Exception handling in the 68000, Part 1

Motorola's 68000 is an example of a microprocessor with sophisticated exception handling facilities. In the first of two tutorial papers, **Alan Clements** presents an overview of the 68000's exceptions and interrupts

The paper gives an overview of the implementation of exception handling in the 68000 microprocessor, starting with an introductory discussion of interrupts. The different types of interrupt are outlined. This is followed by a discussion of privileged states on the 68000. The types of exception supported by the 68000 are described, as are the use and maintenance of the exception vector table. Finally, the response of the 68000 CPU to an exception is covered.

microprocessors exceptions interrupts 68000

This two-part paper examines how the 68000 16-bit microprocessor implements exception handling.

Most of the first- and second-generation 8-bit microprocessors had rather primitive exception handling facilities consisting of little more than one or two interrupt request inputs and some form of software interrupt. Modern high-performance microprocessors have very sophisticated exception handling mechanisms and can deal with multilevel, prioritized, vectored interrupts together with a wide range of software traps and operating system calls. The paper begins with a brief look at interrupts, which are a special case of the more general exception processing capability of a computer. Part 2 of the paper shows in detail how the 68000 handles interrupts.

INTERRUPTS

A computer executes the instructions of a program sequentially unless a jump or conditional branch modifies their order, or unless a subroutine is called. In such cases, any deviation from the sequential execution of instructions is determined by the programmer. Deviations caused by conditional branches or subroutines are said to be synchronous, because they occur at predetermined points in the program. Under certain circumstances this arrangement is very inefficient.

Suppose a microprocessor is reading data from a keyboard at an average rate of 250 characters per minute, corresponding to approximately four characters per second. In a 68000 system, the processor reads the status of a memory-mapped peripheral to determine whether or not a key has been pressed. If no key has been pressed, a branch is made back to the instruction which reads the status of the peripheral, and the cycle continues until a key is pressed. The following program shows how this is done.

```
KEY__STATUS  EQU      $F00000           (Location of input status
                                         word)
KEY__VALUE   EQU      KEY__STATUS + 2   (Location of input data
                                         word)

             LEA.L    KEY__STATUS, A0   (A0 points to Key__status)
             LEA.L    KEY__VALUE, A1    (A1 points to Key__value)
TEST__LOOP   BTST.B   #0, (A0)          (Test status, ie least
                                         significant bit)
             BEQ      TEST__LOOP        (Repeat while least
                                         significant bit clear)
             MOVE.B (A1), D1            (Read the data)
```

The two instructions BTST.B #0, (A0) and BEQ TEST__LOOP constitute a 'polling loop', which is executed until the least significant bit of the status word is true, signifying that the data from the keyboard is valid. These two instructions take 20 clock cycles to execute on the 68000, requiring 2 μs with a 10 MHz clock. Thus, for each key pressed, the polling loop is executed approximately 100 000 times! Quite clearly, this is a grossly inefficient use of CPU time.

In some applications, the time wasted in executing a polling loop is of little significance. If an operator is sitting at the keyboard of a personal computer thinking about the next word to enter, for example, it is of no consequence that the CPU is asking the keyboard if it has a new character every 2 μs or so. In more sophisticated applications, the CPU cannot be allowed to waste time executing polling loops. There may be a queue of programs waiting to be run or some peripheral needing continual attention while another program is being run, or there may be a background task and a foreground task.

A technique for dealing more effectively with I/O transactions has been implemented on all microprocessors: it is called an interrupt handling mechanism. An interrupt request line, IRQ, is connected between the peripheral and the CPU. Whenever the peripheral is ready to take part in an I/O operation, it asserts the IRQ line and invites the CPU to deal with the transaction. The CPU is free to carry out background tasks between interrupt requests from the peripheral.

An interrupt is clearly an asynchronous event, because the processor cannot know at which instant a peripheral such as a keyboard will generate an interrupt. In other words, the activity generating the interrupt bears no particular timing relationship to the activity that the computer is carrying out between interrupts. When an interrupt occurs, the computer first decides whether to deal with it (ie to service it), or whether to ignore it for the time being. If the computer is doing something which

Department of Computer Science, Teesside Polytechnic, Middlesbrough, Cleveland TS1 3BA, UK

0141-9331/86/04202-09 $03.00 © 1986 Butterworth & Co. (Publishers) Ltd

must be completed within a given time, it ignores interrupts. Should the computer decide to respond to the interrupt, it must carry out the following sequence of actions.

(1) Complete its current instruction. All instructions are indivisible, which means they must be executed to completion. A more sophisticated architecture might allow the temporary suspension of an instruction.

(2) The contents of the program counter must be saved in a safe place, so that the program can continue from the point at which it was interrupted after the interrupt has been serviced. The program counter is invariably saved on the stack so that interrupts can, themselves, be interrupted without losing their return addresses.

(3) The state of the processor is saved on the stack. Clearly, it would be unwise to allow the interrupt service routine to modify, say, the value of the carry flag, so that an interrupt occurring before a BCC instruction would affect the operation of the BCC after the interrupt had been serviced. In general, the servicing of an interrupt should have no effect whatsoever on the execution of the interrupted program. (This statement is qualified below in dealing with software interrupts, which are a special type of synchronous event.)

(4) A jump is then made to the location of the interrupt handling routine, which is executed like any other program. After this routine has been executed, a return from interrupt is made, the program counter restored, and the system status word returned to its pre-interrupt value.

Before examining the way in which the 68000 deals with interrupts, it is worthwhile considering some of the key concepts emerging from any discussion of interrupts.

Nonmaskable interrupts

An interrupt request is so called because it is a request, and therefore carries the implication that it may be denied or deferred. Whenever an interrupt request is deferred, it is said to be masked. Sometimes it is necessary for the computer to respond to an interrupt no matter what it is doing. Most microprocessors have a special interrupt request input called a 'nonmaskable interrupt input' (NMI). Such an interrupt cannot be deferred and must always be serviced.

Nonmaskable interrupts are normally reserved for events such as loss of power. In this case, a low voltage detector generates a nonmaskable interrupt as soon as the power begins to decay. This forces the processor to deal with the interrupt and perform an orderly shutdown of the system before the power drops below a critical level and the computer fails completely. The 68000 has a single (level 7) nonmaskable interrupt request.

Prioritized interrupts

In an environment where more than one device is able to issue an interrupt request, it is necessary to provide a mechanism to distinguish between an important interrupt and a less important one. For example, if a disc drive controller generates an interrupt because it has some data ready to be read by the processor, the interrupt must be serviced before the data is lost and replaced by new data from the disc drive. On the other hand, an interrupt generated by a keyboard interface probably has from 250 ms to several seconds before it must be serviced. Therefore an interrupt from a keyboard can be deferred if interrupts from devices requiring urgent attention are pending.

For the above reasons, microprocessors are often provided with prioritized interrupts. Each interrupt has a predefined priority, and a new interrupt with a priority lower than or equal to the current one cannot interrupt the processor until the current interrupt has been dealt with. Equally, an interrupt with a higher priority can interrupt the current interrupt. The 68000 provides seven levels of interrupt priority.

Vectored interrupts

A vectored interrupt is one in which the device requesting the interrupt automatically identifies itself to the processor. Some 8-bit microprocessors lack a vectored interrupt facility and have only a single interrupt request input (IRQ*). When IRQ* is asserted, the processor recognizes an interrupt but not its source. This means that the processor must examine, in turn, each of the peripherals that may have initiated the interrupt. To do this, the interrupt handling routine interrogates a status bit associated with each of the peripherals.

More sophisticated processors have an interrupt acknowledge output line, IACK, which is connected to all peripherals. Whenever the CPU has accepted an interrupt and is about to service it, the CPU asserts its interrupt acknowledge output. An interrupt acknowledge from the CPU informs the peripheral that its interrupt is about to be serviced. The peripheral then generates an 'identification number' which it puts on the data bus, allowing the processor to calculate the address of the interrupt handling routine appropriate to the peripheral. This is called a vectored interrupt. The 68000 provides the designer with both vectored and nonvectored interrupt facilities.

PRIVILEGED STATES AND THE 68000

Having introduced the interrupt the next step is to look at how the 68000 handles exceptions, which are a more general form of interrupt.

The 68000 is an unusual processor because it always operates in one of two states: either supervisor state or user state. User and supervisor states are only relevant to multitasking systems in which several user tasks are run under control of the operating system. By executing the operating system in the supervisor mode and the user tasks in the user mode, it becomes relatively easy to prevent one user task from accessing the memory space of another task or of the operating system.

The supervisor state is the higher state of privilege and is in force whenever the S bit of the status register is true. All the 68000's instructions can be executed while the processor is in this state. The user state is the lower state of privilege, and certain instructions cannot be executed in this state. Each of the two states has its own stack pointer, so that the 68000 has two A7 registers. The user-mode A7 is called the user stack pointer (USP) and the supervisor-mode A7 is called the supervisor stack pointer (SSP). Note that the SSP cannot be accessed from the user state, whereas the USP can be accessed in the supervisor state by means of the instructions MOVE USP,An and MOVE An,USP. Figure 1 shows the register arrangements of the 68000. The program counter, status register, data registers and address registers A0–A6 are common to both operating modes. Only A7 is duplicated.

All exception processing is carried out in the supervisor state, because an exception forces a change from user to supervisor state. Indeed, the only way of entering the supervisor state is by means of an exception. Figure 1 shows how a transition is made between the 68000's two states. Note that an exception causes the S bit in the 68000's status register to be set and the supervisor stack pointer to be selected at the start of the exception, with the result that the return address is saved on the supervisor stack and not on the user stack. The 68000 puts out a function code on its three function code pins, FC0–FC2, which informs any external memory management unit whether the CPU is accessing user or supervisory memory space. This enables the memory management unit to protect the supervisor's memory space from any illegal access by user programs.

