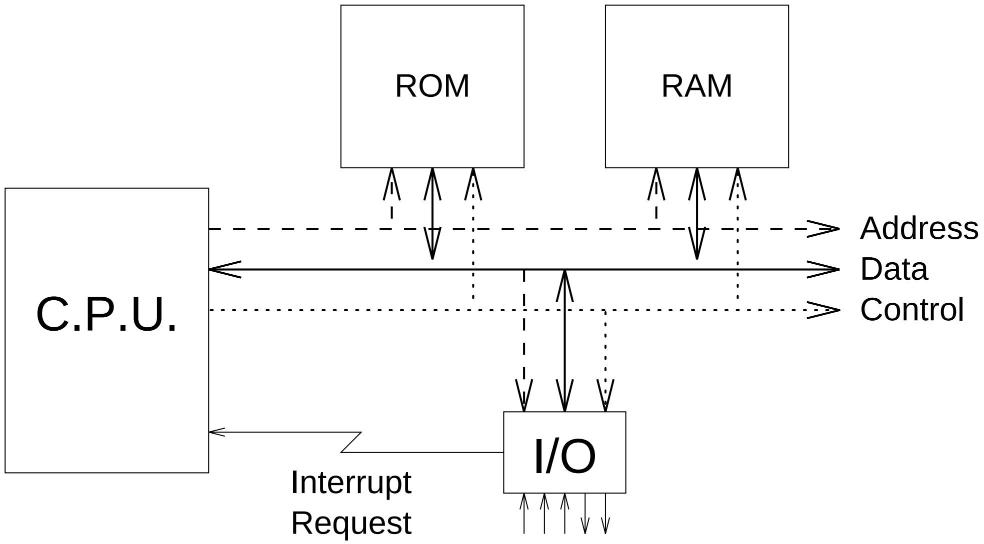
Interrupts

In order to provide faster response to external events, while avoiding inefficient *busy wait*, most CPUs provide an interrupt request line for use by I/O devices.



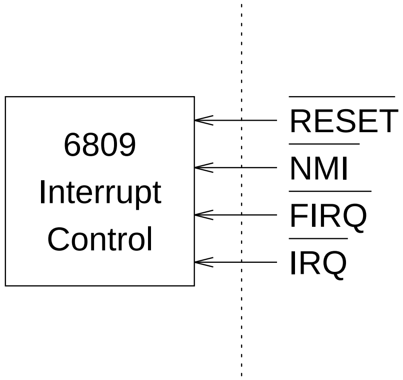
The assertion of the interrupt request line by an I/O device should initiate an interrupt cycle.

Interrupt Cycle

- Disable interrupts
- Store PC
- Jump to service routine
  - Determine source of interrupt
  - Service interrupt
  - Clear interrupt
  - Return from interrupt
- Enable interrupts

6809 Interrupts

The 6809 processor provides a number of interrupt request lines with differing functionality.



The simplest is the Fast Interrupt Request line, $\overline{FIRQ}$.
We shall consider the interrupt cycle initiated by asserting $\overline{FIRQ}$.

6809 Interrupt Cycle

- `FIRQ`
  - Push PC onto system stack

    $\mathtt{SP}\Downarrow\ \mathtt{PC}$

    $\mathtt{SP}\Downarrow\ \mathtt{PC}_L$

    $\mathtt{SP}\Downarrow\ \mathtt{PC}_H$
  - Push CCR onto system stack

    `Clear E`[1]

    $\mathtt{SP}\Downarrow\ \mathtt{CCR}$
  - Disable interrupts

    `Set F,I`
  - Jump to service routine

    $\mathtt{PC} \leftarrow \mathtt{(FFF6):(FFF7)}$

---

[1] E = 0 indicates a *fast* interrupt.

6809 Interrupt Cycle

- `RTI`
  - Restore CCR (and hence enable interrupts)

    `SP⇑ CCR`
  - Check for fast interrupt

    `IF (E == 0)`

    - - Restore PC

      `SP⇑ PC`

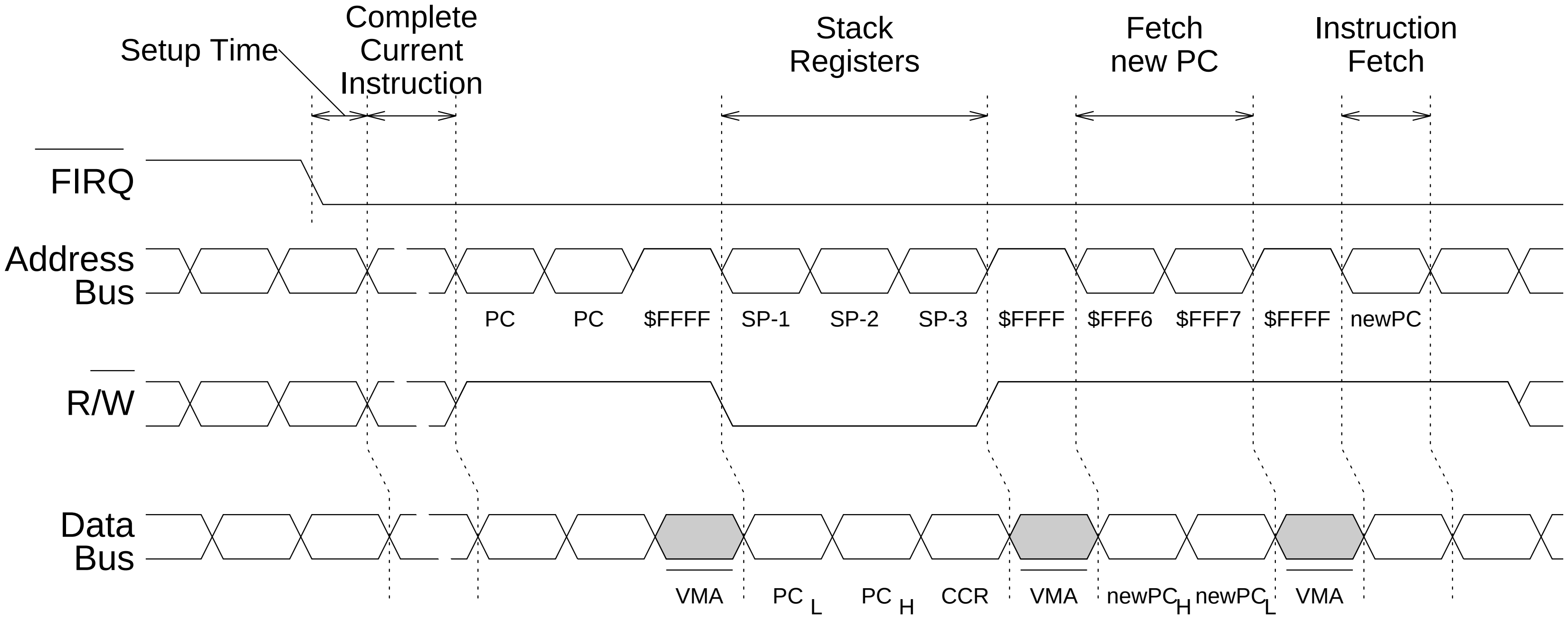
      SP⇑ PC_H
      SP⇑ PC_L
  - Otherwise . . .

