AN1461

Improved ESD immunity against interrupts lock-up

by Pascal Janiin

INTRODUCTION

In a working environment under heavy external interferences like high voltage spikes, the normal behaviour of a microcontroller (MCU) may be corrupted, leading to uncontrolled and unpredictable actions, and even permanent damages if they are not taken care of properly.

This is especially the case inside computer displays, where the vicinity of the very high voltages (up to 20KV) needed to drive the CRT tube, create transient spikes most notably during video mode changes. Those spikes superimpose themselves over the many signals generated by the MCU and, once applied upstream to the MCU pins, often interfere with the normal behaviour of the built-in software.

This leads to various failures such as picture goes blank for a brief time, whole monitor turns off, picture gets distorted, user settings are corrupted, picture freezes (a video mode change is not detected) and so on.

Most of those failures are transient and non-destructive, and most of the time the MCU recovers by itself or, if locked up, after a power off/on cycle. Nevertheless, it is very important to improve the MCU immunity as much as possible against such uncontrolled behaviours which, to the display manufacturer, may not be acceptable.

A whole set of extremely efficient protection guidelines are already described in the other application note AN1015. An additional software hardening technique will be described here to improve sotware robustness even further, when many interrupts are running concurrently and, if corrupted, may lock up the whole sotware.

UNDER PATENT NUMBERS: 00-GR2-241 and 09/714, 326 (US)

Table of contents

1 Typical software architecture

1.1 Normal flow with interrupts

A typical MCU software, not dedicated to computer displays, consists in :

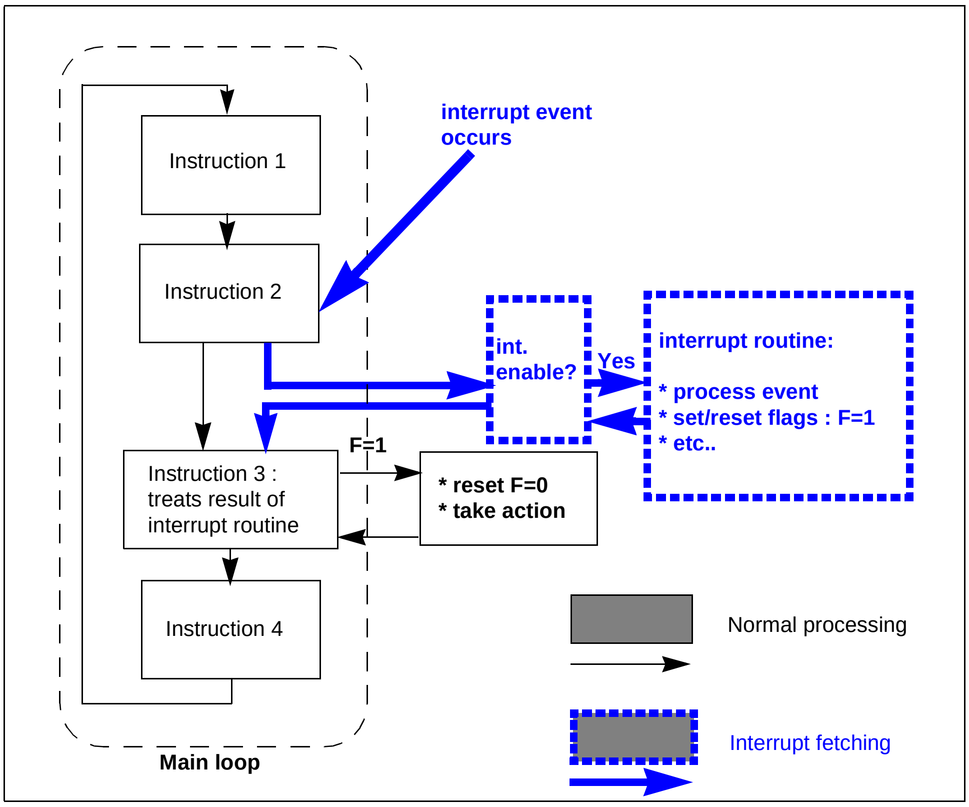
- a main loop, that runs in a neverending fashion and waits for the occurence of external events
- one or several interrupt routines which are triggered on demand by external events such as:
  - ➢ when the user presses a key
  - ➢ when a change is detected (a different video mode is issued by the PC)
  - ➢ when a data transfer is initiated (USB, DDC, I2C etc) and must be handled promptly
  - ➢ when a process must be run at given time intervals
  - ➢ etc

