



SOFTWARE TECHNIQUES FOR IMPROVING MICROCONTROLLER EMC PERFORMANCE

by Microcontroller Division Applications

1 INTRODUCTION

A major contributor to improved EMC performance in microcontroller-based electronics systems is the design of hardened software.

To achieve this goal, you have to include EMC considerations as early as possible in the design phase of your project.

EMC-oriented software increases the security and the reliability of your application. EMC-hardened software is inexpensive to implement, it improves the MCU's immunity performance and saves hardware costs. You should consider EMC disturbances to analog or digital data as parameters that must be managed by the MCU software just like any other application parameter.

Examples of software disturbances:

- Microcontroller not responding
- Program Counter runaway
- Execution of unexpected instructions
- Bad address pointing
- Bad execution of subroutines
- Parasitic reset
- Parasitic interrupts
- I/O deprogramming

Examples of the consequences of failing software:

- Unexpected commands
- Loss of context
- Unexpected branch in process
- Loss of interrupts
- Loss of data integrity
- Wrong input measurement values

This application note describes software techniques divided into two categories:

- Preventive techniques
- Auto-recovery techniques

You can implement preventive techniques in existing programs. Their purpose is to avoid visible disturbances at user level.

The software must include auto-recovery routines. When a runaway condition is detected, a recovery subroutine is used to take the decision to stop program execution, optionally give a warning and then return automatically to normal operations. This operation may be absolutely transparent to the user of the application.

2 PREVENTIVE TECHNIQUES

You can easily implement these techniques in an existing program as they do not require any change to the structure of the software.

2.1 USING THE WATCHDOG CORRECTLY

The watchdog is the most efficient tool you have available for ensuring that the MCU can recover from software runaway failures. Its principle is very simple: it is a timer which generates an MCU reset at the end of count. The only way of preventing the Watchdog resetting the microcontroller is to refresh the counter periodically in the program.

But to make the watchdog work at its full potential, you have to insert the enable and refresh instructions in your software in the right way.

Figure 1 shows the classic examples of bad watchdog implementation:

To do it the right way, the golden rules are:

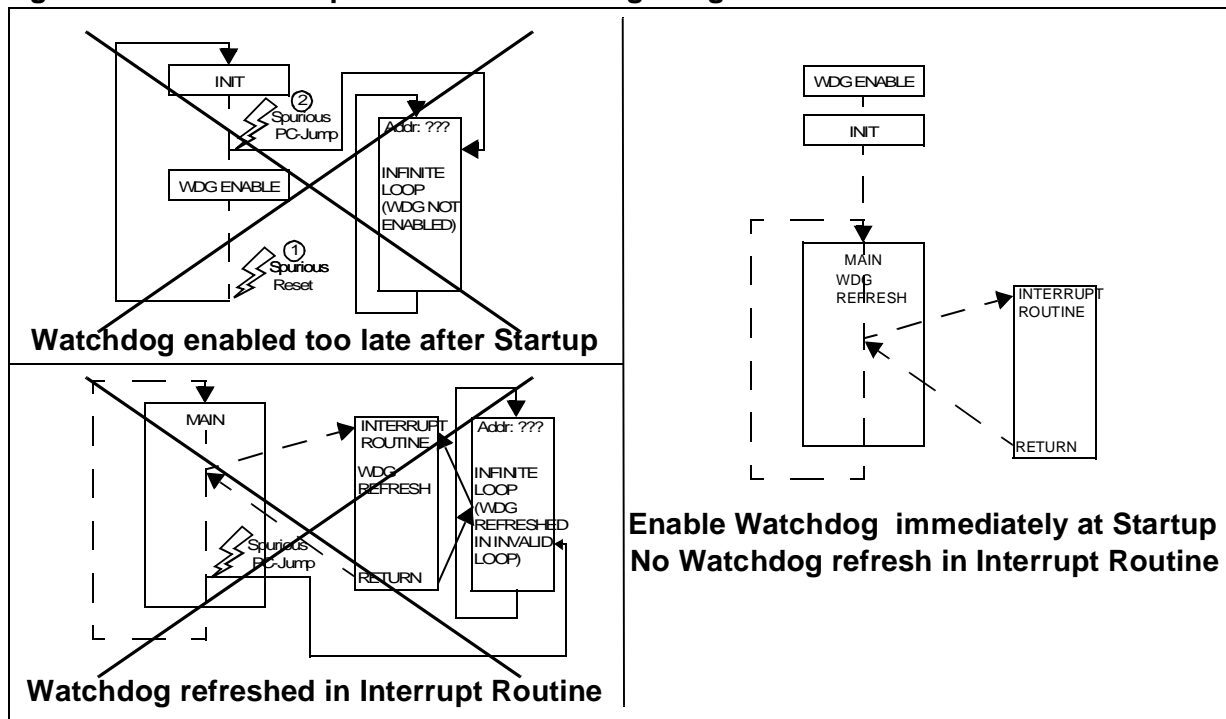
- Enable the watchdog as soon as possible after reset, or use the Hardware Watchdog option if its available.
- Never refresh the watchdog in an interrupt routine.