    ```
    IF (E == 1)
        . . .
        . . .
    ```

6809 Interrupt Cycle

The diagram below illustrates the sequence of events for an FIRQ request including:

- The stacking of the program counter and condition code registers.
- The indirection via the FIRQ interrupt vector, (FFF6):(FFF7).



Interrupt Cycle

Use of registers

The interrupt should not affect the interrupted code. Thus any registers used by the interrupt service routine must be saved.

Where this is not done automatically by the processor it is the responsibility of the service routine.

- **Store registers**
- *Determine source of interrupt*
- *Service interrupt*
- *Clear interrupt*
- **Restore registers**
- *Return from interrupt*

6809 Interrupt Cycle

- `IRQ`
  - Push PC onto system stack

    `SP⇓ PC`
  - Push other registers onto system stack

    `SP⇓ U,Y,X,DPR,B,A`
  - Push CCR onto system stack

    `Set E`[2]

    `SP⇓ CCR`
  - Disable interrupts

    `Set I`
  - Jump to service routine

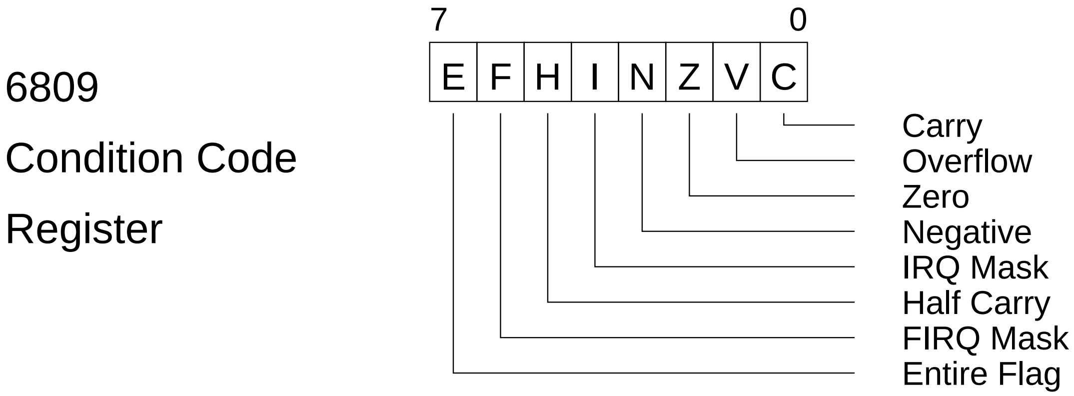
    `PC ← (FFF8):(FFF9)`

---

[2] E = 1 indicates that the *entire* state has been saved. RTI checks this flag and restores registers accordingly.

6809 Interrupt Priority

Each service routine automatically disables interrupts of the same or lesser priority.



- IRQ *low priority*

  sets `I=1` disabling only IRQ requests.

- FIRQ *high priority*

  sets `F=1` and `I=1` disabling FIRQ and IRQ requests.

Masks are reset when CCR is restored from stack by RTI.

6809 Interrupt Priority

The 6809 supports a third interrupt line:

- NMI *highest priority*
  - Non-Maskable Interrupt Request
  - Saves entire state and sets `E=1` like IRQ.
  - NMI sets `F=1` and `I=1` disabling FIRQ and IRQ requests.
  - There is no interrupt mask for an NMI request.
  - An NMI event will interrupt any process (including another NMI service routine).