During normal behaviour, the main loop runs on its own. It is interrupted only when an interrupt event is triggered (by any of the events here above listed).

Following an interrupt event, the main loop is frozen. The Program Counter (PC) of the MCU jumps to the interrupt routine corresponding to the external event, if the interrupt is enabled.

This interrupt routine is executed until the end, after which the PC jumps back to the main loop where it had been interrupted, and normal program execution resumes :

Figure 1: Normal software behaviour and interrupt handling flow

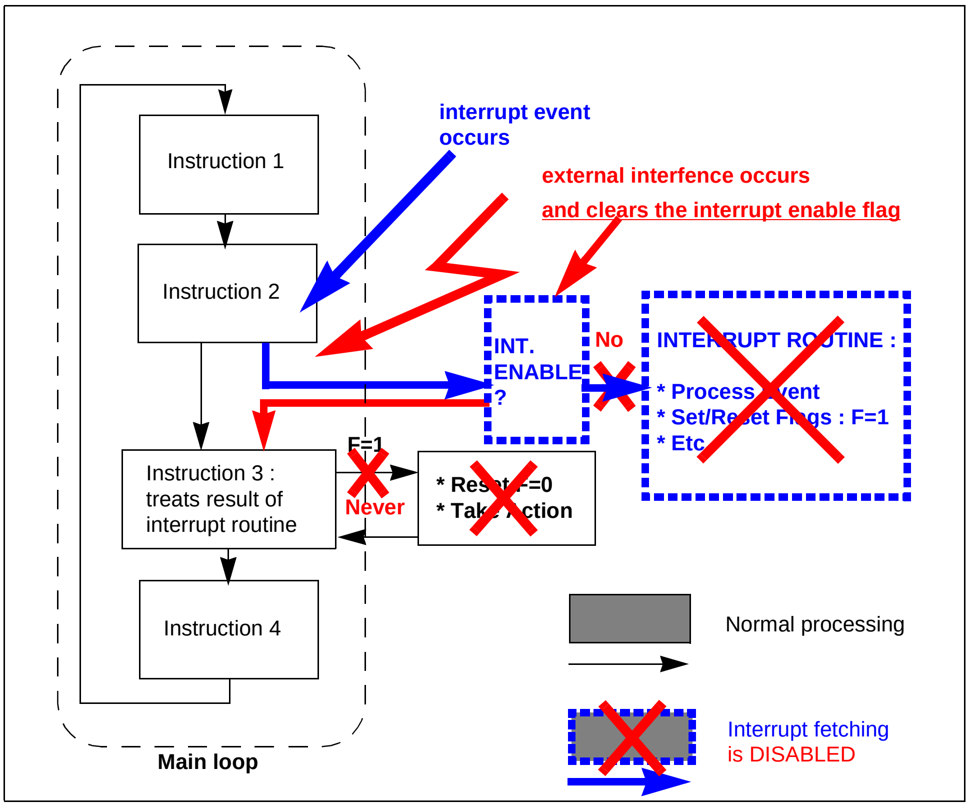


1.2 Transient interference locks interrupt

In *Figure 1*, the “instruction 3” block from the main loop contains a specific processing for an “F” flag set within the interrupt routine. The main loop uses “F” flag to detect that the interrupt truly took place, and subsequently processing occurs accordingly.

