MinPty** in the System Globals). Otherwise it inserts the process in the active queue, to force it to be suspended.

2) If the process was not timed out, the kernel checks whether the process is "condemned" (bit 1 of the **P$State** field of the process descriptor is set) - the process has received a "kill" signal, or a debugged process has died. If so, it terminates the process, and calls the **F$NProc** routine to make the next process in the active queue the current process.

3) The kernel checks whether the process has a signal pending (the **P$Signal** field of the process descriptor is not zero), and the signal mask (**P$SigLvl**) is clear. If so, the kernel terminates the process if it has no signal handler routine installed, otherwise the kernel copies the process's register stack frame to the process's user stack (so that execution continues with the main body of the program when the signal handler finishe, by making an **F$RTE** system call), sets the signal mask to one, and modifies the register stack frame as follows:
d0 = number of signals in the queue (including the current

one).
d1 = the signal code.
d2 = zero.
a6 = signal handler's static storage.
pc = address of signal handler routine.
This causes the signal handler to be executed when the process is restarted.

4) If the system has a floating point unit, the kernel saves the current FPU context, and restores the FPU context of the new current process (unless there was no change of current process).

5) If the system is using the SSM, the kernel calls the **F$AllTsk** routine, to ensure the MMU is correctly set up for the process.

6) The kernel clears bit 7 of the **P$State** field of the process descriptor, to indicate that the process is executing in user state.

7) Lastly, the kernel restores the data and address registers from the register stack frame, adds 4 to the stack pointer to ditch the vector offset value (see the section on the Exception Jump Table), and executes the **rte** instruction. This instruction loads the status register and program counter from the stack, causing execution to continue with the instruction following the system call, or the point at which an interrupt occurred or the process was suspended by a task switch.

If the process is returning to system state (for example, an operating system component makes a system call, or a system call routine is interrupted), only step 7 is executed. It is this that causes system calls to be indivisible.

Additional system calls can be installed, or existing ones replaced using the **F$SSvc** privileged system call. This is normally done by a kernel customization module, but may be done by any operating system component, such as a device driver or file manager.

10.3 SYSTEM CALLS - TRAP #0

This processor instruction is reserved in OS-9 for making operating system calls. A system call instruction has the form:

```
trap    #0
dc.w    function_code
```

The Microware assembler provides a built-in macro **os9** to do this in one instruction. For example:

```
trap    #0
dc.w    F$Link
```

can be expressed as:

```
os9     F$Link
```

When this exception occurs, the kernel reads the function code word pointed to by the saved program counter (on the supervisor stack), and then adds two to the saved program counter, updating it to point at the instruction following the function code.

The kernel uses the function code word as an index into either the User or the System Dispatch table, depending on whether the call was made from user or supervisor state respectively (the kernel tests the supervisor state flag - bit 13 - of the saved status register on the supervisor stack). The table entry is the address of the routine to call to handle the system call.

Unless the call is made from within an interrupt service routine, the stack used for the system call is naturally the System State stack of the calling process (the upper half of the Process Descriptor), because the processor automatically switches from using the user stack to using the supervisor stack when an exception occurs (because the supervisor state bit is set in the status register). When a process is forked, the kernel sets its system state stack pointer (in the **P$sp** field of the process descriptor) to the address of the top of the process descriptor. When a process becomes the current process, the kernel sets the processor's supervisor stack pointer register from the **P$sp** field of the process descriptor. When a process ceases to be the current process (it goes to sleep, or a task switch occurs), the kernel saves the processor's supervisor stack pointer register in the **P$sp** field of the process descriptor.

The kernel's handler for the **trap #0** exception saves all the data and address registers, making a register stack frame on the process's system state stack. The stack frame includes not only the data and address registers, but also above them the vector offset (the vector number times four) as a long word in the **R$a7** field, the status register (a word), and the program counter (updated past the **trap #0** instruction as part of the processor's exception handling). The kernel clears the condition codes register (the low byte of the status register) in the stack frame, so that the default is to return the carry flag clear to the caller, indicating no error.

If the call is made from user state, the kernel also saves the supervisor stack pointer (pointing to the stack frame) and user stack pointer registers in the **P$sp** and **P$usp** fields of the process descriptor respectively. The kernel then saves the values currently in the **P$ExcpPC** and **P$ExcpSP** fields of the process descriptor, and sets them equal to the supervisor stack pointer and the address of the instruction following the kernel's call to the system call handler routine. This causes any hardware exception within the system call handler to return cleanly to the kernel, as described below.

The kernel copies the function code to the high word of the "stack pointer" field (**R$a7**) of the stack frame (the low word contains the vector offset, placed on the stack as a long word by the exception jump table instructions), and sets the **a5** register to point to the stack frame. The system call handler routine will use the stack frame to access the caller's parameters, and to return values to the caller.

Finally, the kernel uses the function code, as described above, to get the address of the appropriate handler routine, and also to get the address of the static storage of the handler routine (see the chapter on the OS-9 Internal Structure). The kernel then calls the handler routine. On return from the handler routine (which could be as the result of a hardware exception), the kernel checks the returned carry flag. If it is set, the kernel sets the carry flag in the condition codes register in the stack frame, clears the high word of the **d1** register in the stack frame, and writes the error code (returned in the low word of the **d1** register) to the low word of the **d1** register in the stack frame. Note that this means that if there is no error, the **d1** register is preserved (unless it was changed in the stack frame by the system call handler routine), otherwise the **d1** register contains the error code as a long word (the high word is zero).

On return from the system call handler, and having set the error status in the stack frame if there was an error, the kernel branches to its "user state return" or its "system state return", depending on whether the call was made from user or system state.

10.4 TRAP HANDLER MODULES - TRAPS #1 TO #15

These processor instructions are used within OS-9 to call trap handler modules. A trap handler module is essentially an OS-9 memory module containing any number of subroutines that can be called by function code rather than by address, and that has its own static storage, separate from that of the process using the trap handler. This provides a mechanism for

calling functions without the need to know the address of the functions, or the need to reserve static storage for them.

Microware provide the **cio** and **math** trap handler modules. The C libraries, in conjunction with 'cstart.a', use **trap #13** and **trap #15** respectively to call these trap handler modules, although in principle any trap number can be used to call any trap handler module. The program wishing to use a trap handler module makes the **F$TLink** system call, specifying the name of the trap handler module, the number of the trap instruction (1 to 15) that will be used to call it, and an "additional static storage" size (usually zero).

The kernel allocates static storage memory for the trap handler. The size of the static storage is the sum of the **M$Mem** and **M$Stack** fields of the trap handler's module header, and the "additional static storage" parameter to the **F$TLink** system call. The trap handler may not require any static storage, in which case its **M$Mem** and **M$Stack** fields are zero. The kernel saves the addresses of the trap handler module and its static storage, and the size of the static storage, in the process descriptor of the calling process. The process descriptor fields used (**P$Traps**, **P$TrpMem**, and **P$TrpSiz**) are each arrays of 15 entries, so a process can use up to 15 trap handlers concurrently, one for each **trap #n** instruction other than **trap #0**.

Note that the static storage for the trap handler is allocated separately for each process that has used the **F$TLink** system call to link to the trap handler. The kernel initializes the static storage in the same way that it does for a program, so initialized data can be used. Also, the kernel adds 32k to the static storage address before saving it in the process descriptor, just as is done for a program. The linker compensates by subtracting 32k from all static storage references when creating a trap handler module. As for a program, this is done to maximize the amount of static storage that can be accessed using the signed 16-bit constant offset indexed addressing mode of the 68000 processor family.