6809 Interrupt Priority

Non–Maskable Interrupts

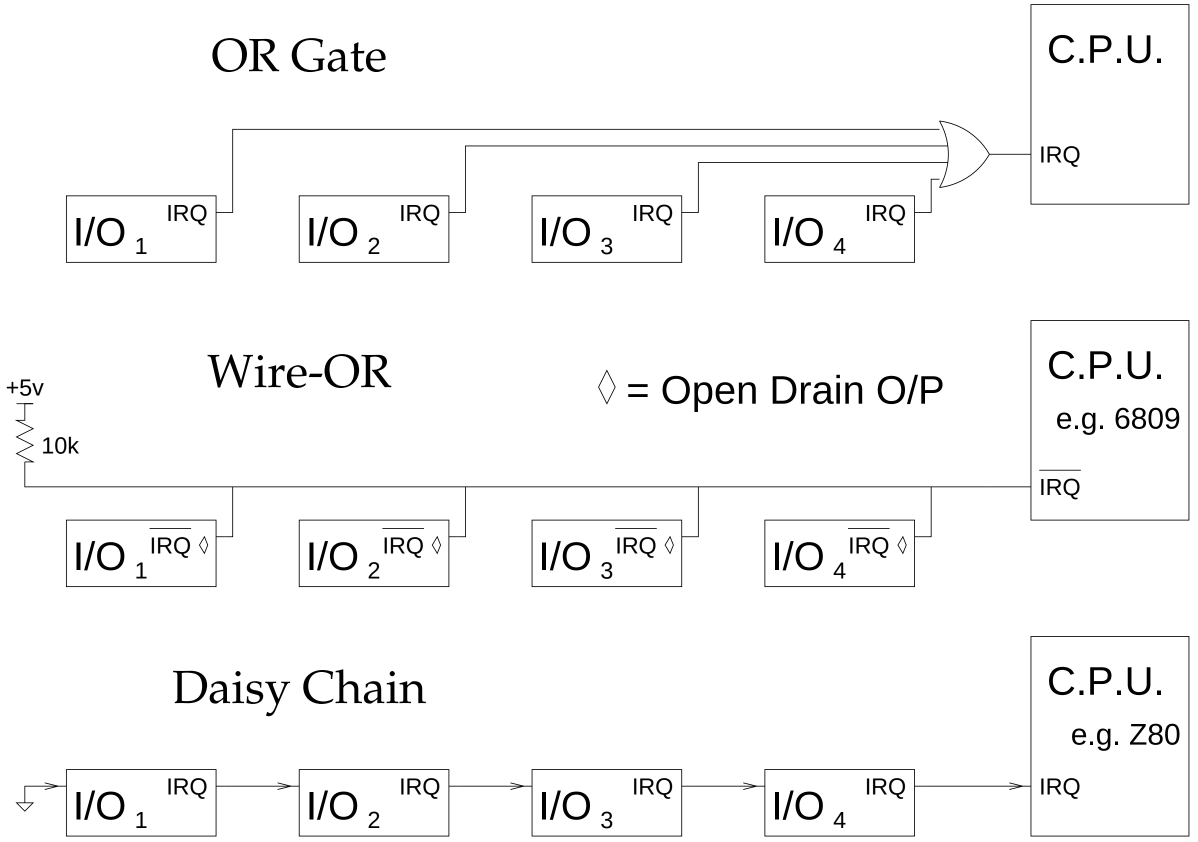
- This interrupt is used for the most urgent tasks, which cannot wait for other interrupt routines to complete.
- Since we cannot disable this interrupt, we must find an other method of preventing recursive service routine calls as a result of a single event.
  - The NMI event is triggered by a *falling edge* on the $\overline{NMI}$ line.
  - The action of clearing the interrupt will enable subsequent interrupts.
  - We must consider the possibility of overlapping NMI requests[3].

---

[3] e.g. poll NMI devices again after clearing the source of an interrupt

Sharing an Interrupt Line

It is seldom possible to have one interrupt line per I/O device.



Note: The wire-OR here is actually a wire-AND of negated signals

Sharing an Interrupt Line

- Determining the source of an interrupt

  Polling

  - The processor must poll each I/O device[4] connected to the line.
  - This may be very time-consuming where many devices share the same IRQ line.

  Priority

  - The order of polling determines the priority where multiple devices are awaiting service.

---

[4] i.e. read the I/O device *status register*

Sharing an Interrupt Line

Nested interrupts

In order to avoid blocking important events we may wish to re-enable interrupts on the same line before an interrupt routine completes.

- *Determine source of interrupt*
- **Clear interrupt**
- **Enable interrupts**
- **Service interrupt**
- *Return from interrupt*

Note: Where request lines are level sensitive, we must clear the source of an interrupt before re-enabling interrupts in order to avoid an endless recursion.

Sharing an Interrupt Line

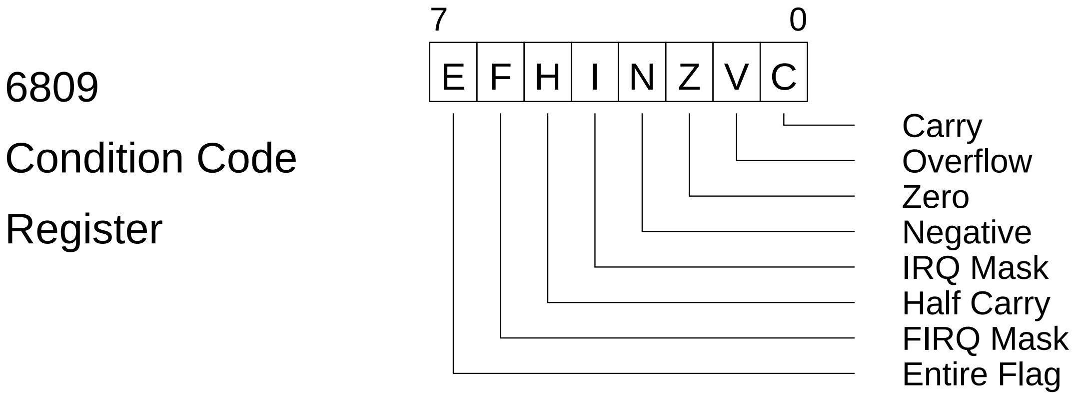
Nested interrupts

Alternatively, where one part of the routine is particularly time critical, interrupts may be enabled after this code has completed.

