AN-807
Application Note

SPECIAL CONSIDERATIONS IN USING THE MC6801 INTERRUPT CAPABILITIES

Prepared by
Clint Bauer
Systems Engineer
Motorola Automotive Electronic Systems
Mesa, Arizona

M6800 Microprocessor family components are used in numerous real-time control applications, many of which use external interrupt, timer and/or ACIA interface devices to increase system capability. The MC6801 microcomputer brings all these capabilities together with ROM and RAM on a single chip, while also providing a dramatically enhanced, yet machine-code compatible version of the MC6800 processor.

The MC6801 interrupt control methods are also enhanced, but still retain the same philosophy of operation used by other M6800 family components. The improvements increase performance, but also make possible several application-dependent constraints which merit consideration in certain systems. It is hoped that the information contained in this application note will aid the reader during his system design, so that a similar education is not required at debug time. It is assumed that the reader is familiar with basic operation of the MC6801 as described in the MC6801 Data Sheet and/or the MC6801 Manual. An optional review of MC6801 interrupt operation is provided first, followed by a discussion of important interrupt design constraints.

The MC6801 interrupt structure is similar to that available in the MC6800. The principle difference is that the MC6801 has four additional interrupt vectors and handshake logic to control them (see Figure 1). MC6800 systems are able to support external circuits that offer this same capability, but normally do so by sharing the single $\overline{IRQ}$ line and interrupt vector. The additional MC6801 vectors allow more efficient interrupt service by eliminating polling requirements for the triple-function timer/counter (where quick response is especially helpful), and reducing them elsewhere.

Having more vectors, MC6801 systems now offer a greater probability that near simultaneous interrupts will occur. For example, the three vector internal timer will often handle multiple asynchronous events. Therefore, it is important that the MC6801 system designer carefully observe the exact rules concerning interrupt recognition, entry, and service. A review of these rules is provided below.

	Vector (MSB:LSB)	Description
Highest Priority	FFFE:FFFF	Reset
	FFFC:FFFD	Non-Maskable Interrupt (NMI)
	FFFA:FFFB	Software Interrupt (SWI)
	FFF8:FFF9	$\overline{IRQ1}$ Interrupt ($\overline{IRQ1}$, IS3 – Mode 7)
	FFF6:FFF7	$\overline{IRQ2}$/Timer Input Capture (ICF)
	FFF4:FFF5	$\overline{IRQ2}$/Timer Output Compare (OCF)
	FFF2:FFF3	$\overline{IRQ2}$/Timer Overflow (TOF)
Lowest Priority	FFF0:FFF1	$\overline{IRQ2}$/SCI (RDRF, ORFE, TDRE)