A trap handler can execute in user or supervisor state. The kernel will execute trap handler functions in supervisor state if the "system state" bit is set in the attributes field of the module header. However, the kernel will give a "no permission" error (**E_PERMIT**) if an **F$TLink** system call is made for a system state trap handler which was not created by a super user - that is, the owner group number (the high word of **M$Owner**) of the module header is not zero.

Once a process has linked to a trap handler using the **F$TLink** system call, it can call the functions of the trap handler using the **trap #n** instruction

followed by a 16-bit function code. For example, to call the **sscanf()** function in the **cio** trap handler:

```
trap    #13
dc.w    $1A
```

Unless the trap handler is a system state trap handler, the kernel builds a parameter frame on the user state stack (as show below), restores the data and address registers, and uses the **rte** instruction to return to the state (system or user) of the caller and jump to the trap handler.

Therefore if a system state process calls a "user state" trap handler, the trap handler is called in system state. However, because the stack frame is built on the user state stack, the trap handler will have no access to the stack frame (unless it knows it is being called from system state, which it cannot check if it could also be called from user state, as reading the high byte of the status register in user state is only possible on the 68000/010). This implies that a system state process cannot successfully call a user state trap handler.

If the trap handler is a system state trap handler, the kernel builds a stack frame as described below, and calls the trap handler entry point as a subroutine (in system state). On return, the kernel calls its "return to user state" function. This implies that a system state process should not call a system state trap handler. Note that prior to OS-9 version 2.3, the kernel jumped directly to the trap handler entry point, so it was the responsibility of the trap handler to finish with an **rte** instruction. Since OS-9 version 2.3 the trap handler returns to the kernel with an **rts** instruction, allowing the kernel to perform its normal "return to user state" checks.

10.4.1 The Trap Handler Routine

Because a user state trap handler returns directly to the calling process, not through the kernel, it is the trap handler's responsibility to preserve or modify the processor registers as required. In effect, the trap handler is acting as a subroutine of the program. The kernel calls a user-state trap handler with the following register parameters and stack frame:

```
d0-d7/a0-a5 = caller's registers
(a6) = trap handler's static storage
a7 = usp
8(a7).l = caller's return program counter
6(a7).w = exception vector offset
4(a7).w = trap function code
0(a7).l = caller's a6
```

The exception vector offset is the offset for the trap instruction vector, which is (32 + trap_number)*4. If the trap handler has no private static storage, the

a6 register passed is the value used by the program when making the **F$TLink** system call – usually the address of its own static storage. The trap handler must use the function code to decide which subroutine to execute. On return from the subroutine it must restore the caller's **a6** register from the stack frame, add 8 to the stack to remove the parameters, and execute an **rts** instruction to return to the program.

Because the trap handler is effectively acting as a subroutine of the program, it can make any system calls, which will be made on behalf of the program. However, C library functions must be used with care, as they may use private static storage variables. These static storage variables will be in the trap handler's static storage, not the program's, which might cause some conflict (for example, when using buffered I/O functions such as **fread()**).

A system state trap handler is called with registers and stack frame as follows:

```
d0-d7/a0-a5 = caller's registers
(a6) = trap handler's static storage
a7 = ssp
8(a7).l = kernel's return program counter
6(a7).w = exception vector offset
4(a7).w = trap function code
0(a7).l = caller's a6
```

The trap handler acts in the same way as a user state trap handler. Note, however, that it is not necessary to restore the caller's **a6** register, as the kernel immediately loads **a6** with the System Globals address. The kernel preserves the caller's **a6** register itself. As with all system state components, the stack used is the calling process's system state stack, in the upper half of the process descriptor.

Prior to OS-9 version 2.3, a system state trap handler was called with a slightly different stack frame:

```
10(a7).l = caller's return program counter
8(a7).w = caller's status register
6(a7).w = exception vector offset
4(a7).w = trap function code
0(a7).l = caller's a6
```

It was the responsibility of the trap handler to restore the caller's **a6** register from the stack frame, add 8 to the stack pointer, and return directly to the caller with an **rte** instruction.

10.4.2 Installing Trap Handlers

A process can install (link to) a trap handler by using the **F$TLink** system call. This can be explicitly executed in the main body of the program. However, to make the use of trap handlers as transparent as possible, the **F$TLink** system call can instead be executed automatically when the first **trap #n** instruction tries to call the trap handler. To do this, a program must have an "uninitialized trap handler entry point". This is a routine in the program module, the offset to which is given in the **M$Excpt** field of the module header. The offset is calculated by the linker, from the entry point symbol given as the seventh parameter to the **psect** directive.

When the program executes a **trap #n** instruction, if the process does not have a trap handler installed for that trap number, the kernel calls the program's uninitialized trap handler routine with the registers and stack frame exactly as for a user state trap handler. The routine should use the vector offset value on the stack to determine which trap handler is required, and execute the **F$TLink** system call to install the trap handler. It must then re-execute the **trap #n** instruction. This is done by subtracting 4 from the return address in the stack frame, to point again at the **trap #n** instruction, and then returning in the same way as a user state trap handler. This is valid for both user and system state trap handlers. The 'cstart.a' file used for C programs contains such a routine – it is worth studying as an example.

The **F$TLink** system call attempts to link to the required trap handler. If the trap handler module is not in the module directory, the kernel attempts to load a file of the given name, relative to the current execution directory, and uses the first module in the file. The kernel then allocates and initializes the trap handler's static storage (if any), and calls the initialization routine of the trap handler. The initialization routine of a user state trap handler is called with the following registers and stack frame:

```
d0-d7 = caller's registers (d0.w = trap number)
(a0) = caller's trap module name string, updated past end of string
a1.l = address of the trap execution routine
(a2) = trap handler module header
a3-a5 = caller's registers
(a6) = trap handler's static storage
8(a7).l = caller's return program counter
4(a7).l = zero
0(a7).l = caller's a6 register
```

The initialization routine returns directly to the calling program, not to the kernel, just as the main trap handler execution routine does. The initialization routine must finish by restoring the caller's **a6** register,

removing the 8 bytes of information on the stack, and returning to the calling program with an **rts** instruction.

The initialization routine of a system state trap handler is called with:

```
d0-d7 = caller's registers (d0.w = trap number)
(a0) = caller's trap module name string, updated past end of string
(a1) = System Globals
(a2) = trap handler module header
a3-a5 = caller's registers
(a6) = trap handler's static storage
8(a7).l = return program counter (into kernel)
4(a7).l = zero
0(a7).l = caller's a6 register
```

On return from the system state trap handler's initialization routine, the kernel copies the returned registers **d0–d7/a0–a5**, and **ccr**, to the calling program's register stack frame. The initialization routine must therefore preserve all the registers other than **a6** that its specification does not explicitly state will be changed, just as for a user state trap handler. Note that normally a trap handler preserves all the registers, but OS–9 permits it to return results to the calling program by changing the registers. The system state trap handler's initialization routine is called in system state, as a subroutine of the kernel. It should finish by restoring the caller's **a6** register, removing the 8 bytes of information on the stack, and returning to the kernel with an **rts** instruction. As with all system state components, the stack used is the calling process's system state stack, in the process descriptor.

Prior to OS–9 version 2.3 the initialization routine of a system state trap handler was called with a slightly different set of parameters and stack frame:

```
d0-d7 = caller's registers (d0.w = trap number)
(a0) = caller's trap module name string, updated past end of string
a1.l = the address of the trap execution routine
(a2) = trap handler module header
a3-a5 = caller's registers
(a6) = trap handler's static storage
10(a7).l = return program counter (to calling program)
8(a7) = caller's status register
4(a7).l = zero
0(a7).l = caller's a6 register
```

This initialization routine returns directly to the calling program, not to the kernel, just as a user state trap handler initialization routine does. The initialization routine must finish by restoring the caller's **a6** register, removing the 8 bytes of information on the stack, and returning to the calling program with an **rte** instruction.