- *Determine source of interrupt*
- **Time critical interrupt service**
- *Clear interrupt*
- *Enable interrupts*
- **Rest of interrupt service**
- *Return from interrupt*

6809 Interrupt Masks

The 6809 allows direct manipulation of the control bits in the CCR.[5]



6809

Condition Code

Register

- Interrupt masks may be set or cleared explicitly by the user.
    - `ANDCC #10111111`$_2$ will enable FIRQ events.
    - `ORCC #01010000`$_2$ will disable IRQ & FIRQ events.

---

[5] Note that greater selectivity may be achieved by masking an interrupt at source, but this is rather dangerous; only the routine responsible for the management of an I/O device should manipulate its interrupt mask.

Interaction with I/O Device

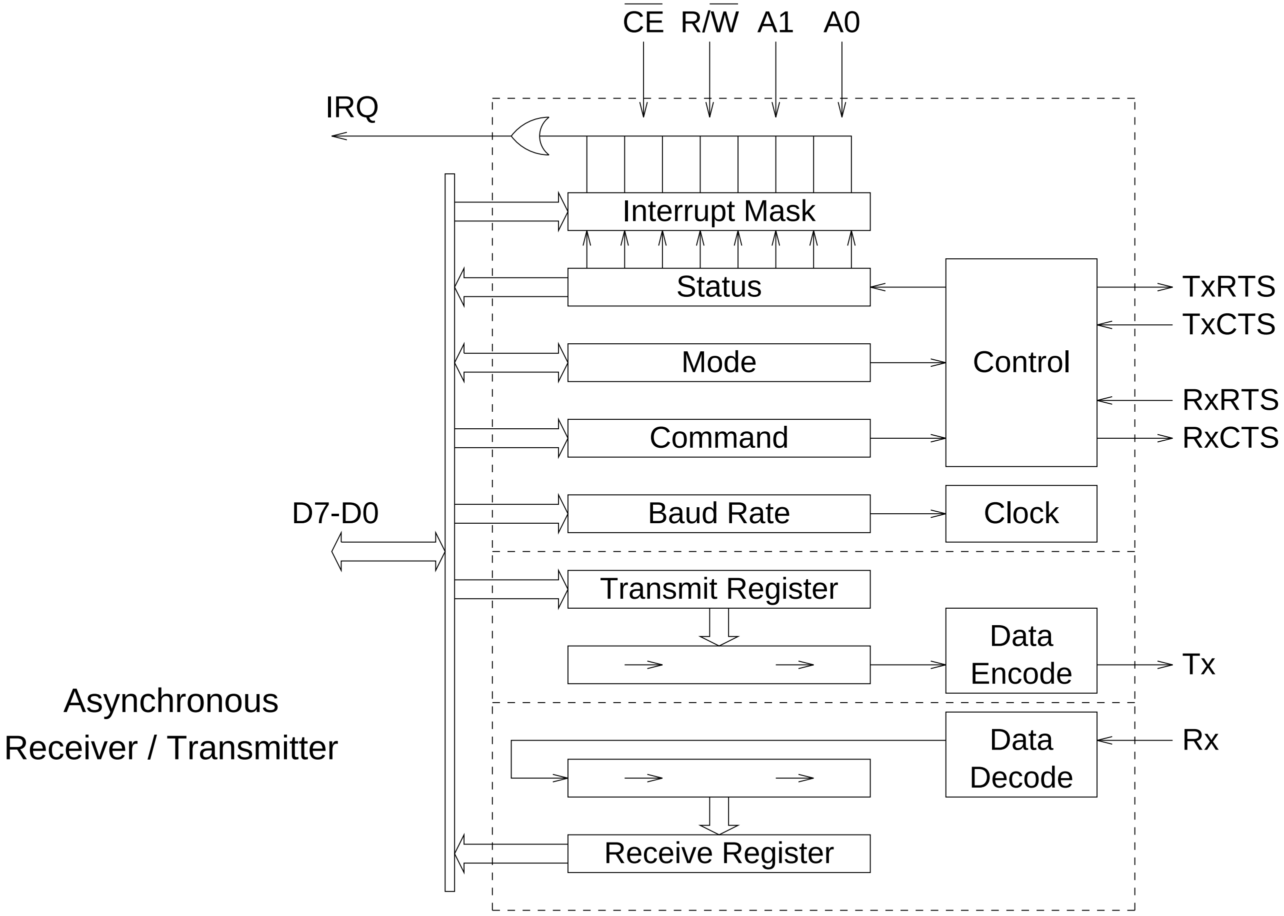
- Interrupt Request asserted by I/O Device
  - request will remain active until explicitly cleared
- Service Routine queries I/O Device
  - read from status register
    - whether the device interrupt is active
    - what is the cause of an active interrupt[6]
- Service Routine services interrupt
  - e.g. reads data from I/O device
- Service Routine clears interrupt
  - Device dependent – consult device data sheet