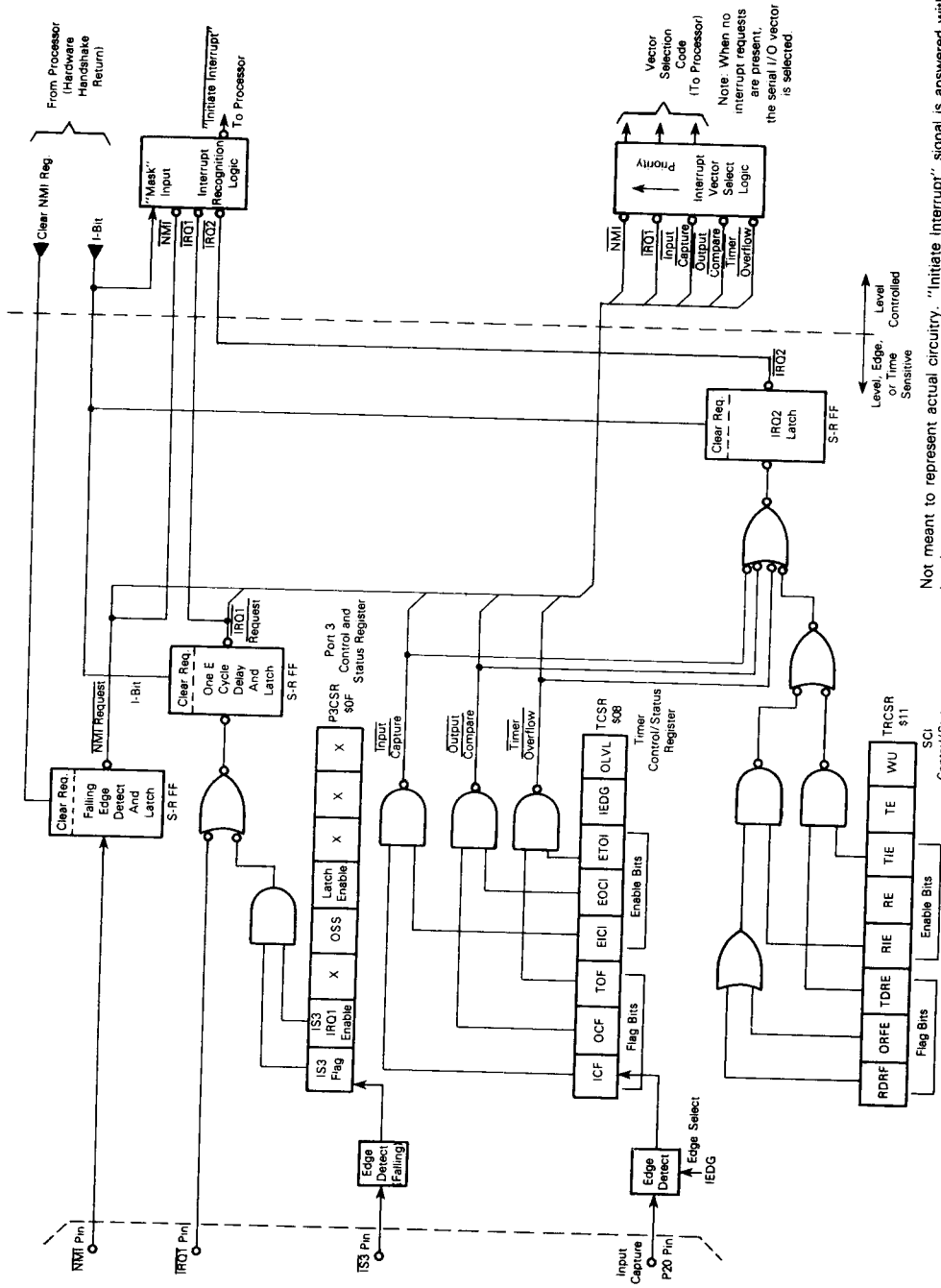
FIGURE 1 — MC6801 INTERRUPT PRIORITY AND VECTOR MEMORY MAP

a. *All interrupt possibilities but two are disallowed, or "masked" when the interrupt-mask bit I is set.* Bit I in the processor condition code register (CCR) is automatically set during MC6801 Power-up/Reset. The I-bit does not "mask" NMI (non-maskable interrupt). SWI (software interrupt) does not interrupt a program but executes like any other machine code and as such it is not maskable.

b. *I-Bit behavior can be summarized as follows:*
   (1) Actions that set the I-bit do so immediately.
   (2) Actions that clear the I-bit do so after one E cycle delay.

   Therefore, the CLI instruction can often be placed one program step sooner than might otherwise be thought, for the I-bit actually clears during the first cycle of the instruction following CLI.

c. *Most MC6801 interrupts can be inhibited at a second level.* Specific control bits in several MC6801 registers (see Figure 2) separately enable or disable the six interrupt possibilities shown in Table 1. All interrupt enable bits are cleared (disabled) during MC6801 Power-up/Reset. User programs can set or clear these bits, the action taking place during E time of an MPU write cycle to the specified register.



Not meant to represent actual circuitry. "Initiate Interrupt" signal is answered with handshake action that sets I-bit (upper right). For $\overline{IRQ1}$ and $\overline{IRQ2}$ interrupts, a software handshake is also necessary to prevent repeat service when I-bit is cleared. This second handshake clears the appropriate interrupt request directly or must indirectly cause line $\overline{IRQ1}$ to return to its inactive state.

FIGURE 2 — CONCEPTUAL REPRESENTATION OF MC6801 INTERRUPT STRUCTURE

TABLE 1 — MC6801 INTERRUPT FLAG AND ENABLE BITS

Interrupt	Interrupt Flag Bits	Interrupt Enable Bits
Input Strobe 3 $\overline{IRQ1}$	IS3 FLAG	IS3 IRQ1 ENABLE
Timer Input Capture	ICF	EICI
Timer Output Compare	OCF	EOCI
Timer Overflow	TOF	ETOI
Serial Receive	RDRF/ORFE	RIE
Serial Transmit	TDRE	TIE

d. *MC6801 interrupts are requested when appropriate actions set particular flag bits (the flag bits are listed in Table 1).* If the matching enable bit is set and the processor I-bit is clear, the flag bit will "request" interrupt service, as shown in Figure 3.

Activating the external $\overline{IRQ1}$ pin sets a non-machine-readable flag that remains latched as long as the I-bit is clear. The negative edge of $\overline{NMI}$ also influences a certain flip-flop to request service, but is serviced so quickly that there is no point in making its state readable.

e. *Interrupt request flags become set at the following times:*

IS3 FLAG: Directly clocked by the negative edge at $\overline{IS3}$ pin.

ICF: During $\overline{E}$ time that the timer capture actually occurs, which is two cycles after the capture pin edge.

OCF: During $\overline{E}$ time but one cycle after timer compare occurs.

TOF: During $\overline{E}$ time that the timer counter would read $FFFF.