It is very important to optimize the period between the two refresh instructions according to the duration of the various routines, including the interrupt routines.

The minimum use of the watchdog resets the MCU, this means that the program execution context is lost as well as the application data's integrity.

After reset, in addition to enabling the watchdog, on some MCUs you can use the reset flags to distinguish between a Power On or Low Voltage reset and a Watchdog reset (refer to Section 3.3. for more details)

Figure 1. Classic Examples of Bad Watchdog Usage



2.2 SECURING THE UNUSED PROGRAM MEMORY AREA

In most applications, program memory space is not used completely. For extra security, fill the unused memory locations with code that forces a watchdog reset or jumps to a known program location if you do not want to generate a reset.

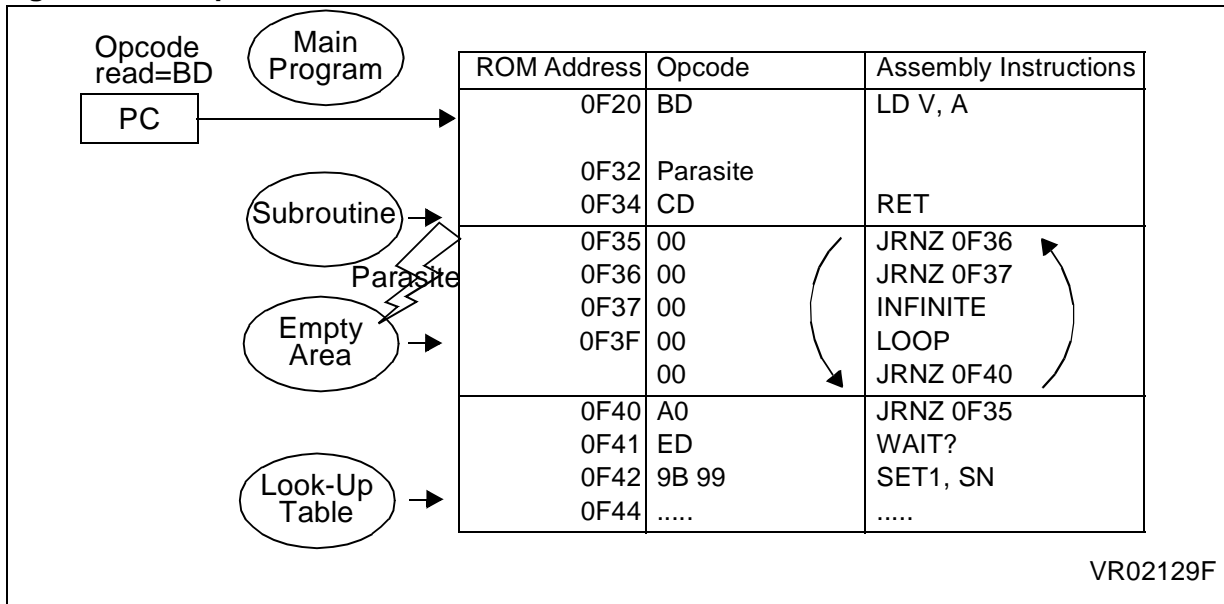
This will ensure that even if the program counter is corrupted and jumps to an unused memory location, the MCU will recover and return to normal operations.

In the example below (using the ST6), at address 0F3F if you put **LDI WDT,01h**, this instruction causes the MCU to be reset by the internal watchdog and thus avoids a microcontroller lock condition.

In this unused area you can also jump to a Parasite Detection subroutine, which allows you to return to normal operations.

For ST7 users, the ST7 "TRAP" instruction is also very convenient (only one instruction byte:83) for generating a software interrupt in order to recover from a jump to an unexpected location in memory.

Figure 2. Example of a Microcontroller lock condition



2.3 INPUT FILTERING

The routine given below (for ST6) checks several times that PB4=1 before continuing the program execution.

This is a simple means of filtering a critical input at no extra cost!

```
MAIN1  LDI LOOP,08h          repeat measurement 8 times
MAIN2  JRR 4,PB,MAIN1        Check bit 4 of port B
      DEC LOOP              Decrement loop
      JNRZ MAIN2            until Loop=0
```

2.4 MANAGEMENT OF UNUSED INTERRUPT VECTORS

To avoid problems caused by unexpected interrupt occurrences (whatever the source) it is recommended to manage all the possible interrupt sources by putting a valid interrupt routine address in the corresponding vector.

In the example below the unused interrupt vectors point to a "dummy" label filled with a simple "return from interrupt" instruction.