---

[6] a single device may have several interrupts active concurrently

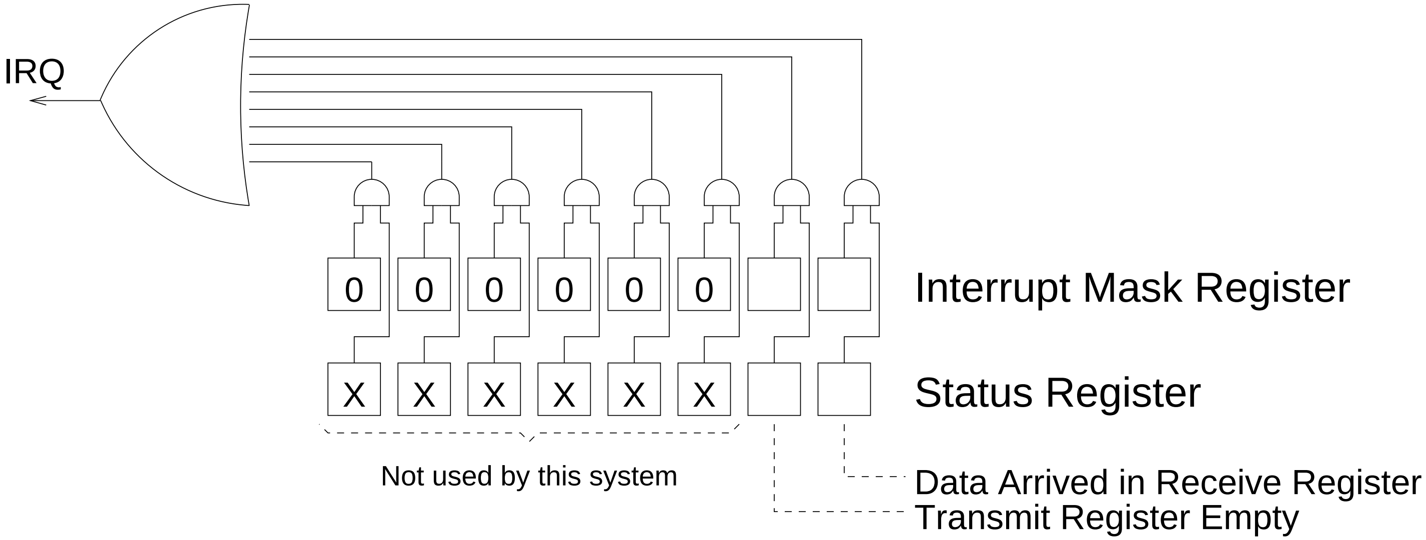
Clearing of Interrupts

Consider our imaginary asynchronous receiver/transmitter device:



Clearing of Interrupts

- The device generates the interrupt request from a logical AND of the Status Register and the Interrupt Mask Register.



- The device supports two interrupts
  - data arrived – needs reading
  - data transmitted – send more

Clearing of Interrupts

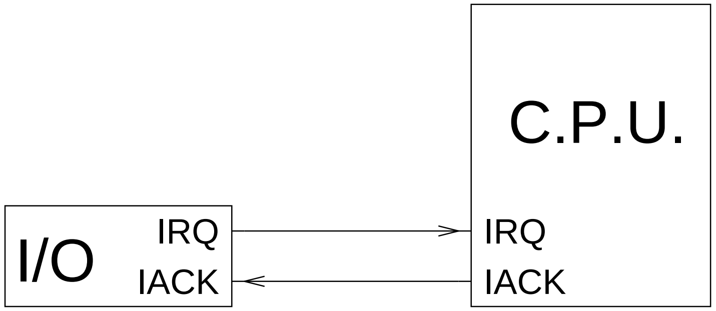
- Explicit clearing of interrupts

  We might clear an interrupt by writing to “Interrupt Clear” command to the Command Register.

- Implicit clearing of interrupts

Automatic Clearing of Interrupts

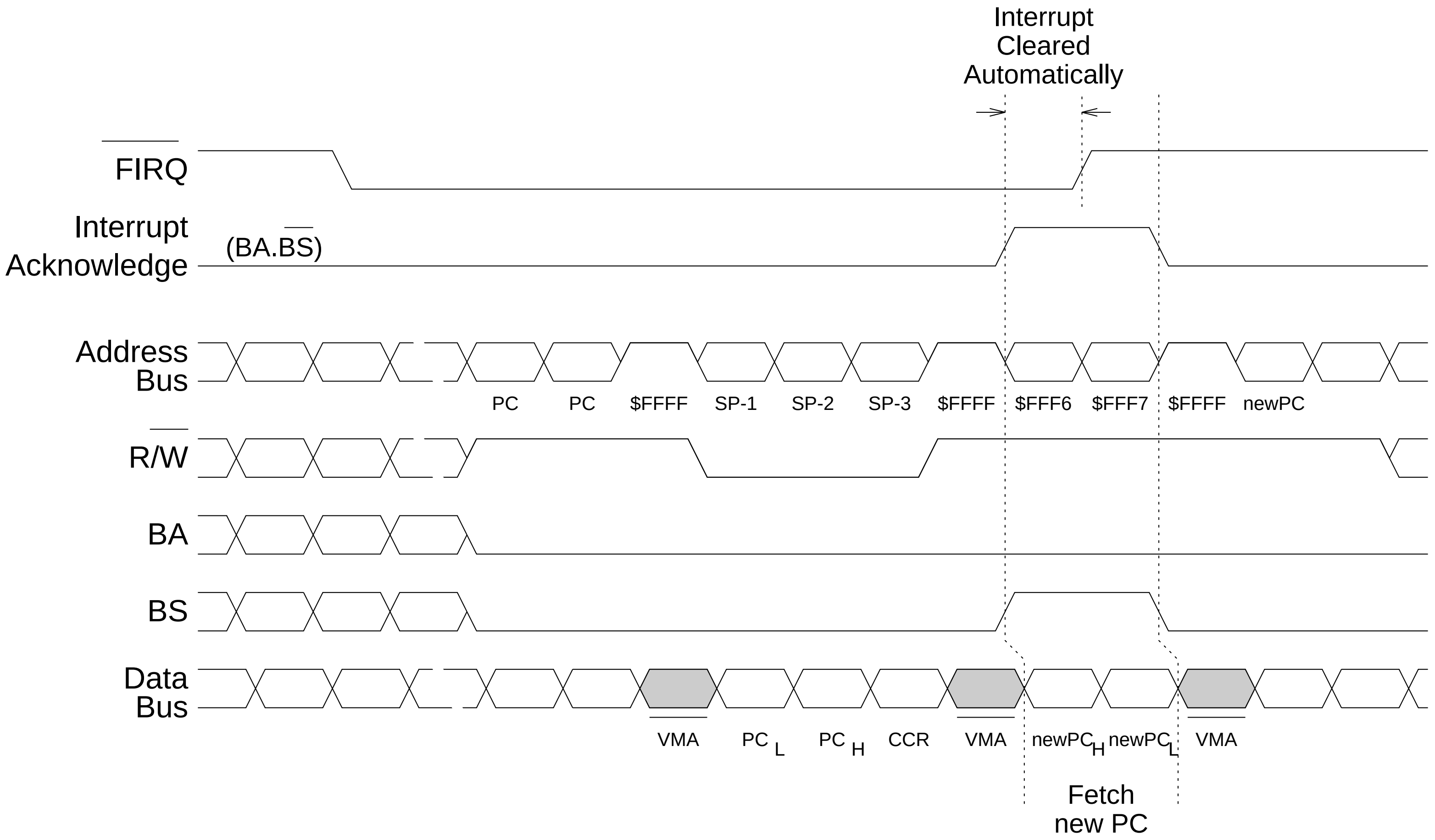
- Many I/O devices support an interrupt acknowledge input.
  - A pulse on the interrupt acknowledge line indicates that the interrupt routine has been started.