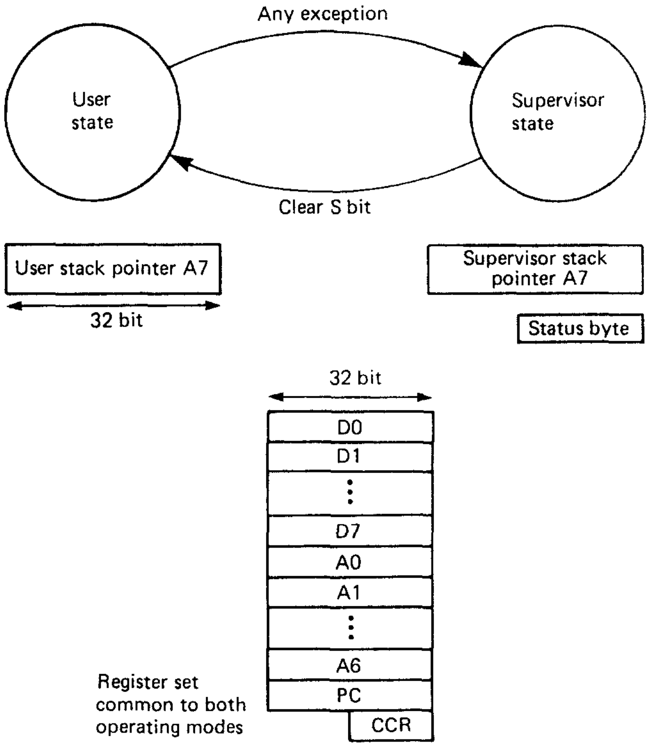


Figure 1. State diagram of user and supervisor state transitions

The change from supervisor to user state is made by clearing the S bit of the status register. This change is carried out by the operating system when it wishes to run a user program. Four instructions are available for this operation: RTE; MOVE.W < ea >, SR; ANDI.W #$XXXX, SR; and EORI.W #$XXXX, SR. The #$XXXX represents a 16-bit literal value in hexadecimal form. The RTE (return from exception) instruction terminates an exception handling routine and restores the value of the program counter and the old status register stored on the stack before the current exception was processed. Consequently, if the 68000 were in the user state before the current exception forced it into the supervisor state, an RTE would restore the processor to its old (ie user) state.

In the user state, the programmer must not attempt to execute certain instructions. For example, the STOP and RESET instructions are not available to the programmer, because the RESET instruction forces the RESET* output of the 68000 low and resets any peripherals connected to this pin. Some of these peripherals may be in use by another program. The whole philosophy behind user and supervisor states is to prevent this type of thing from happening. Similarly, a STOP instruction has the effect of halting the processor until certain conditions are met, and is not allowed in the user state because a user program should not be allowed to bring the entire system to a standstill.

Any instructions affecting the S bit in the upper byte of the status register (ie RTE; MOVE.W < ea>,SR; ANDI.W #$XXXX,SR; ORI.W #$XXXX,SR; EORI.W #$XXXX,SR) are also not permitted in the user mode. Note that no instruction that performs useful computation is barred from the user mode. Only certain system operations are privileged.

Suppose a programmer tries to set the S bit by executing an ORI.W #$2000,SR; obviously, there is nothing to stop him/her from writing the instruction and running the program containing it. When the program is run, the illegal operation violates the user privilege by trying to enter the supervisor mode and forces an exception to be generated. This causes a change of state from user to supervisor, ie the 'punishment' for trying to enter the supervisor state is to be forced into it.

In fact, the effect of attempting to execute an ORI.W #$2000,SR is to raise a software exception called a privilege violation, forcing a jump to a specific routine dealing with this type of exception. (The way in which exception states are entered and processed is dealt with below.) Once the exception handling routine dealing with the privilege violation has been entered, the user no longer controls the processor. The operating system has now taken over and it is highly probable that the exception handling routine will deal with the privilege violation by terminating the user's program.

EXCEPTION TYPES

There are a number of different exception types supported by the 68000, some of which are associated with external hardware events such as interrupts, and some of which are associated with internally generated events such as privilege violations. Below is a list of the exception types currently implemented. The way in which these exceptions are implemented is described later.

Reset

An externally generated reset is caused by bringing the RESET* and HALT* pins low for 10 clock pulses (or 100 ms on power up), and is used to place the 68000 in a known state at start up or following a totally irrecoverable system collapse. The reset is a unique exception, because there is no 'return from exception' following a reset.

Bus error

A bus error is an externally generated exception, initiated by hardware driving the 68000's BERR* pin active low. It is a 'catch-all' exception, because the systems designer may use it in many different ways, and it is provided to enable the processor to deal with hardware faults in the system. A typical use of the BERR* input is to indicate either a faulty memory access or an access to a nonexistent memory.

Interrupt

The 68000 has three interrupt request inputs, IPL0-IPL2, which are encoded and indicate one of seven levels of interrupt. To obtain maximum benefit from the interrupt request inputs, it is necessary to apply an eight-line to three-line priority encoder to convert one of seven interrupt request inputs from peripherals into a 3-bit code. The eighth code represents no interrupt request.

Address error

An address error exception occurs when the processor attempts to read a 16-bit word or a 32-bit longword at an odd address. Attempting to read a word at an odd address would require two accesses to memory — one to access the odd byte of an operand and the other to access the even byte at the next address. Address error exceptions are generated when the programmer makes a mistake. Consider the following fragment of code

```
MOVEA.L #$7000,A0   (Load A0 with $00 7000)
MOVE.B (A0) +,D0    (Load D0 with the byte pointed at by A0, and
                    increment A0 by 1)
MOVE.W (A0) +,D0    (Load D0 with the word pointed at by A0, and
                    increment A0 by 2)
```

The third instruction results in an address error because the previous operation, MOVE.B (A0) + D0, causes the value in A0 to be incremented from $7000 to $7001. Therefore when the processor attempts to execute MOVE.W (A0) +, D0 it finds it is trying to access a word at an odd address. In many ways, an address error is closer to an exception generated by an event originating in the hardware than to one originating in the software. The bus cycle that leads to the address error is aborted, as the processor cannot complete the operation.

Illegal instruction

In 8-bit microprocessors, it was an intriguing diversion to find out what effect 'unimplemented' op. codes had on the processor. For example, if the value $A5 did not correspond to a valid op. code, an enthusiast would try and execute it and then see what happened. This was possible because the control unit (ie the instruction interpreter) of most 8-bit microprocessors was implemented by random logic.

To reduce the number of gates in the control unit of the CPU, some manufacturers have not attempted to deal with illegal op. codes; after all, these are not supposed to be executed. In keeping with the 68000's approach to programming, an exception is generated whenever an operation code is read that does correspond to the bit pattern of the first word of one of the 68000's legal instructions.

Divide by zero

If a number is divided by zero, the result is meaningless and often indicates that something has gone seriously wrong with the program attempting to carry out the division. For this reason, the designers of the 68000 decided to make any attempt to divide a number by zero an exception generating event. Good programs never try to divide a number by zero, so the divide-by-zero exception should not arise; it is intended merely as a failsafe device, to avoid the meaningless result that would occur if a number was divided by zero.

CHK instruction

The 'check register against bounds' instruction (CHK) has the assembly language form CHK < ea >,Dn, and has the effect of comparing the content of the specified data register with the operand at the effective address. If the lower-order word in the register, Dn, is negative, or is greater than the upper bound at the effective address, an exception is generated. For example, when the instruction CHK D1,D0 is executed, an exception is generated if

$$[D0(0{:}\ 15)] < 0$$

or

$$[D0(0{:}\ 15)] > [D1(0{:}15)]$$

The CHK instruction works only with 16-bit words, and therefore cannot be used with an address register as an effective address. The CHK exception has been included to help compiler writers for languages such as PASCAL which have facilities for the automatic checking of array indexes against their bounds.

TRAPV instruction

When the 'trap on overflow' instruction (TRAPV) is executed, an exception occurs if the overflow bit, V, of the condition code register is set. Note that an exception caused by dividing a number by zero occurs automatically, while TRAPV is an instruction equivalent to: IF V = 1 THEN exception ELSE continue.

Privilege violation

If the processor is in the user state (ie the S bit of the status register is clear) and it attempts to execute a privileged instruction, a privilege violation exception occurs. Apart from any instruction that attempts to modify the state of the S bit, the following three instructions cannot be executed in the user state: STOP; RESET; MOVE < ea >,SR.

Trace

A popular method of debugging a program is to operate in a trace mode, in which the contents of all registers are printed out after each instruction has been executed. The 68000 has an inbuilt trace facility. If the T bit of the status register is set, a trace exception is generated after each instruction has been executed. The exception handling routine called by the trace exception can be constructed to offer programmers any facilities they need.

Line 1010 emulator

Operation codes whose four most significant bits (bits 12–15) are 1010 or 1111 are unimplemented in the 68000, and therefore represent illegal instructions. However, the 68000 generates a special exception for op. codes whose most significant nibble is 1010 (also called line ten). The purpose of this exception is to emulate instructions on future versions of the 68000. Suppose a version of the 68000 is designed which includes floating point operations as well as the normal 68000 instruction set. Clearly, it is impossible to run code intended for the floating point processor on a normal 68000. But by using 1010 as the four most significant bits of each of the new floating point instructions, an exception is generated each time the 68000 encounters one, and the line 1010 exception can emulate its more sophisticated counterpart.

Line 1111 emulator

The line 1111 (or line F) emulator behaves in almost exactly the same way as the line 1010 emulator, except that it has a different exception handling routine.

Uninitialized interrupt vector

The 68000 supports vectored interrupts, so that an interrupting device can identify itself and allow the 68000 to execute the appropriate interrupt handling routine without having to poll each device in turn. Before a device can identify itself, it must first be correctly configured by the programmer. If a 68000 series device is unconfigured and yet generates an interrupt, the 68000 responds by raising an 'uninitiated interrupt vector' exception. 68000 series peripherals are designed to supply the initialized interrupt vector number ($0F) during an IACK cycle, if they have not been initialized by software.

Spurious interrupt

If the 68000 receives an interrupt request and sends an interrupt acknowledge, but no device responds, the CPU generates a spurious interrupt exception. To implement the spurious interrupt exception, external hardware is required to assert BERR* following the nonappearance of either DTACK* or VPA* a reasonable time after an interrupt acknowledge has been detected.

TRAP (software interrupt)

The 68000 provides sixteen instructions of the form TRAP #1, where *I* = 0, 1, . . ., 15. When this instruction is executed an exception is generated and one of sixteen exception handling routines called. Thus TRAP #0 causes TRAP exception handling routine 0 to be called, and so on.

The TRAP instruction is very useful. Suppose a program is written which is to run all 68000 systems. The greatest problem comes in dealing with input or output transactions. One 68000 system may deal with input in a very different way to every other 68000 system. However, if everybody agrees that, say TRAP #0 means input a byte and TRAP #1 means output a byte, then the software becomes truly portable. All that remains to be done is for an exception handler to be written for each 68000 system to actually implement the input or output as necessary.