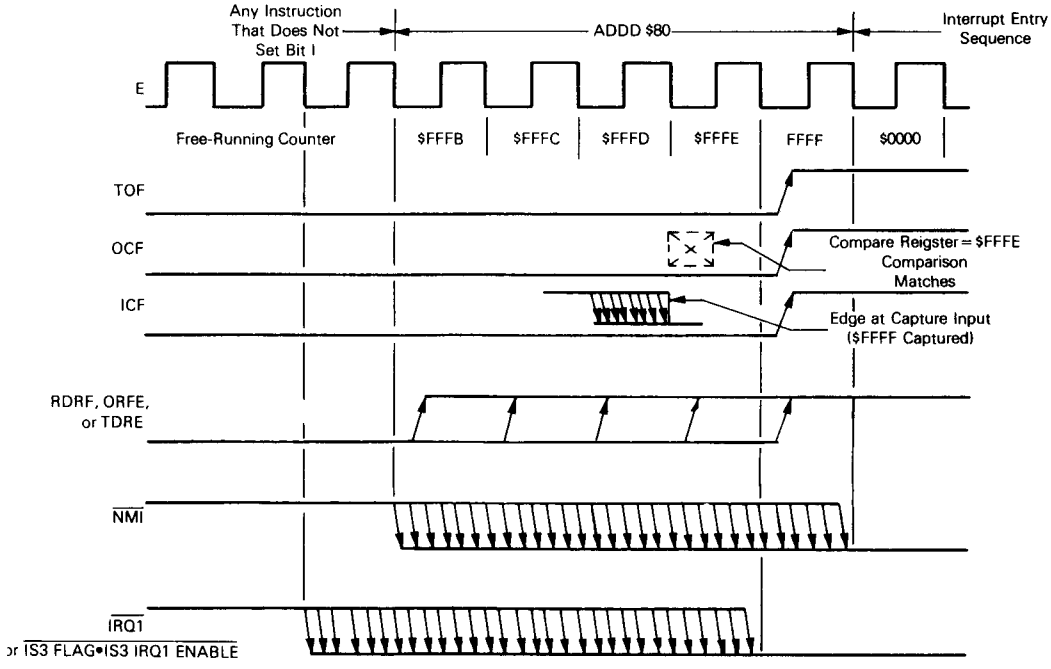
RDRF: During $\overline{E}$ time that received data is latched into buffer.

ORFE: During $\overline{E}$ time that an overrun or framing error is detected.

TDRE: During $\overline{E}$ time that a data word is actually transferred to the serial out shift register.

f. *Once set, the interrupt flag bits are cleared during E time of special memory accesses that occur after the flag is "armed" for clearing:* The $\overline{NMI}$ request flip-flop is automatically cleared during the tenth cycle of the interrupt entry sequence, as described later.

Flag Bit	Arming Mechanism	Bit Clearing Action
IS3 Flag	P3CSR read (at $F)	P3DATA read or write (at $6)
ICF	TCSR read (at $8)	CAPREG read (at $D)
OCF	TCSR read (at $8)	CMPREG write ($B or $C)
TOF	TCSR read (at $8)	COUNTER read (at $9)
RDRF	TRCSR read (at $11)	RDR read (at $12)
ORFE	TRCSR read (at $11)	RDR read (at $12)
TDRE	TRCSR read (at $11)	TDR write (at $13)



The one cycle skew for $\overline{IRQ1}$ results from signal conditioning and synchronization.

FIGURE 3 — INTERRUPT RECOGNITION WINDOWS

g. *Regardless of how interrupts are caused, the end interface between each interrupt request and the processor is level controlled, as shown in Figure 2 (as the Interrupt Vector Select Logic block).* This feature gives an MC6801 program more control over interrupt service than is otherwise possible. For example, if the three timer interrupts were enabled and their flags were to simultaneously set, the input capture interrupt (having the highest priority of the three) would be serviced first. This service routine could temporarily inhibit compare interrupt service by clearing bit EOCI, which allows overflow interrupt service (normally the lower priority) to occur when capture service is complete. If the end interrupt request interface was latch rather than level controlled, clearing bit EOCI in the example would not prevent the compare interrupt from being serviced before timer overflow.

Individual flag bits are separately latched, however. In the example just given, bit OCF is temporarily inhibited but will indeed be serviced when the program restores bit EOCI to its enable state.

h. *Interrupt requests trigger interrupt service at times well defined relative to the end of the instruction in progress, as shown in Figure 3.*

i. *After recognition, all interrupts are initiated by a twelve-cycle interrupt entry sequence (see Figure 4).* The particular request that initiates the interrupt entry sequence will normally be, but is not always, the same one immediately serviced. Exceptions can occur where two or more interrupts occur at nearly the same time, because actual selection of which interrupt to service is delayed until near the end of the resulting interrupt entry sequence. At the ninth cycle, a decision is made as to whether $\overline{NMI}$, $\overline{IRQ1}$, or $\overline{IRQ2}$ will be serviced. If $\overline{IRQ2}$ is selected, the exact selection of which $\overline{IRQ2}$ to service is made during the tenth cycle. Requests not selected remain pending but are masked (I-bit sets during the tenth cycle), allowing the selected service routine to proceed undisturbed. Some example patterns of near-coincidental interrupt service are shown in Figure 5.

j. *Interrupt service is complete when the processor executes an RTI instruction.* This ten cycle instruction simply returns seven bytes from the stack to the processor registers, restoring the original machine state present when the interrupt was serviced (assuming the interrupt routine does not modify stack contents). In particular, the original I-bit is restored. If it returns to a logic "0", the $\overline{IRQ1}$ and $\overline{IRQ2}$ latches of Figure 2 are again enabled so that any pending request can be serviced.

k. *A CLI instruction can be executed during interrupt service to allow prompt processor response to pending $\overline{IRQ1}$ or $\overline{IRQ2}$ requests.* The benefits gained by this are sometimes offset by increased program complexity and greater required stack depth.

l. *All interrupt service routines (except $\overline{NMI}$ and SWI) should take action that removes its interrupt request prior to executing an RTI instruction.*

An $\overline{IRQ2}$ or $\overline{IS3}$ or $\overline{IRQ1}$ request is normally removed by clearing the appropriate flag bit. As an alternative, the matching enable bit can be cleared. External hardware must remove any external $\overline{IRQ1}$ interrupt requests, as this line is not directly controlled by the processor. This is best handled by providing handshake logic similar to that used internally to control the $\overline{IRQ2}$ requests. The MC6821 PIA and MC6846 RIOT devices each provide an excellent $\overline{IRQ1}$ interface, though discrete logic designs will also work.

m. Interrupt service cycle times are well defined:

C_{serv}: Number of cycles taken away from non-interrupt execution by interrupt execution.