  - This enables an interrupt to be cleared at an early stage in the interrupt cycle; well before the interrupt is serviced.

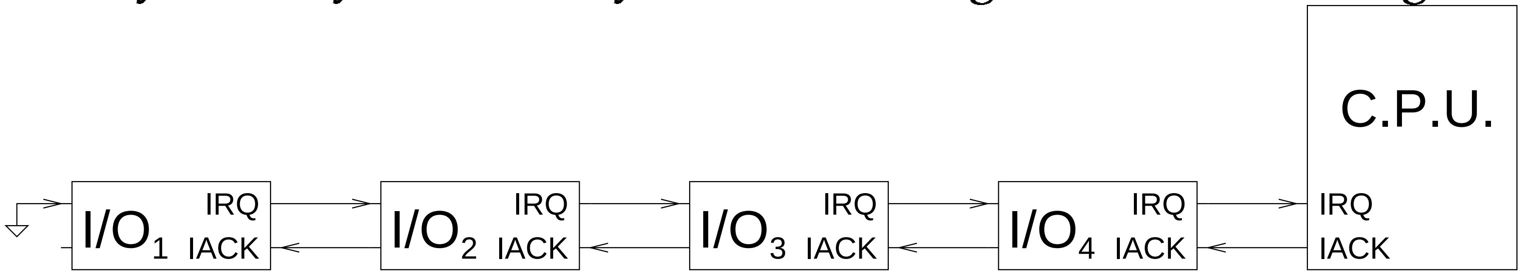
Automatic Clearing of Interrupts

- The 6809 supports an interrupt acknowledge indicated by the state of the Bus Available and Bus Status lines.



Automatic Clearing of Interrupts

A Z80 style Daisy Chain may be used to regulate acknowledge.



Pros:

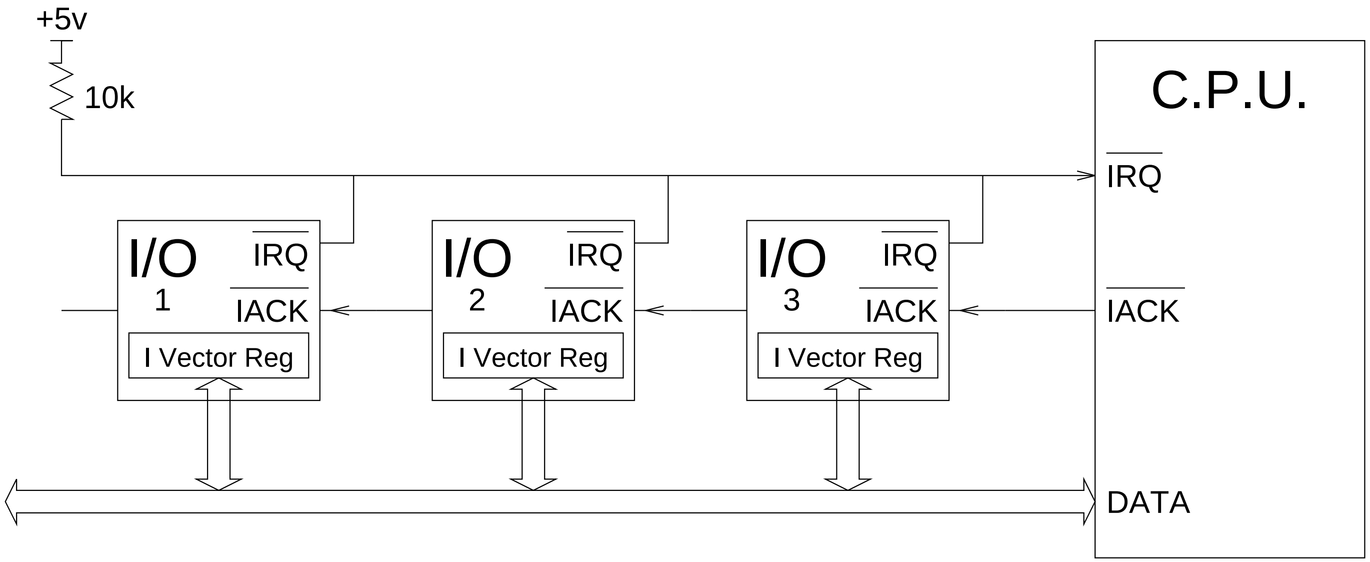
- We ensure that no more than one interrupt is acknowledged.
  - IACK is passed to next device only if local IRQ is inactive.