EXCEPTION VECTORS

Having described the various types of exception supported by the 68000, the next step is to explain how the processor is able to determine the location of the corresponding exception handling routine. Every exception has a vector associated with it, and that vector is the 32-bit absolute address of the appropriate exception handling routine. All exception vectors are stored in a table of 512 words, extending from address $00 0000 to $00 03FF.

A list of all the exception vectors is given in Table 1, and Figure 2 shows the physical location of the 512 vectors in memory. The left-hand column of Table 1 gives the vector number of each entry in the table. The vector number is a value which, when multiplied by four, gives the address, or offset, of an exception vector. For example, the vector number corresponding to a privilege violation is 8, and the

Table 1. Exception vectors and the 68000

Vector number	Vector (hex)	Address space	Exception type
0	000	SP	Reset — initial supervisor stack pointer
1	004	SP	Reset — initial program counter value
2	008	SD	Bus error
3	00C	SD	Address error
4	010	SD	Illegal instruction
5	014	SD	Divide by zero
6	018	SD	CHK instruction
7	01C	SD	TRAPV instruction
8	020	SD	Privilege violation
9	024	SD	Trace
10	028	SD	Line 1010 emulator
11	02C	SD	Line 1111 emulator
12	030	SD	(Unassigned — reserved)
13	034	SD	(Unassigned — reserved)
14	038	SD	(Unassigned — reserved)
15	03C	SD	Uninitialized interrupt vector
16	040	SD	(Unassigned — reserved)
:	:	:	:
23	05C	SD	(Unassigned — reserved)
24	060	SD	Spurious interrupt
25	064	SD	Level 1 interrupt autovector
26	068	SD	Level 2 interrupt autovector
27	06C	SD	Level 3 interrupt autovector
28	070	SD	Level 4 interrupt autovector
29	074	SD	Level 5 interrupt autovector
30	078	SD	Level 6 interrupt autovector
31	07C	SD	Level 7 interrupt autovector
32	080	SD	TRAP #0 vector
33	084	SD	TRAP #1 vector
:	:	:	:
47	0BC	SD	TRAP #15 vector
48	0C0	SD	(Unassigned — reserved)
:	:	:	:
63	0FC	SD	(Unassigned — reserved)
64	100	SD	User interrupt vector
:	:	:	:
255	3FC	SD	User interrupt vector

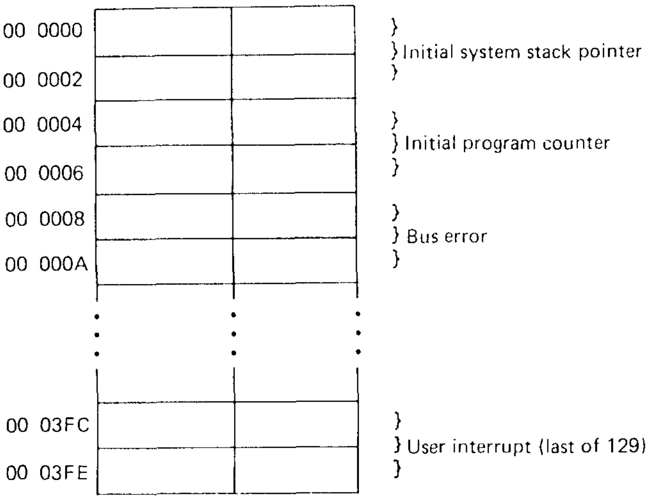

00 0000
00 0002
00 0004
00 0006
00 0008
00 000A
00 03FC
00 03FE
} Initial system stack pointer
} Initial program counter
} Bus error
} User interrupt (last of 129)

Figure 2. Memory map of the 68000 vector table

appropriate exception vector is to be found at memory location 8 × 4 = 32 = $20. Therefore, whenever a privilege violation occurs, the CPU reads the longword at location $20 and loads it into its program counter.

Although there are normally two words of memory space devoted to each 32-bit exception vector, as stated above, the reset exception (vector number 0) is a special case. The 32-bit longword at address $00 0000 is not the address of the reset handling routine, but the initial value of the supervisor stack pointer. The actual reset exception vector is at address $00 0004. Thus the reset exception takes four words of memory instead of the usual two.

The first operation performed by the 68000 following a reset is to load the system stack pointer. This is important because, until a stack is defined, the 68000 cannot deal with any other type of exception. Once the stack pointer has been set up, the reset exception vector is loaded into the program counter and processing continues normally. The reset exception vector is, of course, the initial (or 'cold start') entry point into the operating system.

There is yet another difference between the reset vector and all other exception vectors. The reset exception vector and the intial value of the supervisor stack pointer both lie in the supervisor program space, denoted by SP in Table 1. All other exception vectors lie in supervisor data space.

USING THE EXCEPTION TABLE

In any 68000 system, an exception vector table must be maintained in memory. Although the complete table is 512 words (1024 byte) long, it is not necessary to fill it entirely with exception vectors. For example, if the system does not implement vectored interrupts, the memory space from $00 0100 to $00 03FF does not need to be populated with user interrupt vectors. It is a good idea to reserve the memory space $00 0000 to $00 03FF for the exception vector table unless forced to do otherwise, even if the whole of the table is not being used. All unused vectors can be preset to the spurious exception handling routine vector. This approach is in line with the philosophy of always providing a recovery mechanism for events which could possibly happen, and which would cause the system to crash if not adequately catered for.

The 8-bit microprocessor also has its own vector table; these contain relatively few vectors corresponding to the limited exception handling facilities of most 8-bit devices. The table is invariably maintained in the same ROM that holds the processor's operating system or monitor. An advantage of putting exception vectors in ROM is that the table is always there immediately after power up, but a disadvantage is its inflexibility. Once a table is in ROM, the vectors cannot be modified to suit changing conditions. In practice, whenever a vector has to be variable, 8-bit processors use a vector pointing to another table in read/write memory containing the actual exception handling vector.

As the 68000 is so much more sophisticated than 8-bit devices and a dynamic or flexible response is sometimes required for the treatment of exception handling routines, the exception vector table is frequently held in read/write memory rather than in ROM. The operating system, held either in ROM or loaded from disc, sets up the exception vector table early in the initialization process following a reset.

A problem here is the reset vector. The two things that *must* be in ROM are the reset vector and the system monitor or bootstrap loader. Clearly, when the system is powered up and the RESET* input asserted, the reset exception vector and supervisor stack pointer, loaded from $00 0004 and $00 0000 respectively, must be in ROM. At first sight, it might be thought that it is necessary to place the whole exception vector table in ROM, as it is not possible to get a four-word ROM just for the reset vector, and a (512 – 4)-word read/write memory for the rest of the table. Hardware designers have solved the problem by locating the exception vector table in read/write memory and overlaying this with ROM whenever an access in the range $00 0000 to $00 0007 is made.

Consider a situation in which 4 kbyte of memory in the range $00 0000–$00 0FFF are provided by read/write memory. As explained later, the region of RAM at $00 0000–$00 0007 cannot be accessed by the processor. Read/write memory extending from $00 0008 to $00 03FF holds the exception vector table, which is loaded with vectors as dictated by the operating system. The remaining read/write memory from $00 0400 to $00 0FFF, is not restricted in use, and is freely available to the user or the operating system.

The 4 kbyte of memory space from $00 1000 to $00 1FFF is populated by ROM. The first 8 byte of this ROM, from $00 1000 to $00 1007, contains the reset vectors. It is necessary to arrange the hardware of the 68000 system so that a read access to the reset vectors automatically fetches them from the ROM rather than the read/write memory. One way of achieving this is demonstrated by the circuit in Figure 3. The read/write memory and ROM elements are supplied by conventional 2k × 8 bit chips, and their circuitry is entirely straightforward. Read/write memory is selected when CSRAM* is active low, and ROM when CSROM* is active low. Whenever an access is made to the region $00 0000–$00 0007, the output of IC6 (RVEC*) goes active low. The effect of this is to deselect the read/write memory and to select the ROM containing the reset vectors.

EXCEPTION PROCESSING

The 68000 responds to an exception in four identifiable phases.

In phase one, the processor makes a temporary internal copy of the status register and modifies the current status register ready for exception processing. This involves setting the S bit and clearing the T bit (ie trace bit). The S bit is set because all exception processing takes place in the supervisor mode. The T bit is cleared because it is undesirable to have the processor in the trace mode during exception processing. (The trace mode forces an exception after the execution of each instruction. If the T bit were set, an instruction would trigger a trace exception which would, in turn, cause a trace exception after the first instruction of the trace handling routine had been executed. In this way an infinite series of exceptions would be generated.) Two specific types of exception have a further effect on the contents of the status word. After a reset, the interrupt mask bits are automatically set to indicate an interrupt priority of level 7. An interrupt causes the interrupt priority to be set to the same value as the interrupt currently being processed.

In phase two, the vector number corresponding to the exception being processed is determined. Apart from interrupts, the vector number is generated internally by the 68000 according to the exception type. If the exception is an interrupt, the interrupting device places the vector number on data lines D00–D07 of the processor data bus during the interrupt acknowledge cycle, signified by a function code (FC2, FC1, FC0) of 1, 1, 1. Once the processor has determined the vector number, it multiplies it by four to extract the location of the exception processing routine within the exception vector table.

In phase three, the current 'CPU context' is saved on the system stack. The CPU context is all the information required by the CPU to return to normal processing after an exception. Phase three of the exception processing is complicated by the fact that the 68000 divides exceptions into two categories, and saves different amounts of information according to the nature of the exception. The information saved by the 68000 is called the 'most volatile portion of the current processor context', and is saved in a data structure called the exception stack frame.

Figure 4 shows the structure of the exception stack frame. Note that exceptions are classified into three groups. The information saved during group 1 or group 2 exceptions is only the program counter (two words) and the status register, temporarily saved during phase one. This is the minimum of information required by the processor to restore itself to the state it was in prior to the exception.

The three groups of exception are categorized in Table 2. A group 0 exception originates from hardware errors (the address error has all the characteristics of a bus error but is generated internally by the 68000) and often indicates that something has gone seriously wrong with the system. For this reason the information saved in the stack frame corresponding to a group 0 exception is more detailed than that for groups 1 and 2. Figure 4b shows the stack frame of group 0 exceptions.