10.4.3 Terminating Trap Handlers

Although the module header format for a trap handler includes the offset to a termination routine, the termination routine is never called. The trap handler is simply unlinked and its static storage memory is returned when the program makes the **F$TLink** system call with a module name pointer of zero (for the appropriate trap number), or when the program terminates.

It is possible that future releases of OS-9 will call the termination routine, so a trap handler should include a termination routine, expecting the same registers and stack frame passed to the initialization routine, and returning to the caller with the carry flag clear. Alternatively, and following Microware's example in the OS-9 Technical Manual, the termination routine could execute an **F$Exit** system call with a suitable error number (Microware suggest 455).

10.4.4 Writing a Trap Handler in C

The OS-9 Technical Manual gives an example of a user state trap handler in assembly language. However, just as with device drivers and file managers, it is also possible to write a trap handler in C, provided a suitable "skeleton" in assembly language is provided. Such a skeleton (for a system state trap handler) is shown below. Note that by saving the registers on the stack, the skeleton is providing a complete stack frame to the C function that handles the trap instructions, including the condition codes register (**ccr**). From OS-9 version 2.4 onwards, Microware provides the source code of a skeleton user state trap handler, in the directory 'C/SOURCE'.

The trap handler skeleton below presets the **ccr** image to zero, so that the default return is with the carry flag clear, but the C function can set any ccr bits (such as the V bit, to indicate arithmetic overflow). Note that in order not to need to reconstruct the stack frame passed by the kernel, the **ccr** image is held in the stack frame location normally used for the **a7** register (**R$a7** in assembly language, or **a[7]** in C), and is manipulated as a long word (that is, the actual **ccr** image is in the last byte of the four byte field).

```
* File: trapskel.a
* System state trap handler skeleton for a trap handler in C
            use     /dd/defs/oskdefs.d
Typ_Lang    set     (TrapLib<<8)+Objct
Att_Revs    set     (ReEnt+SupStat)<<8
Edition     set     2
            psect   trapskel,Typ_Lang,Att_Revs,Edition,0,TrapEnt
*****************************************
* Static storage (local to trap handler)
*
            vsect
errno:      ds.l    1               standard C error number location
            ends

*****************************************
* Entry point offset table:
*
            dc.l    TrapInit        initialization routine
            dc.l    TrapTerm        termination routine

*****************************************
* TrapInit
* Initialize trap handler
*
* Passed:   d0.w = trap number
*           d1.l = additional static storage allocated (caller's d1)
*           d2-d7 = caller's registers
*           (a1) = trap handler execution entry point
*           (a2) = trap handler module header
*           a3-a5 = caller's registers
*           (a6) = trap handler static storage
*           4(a7) = 0
*           0(a7) = caller's a6 (static storage ptr)
* Returns: carry set if error, with error code in d1.w
* May destroy: ccr
*
* Parameters passed to C function 'trapinit':
*       int trapinit()
* The C function returns zero if no error, else the OS-9 error code.
*
TrapInit
            movem.l d0-d1/a5,-(a7)  save caller's regs
            move.w  #0,a5           reset stack trace pointer
            bsr     trapinit        call C function
            tst.l   d0              any error?
            beq.s   TrapInit10      ..no
            move.l  d0,4(a7)        overwrite saved d1 with error
            ori     #Carry,ccr      show error
TrapInit10  movem.l (a7)+,d0-d1/a5-a6   retrieve caller's regs
            addq.l  #4,a7           ditch zero parameter
            rts
```

```
*****************************************
* TrapTerm
*  Trap handler termination function
* NOTE: at present OS-9 never calls the termination function of a trap
* handler.
*
TrapTerm
            move.w  #1<<8+199,d1    Microware's suggested 'crash'
            os9     F$Exit

*****************************************
* TrapEnt
*  Trap handler main entry point
*
* Passed:   d0-d7 = caller's registers
*           a0-a5 = caller's registers
*           (a6) = trap handler static storage
*           6(a7) = trap vector offset (word)
*           4(a7) = trap function code (word)
*           0(a7) = caller's a6 (static storage ptr)
* Returns:  depends on C function 'trapent'
* May destroy:  depends on C function
*
* Parameters passed to C function 'trapent':
*       void trapent(x,r)
*       int x;          /* function number */
*       REGISTERS *r;   /* caller's stack frame ptr */
* The C function may return values to the caller by modifying the stack
* frame (d0-d7/a0-a5 and ccr in R$a7 only).
*
TrapEnt
        movem.l d0-d7/a0-a5,-(a7)   make stack frame
        move.w  #0,a5               reset stack trace pointer
        moveq.l #0,d0
        move.w  60(a7),d0           get function code
        move.l  a7,d1               copy stack frame address
        clr.l   R$a7(a7)            clear "ccr"
        bsr     trapent             call C function
        movem.l (a7)+,d0-d7/a0-a6   retrieve registers
        move.b  3(a7),ccr           set return ccr
        addq.l  #4,a7               ditch ccr image
        rts

        ends
```

The main body of the trap handler – written in C – must be in a separate source file. Below is a "do nothing" example, compatible with the skeleton above:

```
/* File: trap.c
    Trap handler main body
*/

#include <errno.h>                      /* error numbers */
#include <modes.h>                      /* file modes */
#include <types.h>                      /* unsigned data types */
#include <MACHINE/reg.h>                /* register stack frame */

/* Static storage: */
int call_count=0;                       /* number of calls received */

/* Initialize trap handler */
int trapinit()
{
    return(0);                          /* no error */
}

/* Main trap handler function */
void trapent(x,r)
int x;                                  /* function number */
REGISTERS *r;                           /* caller's stack frame ptr */
{
    call_count++;           /* count calls (for something to do) */
    switch (x) {                        /* act on function code */
        case 1:                         /* request call count */
            r->d[0]=call_count;         /* return call count in d0 */
            break;
        default:                        /* unknown request */
            r->d[1]=E_UNKSVC;           /* error code */
            r->a[7]|=1;                 /* set carry flag in ccr */
            break;
    }
}
```

As when writing a device driver or file manager in C, a **make** file should be used. However, for the purposes of the example, equivalent command lines to assemble, compile, and link the trap handler are shown below:

```
$ r68 trapskel.a -qo=RELS/trapskel.r
$ cc -qr=RELS trap.c
$ l68 RELS/trapskel.r RELS/trap.r -l=/dd/LIB/clibn.l
-l=/dd/LIB/math.l -l=/dd/LIB/sys.l -O=OBJS/trap
```

A simple **make** file to do the same thing is shown below:

```
# make file to make 'trap' module
RDIR = RELS                 # directory for ROFs
ODIR = OBJS                 # directory for object modules
LDIR = /dd/LIB              # directory for libraries
CFLAGS = -q                 # compiler flags for automatically \
                              generated command lines
RFLAGS = -q                 # assembler flags for automatically \
                              generated command lines
RFILES = trapskel.r trap.r  # names of ROFs

trap: $(RFILES)             # root dependency - make 'trap'
    chd $(RDIR);l68 $(RFILES) -l=$(LDIR)/clibn.l -l=$(LDIR)/math.l \
        -l=$(LDIR)/sys.l -O=../$(ODIR)/$@
```

10.5 HARDWARE EXCEPTIONS

These exceptions – such as bus error, address error, and illegal instruction – always occur as a result of a problem in executing a program instruction. The exception may be a normal part of program execution (for example, a **TRAPV** instruction generating an exception as the result of overflow in an arithmetic operation), or it may indicate a programming error (for example, if an illegal instruction exception occurs). Normally, if one of these exceptions occurs during the execution of a process, the kernel will immediately terminate the process, as such an exception implies an unexpected catastrophic error. The exit status of the program is calculated as 100 plus the exception number. For example, a bus error will give an exit status of 102, and an address error will give an exit status of 103. However, in some circumstances the programmer may be able to anticipate and cope with the error. OS-9 therefore provides a means for a program to intercept these exceptions.