C_{entry}: 12 cycles to enter interrupt service.

C_{clrflg}: 4 to 9 cycles to clear interrupt request (zero for NMI or SWI)

C_{task}: number of cycles to perform desired service.

C_{exit}: 10 cycles to execute RTI instruction.

$C_{\overline{IRQ1},\ IRQ2} = C_{task} + 26$ to 31 cycles

$C_{\overline{NMI},\ SWI} = C_{task} + 22$ cycles

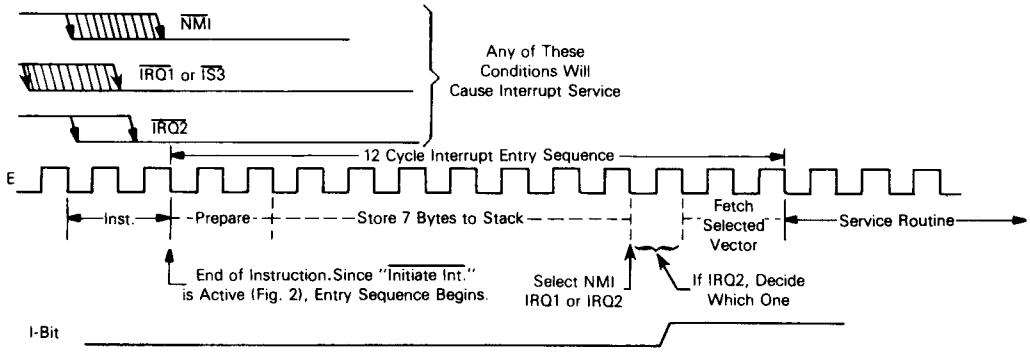
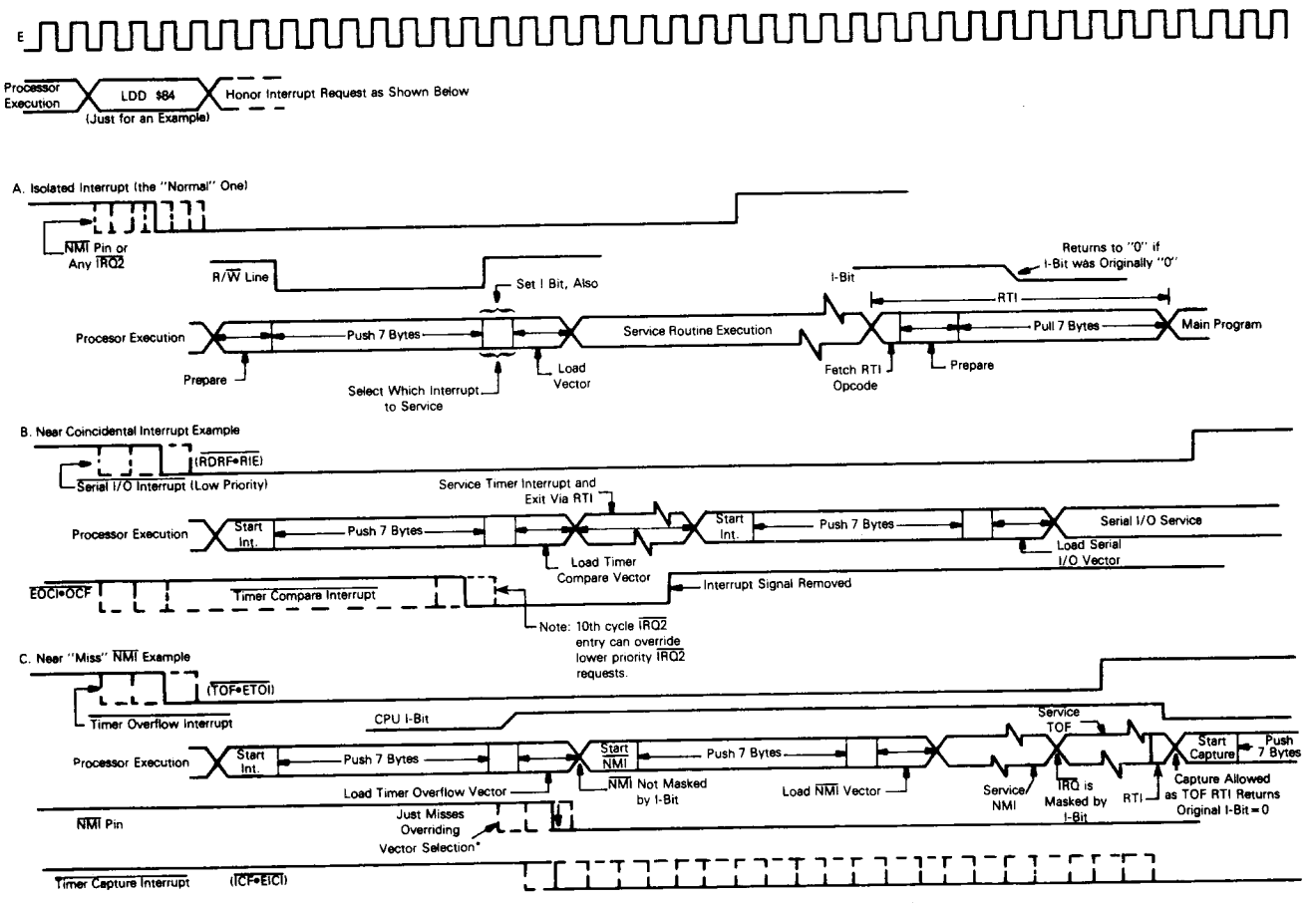


FIGURE 4 — MC6801 INTERRUPT ENTRY SEQUENCE



*Note: NMI or IRQ1 requests must be present by end of 9th cycle to participate in vector selection.