The additional information saved in the stack frame by a group 0 exception is a copy of the first word of the instruction being processed at the time of the exception and the 32-bit address that was being accessed by the aborted memory access cycle. The third new item saved is a 5-bit value giving the function code which was displayed on FC2, FC1 and FC0 when the exception

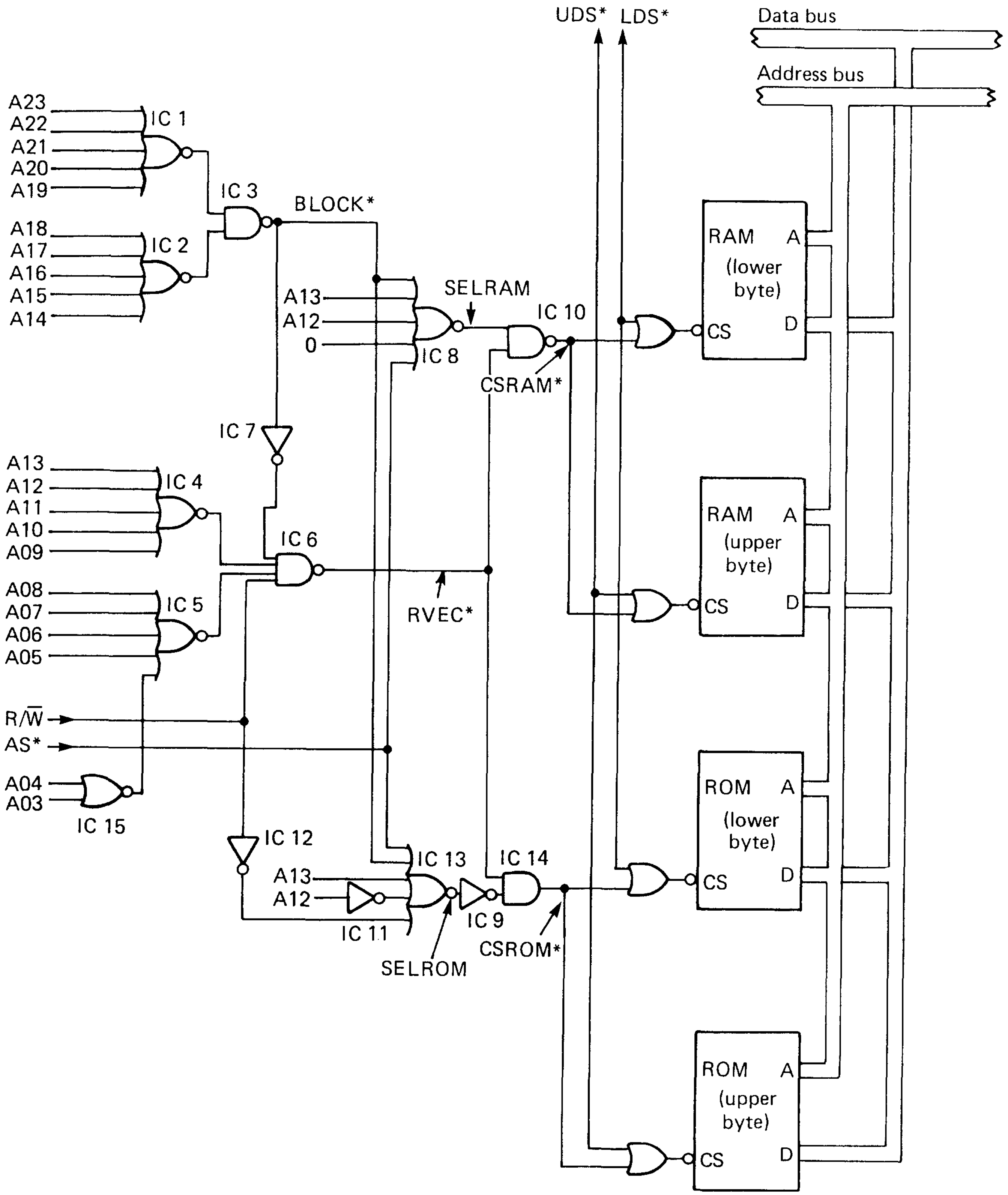


Figure 3. Implementation of an overlaid reset vector

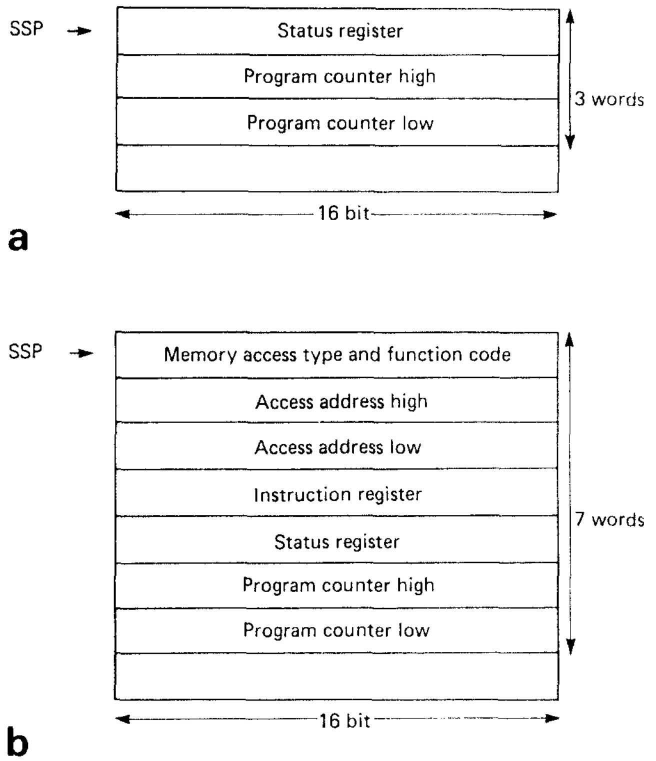


Figure 4. Stack frames: a, for group 1 and group 2 exceptions; b, for group 0 exceptions

occurred, together with an indication of whether the processor was executing a read or a write cycle, and whether is was processing an instruction or not. This is diagnostic information which may be used by the operating system when dealing with the cause of the exception.

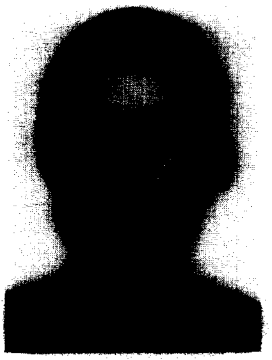
The fourth and final phase of the exception processing sequence consists of a single operation — the loading of the program counter with the 32-bit address pointed at by the exception vector. Once this has been done, the processor continues executing instructions normally.

When an exception handling routine has been run to completion, the instruction 'return from exception' (RTE) is executed to restore the processor to the state it was in prior to the exception. RTE is a privileged instruction and has the effect of restoring the status register and program counter from the values saved on the system stack. The contents of the program counter and status register, just prior to the execution of the RTE, are lost. RTE cannot be used after a group 0 exception to execute a return.

Table 2. Exception grouping according to type and priority

Group	Exception type	Time at which processing begins
0	Reset Bus error Address error	Exception processing begins within two clock cycles
1	Trace Interrupt Illegal op. code Privilege	Exception processing begins before the next instruction
2	TRAP TRAPV CHK Divide by zero	Exception processing is by normal instruction execution

The second part of this paper, to be published in *Microprocessors Microsyst.* Vol 10 No 5 (June 1986), looks at the way in which interrupt handling is actually implemented in the 68000.



Alan Clements obtained an honours degree in electronics at the University of Sussex, UK, in 1971 and a doctorate from Loughborough University, UK, in 1976. During the two years he spent at Loughborough University as a research fellow, he studied the application of microprocessor technology to adaptive equalizers for distorted digital signals. In 1977 he joined the Department of Computer Science at Teesside Polytechnic, UK. In the last few years, he has devoted much of his spare time to writing. His first book, Microprocessor systems design and construction, *was published in 1982 and this was followed by* Principles of computer hardware *in 1985.*

Exception handling in the 68000, Part 2

In the second of two tutorial papers, **Alan Clements** examines in detail the implementation of hardware and software initiated exceptions on the 68000, with particular emphasis on interrupts

The paper discusses the implementation by the 68000 microprocessor of hardware and software initiated exceptions. The three categories of hardware exception — resets, bus errors and interrupts — are covered first, with detailed discussion of how vectored and autovectored interrupts are processed. The software initiated exceptions discussed are illegal operation codes, traces, emulator-mode exceptions and traps. An example of the use of a trap handler is given.

microprocessors exceptions interrupts 68000

Part 1 of this paper[1] introduced the concept of the exception, a mechanism employed by computers to deal with certain events. These events cannot easily be handled by normal 'inline' programming techniques and require the processor to temporarily interrupt its normal processing. Part 2 of the paper examines how the 68000 implements asynchronous exceptions, and introduces the software exception.

HARDWARE INITIATED EXCEPTIONS

Three types of exception are initiated by events taking place outside the 68000 and are communicated to the CPU via its input pins; they are the reset, the bus error and the interrupt. Each of these three exceptions has a direct effect on the hardware design of a 68000-based microcomputer. This section examines each of these systems in more detail, beginning with a look at the control of the RESET* pin.

Reset

It was stated in Part 1 of the paper[1] that the exception vector table is frequently located in read/write memory, and that the circuit shown in Figure 3 of Part 1 can be used to overlay the first 8 byte of the read/write memory with ROM containing the reset vectors. The only other circuitry associated with the 68000's reset mechanism is that connected to the RESET* pin itself.

The designers of the 68000 have made RESET* a bidirectional I/O, which complicates the design of the reset circuitry. In normal operation the RESET* pin is an input. Almost all microprocessor systems connect every device that can be reset to the RESET* pin. This means that all devices are reset along with the 68000 at power up or following a manual reset. The 68000 is also capable of executing a software reset instruction which forces its RESET* pin active low, resetting all devices connected to it. This facility was provided to permit a system reset under software control that does not affect the processor itself. Consequently the RESET* pin of the 68000 cannot be driven by gates with active pull-up circuits. RESET* must be driven by open-collector or open-drain outputs.

Another aspect to note about hardware initiated resets is that both RESET* and HALT* must be asserted simultaneously. Like RESET*, the HALT* pin is bidirectional and must also be driven by an open-collector or open-drain output. If RESET* and HALT* are asserted together following a system crash, they must be held low for at least 10 clock cycles to ensure satisfactory operation. However, at power up they must be held low for at least 100 ms after the V_{CC} supply to the 68000 has become established.