Hardware exceptions in system state usually indicate a fatal system error. However, the error may be recoverable or ignorable – preferable to crashing the system. OS-9 therefore provides a separate means for such exceptions to be handled, and the kernel uses this to provide a basic protection against system state exceptions in system calls. If this mechanism has not been reset

or changed by the system call, and a hardware exception occurs during the system call, the kernel returns the exception as an error to the calling program. The error code is 100 plus the exception number. For example, a bus error gives error code 102.

In systems which include one of the Microware ROM-based debuggers, the exception vector table entries for some of the hardware exceptions (bus error, address error, and illegal instruction) point to handlers in the debugger, rather than the corresponding entries in the exception jump table (see the section on the Exception Jump Table). The debuggers have an "enable" command - "e[CR]". If the debugger is "enabled", and one of these exceptions occurs, the debugger is entered and performs a register dump. This allows the programmer to investigate the cause of these potentially fatal error conditions. If the debugger is not "enabled", the debugger jumps to the appropriate entry in the exception jump table, causing the exception to be processed in the normal way by the kernel.

10.5.1 Hardware Exceptions in User State

A process can install handler routines for one or more of the hardware exceptions. The handler routine will only be called if the exception occurs while the process is executing in user state. The handler will not be called if the exception occurs while another process is the current process, or if the exception occurs while the processor is in supervisor state. Once installed, a handler routine may be removed by a further program request, in which case a subsequent exception of that type will cause the program to be terminated.

The handler routines are installed and removed using the **F$STrap** system call. The program passes a pointer to a list of structures, each describing a handler to be installed. Each structure consists of two 16-bit words. The first word gives the 68000 exception vector offset. For example, the bus error exception is exception number 2, so it has an exception vector *offset* of 8, four times the exeception number. Microware have provided symbolic definitions for the exception vector offsets in the file 'DEFS/sysglob.a' (so the symbols are available from the library 'LIB/sys.l', included by the **cc** executive when linking a C program). The symbols all start with the characters **T_**. For example, the bus error exception vector offset has the symbol **T_BusErr**. The following table shows the exceptions that can be intercepted, with their numbers, offsets, and symbolic names for the offsets.

Description	Number	Offset	Symbol
Bus error	2	8	T_BusErr
Address error - odd address when even required	3	12	T_AddErr
Illegal instruction	4	16	T_IllIns
Divide by zero	5	20	T_ZerDiv
CHK instruction	6	24	T_CHK
TRAPV instruction - arithmetic overflow	7	28	T_TRAPV
Privilege violation - supervisor state instruction executed in user state	8	32	T_Priv
Line 1010 emulator	10	40	T_1010
Line 1111 emulator	11	44	T_1111
FPU Branch or set on unordered condition	48	192	T_FPUnordC
FPU inexact result	49	196	T_FPInxact
FPU divide by zero	50	200	T_FPDivZer
FPU underflow	51	204	T_FPUndrFl
FPU operand error	52	208	T_FPOprErr
FPU overflow	53	212	T_FPOverFl
FPU not a number	54	216	T_FPNotNum

The second word in the structure gives the offset to the routine. The offset is calculated from the end of the structure. The following assembly language line would produce the appropriate structure for a bus error handler function **BusError**:

```
        dc.w    T_BusErr,BusError-*-4
```

The list is terminated by a word of −1 ($FFFF). The list shown below would provide handlers for the bus error, address error, and illegal instruction exceptions:

```
Handlers    dc.w    T_BusErr,BusError-*-4
            dc.w    T_AddErr,AddError-*-4
            dc.w    T_IllIns,IllError-*-4
            dc.w    -1
```

The order of the structures in the list is not important (unless there are two for the same exception vector offset!). The calling program passes a pointer to the list in the **a1** register, and a stack frame space pointer in the **a0** register. The kernel saves the handler address and the stack frame space address in two tables in the caller's process descriptor. The handler addresses are saved in the table at **P$Except**, and the stack frame addresses are saved in the table at **P$ExStk**.

If the routine offset in a list structure is zero, the kernel clears the handler address in the appropriate entry of the **P$Except** table. This is the way the handler for an exception can be removed:

```
NoHandlers  dc.w    T_BusErr,0
            dc.w    T_AddErr,0
            dc.w    T_IllIns,0
            dc.w    -1
```

Note that because the offsets are word values relative to the address of the table entries, the handler routines must be located in the program module containing the table, and cannot be more than plus or minus 32k bytes away from the table entry.

The 68020/030 with FPU (68881 or 68882), and the 68040 (which has an internal FPU), can generate additional floating point exceptions. These are supported in the same way by OS-9. The handler addresses and stack frame addresses are saved in additional tables in the process descriptor, **P$FPExcpt** and **P$FPExStk** respectively.

When a hardware exception occurs in user state, the kernel uses the vector offset of the exception as an index into the table in the process descriptor of the current process. If the handler address is zero, the kernel terminates the process, giving it an exit status of 100 plus the exception number. Otherwise, the kernel builds a register stack frame in the memory whose address was given by the **F$STrap** system call. If the stack frame space address is zero, the kernel uses the process's current user stack pointer.

Note that in either case the kernel builds the stack frame *below* the address given (simulating a push down stack). Therefore when giving a fixed address at which to build the stack frame, the program must add the size of the stack frame to the base address of the storage before passing it to the **F$STrap** system call. The size used by the kernel is calculated as **R$Size-2** (70 bytes) - that is, no exception format and vector word is written to the stack frame. Before building the stack frame, the kernel checks that the user has write permission for the memory. If not, the process is terminated, with a stack overflow exit status (**E$StkOvf**).

The kernel then pushes the status register at exception and the address of the handler onto the system state stack, and so calls the handler by executing an **rte** instruction. The exception handler is called in user state (the state at the time of exception). It does not return to the kernel. The effect is as if the program had built the stack frame and jumped (not a subroutine call) to the handler routine, instead of executing the instruction that caused the exception.

The handler routine is passed:

```
d7.l = exception vector offset (exception number times 4)
a0.l = program counter at exception
a1.l = user stack pointer at exception
(a5) = register stack frame
a7.l = a5.l unless explicit stack frame space specified
66(a5) = program counter at exception (= a0)
64(a5) = status register at exception
60(a5) = user stack pointer at exception (= a1)
0(a5) = d0-d7/a0-a6 at exception
```

The exception handler is effectively jumped to as a change of flow of control in the program. It must decide whether and how to continue execution of the program. It may decide that it can fix the problem, and allow the main program to continue execution. Having fixed the problem, the exception handler would restore the program's registers from the stack frame (including the condition codes register - **ccr**), restore the program's stack pointer (passed to the exception handler in the **a1** register), and then jump back to the program, using the program counter passed in the **a0** register. Note that the program counter will not normally point at the instruction that caused the exception. Usually it will have been incremented by the processor to point at the next instruction, but for exceptions caused by a memory access (bus error and address error) the program counter may point part of the way through the instruction that caused the exception.