Cons:

- Must still poll to discover source of interrupt.
  - although the I/O devices know which interrupt is being serviced!
- Priority system is fixed in hardware.

Automatic Detection of Interrupt Source

Vectored Interrupts



- An alternative strategy allows the I/O device to identify itself during the IACK cycle.

Automatic Detection of Interrupt Source

Vectored Interrupts

- Initialization
  - I/O devices have vectors programmed at start up
- IACK cycle
  - IACK signal requests vector (typically 8 bits only)
  - Responding device places vector on data bus
  - C.P.U. grabs service routine address from vector table
- No requirement for device polling
- Priority scheme still fixed in hardware

6809 Vectored Interrupts

The 6809 supports different interrupt vectors for the different interrupt request lines:[7]

- FIRQ vector (FFF6):(FFF7)
- IRQ vector (FFF8):(FFF9)
- NMI vector (FFFC):(FFFD)

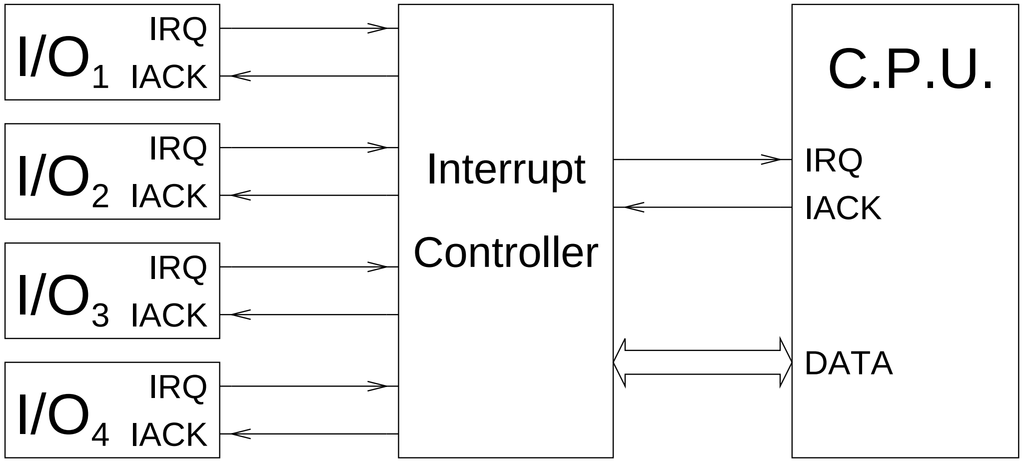
Since the 6809 IACK cycle corresponds to the reading of the interrupt vector, it is possible for the vector to be provided by the I/O device.

- This task is rather too complex for most I/O devices, we would need an Interrupt Controller.

---

[7] other vectors are used for the RESET input and the SWI instructions.

External Interrupt Controllers



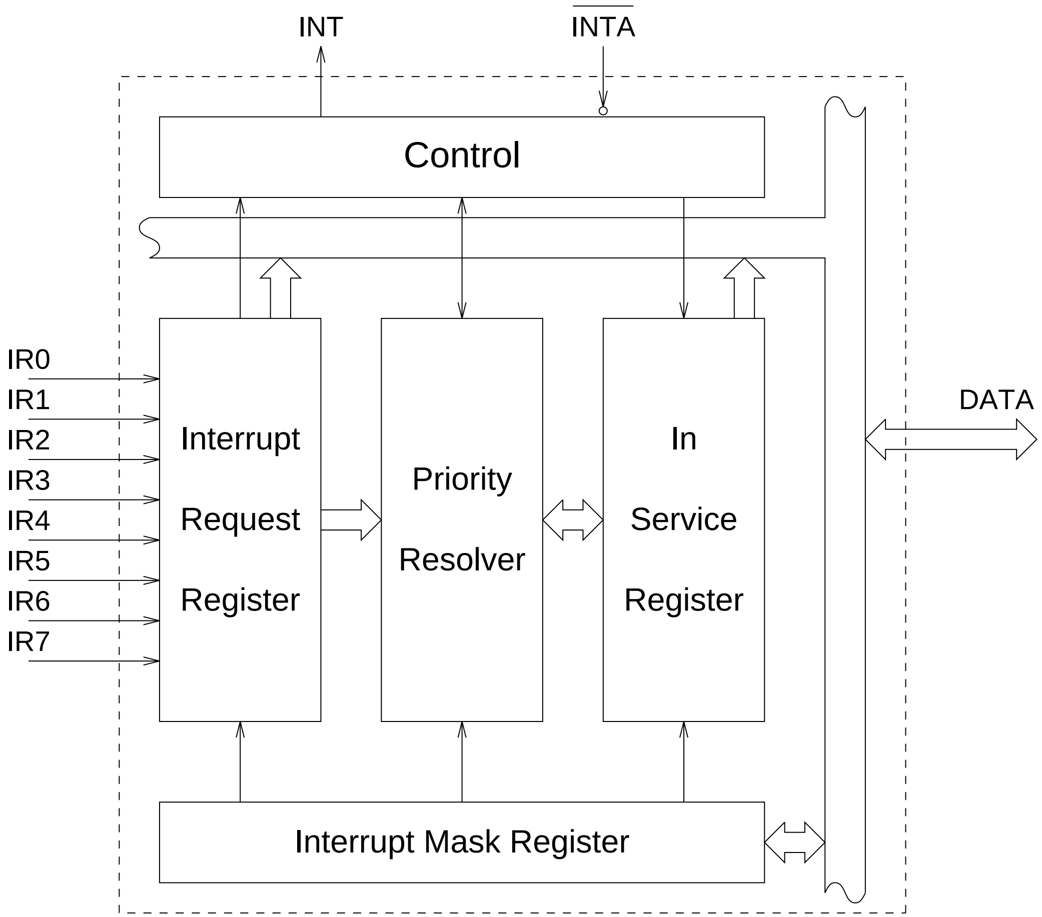
External interrupt controllers are used where we wish to deal efficiently with large numbers of I/O devices.