A possible arrangement of a reset circuit for the 68000 is given in Figure 5. IC1, a 555 timer, generates an active-high pulse at its output terminal shortly after the initial application of power. The timer is configured to operate in an astable mode, generating a single pulse whenever it is triggered. The time constant of the output pulse (ie the duration of the reset pulse) is determined by resistor R_2 and capacitor C_2. R_1 and C_1 trigger the circuit on the application of power. The output is buffered and inverted by IC2a to become the system active-low power-on-reset pulse (POR*), which can be used by the rest of the system as appropriate. A pair of crosscoupled NAND gates provides a manual reset facility.

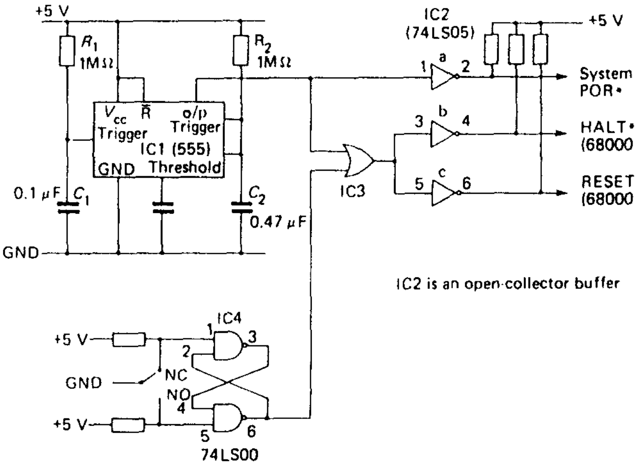


Figure 5. Control of RESET* in a 68000-based system

Department of Computer Science, Teesside Polytechnic, Middlesbrough, Cleveland TS1 3BA, UK

0141-9331/86/05258-10 $03.00 © 1986 Butterworth & Co. (Publishers) Ltd

Bus error

A bus error is an exception raised in response to a failure by the system to complete a bus cycle. As there are many possible failure modes, the details of each depending on the type of hardware used to implement the system, the detection of a bus error is left to the systems designer rather than to the 68000 itself. All the 68000 provides is an active-low input, BERR*, which generates a bus error exception when asserted.

Figure 6 gives the timing requirements that the BERR* input must satisfy. To be recognized during the current bus cycle, BERR* must fulfil one of two conditions: it must either be asserted at least time t_{ASI} (the asynchronous input set-up time) before the falling edge of state S4; or it must be asserted at least time t_{BELDAL} (BERR* low to DTACK* low) before the falling edge of DTACK*. It is necessary to maintain BERR* active low until time t_{SHBEH} (AS* high to BERR* high) after the address and data strobes have become inactive. The minimum value of t_{SHBEH} is zero, implying that BERR* may be negated concurrently with AS* or LDS*/UDS*. It is important to realize that, if BERR* meets the timing requirement t_{ASI}, it will be processed in the current bus cycle irrespective of the state of DTACK*.

There are a number of reasons why BERR* may be asserted in a system. Typical applications of BERR* are as follows.

- Illegal memory access. If the processor tries to access memory at an address not populated by memory, BERR* may be asserted. BERR* may also be asserted if an attempt is made to write to a read-only memory address. The decision as to whether to assert BERR* in these cases is a design decision. (All 8-bit microprocessors are quite happy to access nonexistent memory or to write to ROM!) The philosophy of 68000 systems design is to trap events which may lead to unforeseen circumstances. If the processor tries to write to ROM, the operating system can intervene because of the exception raised by BERR*.
- Faulty memory access. If error-detecting memory is employed, a read access to a memory location at which an error is detected can be used to assert BERR*. In this way the processor will never try to process data that is in error due to a fault in the memory.
- Failure to assert valid peripheral address (VPA*). If the processor accesses a synchronous bus device and VPA* is not asserted after some time-out period, BERR* must be asserted to stop the system hanging up and waiting for VPA* forever.
- Memory privilege violation. When the 68000 is used in a system with some form of memory management, BERR* may be asserted to indicate that the current memory access is violating a privilege. This may be an access by one user to another user's program. In a system with virtual memory, it may result from a page fault, indicating that the data being accessed is not currently in read/write memory.

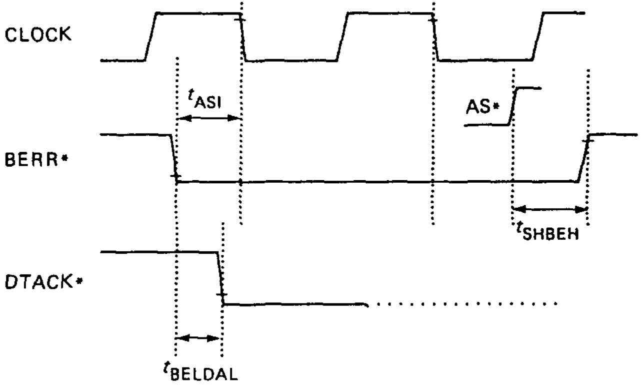


Figure 6. Bus error input (BERR) timing diagram: t_{ASI} = 20 ns (min.); t_{BELDAL} = 20 ns (min.); t_{SHBEH} = 0 ns (min.)*

Bus error sequence

When BERR* is asserted by external logic and satisfies its set-up timing requirements, the processor negates AS* in state S6. As long as BERR* remains asserted the data and address buses are both floated. When the external logic negates BERR*, the processor begins a normal exception processing sequence for a Group 0 exception. Figure 4b (Part 1 of the paper[1]) shows that additional information is pushed onto the system stack to facilitate recovery from the bus error. Once all phases of the exception processing sequence have been completed, the 68000 begins to deal with the problem of the error in the BERR* exception handling routine. It must be emphasized that the treatment of the hardware problem which led to the bus error takes place at a software level within the operating system. For example, if a user program generates a bus error, the exception processing routine may abort the user's program and provide him/her with diagnostic information to help deal with the problem. The information stored on the stack by a bus error exception (or an address error) is to be regarded as diagnostic information only and should not be used to institute a return from exception. In other words, the 68000 does not support a return from a Group 0 exception (although the 68010 does).

Rerunning the bus cycle

It is possible to deal with a bus error in a way which does not involve an exception. If, during a memory access, the external hardware detects a memory error and asserts both BERR* and HALT* simultaneously, the processor attempts to rerun the current bus cycle.

Figure 7 illustrates a rerun cycle. A bus fault is detected in the read cycle and both BERR* and HALT* are asserted simultaneously. As long as the HALT* signal remains asserted, the address and data buses are floated and no external activity takes place. When HALT* is negated by the external logic, the processor will rerun the previous bus cycle using the same address, the same function codes, the same data (for a write operation) and the same

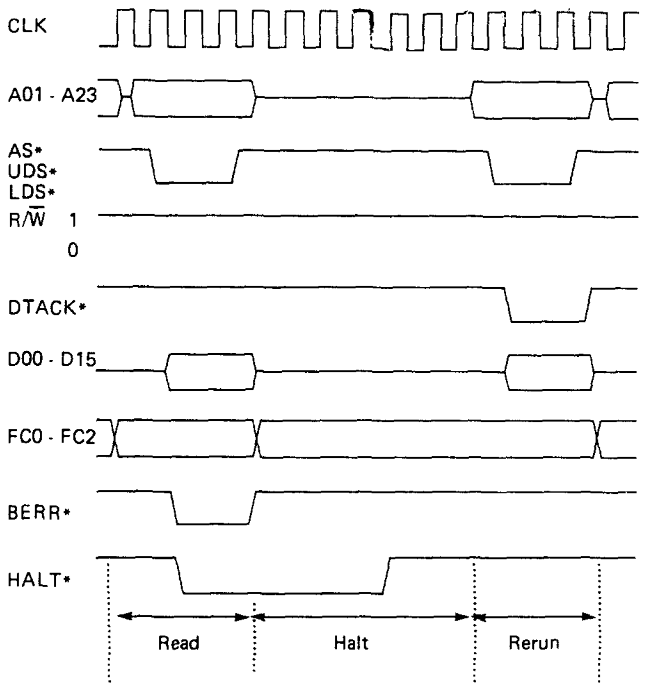


Figure 7. Rerunning the bus cycle

control signals. For correct operation, the BERR* signal must be negated at least one clock cycle before HALT* is negated.

A possible implementation of bus error control in a sophisticated 68000-based system might detect a bus error and assert BERR* and HALT* simultaneously. The rising edge of AS* can be used to release BERR* and HALT* at least one clock cycle later. This guarantees a rerun of the bus cycle. Of course, if the error is a 'hard' error (ie is persistent), rerunning the bus cycle will achieve little and external logic will once again detect the error. A reasonable strategy would be to permit, say, three reruns and assert BERR* alone on the next cycle, forcing a conventional bus error exception.

Interrupt

As discussed in Part 1[1], an interrupt is a request for service generated by some external peripheral, which the 68000 implements within the framework of its overall exception handling procedures. The 68000 offers two schemes for dealing with interrupts. One is intended for peripherals specifically designed for 16-bit processors, while the other is more suited to earlier 8-bit (6800 series) peripherals.

An external device signals its need for attention by placing a 3-bit code on the 68000's interrupt request inputs (IPL0*, IPL1* and IPL2*). The code corresponds to the priority of the interrupt and is numbered from 0 to 7. A level 7 code indicates the highest priority, level 1 the lowest priority and level 0 the default state of no interrupt request. While it would be possible to design peripherals with three interrupt request output lines on which they put a 3-bit interrupt priority code, it is easier to have a single interrupt request output and to design external hardware to convert its priority into a 3-bit code suitable for the 68000.

Figure 8 shows a typical scheme for handling interrupt requests in a 68000 system. A 74LS148 eight-line to three-line priority encoder is all that is needed to translate one of the seven levels of interrupt request into a 3-bit code. Table 3 gives the truth table for this device. Input EI* is an active-low enable input, used in conjunction with outputs GS and EO to expand the 74LS148 in systems with more then seven levels of priority.

As the enable input and expanding outputs are not needed in this application of the 74LS148, Table 3 has been redrawn in Table 4 with inputs 1-7 renamed IRQ1*-IRQ7*, respectively, and outputs A0-A2 renamed IPL0*-IPL2*. It must be appreciated that all inputs and all outputs are active low, so that an output value 0, 0, 0 denotes an interrupt request of level 7, while an output 1, 1, 1 denotes a level 0 interrupt request (ie no interrupt).