Alternatively, the exception handler may decide to continue execution at a different point in the program, or to terminate the program (perhaps preceded by a "clean up" sequence). Just as with a signal handler routine, the exception handler can execute any system calls - it is executing as a part of the program, in user state - but because it is called asynchronously, it must be careful not to use program variables that may be in use by the main body of the program.

The main concepts to understand in order to write a user state exception handler under OS-9 are:

- The exception handler is effectively asynchronously jumped to (not a subroutine call).
- The kernel builds a register stack frame *below* the given memory address, or on the user stack if no address was specified. The stack frame contains the data and address registers (including the stack pointer), the status register, and the program counter, as they were at exception.

10.5.2 Example - Bus Error Handler

The "bus error" exception is probably the exception most commonly required to be intercepted by a program. A bus error exception occurs if the processor "bus error" input signal is asserted in response to a memory access, instead of the normal termination of a memory access. External circuitry on the processor board normally asserts the bus error input if a memory access attempted by the processor has not completed within a certain time, indicating that no device is responding to the address that has been put out by the processor.

The timeout depends on the processor board (and some boards have a programmable timeout), but a time of the order of 200 microseconds is typical. The timeout may occur because the address does not match the address of any device (memory chip or I/O interface chip) in the system, or because the device is currently busy, and refuses to respond. The bus error signal is also asserted by the Memory Management Unit (MMU) if one is in use (by the System Security Module, under OS-9), and a program tries to access a memory location that is not within its current memory map, or it tries to write to a location for which it does not have write permission.

A program that checks for the existence of an area of memory, or an I/O interface, will need to install a bus error exception handler, to handle the exception that will occur if the memory or interface chip is not present. Also, a program that directly accesses an I/O interface that is sometimes busy will need to install a bus error exception handler to retry an access to the interface. In the first case the program will not want to retry the instruction that caused the bus error - it will set a "device does not exist" flag. In the second case the program will want to retry the instruction, perhaps with a maximum number of attempts. This can be done by resetting the program counter to point again at the instruction, or by setting a flag and jumping to the end of a loop in the program to test for success or failure.

The example below shows a bus error exception handler for the first case. The program is attempting to determine whether an I/O interface chip is present at the given address in this system. The main body of the program is in C, but the function to make the **F$STrap** system call must be in assembly language, as must the exception handler (or at least a skeleton function). In this example the assembly language function **probe_byte** attempts to read a byte from the given memory address. If it succeeds (no bus error), the function finishes in the normal way, returning the **d0** register set to zero. Otherwise, the exception handler is called, which sets the **d0** register to -1,

and jumps to the instruction in the **probe_byte** function following the instruction to set **d0** to zero.

```
#include <stdio.h>
#include <errno.h>
#include <MACHINE/reg.h>

#define ERROR (-1)

REGISTERS stack_frame;          /* structure for stack frame */

int f_strap(),probe_byte();     /* declare functions */

main(argc,argv)
int argc;
char **argv;
{
    char *check_addr;

    /* The address to test is a command line parameter: */
    if (argc!=2 || sscanf(argv[1],"%x",&check_addr)!=1)
        exit(_errmsg(1,"Invalid board address\n"));
    if (f_strap(&stack_frame)==ERROR)       /* install handler */
        exit(_errmsg(errno,"Can't install handler\n"));
    /* Test for the existence of a device at the address given: */
    if (probe_byte(check_addr)==-1)
        _errmsg(1,"No board at address %08x\n",check_addr);
    else
        _errmsg(1,"Board exists at address %08x\n",check_addr);
}
/* Function to install bus error handler
    Passed:     address of stack to use (zero to use program stack)
*/
#asm
f_strap:    movem.l d1/a0-a1,-(a7)  save registers
            lea     ExcpTbl(pc),a1  point at table of handlers
            tst.l   d0              any stack given?
            beq.s   f_strap10       ..no
            addi.l  #R$Size-2,d0    convert to pointer to top of stack
f_strap10   movea.l d0,a0           copy top of stack address
            os9     F$STrap         make the system call
            bcs.s   f_strap20       ..error
            moveq   #0,d0           show no error
            bra.s   f_strap30       ..and return
f_strap20   move.l  d1,errno(a6)    save error code
            moveq   #-1,d0          show error occurred
f_strap30   movem.l (a7)+,d1/a0-a1  retrieve registers
            rts                     return to C program
* Table of exceptions to handle, and handler offsets:
ExcpTbl     dc.w    T_BusErr,bus_hand-*-4
            dc.w    -1              end of table marker
```

```
* Read a byte from a specified address:
* Passed:   d0.l = address to test
probe_byte: move.l  a0,-(a7)        save register
            move.l  d0,a0           copy address to use
            move.b  (a0),d0         read byte
* The following instruction is only executed if no
* bus error occurred:
            moveq   #0,d0           show no bus error

probe_byte10
            movea.l (a7)+,a0        retrieve register
            rts                     return to C program

* Bus error handler:
bus_hand:   movea.l a1,a7           restore stack pointer
            movem.l (a5),d0-d7/a0-a4    restore regs from stack frame
            movea.l R$a5(a5),a5     restore a5 from stack frame
            moveq   #-1,d0          show bus error occurred
            bra.s   probe_byte10    ..finish off
#endasm
```

In the more generalized case, where the bus error exception could occur in more than one place (for example, separate functions might be used to try reading a byte, or a word, or a long word), the program could set a static storage variable to indicate which function is executing, or the program could save in static storage the program address at which to continue execution after a bus error.

10.5.3 'move from sr' and 'move from ccr'

The **move from sr** (copy the status register) instruction is not a privileged instruction on the 68000/010, and these members of the 68000 family do not have a separate **move from ccr** (copy the condition codes register) instruction. In contrast, the higher members of the 68000 family (68020/030/040) have a **move from ccr** instruction, and on these processors the **move from sr** instruction is privileged – a "privilege" exception occurs if this instruction is executed in user state.

In order to be compatible with both groups of processors, the kernel checks the "illegal instruction" and "privilege" exceptions, to see if they are due to a **move from ccr** or **move from sr** instruction respectively. If so, the kernel emulates the instruction, by moving the **ccr** register to the destination specified in the instruction, rather than passing the exception to the process's exception handler (or terminating the process if it has no handler). This makes programs written with either instruction execute correctly on both groups of processors.

10.5.4 Hardware Exceptions in System State

The processor is in supervisor state during the execution of a system call, a system state program, a system state trap handler, or an interrupt service routine. When a hardware exception occurs in system state, the kernel handles the exception in a different manner from a hardware exception in user state.

A hardware exception during interrupt handling (which is always in system state) is considered a special case. If the hardware exception occurs during an interrupt service routine (the **D_IRQFlag** field in the System Globals is not negative), the operating system gives up through a controlled system crash. It prints a "System state exception" message on the system console (reporting the exception vector offset), and attempts a soft reset (jump to the bootstrap ROM entry point). If a ROM-based debugger is available, the kernel calls the debugger, rather than jumping straight to a soft reset. The kernel "crashes" the system because it cannot know whether the interrupt has been successfully handled, or whether the I/O device is now in a non-functional state, with further use possibly resulting in a corruption of a filing system.

Therefore if an interrupt service routine anticipates that it may cause a hardware exception, it should temporarily patch the exception jump table before executing the instruction that may cause the exception. This is a perfectly valid mechanism - it cannot cause conflict, because interrupts are handled in a purely hierarchical prioritized order (this is a function of the processor) - they cannot sleep or be switched out.