FIGURE 5 — MC6801 INTERRUPT EXAMPLES

DESIGN CONSTRAINTS OF THE MC6801 INTERRUPT SYSTEM

The expanded interrupt system of the MC6801 offers important benefits when software is constructed to utilize it properly. However, certain specific software practices should be avoided because unexpected program behavior may result. These practices are now described, along with alternatives that will aid in better achieving the desired results.

AVOID $\overline{IRQ2}$ HANDSHAKE VIOLATIONS

MC6801 interrupt requests (except for NMI and SWI) are cleared with a software handshake during interrupt service to avoid repetitive service of the same interrupt. The programmer should avoid several improper procedures that can clear these requests at the wrong time, or several difficulties may occur that can cause unexpected system performance.

What happens if an $\overline{IRQ2}$ request is somehow removed prior to actual service (a handshake violation)? If the request had not yet triggered an interrupt entry sequence, nothing unusual takes place. If indeed triggered, however, the following rule will apply: *An $\overline{IRQ2}$ interrupt entry sequence that finds no request present during its tenth cycle will always select the serial I/O vector for service.* This may or may not be a problem if the original request was for serial I/O service. On the other hand, programs that allow $\overline{IRQ2}$ requests to be cleared between interrupt sequence triggering and actual vector selection will service the serial I/O vector in lieu of that desired. Two methods exist that allow this to occur, which are described below and then summarized in Table 2.

1. *Clearing $\overline{IRQ2}$ Enable Bits While I-bit is Clear* — Programs are often structured such that mask-bit I is clear during background or non-interrupt execution. Some programs will also purposely clear the I-bit during interrupt service routines. At either time, software that clears an $\overline{IRQ2}$ enable bit should be avoided because the corresponding interrupt flag may have just become set. Figure 6 shows that an $\overline{IRQ2}$ interrupt only momentarily requested can result in erroneous selection of the serial I/O vector. To prevent this, use instruction SEI to mask all interrupt requests for the short time that it takes to clear the desired enable bit, then clear the I-bit again with instruction CLI. The SEI/CLI combination is unnecessary when the programmer knows that the I-bit is already set, as is usually true within interrupt service routines that do not themselves alter the I-bit.
2. *Clearing Enabled $\overline{IRQ2}$ Flag Bits while I-bit is Clear* — $\overline{IRQ2}$ requests can also be removed by clearing the interrupt-flag itself. Doing so just as the interrupt is to be serviced should be avoided to prevent improper serial I/O vector selection, as demonstrated in Figure 7a.

   Two special cases of programming practice can also generate this undesirable result. The double-byte read instructions "LDD TCSR" ($8) and "LDD TRCSR" ($11) are used to arm and clear interrupt flags TOF, RDRF, and ORFE. As such, they are excellent for use as the software handshake needed during service of these flags, but altogether improper any time their interrupts are enabled and the I-bit is clear.

   For example, TOF might set, arm, and clear within the four cycles of "LDD TCSR" execution. Though the request is removed, it is still able to initiate an interrupt entry sequence, resulting in erroneous service of the serial I/O routine (see Figure 7b). Good programming practice would clear interrupt flags only during the appropriate service routine, which is the best solution to this difficulty. "LDD TRCSR" can similarly clear RDRF and/or ORFE while simultaneously initiating an interrupt sequence. Again, the serial I/O vector is selected, which is seemingly proper in this special case. However, the serial interrupt service routine normally polls flags RDRF, ORFE, and TDRE to determine the actual interrupt source. It is possible, then, that RDRF or ORFE service be skipped due to improper flag-clearing.

Table 2 summarizes the several methods by which the serial I/O vector may be improperly selected.

TABLE 2 — METHODS OF GENERATING IMPROPER SERIAL I/O VECTOR SELECTION

The Cause	Control or Flag Bits Affected	The Solution
Clearing IRQ2 enable bit just as interrupt entry sequence begins.	EICI EOCI ETOI TIE RIE	Disable these enable bits only while I-bit is set.
Clearing IRQ2 flags just as interrupt entry sequence begins	All IRQ2 Flags	Do not clear flags directly after CLI instruction.
	TOF RDRF ORFE	Execute these instructions only if I-bit is set. LDD TCSR LDD TRCSR LDX TCSR LDX TRCSR ADDD TCSR ADDD TRCSR SUBD TCSR SUBD TRCSR CPX TCSR CPX TRCSR LDS TCSR LDS TRCSR

AVOID IRQ1 HANDSHAKE VIOLATIONS

$\overline{IRQ1}$ requests are latched as long as the I-bit is clear (see Figure 2) and will not cause improper selection of the serial I/O vector. However, it is still wise to observe the precautions described for $\overline{IRQ2}$ to prevent any unexpected system performance. For example, handshake violations can clear $\overline{IRQ1}$ request flags just as interrupt service is being initiated. As with IRQ2, programmers should avoid clearing $\overline{IRQ1}$ flags during an instruction that follows CLI. Any of the "LDD-type" violations described previously should also be avoided any time the I-bit is clear, for $\overline{IRQ1}$ flags can also set, arm and clear during a single instruction. These violations allow $\overline{IRQ1}$ service to take place, but prevent recognition of the calling flag during IRQ1 polling.