The interrupt enable flag enables or disables the interrupt event. If the external interference permanently corrupts either the interrupt enable flag or the whole interrupt management circuitry, any further processing of the incoming interrupt events is disabled. The interrupt is no longer processed. Therefore, block #3 from the main loop no longer processes any eventual crucial event that is no longer detected (*Figure 2*). Then, the whole part of the software is no longer executed.

Figure 2: When the interrupt enable flag is corrupted by interference



A simple solution consists of regularly rewritting the most important registers in the main loop, ensuring their right configuration. Although this solution is good, it is not always applicable as the events that have already happened can be cleared (ignored) in the process (such as certain flags in I2C or TIMER registers that are cleared after a mere read or write access).

Also, certain misbehaviours were already identified in the past as being due to internal MCU circuitry over which the user software has limited or no control. One example is the built-in interrupt management cell : even if all the interrupt flags are enabled, some interference causes interrupts to be permanently disabled, no matter how many times the interrupt enable registers are reprogrammed.

The definite solution for this lock-up situation is to force the MCU with a self reset. This restores the entire core circuitry to a known working state and the application restarts on safe grounds. Reset is usually achieved with the hardware watchdog register. Still, the user software first needs to understand **when** the operation has become critical and locked up, in order to force a reset in a 2nd step.

2 Interrupt failure processing

2.1 Principle

To be able to “understand” when the program flow has deviated from its normal behaviour, the main loop must have the ability to distinguish between:

- A normal event that has duly triggered an interrupt
- A normal even that should have triggered an interrupt but has not

The 1st case is the normal operation of the software. Under normal circumstances, all interrupts work the way they should, therefore no specific supervision is needed from the main loop.

The 2nd case is the one described earlier, where an event occurs but the corresponding interrupt routine is not fetched due to some internal circuitry misbehaviour. If the main loop is not “informed” that the interrupt routine was not run after the event, it still assumes the operation is normal. Therefore an additional information status should be passed on to the main loop.

2.2 Event detection

There are two kinds of events :

- Events whose occurence can be predicted, like interval timers: the interrupt routine is fetched at regular intervals that are estimated in a software delay routine fashion
- Unpredictable events that occur at any time, like external signals (synchro signals in a computer display, I2C communication in progress etc)

The 1st kind of predictable events is easy to monitor. If the main loop is waiting for a precise delay, this delay duration is known by the programmer so that a software delay routine is implemented in parallel, and if the interrupt has not occured in the given time, there is a chance that the interrupt had locked up.

Example : a delay routine that uses one of the timer interrupts to compute a precise duration

The 2nd kind of events are unpredictable by nature and rely on external signals coming to the MCU. In that case, a preliminary requirement for this software hardening technique is that the external event to monitor not only triggers an interrupt but also sets a flag that can be freely read.

That way, the flag matching the event is regularly polled from inside the main loop, and a decision is made if no interrupt occured.

Example 1: the flag VSYN in the Syncprocessor cell is set whenever a falling edge was sensed on the VSYNCI input pin

Example 2: the flag EVF in the I2C cell is set whenever an I2C event has occured (several possible causes)

Both kinds of events, predictable or not, are monitored during the main loop execution. Obviously, cases where a flag truly indicates that an event occured produce the best results.

2.3 Cross-counters

This software hardening technique is entirely based on a counter, assigned to an unique kind of event, which is:

- incremented in all routines but its own interrupt routine
- cleared in its own interrupt routine only