If the processor is not executing an interrupt handler at the time of the system state exception, the kernel attempts to call a system state exception handler for the current process. Like the user state exception handlers, the address of the exception handler and the address of the stack to use are held in the process descriptor, so separate exception handlers can be set for each process. This allows a process to go to sleep without having to save and restore the handler address and stack pointer. Unlike the user state exception handlers, there is only one system state hardware exception handler address field in the process descriptor (**P$ExcpPC**) for handling all system state hardware exceptions, and only one field for the stack pointer to use at exception (**P$ExcpSP**).

If the stack pointer field (**P$ExcpSP**) of the process descriptor of the current process is zero the kernel gives up through a controlled system crash, as described above. This is the default case while executing a system-state process or system-state trap handler. Otherwise the kernel loads the **a7** register (the stack pointer) with the value in the **P$ExcpSP** field of the

process descriptor, unmasks all interrupts, and jumps to the address given in the **P$ExcpPC** field.

When called to execute a system call (**trap #0** instruction) the kernel installs a default exception handler, which simply returns the exception as an error to the caller. The error code is the exception number plus 100. The kernel restores the original handler (usually "none") before returning to the caller.

However, the **I$Attach** system call routine temporarily installs its own exception handler before calling the initialization routine of the device driver, such that an exception is treated as an initialization error (returning an error code of 100 plus the exception number). This causes a program to get an error **E_BUSERR** if it attempts to open a path to a device for which the hardware is not present. In addition, the device driver has the opportunity to de-allocate any allocated resources, because its termination routine is called by the **I$Attach** system call, as happens if the initialization routine returns an error in the normal way.

Any system state routine can intercept hardware exceptions by temporarily replacing the stack pointer and exception handler fields in the process descriptor of the current process. On exception, the kernel jumps to the exception handler address, effectively causing an asynchronous change of flow of control in the system state routine that caused the exception. The handler is called as follows:

```
d1.w = exception number plus 100
d2-d6/a1-a3/a5 = registers at exception
d7.l = exception vector offset
(a4) = current process's Process Descriptor
(a6) = System Globals
a7.l = value taken from P$ExcpSP
sr = interrupt mask is clear
```

Registers **d0-d1/d7/a0/a4/a6** and **ccr** are lost. Note that the kernel does not place any parameters on the stack. The example below shows a system state routine recovering from a bus error:

```
* The address of the location to test is passed in the d0 register:
ProbeByte:  movem.l d1-d2/a0,-(a)       save registers
            move.l  P$ExcpPC(a4),-(a7)  save current values
            move.l  P$ExcpSP(a4),-(a7)
            lea     ProbeByte10(pc),a0  build recovery PC
            move.l  a0,P$ExcpPC(a4)     set recovery PC
            move.l  a7,P$ExcpSP(a4)     and stack pointer
            movea.l d0,a0               copy the address to test
            moveq   #-1,d2              default to bus error occurred
            tst.b   (a0)                test the memory location
* The next instruction is only executed if no exception occurred:
```

```
            moveq   #0,d2               no bus error
ProbeByte10 move.l  (a7)+,P$ExcpSP(a4)  restore old values
            move.l  (a7)+,P$ExcpPC(a4)
            move.l  d2,d0               copy the result
            movem.l (a7)+,d1-d2/a0      retrieve registers
* The d0 register now contains 0 if no bus error (or other hardware
* exception) occurred, otherwise it contains -1.
            rts
```

10.6 INTERRUPTS

10.6.1 How 68000 Interrupts Work

An interrupt is an external signal to the processor requesting the asynchronous execution of a subroutine, known as an interrupt handler. In general, interrupts are generated by I/O interface chips when they require servicing by the processor – for example, when a serial port interface has received a character. The 68000 family processors respond to an interrupt by an exception, allowing the interrupt handler to be executed in supervisor state, and afterwards the interrupted program to continue execution. These processors do not provide just a single interrupt input signal. Instead, they have a 3-bit binary coded input. This is generated by an external priority encoder chip, that takes 7 interrupt inputs (numbered 1 to 7), and outputs the 3-bit binary value indicating the number of the highest active input. If no input is active, the priority encoder generates a code of zero, meaning no interrupt handling is currently required.

This mechanism provides a prioritized system of seven levels of interrupts. If the input interrupt code exceeds the current interrupt mask value in the processor's status register, the processor initiates exception processing of the interrupt, with a special memory access cycle known as an interrupt acknowledge cycle. The processor (having saved the current status register, as in all exceptions), also sets the interrupt mask in the status register to equal the level of the interrupt being serviced. Thus, until the interrupt handler finishes, any other interrupt on the same or a lower level is ignored, but a higher level interrupt can cause a further exception, interrupting the interrupt handler of the lower level interrupt. Note that it requires a privileged instruction to change the interrupt mask in the status register (as with any part of the high byte of the status register word), so user state programs cannot mask interrupts.

Interrupt level 7 is a special case. Setting the interrupt mask to 7 does not prevent the processor responding to a level 7 interrupt, making such interrupts "non-maskable". Note, however, that because the processor only

starts interrupt processing if the interrupt level goes above the interrupt mask, or the interrupt mask is lowered below the current interrupt level, if the interrupt handler completes without clearing the level 7 interrupt, and the interrupt mask restored by the **rte** instruction at the end of the handler is 7, a further exception is not taken.

The processor must have some means of determining which device caused the interrupt, because most systems will have more than one device generating an interrupt. Simple processors require that software poll the status register of all devices that may be interrupting, to see which is currently generating an interrupt. However, the 68000 family processors can take distinct exceptions for different interrupt sources, by using separate interrupt exception vectors. These processors support two methods by which the interrupt vector is generated.

The first method is known as normal vectoring. The device (or some associated circuit) that is generating the interrupt responds to the interrupt acknowledge cycle by returning a vector number. The second method is known as auto-vectoring. This allows the use of devices that cannot themselves return a vector number. External circuitry detects that the interrupting device is of this type, and asserts an "auto-vector" input signal to the processor in response to the interrupt acknowledge cycle. The processor then generates the vector number internally, by adding 24 to the level of the interrupt. For example, a level 3 auto-vector interrupt generates a vector of 27.

10.6.2 Using Interrupts Under OS-9

For compatibility with all members of the 68000 family, and with most I/O devices, vector numbers are limited to 8 bits. Most devices will allow any vector number to be programmed into their interrupt vector register, to be used in response to a future interrupt acknowledge cycle. However, Motorola have reserved vector numbers 0 to 63 for other types of exception (including auto-vectored interrupts). Therefore 192 vector number are available for normal vectored interrupts, and 7 for auto-vectored interrupts, making a total of 199 interrupt vector numbers.

Most systems will use only a few of these vectors, and some will use the same vector for more than one device. This is particularly true of auto-vectored interrupts, as only 6 maskable levels are available, but it should be avoided with normal vectored interrupts. Therefore OS-9 provides a simple mechanism to allow any number of interrupt handlers to be installed on any

number of vectors, without absolutely requiring that any handlers be installed at all (the kernel has a default handler for unexpected interrupts).

In accordance with the OS-9 philosophy of dynamic configurability, interrupt handlers are installed when needed, and removed when no longer required. The **F$IRQ** privileged system call is used to install an interrupt handler. The caller passes the address of the handler, the address of the static storage to be used by the handler, a "port address", the interrupt vector number, and a software polling priority value. Usually, an interrupt handler is part of a device driver. In this case, the static storage is normally the Device Static Storage, the "port address" is normally the base address of the registers of the interface chip, and the vector number and polling priority are taken by the device driver from the device descriptor.