Additionally, the MC6821 and MC6850 offer interrupt request flags that need not be "armed" before clearing — a single memory access does the job. Therefore, limit these accesses to the appropriate service routine so that no request can be missed.

There is no hardware oriented reason to avoid clearing $\overline{IRQ1}$ interrupt enable bits while the I-bit is clear. However, a polling routine cannot reliably test both flag and enable bits when this is the case.

Pulsing the external $\overline{IRQ1}$ line by any form of signal generator without a handshake should normally be avoided.* Edge triggered interrupt lines $\overline{NMI}$, IS3, and Input Capture are better used for such signals. Or, an MC6821 or MC6846 can transform these into level-sensitive, handshake controlled request signals which are more suitable for $\overline{IRQ1}$.

AVOID CLEARING THE I-BIT DURING NMI SERVICE

There is need to be cautious about clearing the I-bit during $\overline{NMI}$ service because this interrupt can occur at virtually any point in program execution. Some programs that use this technique are likely to service occasional $\overline{IRQ1}$ or $\overline{IRQ2}$ interrupts twice per request.

Double service occurs whenever an I-bit clearing $\overline{NMI}$ service routine is executed before the flag-clearing handshake of an already entered $\overline{IRQ1}$ or $\overline{IRQ2}$ service routine. For example, Figure 8 shows that an $\overline{NMI}$ occurrence during a particular window of time prevents the quick handshake that clears ICF. When $\overline{NMI}$ service executes instruction CLI, flag ICF teams with enable bit EICI to again request capture service. As shown, all routines will execute properly and to completion, including *double* service of the twice-called capture routine.

Clearing the I-bit during other service routines will not generate this situation, although doing so before clearing the calling interrupt request is disastrous. The best way to avoid any problem is to leave the I-bit set throughout $\overline{NMI}$ service. Where this is undesirable, additional software can be added to the $\overline{NMI}$ routine stack and compare it to all possibilities that lead to double service. Where such is indicated, clearing the I-bit should be skipped. If the I-bit must be cleared every time, additional software should first clear the interrupt flag scheduled for double service. Clearly, the benefits desired when clearing the I-bit during $\overline{NMI}$ service are potentially offset by the added software required to support this technique. For the same reasons, do not program an $\overline{NMI}$ interrupt service routine to clear the I-bit record contained on its stack. This would allow all portions of a program to be subjected to I-bit clear execution, resulting in potential double service of interrupts.

*Appendix A offers an application of $\overline{IRQ1}$ pulsing that does work, but only under special circumstances.

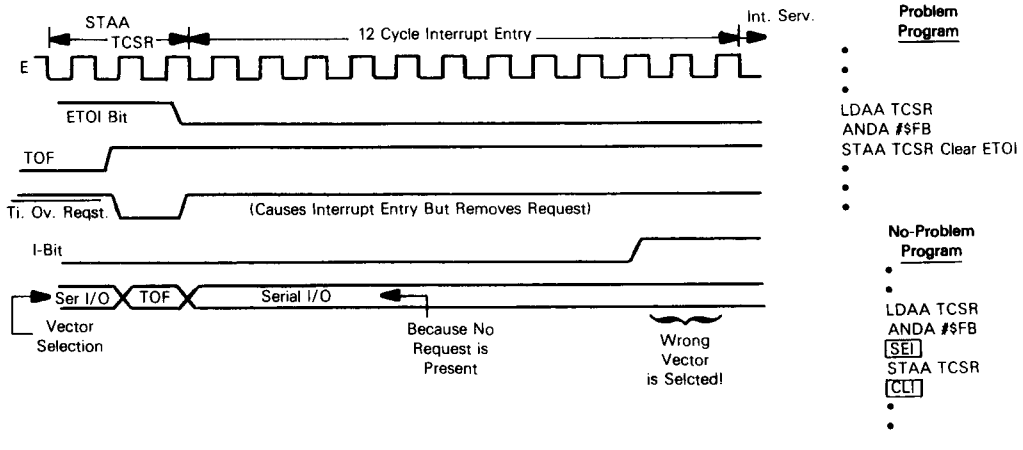
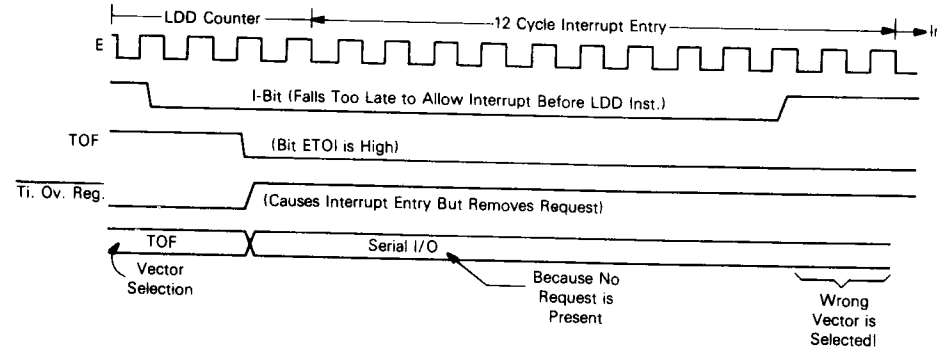


FIGURE 6
Programs that clear $\overline{IRQ2}$ enable bits while I-bit is clear risk improper vector selection.