Inspecting Table 4 reveals that a logical zero on interrupt request input *i* forces interrupt request inputs 1 to $i - 1$ into 'don't care' states (ie if interrupt IRQi* is asserted, the state of interrupt request inputs IRQ1* to

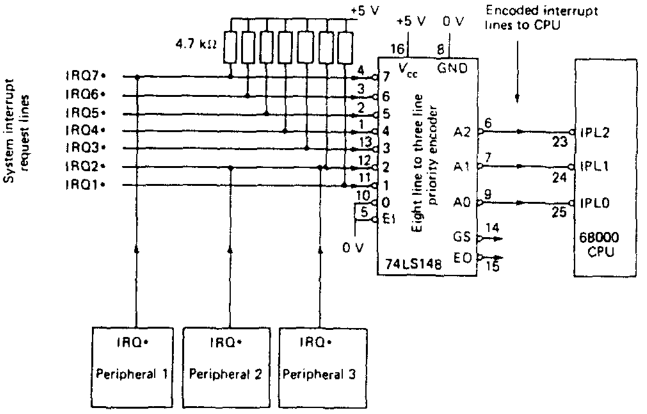


Figure 8. Interrupt request encoding

Table 3. Truth table for the 74LS148 eight-line to three-line priority encoder

Inputs									Outputs				
EI*	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
1	X	X	X	X	X	X	X	X	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	X	X	X	X	X	X	X	0	0	0	0	0	1
0	X	X	X	X	X	X	0	1	0	0	1	0	1
0	X	X	X	X	X	0	1	1	0	1	0	0	1
0	X	X	X	X	0	1	1	1	0	1	1	0	1
0	X	X	X	0	1	1	1	1	1	0	0	0	1
0	X	X	0	1	1	1	1	1	1	0	1	0	1
0	X	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1

0 = low-level signal (< VIL); 1 = high-level signal (> VIH);
X = 'don't care'

Table 4. Truth table for a 74LS148 configured as in Figure 8

Inputs								Outputs		
Level	IRQ1*	IRQ2*	IRQ3*	IRQ4*	IRQ5*	IRQ6*	IRQ7*	IPL2*	IPL1*	IPL0*
7	X	X	X	X	X	X	0	0	0	0
6	X	X	X	X	X	0	1	0	0	1
5	X	X	X	X	0	1	1	0	1	0
4	X	X	X	0	1	1	1	0	1	1
3	X	X	0	1	1	1	1	1	0	0
2	X	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1

0 = low-level signal (< VIL); 1 = high-level signal (> VIH); X = 'don't care'

IRQ[$i - 1$]* has no effect on the output code IPL0*–IPL2*). It is this property that the microprocessor systems designer relies on. Devices with high-priority interrupts are connected to the higher-order inputs. Should two or more levels of interrupt occur simultaneously, only the higher value is reflected in the output code to the 68000's IPL pins.

Figure 8 demonstrates that the 74LS148 does not restrict the system to only seven devices capable of generating interrupt requests. More than one device can be wired to a given level of interrupt request as illustrated by peripherals 2 and 3. If peripheral 2 or 3 (or both) asserts its interrupt request output (IRQ*), a level 2 interrupt is signalled to the 68000, provided that levels 3–7 are all inactive. The mechanism used to distinguish between an interrupt from peripheral 2 and one from peripheral 3 is called 'daisychaining' and enables several devices to share the same level of interrupt priority while permitting only one of them to respond to an IACK cycle. Daisychaining is implemented by feeding the IACK* response from the CPU into the first peripheral in the daisychain. When an IACK cycle is executed, the first peripheral receives the IACK* signal from the CPU. If this peripheral generated the interrupt, it responds to the IACK*. If it did not generate the interrupt, it passes the IACK* signal onto the next peripheral in the chain. Note that, in this arrangement, each peripheral requires an IACK_IN* pin and an IACK_OUT* pin.

Processing the interrupt

All interrupts to the 68000 are latched internally and made pending. Group 0 exceptions (reset, bus error, address error) take precedence over an interrupt in Group 1. Therefore, if a Group 0 exception occurs, it is serviced before the interrupt. A trace exception in Group 1 takes precedence over the interrupt, so that if an interrupt request occurs during the execution of an instruction while the T bit is asserted, the trace exception has priority and is serviced first. Assuming that none of the above exceptions have been raised, the 68000 compares the level of the interrupt request with the value recorded in the interrupt mask bits of the processor status word.

If the priority of the pending interrupt is lower than or equal to the current processor priority denoted by the interrupt mask, the interrupt request remains pending and the next instruction in sequence is executed. Interrupt level 7 is treated slightly differently, as it is always processed regardless of the value of the interrupt mask bits (ie a level 7 interrupt always interrupts a level 7 interrupt if one is currently being processed). Any other level of interrupt can be interrupted only by a higher level of priority. Note that a level 7 interrupt is edge sensitive and is interrupted only by a high-to-low transition on IRQ7*.

Once the processor has made a decision to process an interrupt, it begins an exception processing sequence as described earlier. The only deviation from the normal sequence of events dictated by a Group 1 or Group 2 exception is that the interrupt mask bits of the processor status word are updated before exception processing continues. The level of the interrupt request being serviced is copied into the current processor status. This means that the interrupt cannot be interrupted unless the new interrupt has a higher priority.

Example. Suppose that the current (ie pre-interrupt) interrupt mask is level 3. If a level 5 interrupt occurs, it is processed and the interrupt mask is set to level 5. If, during the processing of this interrupt, a level 4 interrupt is requested, it is made pending even though it has a higher priority than the original interrupt mask. When the level 5 interrupt has been processed, a return from exception is made and the former processor status word is restored. As the old interrupt mask was level 3, the pending interrupt of level 4 is then serviced.

Unlike other exceptions, an interrupt may obtain its vector number externally from the device that made the interrupt request. As stated above, there are two ways of identifying the source of the interrupt, one vectored and one autovectored.

Vectored interrupt

After the processor has completed the last instruction before recognizing the interrupt and stacked the low-order word of the program counter, it executes an interrupt acknowledge cycle (IACK cycle). During an IACK cycle, the 68000 obtains the vector number from the interrupting device, with which it will later determine the appropriate exception vector.

Figure 9 shows the sequence of events taking place during an IACK cycle; it can be seen that an IACK cycle is just a modified read cycle. Because the 68000 puts out the special function code 1, 1, 1 on FC2, FC1 and FC0 during an IACK cycle, the interrupting device is able to detect the IACK cycle. At the same time, the level of the interrupt is put out on address lines A01-A03. The IACK cycle should not decode memory addresses A04-A23 and memory components should be disabled when FC2-FC0 = 1, 1, 1. The device that generated the interrupt at the specified level then provides a vector number on D00-D07 and asserts DTACK*, as in any normal read cycle. The remainder of the IACK cycle is identical to a read cycle. Figure 10 gives the timing diagram of an IACK cycle. Note that the IACK cycle falls between the stacking of the low-order word of the program counter and the stacking of the high-order word.

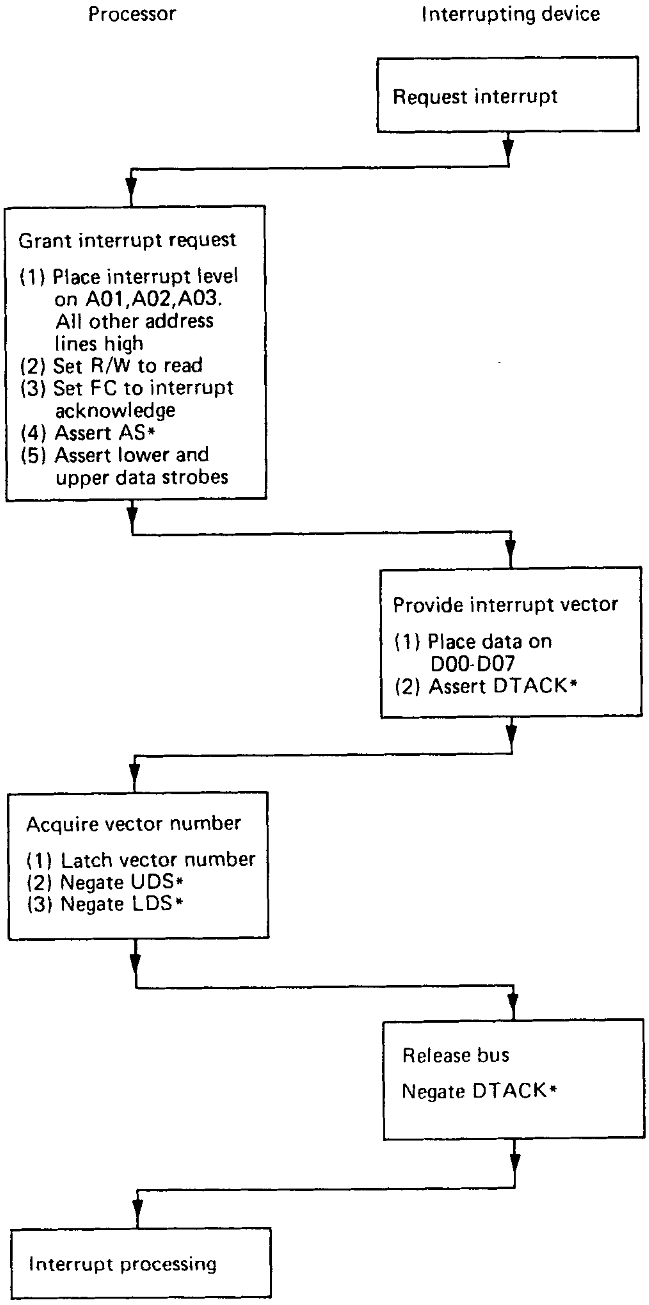


Figure 9. The interrupt acknowledge sequence

After the peripheral has provided a vector number on D00-D07, the processor multiplies it by four to obtain the address of the entry point to the exception processing routine from the exception vector table. Although a device can provide an 8-bit vector number giving 256 possible values, there is space reserved in the exception vector table for only 192 unique vectors. This is more than adequate for the vast majority of applications. (However, a peripheral can put out vector numbers 0-63, as there is nothing to stop these numbers being programmed into the peripheral and the processor does not guard against this situation. In other words, if a programmer programs a peripheral to respond to an IACK cycle with, say, a vector number 5, then an interrupt from this device would cause an exception corresponding to vector number 5 — the value also appropriate to a divide-by-zero exception. While at times this might be useful, it seems an oversight to allow interrupt vector numbers to overlap with other types of exceptions.)