- The interrupt vector is provided by the controller during the IACK cycle.

  Thus we can make use of I/O devices which don't provide vector registers.

External Interrupt Controllers - 8259

The 8259 supports the 80x86 family of microprocessors:



External Interrupt Controllers - 8259

- Behaviour on interrupt
  - I/O device(s) assert one (or more) of IR0-7 inputs to request interrupt.
  - 8259 propagates interrupt to INT pin of CPU unless locally masked.
  - CPU performs an IACK cycle, asserting $\overline{INTA}$.
  - 8259 places 8 bit vector corresponding to highest priority request on data bus.
    - - most significant 5 bits are 8259 programmable vector
    - - remaining 3 bits indicate IR number

External Interrupt Controllers - 8259

- Flexible priority system
  - Any combination of inputs may be masked
  - Priority resolver has 3 modes of operation
    - - Fixed: IR0$_{highest}$ $\ldots \rightarrow \ldots$IR7$_{lowest}$
    - - Rotating: most recently serviced becomes lowest priority, rest rotate from there.
      e.g. IR4→IR5→IR6→IR7→IR0→IR1→IR2→IR3$_{mostrecent}$
    - - Specific: program lowest priority, rest as for rotating.
  - Priority is resolved during the IACK cycle.

External Interrupt Controllers

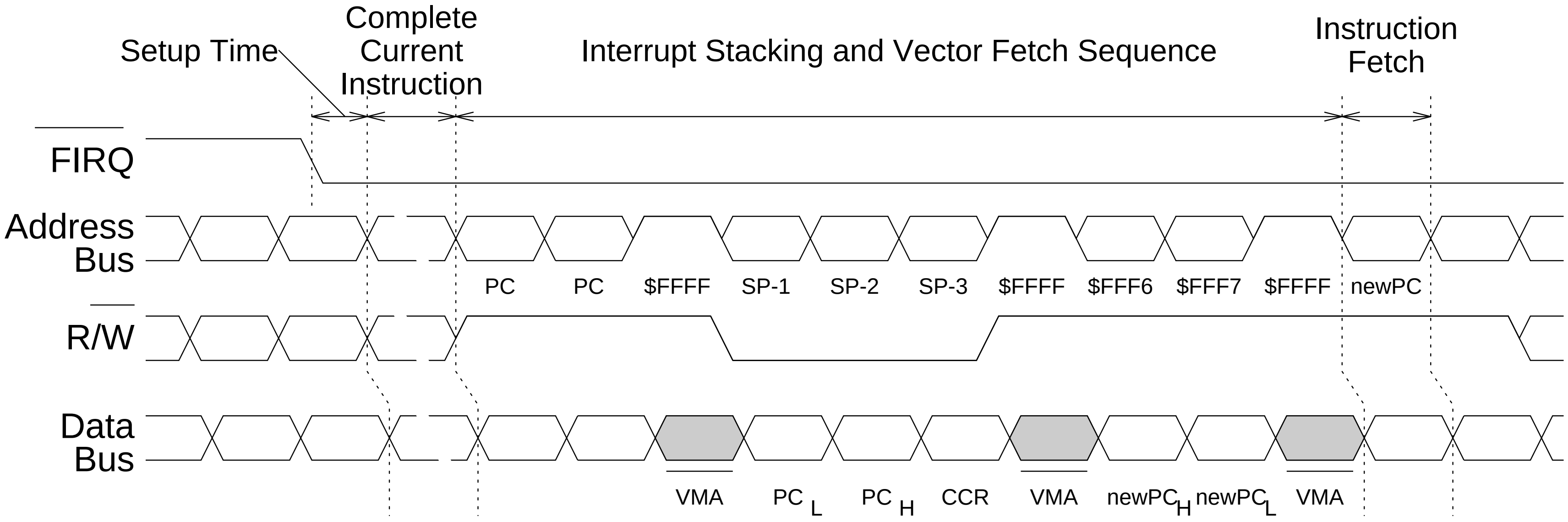
- Summary
  - CPUs provide only limited facilities for interrupt handling.
  - I/O devices do not always support CPU facilities.

  - Interrupt controllers provide hardware support for rapid interrupt servicing
    - Separate masking of I/O devices
    - Dynamically adjustable priority system
    - Efficient vector management

  - Essential for advanced Real-Time software

Response times

- Response time is the sum of:
  - Time to execute service routine code.
  - *Interrupt latency*.
    Interrupt latency is the *hidden* time between the IRQ event and start of the interrupt service routine.[8]



---

[8] i.e. the overhead associated with an interrupt request.

Guaranteed Response Times

A guaranteed response time must include:

- Worst case for variable delays
  - Time for SPARC context storage will depend on the availability of a free register window.
  - Even the time to complete the current instruction may sometimes be large.
- Delay due to nested interrupts.
  - Use a priority system
    - - thus ensuring that time critical interrupts are unaffected by those of lesser priority.
  - Keep all service routines as short as possible
    - - this benefits the interrupted routines as well as the service routine itself.