The kernel maintains an "interrupt polling table". This is an array of structures, initially all free, which are used to link interrupt handlers to interrupt vectors. The size of the table - which is not dynamically expandable - is determined by an entry in the **init** configuration module. The **F$IRQ** system call searches for a free entry in this table, and stores the parameters there. The kernel then uses the vector number to select one of 199 pointers in the System Globals. This pointer is the root of a linked list of polling table entries for that vector number. The kernel then searches the linked list, which is sorted by the software polling priority value - a low value means the entry is placed nearer the start of the list. It inserts the new entry after all entries with a lower or equal software polling priority. If the root pointer was null (zero), the kernel knows that the linked list for the given vector number was empty, and makes the new entry the first entry in the linked list, placing its address in the root pointer.

A software polling priority of zero is a special case. It is used to ensure that the handler is the only handler on the given vector number. If there is already a handler installed (the root pointer is not null), the caller is returned an error - **E$VctBsy**. If an **F$IRQ** system call attempts to install a handler on a vector which already has an entry of software polling priority zero, it is returned the same error. This mechanism is necessary to ensure correct support for some devices that have no status flag to indicate that they are generating an interrupt. The only way of knowing that it is this device that is interrupting is by the unique vector number returned by the interrupt acknowledge cycle.

The **F$IRQ** system call is also used to remove an interrupt handler from the polling table. The caller passes zero in place of the interrupt handler address. The kernel again uses the given vector number to identify the appropriate

root pointer. It then searches the linked list for that vector until it finds an entry whose static storage pointer matches that passed to the **F$IRQ** routine. Having found the correct entry, the kernel unlinks it from the linked list, and marks it as free for use by a subsequent **F$IRQ** call to install a new handler. This has the corollary effect that two interrupt handlers must not be installed on the same vector with the same static storage address. However, this is not a restriction, as there are almost no circumstances imaginable where a programmer would wish to do this.

Also note that two devices using different interrupt levels should not use the same interrupt vector. Otherwise, as the kernel calls each handler in the linked list for the vector in turn, an interrupt handler could be called recursively.

The interrupt polling table structure is described in more detail in the chapter on the OS-9 Internal Structure.

The kernel has a single "core" interrupt handler, and the exception jump table entries (see below) for all of the interrupt exceptions jump to this interrupt handler. The core handler uses the vector offset pushed on the stack by the jump table entry to select the appropriate root pointer for this interrupt exception number. It then calls each handler in the linked list in turn, passing the static storage and port addresses as specified in the **F$IRQ** call, until a handler returns the carry flag clear, indicating that the handler has recognized and serviced the interrupt. Finally, the kernel returns from the exception, using the **rte** instruction. Note that if the interrupt occurred while the processor was executing in user state, the kernel first performs its "return to user state" checks on the current process, such as whether the process is now marked as "timed out". This permits functions such as task switching after a clock tick interrupt, and the calling of the process's signal handler routine if the process is sent a signal by an interrupt handler.

If there is no handler for the interrupt vector (the root pointer is null), or all handlers on the vector return the carry flag set, the kernel increments the byte field **D_UnkIRQ** ("unknown interrupt request") in the System Globals, and then returns from the exception. If the interrupt persists the kernel's interrupt handler will be called again, and the count will eventually roll over to zero (after 256 attempts). When this happens the kernel masks interrupts up to the level of the offending interrupt, preventing the processor from responding to the interrupt again. The **D_UnkIRQ** field is cleared whenever any interrupt is successfully processed. This is a measure to protect against hardware glitches in the external interrupt circuitry - normally an unrecognized interrupt is fatal for any system.

Note that unless bit zero of the first compatibility byte in the **init** configuration module is set, the kernel only saves the registers **d0-d1/a0/a2-a3/a6** on interrupt, to speed up the response to the interrupt. Therefore interrupt handlers that use other registers must save and restore them. The current version of the C compiler generates code that preserves all of the data and address registers not preserved by the kernel, including any floating point unit (FPU) data registers. However, if the interrupt handler does use the FPU, it must save and restore the FPU context, as an interrupt can break into an FPU instruction:

```
IRQSvc      tst.b   D_68881(a6)     does the system have an FPU?
            beq.s   IRQSvc10        ..no
            fsave   -(a7)           save FPU context
            fmovem.l fpcr/fpsr/fpiar,-(a7)   save FPU control registers
IRQSvc10    bsr     IRQSvcMain      service the interrupt
            move    sr,d0           save carry flag
            tst.b   D_68881(a6)     does the system have an FPU?
            beq.s   IRQSvc20        ..no
            fmovem.l (a7)+,fpcr/fpsr/fpiar   restore control registers
            frestore(a7)+           restore FPU context
IRQSvc20    move    d0,sr           restore carry flag
            rts
```

An interrupt handler terminates with an **rts** instruction, not an **rte** instruction. This is because the handler is returning to the kernel's core interrupt handler, which itself executes the **rte** instruction to finish the exception processing.

During interrupt processing the kernel switches to a different stack - the interrupt stack - to avoid the need for stack to be reserved for interrupt processing in the system state stack of every process descriptor. The size of the interrupt stack (which is not dynamically expandable) is specified in the **init** configuration module. The kernel's interrupt handler saves the system stack pointer, and then increments the **D_IRQFlag** field of the System Globals. This field is initialized to -1 during the kernel's coldstart. If it is now zero, the kernel knows that this interrupt is not breaking into the service of another interrupt (which will have already switched to the interrupt stack), so it switches to the interrupt stack by loading the **a7** register from the **D_SysStk** field of the System Globals. At the end of the interrupt service the kernel decrements the **D_IRQFlag** field, and restores the original stack pointer.

Because an unrecognized (and therefore unserviced) interrupt is potentially fatal for the system, interrupt handlers must not be subroutines in modules that can be unexpectedly terminated (such as trap handlers, and user state programs), and must not use static storage that can be unexpectedly

de-allocated (such as program or trap handler static storage). Therefore only operating system components should contain and install interrupt handlers. The use of interrupts within device drivers is explained in the section on "Device Drivers".

10.6.3 Interrupts OS-9 Cannot Handle

Because OS-9 only maintains root pointers for the 199 normally expected interrupt exception vectors, there are two types of interrupt exception that OS-9 cannot handle. The first is the case in which a device returns an interrupt vector (in response to an interrupt acknowledge cycle) that is not in the range 64 to 255. This should never happen. It indicates that there is a hardware fault, or that the device has been programmed with an improper vector by the device driver. If the vector corresponds to the vector for a different type of exception (such as an illegal instruction), the kernel will act as if that exception had occurred - it has no way of knowing that in fact an interrupt generated the exception.

If the exception occurred in user state and the vector does not match any known exception vector, the kernel kills the current process, giving it an exit status of 100 plus the exception vector number. If the exception occurred in system state the kernel treats it like a normal "hardware" exception in system state (see above).

This generation of an invalid vector may happen for certain devices that, on reset, set their interrupt exception vector register to a value of 15 - the 68000 "uninitialized interrupt" vector. If such a chip is then programmed to generate an interrupt without first writing a valid vector number to its interrupt exception vector register, OS-9 will be unable to handle the interrupt.

The second type of interrupt that OS-9 cannot handle is the Spurious Interrupt exception (vector 24). This exception is taken by the processor if the Bus Error input signal is asserted in response to the interrupt acknowledge cycle. Usually this is because no device has responded to the interrupt acknowledge cycle, so the memory access timeout circuit on the processor board asserts the Bus Error signal. This happens on VME systems and other bus based systems that use a daisy-chained interrupt acknowledge signals if the backplane jumper that by-passes the daisy chain for a particular backplane slot is left out when the slot is empty. If a board further down the backplane generates an interrupt, it will not receive the interrupt acknowledge signal from the processor board (because the daisy chain is broken by the empty slot and the missing jumper), and so will not respond.