A possible arrangement of hardware needed to implement a vectored interrupt scheme is given in Figure 11. A peripheral asserts its interrupt request output, IRQ5*, which is encoded by IC3 to provide the 68000 with a level 5 interrupt request. When the processor acknowledges this request, it places 1, 1, 1 on the function code output, which is decoded by the three-line to eight-line decoder IC1. The interrupt acknowledge output (IACK*) from IC1 enables a second three-line to eight-line decoder, IC2, which decodes address lines A01-A03 into seven levels of interrupt acknowledge. In this case, IACK5* from IC2 is fed back to the peripheral, which then responds by placing its vector number onto the low-order byte of the system data bus. If the peripheral has not been programmed to supply an interrupt vector number it should place $0F on the data bus, corresponding to an uninitialized interrupt vector exception.

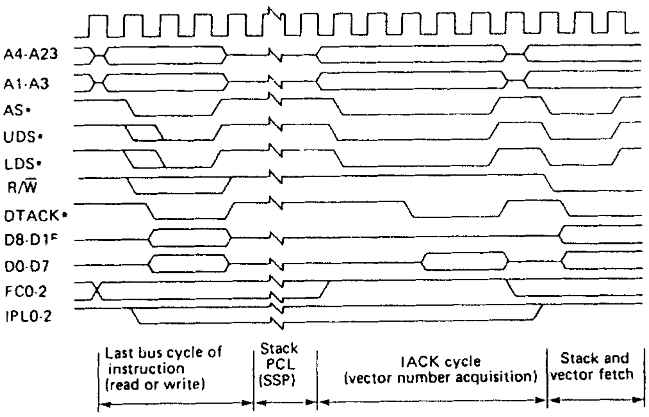


Figure 10. Interrupt acknowledge and the IACK cycle

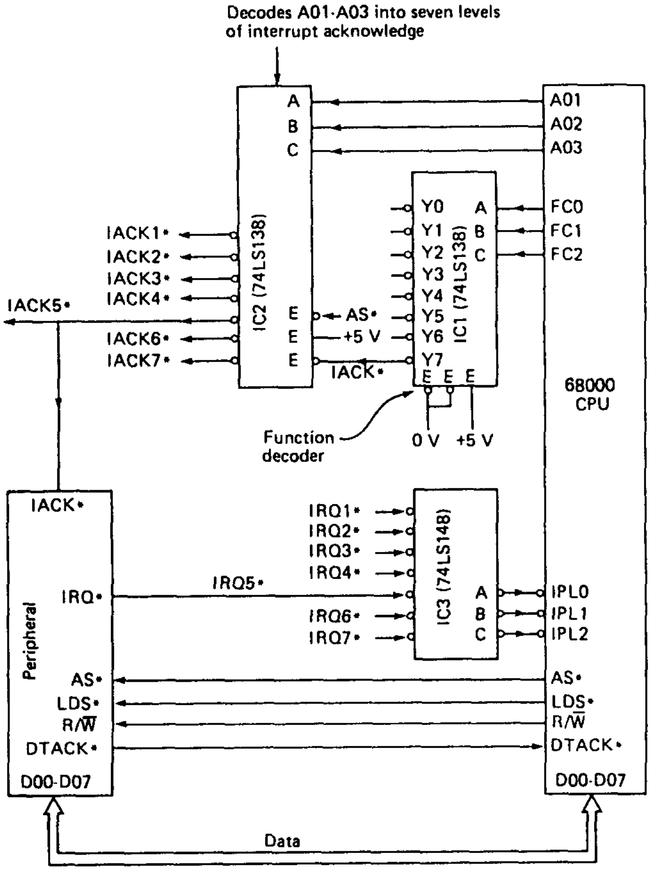


Figure 11. Implementing the vectored interrupt

Autovectored interrupt

As set out above, a device which generates an interrupt request must be capable of identifying itself when the 68000 carries out an interrupt acknowledge sequence. This presents no problem for modern 68000-based peripherals such as the 68230 parallel interface-timer.

Unfortunately, older peripherals originally designed for 8-bit processors do not have interrupt acknowledge facilities and are unable to respond with the appropriate vector number on D00-D07 during an IACK cycle. The systems designer could overcome this problem by designing a subsystem which supplied the appropriate vector as if it came from the interrupting peripheral. Such an approach is valid but a little messy: a single-chip peripheral would need several components just to provide a vector number in an IACK cycle.

An alternative scheme is available for peripherals that cannot provide their own vector numbers. An IACK cycle, like any other memory access, is allowed to continue to state S5 by the assertion of DTACK*. If, however, DTACK* is not asserted but VPA* is asserted, the 68000 carries out an autovectored interrupt.

Valid peripheral address, VPA*, belongs to the 68000's synchronous data bus control group of signals. When asserted, VPA* informs the 68000 that the present memory access cycle is to be synchronous and to 'look like' a 6800 series memory access cycle. If the current bus cycle is an IACK cycle, the 68000 executes a 'spurious read cycle', ie an IACK cycle is executed but the interrupting device does not place a vector number on D00-D07. Nor does the 68000 read the contents of the data bus; instead, it generates the appropriate vector number internally.

The 68000 reserves vector numbers 25-31 (decimal) for its autovector operation (see Table 1). Each of these autovectors is associated with an interrupt on IRQ1*-IRQ7*. For example, if IRQ2* is asserted followed by VPA* during the IACK cycle, vector number 26 is generated by the 68000 and the interrupt handling routine address read from memory location $000068.

Should several interrupt requesters assert the same interrupt request line, the 68000 will not be able to distinguish between them. The appropriate interrupt handling routine must poll each of the possible devices in turn (ie the status register of each peripheral must be read to determine the source of the interrupt).

The timing diagram of an autovector sequence is given in Figure 12 and is almost identical to the vectored IACK sequence of Figure 10, except that VPA* is asserted shortly after the interrupter has detected an IACK cycle from FC0-FC2. Because VPA* has been asserted, wait states are introduced into the current read cycle in order to synchronize the cycle with VMA*. Note that this is a dummy read cycle as nothing is read. (The autovector is generated internally and no device places data on D00-D07 during the cycle.)

The hardware necessary to implement an autovectored interrupt is minimal. Figure 13 shows a possible arrangement involving a typical 6800 series peripheral which requests an interrupt in the normal way by asserting its IRQ* output. This is prioritized by IC3 and an acknowledge signal is generated by ICs 1 and 2.

The interrupting device cannot, of course, respond to an IACK* signal. Instead, the appropriate interrupt acknowledge signal from the 68000 is combined with the

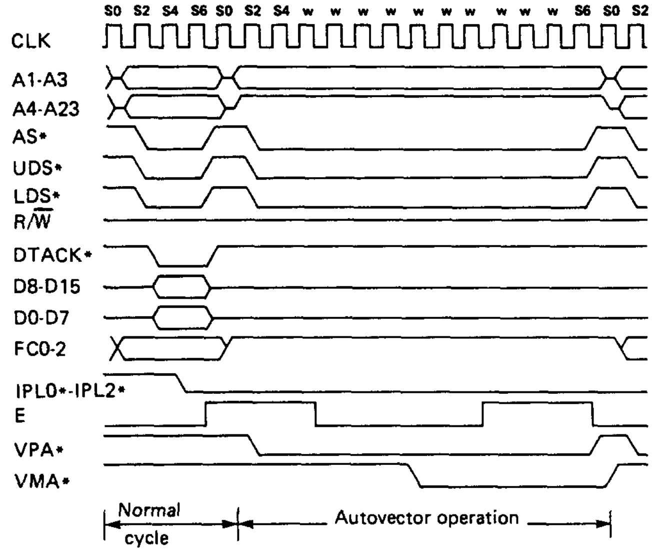


Figure 12. Timing diagram of an autovectored interrupt

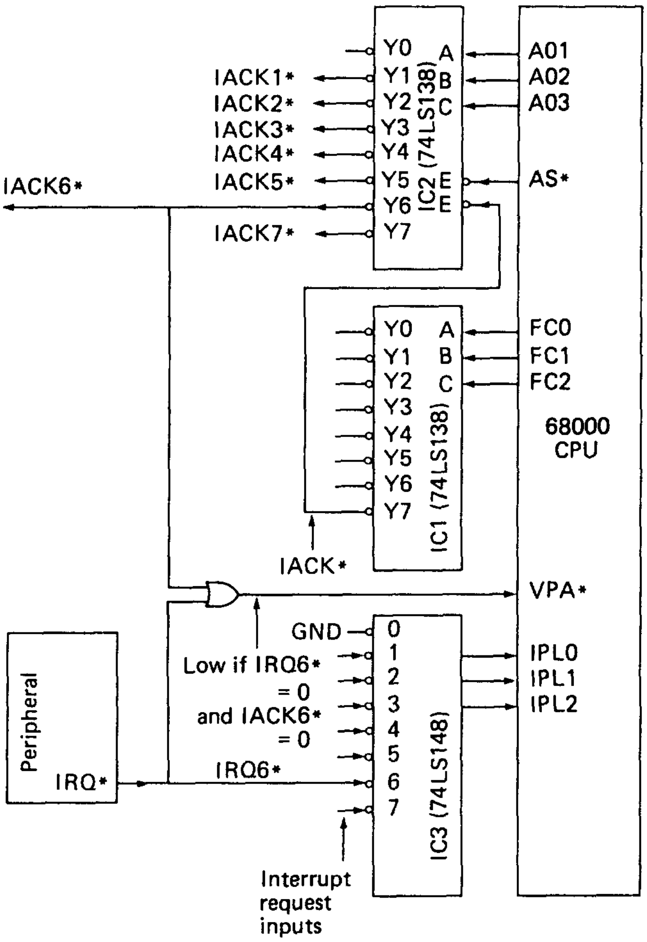


Figure 13. Hardware needed to implement an autovectored interrupt

interrupt request output from the peripheral in an OR gate. Only when the peripheral has asserted its IRQ* and the correct level of IACK* has been generated does the output of the OR gate go low to assert VPA* and force an autovectored interrupt.

SOFTWARE INITIATED EXCEPTIONS

A software initiated exception is one that occurs as the result of an attempt to execute certain types of instruction (not the address error which is classified as a hardware initiated interrupt). Software initiated interrupts fall into two categories: those executed deliberately by the programmer and those representing a 'cry for help'.

The 'help' group comprises the illegal op. code, privilege violation, TRAPV and divide-by-zero exceptions. These are all exceptions that are normally generated by something going wrong; therefore the operating system needs to intervene and sort things out. The nature of this intervention is very much dependent on the structure of the operating system. Often, in a multiprogramming environment, the individual task creating the exception will be aborted, leaving all other tasks unaffected.