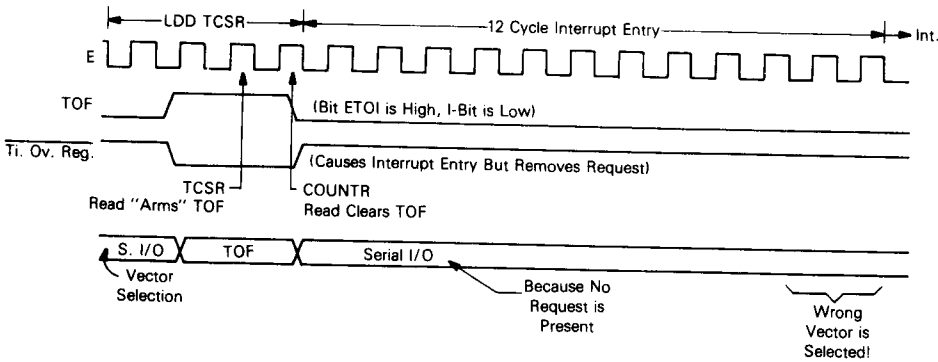


7a Problem Program

```
*ETOI bit is set
 •
 SEI   Mask Interrupts
 •
 •
*Assume that TOF sets in here
 •
 LDAA TCSR  arms TOF
 •
 STAB $80
 CLI
*I-bit clears in 1st cycle of
 next instruction
 LDD COUNTR  clears TOF
 •
 •
```

No-Problem Program

```
*ETOI bit is set
 •
 SEI
 •
*TOF sets here
 •
*LDAA TCSR
 •
*CLI          } or: STAB $80
 STAB $80}          CLI
                    NOP
*Timer Overflow Interrupts
 LDD COUNTR
```



7b Problem Program

```
*I-Bit is clear
*ETOI bit is set
 •
 •
 •
 LDD TCSR
 •
 •
 •
```

No-Problem Program

Best Solution: Do not write code that allows flag clearing outside of interrupt service routine!

Poor Alternatives:

```
1. •
   •
   LDAA TCSR
   LDAB COUNTR
   •
   •
2. •
   •
   SEI
   LDD TCSR
   CLI
   •
   •
```

FIGURE 7
Programs that clear "enabled" IRQ2 flag bits while I-bit is clear risk improper vector selection.

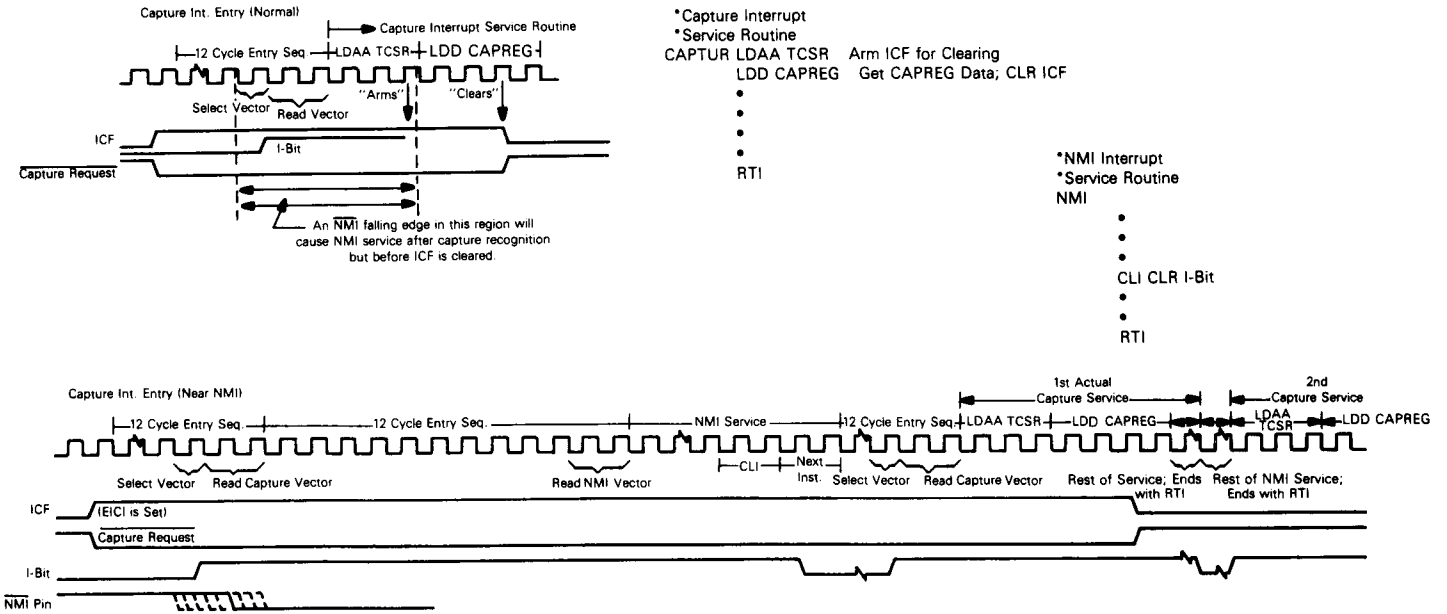


FIGURE 8
Clearing I-bit during NMI service can lead to double service of IRQ1 or IRQ2 interrupt.