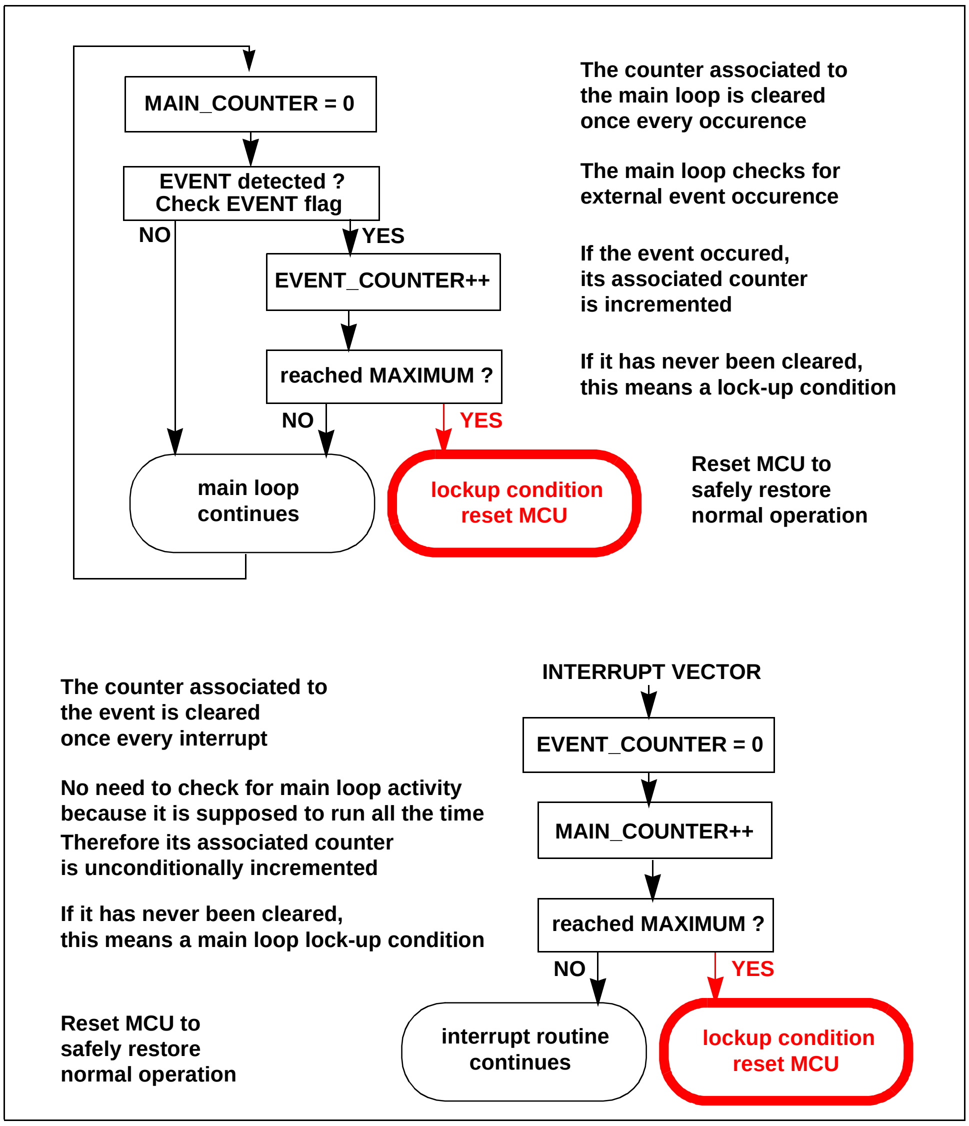
In addition, the main loop is also assigned to its own event counter, which, like the others, is incremented at all interrupt routines and cleared at every main loop iteration.

When a certain maximum value is reached after incrementing a counter, it means that no clearance occured. The related interrupt routine linked to that event never cleared its own counter: that routine was never run, although the event occured and the interrupt was enabled.

This is characteristic of an interrupt lock-up condition. Only the MCU reset using the harware watchdog (or by other means) exits from this condition.

A simple example with 1 interrupt and the main loop is shown in *Figure 3*.