Software exceptions initiated by the programmer are the trace, the trap and the emulator. The trace exception mode is in force whenever the T bit of the status word is set. After each instruction has been executed, a trace exception is automatically generated if the T bit is set. This is done to allow the user to monitor the execution of a program.

Illegal op. code exceptions

Consider the illegal op. code exception. This is raised when the 68000 attempts to execute an op. code that does not form part of the 68000's instruction set. The only way that this can happen is when something has gone seriously wrong — an op. code has been corrupted in memory or a jump has been made to a region containing nonvalid 68000 code. The latter event frequently results from wrongly computed GOTOs. Clearly, once such an event has occurred, it is futile to continue trying to execute further instructions as they have no real meaning. By generating an illegal op. code exception, the operating system can inform users of the problem and invite them to do something about it.

Trace exceptions

The simplest trace facility would allow the user to dump the contents of all registers on the CRT terminal after the execution of each instruction. Unfortunately, this leads to the production of vast amounts of utterly useless information. For example, if the 68000 were executing an operation to clear an array by executing a CLR.L (A4)+ instruction 64k times, the human operator would not wish to see the contents of all registers displayed after the execution of each CLR.

A better approach is to display only the information needed. Before the trace mode is invoked, the user informs the operating system of the conditions under which the results of a trace exception are to be displayed. Some of the events which can be used to trigger the display of registers during a trace exception are

- execution of a predefined number of instructions (eg contents of registers may be displayed after, say, 50 instructions have been executed)
- execution of an instruction at a given address (equivalent to a break point)
- execution of an instruction falling within a given range of addresses, or the access of an operand falling within the same range
- as last event, but with the contents of the register displayed only when an address generated by the 68000 falls outside the predetermined range
- execution of a particular instruction (eg contents of the registers may be displayed following the execution of a TAS instruction)
- any memory access which modifies the contents of a memory location (ie any write access)

It is possible to combine several of the above conditions to create a composite event. For example, the contents of registers may be displayed whenever the 68000 executes write accesses to the region of memory space between $3A 0000-$3A 00FF.

Emulator-mode exceptions

Emulator-mode exceptions provide the systems designer with tools to develop software for new hardware before that hardware has been fully realized. Suppose a company is working on a coprocessor to generate the sine of a 16-bit fractional operand. For commercial reasons, it may be necessary to develop software for this hardware long before the coprocessor is in actual production.

By inserting an emulator op. code (ie an exception call) at the point in a program at which the sine is to be calculated by the hardware, the software can be tested as if the coprocessor were actually present. When the emulator op. code is encountered, a jump is made to the appropriate emulator handling routine. In this routine, the sine is calculated by conventional techniques.

Trap exception

The trap is the most useful software user initiated exception available to the programmer. Indeed, it is one of the more powerful functions provided by the 68000. There are no significant differences between traps and emulator exceptions; they differ only in their applications. There are sixteen traps, TRAP #0-TRAP #15, which are associated with exception vector numbers 32-47 (decimal) respectively.

Just as emulator exceptions are used to provide functions in software that will later be implemented in hardware, trap exceptions create new operations or 'extra codes' not provided directly by the 68000 itself. However, the purpose of the trap is to separate the details of certain 'housekeeping' functions from the user- or applications-level program.

Consider I/O transactions. These involve real hardware devices and the precise nature of an input operation on system A may be very different from that on system B, even though both systems put the input to the same use. System A may operate a 6850 ACIA in an interrupt-driven mode to obtain data, while system B may use in Intel 8055 parallel port in a polled mode to carry out the same function. Clearly, the device drivers (ie the software which controls the ports) in these systems differ greatly in their structures.

Applications programmers do not wish to consider the fine details of I/O transactions when writing their programs. One solution is to use a jump table and to thread all I/O through this table. Table 5 illustrates this approach. It can be seen that the applications programmer deals with all device-dependent transactions by indirect jumps through a jump table. For example, all console input at the applications level is carried out by BSR GETCHAR. At the address GETCHAR in the jump table, the programmer inserts a link (JSR INPUT) to the actual routine used in his/her own system.

This is a perfectly acceptable approach to the problem of device dependency. Unfortunately, it suffers from the limitation that the applications program must be tailored to fit on to the target system. This is done by tagging on the jump table. An alternative approach, requiring no modification whatsoever to the applications software, is provided by the trap exception. This leads to truly system-independent software.

When a trap is encountered, the appropriate vector number is generated and the exception vector table interrogated to obtain the address of the trap handling routine. Note that the exception vector table fulfils the same role as the jump table (Table 5). The difference is that the jump table forms part of the applications program while the exception vector table is part of the 68000's operating system.

An example of a trap handler is found on the Motorola educational single board (ECB) computer. This is known as the 'TRAP #14 handler' and provides the user with a method of accessing functions within the ECB's monitor software without the user having to know their addresses.

The versatility of a trap exception can be increased by passing parameters from the user program to the trap handler. The TRAP #14 handler of Tutor (the monitor on the Motorola ECB) provides for up to 255 different functions to be associated with TRAP #14. Before the trap is invoked, the programmer must load the required function code into the least significant byte of D7. For example, to transmit a single ASCII character to port 1, the following calling sequence is used.

```
OUTCH  EQU     248           (Equate the function
                              code to name of
                              activity)
       MOVE.B  #OUTCH, D7    (Load function code
                              in D7)
       TRAP    #14           (Invoke TRAP #14
                              handler)
```

Table 6 gives a list of the functions provided by the TRAP #14 exception handler of the Tutor monitor on the ECB.

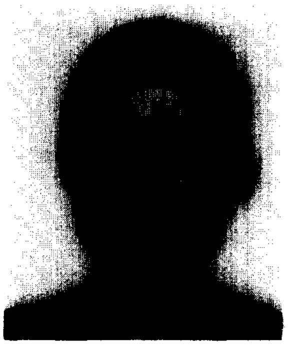
Table 5. The jump table

```
           ORG  $001000    Jump table
GETCHAR    JMP  INPUT
OUTCHAR    JMP  OUTPUT     INPUT, OUTPUT
GETSECTOR  JMP  DISK_IN    DISK_IN, DISK_OUT
PUTSECTOR  JMP  DISK_OUT   Provided by user

.
.
BSR  GETCHAR    input a char
.
.
.                           Application program
BSR  PUTSECTOR  write sector  (address of subroutines
.                              not system dependent)
.
```

Table 6. **Functions provided by the TRAP #14 handler on the EBC**

Function value	Function name	Function description
255	—	Reserved functions — end of table indicator
254	—	Reserved function — used to link tables
253	LINKIT	Append user table to TRAP14 table
252	FIXDAOD	Append string to buffer
251	FIXBUF	Initialize A5 and A6 to BUFFER
250	FIXDATA	Initialize A6 to BUFFER and append string to BUFFER
249	FIXDCRLF	Move CR, LF string to buffer
248	OUTCH	Output single character to port 1
247	INCHE	Input single character from port 1
246	—	Reserved function
245	—	Reserved function
244	CHRPRNT	Output single character to port 3
243	OUTPUT	Output string to port 1
242	OUTPUT21	Output string to port 2
241	PORTIN1	Input string from port 1
240	PORTIN20	Input string from port 2
239	TAPEOUT	Output string to port 4
238	TAPEIN	Input string from port 4
237	PRCRLF	Output string to port 3
236	HEX2DEC	Convert hex values to ASCII-encoded decimal
235	GETHEX	Convert ASCII character to hex
234	PUTHEX	Convert one hex digit to ASCII
233	PNT2HX	Convert two hex digits to ASCII
232	PNT4HX	Convert four hex digits to ASCII
231	PNT6HX	Convert six hex digits to ASCII
230	PNT8HX	Convert eight hex digits to ASCII
299	START	Restart Tutor; perform initialization
228	TUTOR	Go to Tutor; print prompt
227	OUT1CR	Output string plus CR, LF to port 1
226	GETNUMA	Convert ASCII encoded hex to hex
225	GETNUMD	Convert ASCII encoded decimal to hex
224	PORTIN1N	Input string from Port 1; no automatic line feed
223–128	—	Reserved
127–0	—	User-defined functions



Alan Clements obtained an honours degree in electronics at the University of Sussex, UK, in 1971 and a doctorate from Loughborough University, UK, in 1976. During the two years he spent at Loughborough University as a research fellow, he studied the application of microprocessor technology to adaptive equalizers for distorted digital signals. In 1977 he joined the Department of Computer Science at Teesside Polytechnic, UK. In the last few years, he has devoted much of his spare time to writing. His first book, Microprocessor systems design and construction, *was published in 1982 and this was followed by* Principles of computer hardware *in 1985.*

CONCLUSIONS

These two papers have looked at the way in which the 68000 implements exception handling.

Over the past few years, microprocessors have become faster and are able to access much larger memory spaces than those available to earlier 8-bit machines. However, the improvements in the exception handling mechanisms of today's microprocessors are as significant as advances in microprocessor performance (ie throughput). Exception handling carried out within the framework of the 68000's user/supervisor operating modes brings new security mechanisms to microcomputers. Modem multiuser, multitasking systems require more than mere performance. They must have mechanisms which protect one task from illegal access by another task. Equally, they require mechanisms which protect the system from a wide range of 'abuses'.

The 68000 provides all these facilities. The operating system interface is furnished by the trap; protection from some forms of abuse is provided by invalid instruction exceptions, uninitialized interrupt exceptions etc. Hardware exceptions (eg bus error) protect the system from faulty hardware.

REFERENCE

1 **Clements, A** 'Exception handling in the 68000, Part 1' *Microprocessors Microsyst.* Vol 10 No 4 (May 1986) pp 202-210

BIBLIOGRAPHY

Bacon, J *The Motorola MC68000* Prentice-Hall, Englewood Cliffs, NJ, USA (1986)

Eccles, W J *Microprocessor systems — a 16-bit approach Addison-Wesley, Wokingham, UK (1985)*

Jaulent, P *The 68000 hardware and software* Macmillan, London, UK (1985)

Triebel, W A and Singh, A *The 68000 microprocessor — architecture, software and interfacing techniques* Prentice-Hall, Englewood Cliffs, NJ, USA (1986)

The VMEbus specification manual Printek, Benton Harbour, MI, USA (1985)

MC68000 educational computer board user's manual (MEXKECB/D2) Motorola, Austin, TX, USA (1982)

The MC68000 data manual Motorola, Austin, TX, USA (1983)