APPENDIX A
USING IRQ1 AS AN INPUT PIN

If the circumstances are right, an I/O limited MC6801 system may be able to use the $\overline{IRQ1}$ pin as an extra input. Where no other interrupts are used, this can be accomplished with the simple program of Figure A-1 to clear RAM byte IRQTST while also clearing the I-bit. The state of the $\overline{IRQ1}$ pin then determines whether IRQTST will be changed (interrupt occurs) or remain constant (no interrupt occurs). The background program discovers which is the case by simply reading IRQTST. Notice that the processor has no control of input pin $\overline{IRQ1}$ in this method, but can still perform the necessary interrupt handshake by setting the I-bit record stored on the stack. This prevents repeat service that would otherwise tie up the processor as long as the $\overline{IRQ1}$ pin is held low.

The same basic method can also be used when other interrupts are to be serviced as well. The $\overline{IRQ1}$ pin is again tested in the manner just described, but now routine IRQSRV must also poll other interrupt requests in case they need service, as shown in Figure A-2. If it is important that the various interrupts be serviced promptly, the programmer can scatter CLI instructions through his background software. This still allows the $\overline{IRQ1}$ pin to be used as an input, and also permits normal service of all interrupts while $\overline{IRQ1}$ is high. Whenever $\overline{IRQ1}$ is low, IRQSRV becomes an alternate entry path for other maskable interrupt requests.

```
                0080  A IRQTST EQU    $80       RAM BYTE AT $0080
                        *BACKGROUND PROGRAM, I-BIT IS SET.
2000                    ORG    $2000
                2000  A BKGRND EQU    *
                        *       .
                        *FIND OUT IF IRQ1 PIN IS HIGH OR LOW
2000  0E                CLI             I CLRS DURING NEXT INST.
2001  7F  0080  A       CLR    IRQTST   WILL IT STAY ZERO?
                        *IF IRQ1 IS LOW, SERVICE OCCURS AT THIS MOMENT
2004  96  80    A       LDAA   IRQTST   IS NOW #$FF IF IRQ1 IS LOW
2006  26  00 2008      BNE    IRQLOW   IRQ1 DID OCCUR
                        *IRQ1 PIN WAS HIGH
                        *       .
                2008  A IRQLOW EQU    *        IRQ1 PIN WAS LOW
                        *       .
2008  7E  2000  A       JMP    BKGRND   END OF BACKGROUND LOOP


                      *IRQ1 SERVICE ROUTINE
200B  73  0080  A IRQSRV COM    IRQTST   CHANGE IRQTST!
200E  30                TSX             X=SP+1
200F  A6  00    A       LDAA   0,X      THE CCR BYTE ON STACK
2011  8A  10    A       ORAA   #$10     SET I-BIT FOR RETURN
2013  A7  00    A       STAA   0,X
2015  3B                RTI
                    *I-BIT IS SET TO PREVENT IRQ1 RESERVICE
```

FIGURE A-1
Using $\overline{IRQ1}$ as an input pin.

```
                0080  A TCSR    EQU     $8        TIMER C/S REGISTER
                000D  A CAPREG  EQU     $D        CAPTURE REGISTER
                0011  A TRCSR   EQU     $11       TX/RX C/S REGISTER
                0012  A RXBUF   EQU     $12       RECEIVE BUFFER
                0080  A IRQTST  EQU     $80       RAM BYTE AT $0080
2000                            ORG     $2000
                              *IRQ1 SERVICE ROUTINE
2000  73        0080  A IRQSRV  COM     IRQTST    CHANGE IRQTST!
2003  30                        TSX               X=SP+1
2004  A6        00    A         LDAA    0,X       THE CCR BYTE ON STACK
2006  8A        10    A         ORAA    #$10      SET I-BIT FOR RETURN
2008  A7        00    A         STAA    0,X
                              *BEFORE RTI, SEE IF OTHER INTERRUPTS ARE PENDING
200A  96        08    A         LDAA    TCSR      CHECK INPUT CAPTURE
200C  2B        09 2017         BMI     TIMIC1    TIMER INPUT CAPTURE PENDING
200E  DC        11    A         LDD     TRCSR     CHECK SCI IRQ2 REQUESTS
2010  85        E0    A         BITA    #$E0      CHK RDRF,ORFE,TDRE FLAGS
2012  26        08 201C         BNE     SCIIN2    SERVICE SCI INTERRUPT
2014  3B                        RTI               EXIT:NO INTERRUPTS PENDING
                              *I-BIT IS SET TO PREVENT IRQ1 RESERVICE

2015  96        08    A TIMIC   LDAA    TCSR      ARM ICF FOR CLEARING
2017  DC        0D    A TIMIC1  LDD     CAPREG    CLR ICF, GET CAPTURE DATA
                              *         .
2019  3B                        RTI

201A  DC        11    A SCIINT  LDD     TRCS      ACCA=TRCS, ACCB=RXBUF
201C  48                  SCIIN2 ASLA              SORT OUT SCI FLAGS
                              *         .
201D  3B                        RTI
```

FIGURE A-2
Routine IRQSRV can also poll other interrupt requests when using $\overline{\text{IRQ1}}$ as an input.