Figure 3: Cross-check counter principle overview (main loop and 1 interrupt)


MAIN_COUNTER = 0
EVENT detected ? Check EVENT flag
NO
YES
EVENT_COUNTER++
reached MAXIMUM ?
NO
YES
main loop continues
lockup condition reset MCU
The counter associated to the main loop is cleared once every occurence
The main loop checks for external event occurence
If the event occured, its associated counter is incremented
If it has never been cleared, this means a lock-up condition
Reset MCU to safely restore normal operation
INTERRUPT VECTOR
EVENT_COUNTER = 0
MAIN_COUNTER++
reached MAXIMUM ?
NO
YES
interrupt routine continues
lockup condition reset MCU
The counter associated to the event is cleared once every interrupt
No need to check for main loop activity because it is supposed to run all the time
Therefore its associated counter is unconditionally incremented
If it has never been cleared, this means a main loop lock-up condition
Reset MCU to safely restore normal operation

The counter associated to the main loop is called MAIN_COUNTER. It is reset upon every main loop occurence, and incremented at every interrupt occurence.

The counter associated to the external event is called EVENT_COUNTER. It is reset upon every interrupt occurence, and incremented at every main loop occurence via the event flag.

There are 3 possible cases:

- The main loop runs fine and so does the interrupt routine (every external event): both counters are regularly incremented but also cleared, no reset ever occurs
- The main loop runs fine, the external events occur but the interrupt routine is never fetched: only the EVENT_COUNTER is incremented (from inside the main loop) until it reaches its maximum value and then, the MCU resets itself
- The interrupt works fine but the main loop has hung up: only the MAIN_COUNTER is incremented (from inside the interrupt routine) until it reaches its maximum and then, the MCU resets itself

Note: The case where both the main loop and the interrupt routine have hung up is usually covered by the hardware watchdog itself.

2.4 Any number of events

This software technique can be used for as many routines as needed, as long as the basic principles described at the beginning of *Section 2.3* are strictly enforced. This also means that there is 1 dedicated counter per interrupt + 1 for the main loop.

Also, the lock-up detection coverage is drastically improved with the increased number of routines that cross-check each other.

2.5 Maximum value for counters until reset

The theoretical MAXIMUM value to be reached by any counter before triggering the MCU reset solely depends on the slowest and the fastest of all the routines:

- the fastest routine because this will be the shortest time between two consecutive increments of a counter
- the slowest routine because this will be the counter to be cleared the least often

In the previous example, if the main routine takes 10ms to execute one loop, while the interrupt is fetched every 500us, the interrupt routine can be run up to 20 times (10ms/500us) until the main counter is cleared in the main loop. So the main counter can reach the value of 20 before being cleared.

A certain guardband is needed in case the main loop duration is not always the same (which is often the case) therefore a MAXIMUM value of 25 to 30 seems safe enough for the example.

Taking a significantly *higher* value does not impede the lock-up condition detection but *delays* it: the counters take longer to reach the maximum value that triggers the MCU reset.

3 Computer display software application

When applying this software technique to a computer display environment, and if the computer runs with either an ST7275 or an ST72774 Monitor MCU for example, several possible critical interrupt sources must be taken into account and cross-checked, in addition to the main loop itself:

- the Input Capture 1 interrupt event, that is triggered upon VSYNCI incoming signal
- the Input Capture 2 interrupt event, that is triggered upon HSYNCI incoming signal
- the Output Compare 1 and/or 2 and/or Timer Overflow interrupt events, that are commonly used to compute various delays (key debouncing, video mode out of range, flash a LED etc)

Other possible interrupt sources are from I2C, DDC and USB cells, but a simple timeout can be implemented inside the routines themselves, since those communications are based on the ACK/NAK mechanism.

Figure 4 includes the sample source code in C language that cross-checks:

- Input Capture 1 (ICAP1) Interrupt
- Input Capture 2 (ICAP2) Interrupt
- Main Loop

3 counters (1 byte each) are implemented:

- CountICAP1 for Input Capture 1 (ICAP1) Interrupt
- CountICAP2 for Input Capture 2 (ICAP2) Interrupt
- CountMAIN for Main Loop

The MAXCOUNT value strongly depends on the range of horizontal and vertical frequencies of the computer display. As an example:

- with horizontal frequencies included in the range of [31 to 100] kHz, the delay between consecutive ICAP2 interrupts varies between 2.6 ms and 8.3 ms (since ICAP2 is triggered by 256 after a prescaler)
- with vertical frequencies included in the range of [50 to100] Hz, the delay between consecutive ICAP1 interrupts varies between 10 ms and 20 ms
- the main loop lasts between 1ms and 10ms

So the slowest routine is when ICAP1 occurs every 20ms, and the fastest routine is when the Main loop lasts 1ms. The maximum value any counter can ever reach is 20ms/1ms = **20**. (safe value = 30)

Figure 4: Code for ICAP1 and ICAP2 interrupts routine

```
void Input_Capture(void)
{
if (ValBit(TIMSR, ICF2))// ICAP2 interrupt (HSYNC)
  {
  if (ValBit(SYNCLATR,VSYN))// if VSYNC signal detected
(event flag)
    {
    ClrBit(SYNCLATR,VSYN);// clear event flag
    if (CountICAP1<MAXCOUNT)// increment ICAP1 counter
until max
      CountICAP1++;
    else  // if max is reached
      WDGCR = 0x80;// immediate RESET
    }
  if (CountMAIN<MAXCOUNT)// increment MAIN counter till max
    CountMAIN++;
  else    // if max is reached
    WDGCR = 0x80;// immediate RESET

  CountICAP2=0;// reset own ICAP2 counter
  .... <rest of the ICAP2 interrupt routine> ....
  }
if (ValBit(TIMSR, ICF1))// ICAP1 interrupt (VSYNC)
  {
  if (ValBit(SYNCLATR,HSYN))// if HSYNC signal detected
(event flag)
    {
    ClrBit(SYNCLATR,HSYN);// clear event flag
    if (CountICAP2<MAXCOUNT)// increment ICAP2 counter
until max
      CountICAP2++;
    else  // if max is reached
      WDGCR = 0x80;// immediate RESET
    }
  if (CountMAIN<MAXCOUNT)// increment MAIN counter till max
    CountMAIN++;
  else    // if max is reached
    WDGCR = 0x80;// immediate RESET

  CountICAP1=0;// reset own ICAP1 counter
  .... <rest of the ICAP1 interrupt routine> ....
  }
}         // end of ICAP1/ICAP2 interrupt
```

Figure 5: Code for Main loop routine

```
void main(void)
{
if (ValBit(SYNCLATR,VSYN))// if VSYNC signal detected (event
flag)
    {
    ClrBit(SYNCLATR,VSYN);// clear event flag
    if (CountICAP1<MAXCOUNT)// increment ICAP1 counter
until max
      CountICAP1++;
    else  // if max is reached
      WDGCR = 0x80;// immediate RESET
    }
if (ValBit(SYNCLATR,HSYN))// if HSYNC signal detected (event
flag)
    {
    ClrBit(SYNCLATR,HSYN);// clear event flag
    if (CountICAP2<MAXCOUNT)// increment ICAP2 counter
until max
      CountICAP2++;
    else  // if max is reached
      WDGCR = 0x80;// immediate RESET
    }

CountMAIN=0;// reset own MAIN counter
  .... <rest of the main loop> ....
}
```

4 “Light” solution without main loop check

The software technique previously described considered both the interrupts and the main loop. Due to the cross-check between them, the lock-up condition arises if any of them fails.

However, a good software programming scheme usually enables the hardware watchdog , when present, as it is only refreshed within the main loop. The main loop supervision is therefore unnecessary because if the main loop hangs up, the watchdog counter automatically resets the MCU after some time. There is no need for any additional software handling.

Therefore, the code lines managing the counter which are associated to the main loop can simply be omitted. This is particularly significant when only 1 interrupt is managed: in that case, since the main loop counter needs not be managed, there is no extra code to add to the interrupt routine, and everything is done from the main loop.

Considering the example from *Section 3*, all references to CountMAIN can be simply removed. This makes a total of 2 counters to manage only: CountICAP1 and CountICAP2.