The timeout circuit on the processor board eventually times out, and asserts the Bus Error signal.

Although OS-9 has a root pointer in the System Globals corresponding to the Spurious Interrupt exception vector (which is equivalent to an auto-vector of level zero - 24), the **F$IRQ** system call will not permit a handler to be installed on this vector, and the exception jump table entry for this exception jumps to the kernel's "hardware" exception handler. The kernel handles the exception as described above for invalid interrupt vectors.

However, the interrupt from the interrupting device has not been acknowledged or serviced, so the interrupt signal remains asserted, and a further exception is taken once the interrupt mask is cleared (either explicitly by the kernel in its handler for hardware exceptions in system state, or implicitly by the **rte** "return from exception" instruction). Therefore the system will "hang", or - if the interrupt occurred (at a higher level) while another interrupt was being serviced - the kernel will give up through a controlled system crash. In the latter case the exception is recognizable because the system state exception message reports the exception vector offset for a Spurious Interrupt exception - $0060.

10.6.4 The Level 7 Interrupt

As mentioned above, a level 7 interrupt is non-maskable. Therefore it should never be used for normal interrupt handling, as the main body of the module (such as a device driver) initiating the interrupt cannot mask interrupts while manipulating variables or device registers also used by the interrupt handler.

Similarly, a level 7 interrupt handler must not make any system calls, as the kernel cannot protect its data structures by masking the interrupt.

If one of the Microware ROM-based debuggers is installed, the vector for auto-vector level 7 in the exception vector table is set to point to a handler in the debugger, rather than to the kernel's core interrupt handler. Many processor boards provide a front panel "abort" switch that generates a level 7 auto-vector, so this facility can be used to call the ROM-based debugger in the event that the system "hangs", in order to try to determine why the "hang" occurred. Therefore, if the level 7 auto-vector is to be used for some other purpose (such as processor emulation of DMA), the exception vector table entry for auto-vector level 7 must be overwritten, unless it is certain that no ROM-based debugger will be used.

10.7 THE EXCEPTION VECTOR TABLE

The processor selects which handler routine to call on exception by using the exception number as an index into a table of addresses, known as the exception vector table. Because there are 256 possible vector numbers, the table is 256 long words in length - that is, 1k bytes. The first entry (corresponding to vector zero) is reserved for the address to load into the stack pointer on reset, and is used in OS-9 to point to the System Globals. The 68000 processor always locates the exception vector table at address zero, while the 68020/030/040 have a vector base register (**vbr**) that gives the address of the table. The **vbr** is set to zero when the processor is reset, but may be re-programmed by the bootstrap ROM (the OS-9 kernel does not write to this register).

The reset vector and reset stack pointer values must be in ROM, so that they are present on power-on. However, it is also convenient to have the exception vector table in RAM, so that it can be dynamically modified. Processor boards address this dichotomy in a number of different ways:

a) The exception vector table is in ROM, at address zero, and cannot be modified. It is part of the bootstrap ROM. The system RAM starts at some other address.

b) The system RAM is mapped to start at address zero, but the first two long words are overlaid by a reflection of the first two long words of the ROM. The reset vector and reset stack pointer are fixed, but the other vectors are dynamically modifiable.

c) The system RAM is mapped to start at address zero, but is overlaid by the ROM on reset. Either the termination of the reset cycle, or an explicit processor board register write, causes the ROM reflection to disappear. The vectors are then dynamically modifiable.

d) The ROM is mapped to start at address zero, and contains the reset vector and reset stack pointer. However, the processor contains a vector base register, so the exception vector table can be relocated to any part of the system RAM.

If the exception vector table is in ROM only, it forms the first part of the bootstrap ROM, and cannot be modified. Otherwise, the bootstrap program builds the exception vector table in RAM. Each entry in the exception vector table (except the reset stack pointer and reset vector) points to the corresponding entry in the exception jump table (see below). However, as

described above, if the bootstrap ROM contains one of the ROM-based debuggers, then the bus error, address error, illegal instruction, and auto-vector level 7 interrupt exception vectors point to handlers in the debugger. Because the kernel must support all types of system, it does not attempt to modify the exception vector table. Instead, it assumes that the table entries point to the appropriate entries in the exception jump table, and writes the addresses of its handlers in the exception jump table.

If it is known that the exception vector table is in RAM, a vector may be overwritten to cause the exception to call a user-installed handler directly. This by-passes all the kernel mechanisms (such as the use of the interrupt stack) and protections, and so is not recommended by Microware, but gives a slightly faster interrupt response that may be necessary in some critical applications.

10.8 THE EXCEPTION JUMP TABLE

The exception jump table is needed for two reasons. Firstly, the 68000 does not save a record of which exception is being serviced. Although the higher members of the family do (the exception processing pushes the exception number on the stack), OS-9 must be compatible with all members of the family. Secondly, the exception vector table may be in ROM, so the vector addresses cannot be modified, yet the bootstrap ROM cannot know the addresses of the kernel's exception handlers. Therefore each exception vector points to its own entry in the exception jump table. Each entry in the exception jump table consists of an instruction to push the exception vector offset onto the stack, followed by an absolute jump instruction, which jumps to the exception handler. The first two entries in the exception vector table (reset stack pointer, and reset vector) do not need corresponding exception jump table entries, so the first entry is for exception number 2 (bus error). Therefore if the exception jump table is disassembled, the first few instructions look like this:

```
pea     $0008.w
jmp     $xxxxxxxx.l
pea     $000C.w
jmp     $yyyyyyyy.l
pea     $0010.w
jmp     $zzzzzzzz.l
```

The kernel builds the exception jump table as part of its coldstart routine. The entries in the table are each 10 bytes, so the table is 2540 bytes in size. Its start address is usually 4k below the address of the System Globals, but to

allow for variations in the size of this memory area the kernel writes the jump table base address to the **D_ExcJmp** field of the System Globals[12].

The exception jump table is always in RAM, so an operating system component that wishes to permanently or temporarily redirect an exception to its own handler can overwrite the jump address (the last 4 bytes) in the corresponding jump table entry. If the change is to be temporary, the routine should save the current address, set its own address, perform the function that might generate the exception, and then replace the original address of the kernel's handler. This strategy also allows this method of redirection to be nested. For example, a high level interrupt handler could temporarily redirect the bus error vector, even though this interrupted a similar attempt by a low level interrupt handler. The example below temporarily replaces the bus error handler while accessing an I/O device register:

```
            move.l  D_ExcJmp(a6),a0     get address of jump table
* Each entry is 10 bytes, and the jump address is the last long word
* of the entry. The first entry is for exception 2. Bus error is
* exception 2:
            adda.w  (2-2)*10+6,a0   point at jump address for bus error
            move.l  (a0),-(a7)      save the jump address
            lea     BusError(pc),a1 point at our handler
            move.l  a1,(a0)         set the new jump address
            move.l  a7,d1           save the stack pointer
            moveq   #-1,d0          default to "bus error occurred"
            move.w  (a3),d2         the instruction that may cause a
*                                   bus error
* The next instruction is only executed if no bus error occurred:
            moveq   #0,d0           show "no bus error"
BusError    movea.l d1,a7           restore the stack pointer
            move.l  (a7)+,(a0)      restore the original jump address
            tst.l   d0              did a bus error occur?
```

As with overwriting the exception vector table, this mechanism should only be used when strictly necessary, but it is available when needed.

12 The 'CBOOT' and **ROMBug** options for the boot program from OS-9 version 2.4 onwards may allocate static storage that increases this memory area above 4k bytes.