Example of unused interrupt management (ST7):

```
.dummy
    ired

segment 'vectit'

.pwm_it      DC.W    dummy          ;location FFE0-FFE1h
            DC.W    dummy          ;location FFE2-FFE3h
.i2c_it      DC.W    i2c_rt         ;location FFE4-FFE5h
.sci_it      DC.W    dummy          ;location FFE6-FFE7h
.tb_it       DC.W    dummy          ;location FFE8-FFE9h
.ta_it       DC.W    dummy          ;location FFEA-FFEBh
.spi_it      DC.W    dummy          ;location FFEC-FFEDh
.can_it      DC.W    can_rt         ;location FFEE-FFEFh
.ext3_it     DC.W    dummy          ;location FFF0-FFF1h
.ext2_it     DC.W    dummy          ;location FFF2-FFF3h
.ext1_it     DC.W    dummy          ;location FFF4-FFF5h
.ext0_it     DC.W    dummy          ;location FFF6-FFF7h
.mcc_it      DC.W    dummy          ;location FFF8-FFF9h
.nmi_it      DC.W    dummy          ;location FFFA-FFFBh
.softit     DC.W    pc_jp          ;location FFFC-FFFDh
.reset       DC.W    init          ;location FFFE-FFFFh
```

2.5 REMOVING ILLEGAL AND CRITICAL BYTES FROM YOUR CODE

2.5.1 Critical Bytes

A critical byte is an instruction like WAIT or STOP which is decoded by the microcontroller and forces it to stop executing any further instructions.

When the PC is corrupted it often becomes desynchronized (as most of the instructions have several bytes), and as a result it may read and decode critical bytes.

To check and minimize the occurrence of these critical bytes you can edit the program ".list" file.

Very often critical bytes are generated by the compiler as label address bytes. In this case, if you simply insert one or several NOP instructions, all the label addresses will shift and this will change the critical byte value to another value.

Example:

In the ST7 instruction sequence shown below, the "main" label address bytes contain the HALT op-code (8E) and the "loop1" label address bytes contain the WFI op-code (8F).

If you add two "NOP" instructions before "loop1", the addresses are shifted from C18E to C190 for "main" and from C08F to C091 for "loop1" and the critical bytes disappear!

```
.loop1
C08F      . . . . .
C09A      81                                ret

.main                                ;Led PB3 freezed on

C18E      1415                                bset      PBDR,#2
C190      1715                                bres      PBDR,#3
C192      CDC08F                            call      loop1
C195      CDC08F                            call      loop1
C198      1515                                bres      PBDR,#2
C19A      CDC08F                            call      loop1
C19D      CDC08F                            call      loop1
C1A0      CCC18E                            jp         main
```

2.5.2 Illegal Bytes

Illegal bytes are defined as any byte value which is not part of the instruction set. They will either be executed as a NOP instruction or (on some MCUs) a reset is generated if an illegal byte is encountered. In some ST6 devices however "E5h" is executed as a WAIT and "65h" as a STOP. In this case, use the techniques described above (for critical bytes) to remove illegal bytes from your code.

2.6 AVERAGING THE A/D CONVERTER RESULTS

If you are performing A/D conversion, you can repeat conversions several times, store the results in the RAM and then average them (or select the most frequently-occurring value) to obtain accurate results in spite of any potential noise errors.

2.7 REGISTER REPROGRAMMING

It rarely happens that EMC disturbances alter the content of the registers. Generally the registers concerned are clock control registers or I/O configuration and data registers because they are close to the chip output pads.

In such cases a good security measure is to refresh these registers frequently.

Table 1. Summary of Preventive Techniques

Software Quality Preventive Methods	Advantage	Disadvantage	Implementing
Watchdog (Hardware or Software)	Control is CPU-independent Avoids MCU lock	Not compatible with Halt mode	Easy but the activation and refresh instructions must be carefully placed in the code for maximum efficiency
Force a watchdog reset in unused program memory	More direct and quicker than waiting for a watchdog timeout	Loss of previous context	Clear the WDG reset bit (see device spec.)
Fill unused program memory with software interrupt instructions	Single byte instruction. More direct and quicker than wait for a WDG timeout.	Instruction available only on ST7 devices.	Fill unused area with "TRAP"(83h) op-code and manage the failure in the corresponding interrupt routine.
A/D Converter averaging	Ensure the ADC performance in a noisy surrounding.	Processing time	Perform an iterative loop for ADC acquisitions and averaging.
Removal of illegal or critical opcode	Avoid MCU locks due to unexpected readings of WAIT or STOP opcodes	none except restriction on using these opcodes	String search in the ".LIST" file (See §2.5).
Input filtering	Data acquisition stability	Processing time	Repeat measurement several times and perform a statistical choice between "0" or "1".
Unused interrupt management	Avoid runaways due to unexpected interrupts	None	Very easy (see section 2.4
Refreshing of critical registers	Safe running	Uses MCU resources	Refresh critical registers in frequently-executed loops

3 AUTO-RECOVERY TECHNIQUES

This section gives some techniques for quickly recovering your application context after an EMC failure.

Unexpected resets, Program Counter jumps and parasitic interrupts are the most common EMC failures observed in the MCU whatever the source of the disturbance.

In both any of cases the RAM (or EEPROM data memory when available) remains unchanged and can be used as very efficient way to save the application context and parameters.

Note that the RAM will lose its contents if the device is powered-off. The EEPROM data keeps its content at power-off but the writing time is much longer.

3.1 SAVING YOUR CONTEXT IN RAM

Figure 3 shows an example of a software auto-recovery implementation:

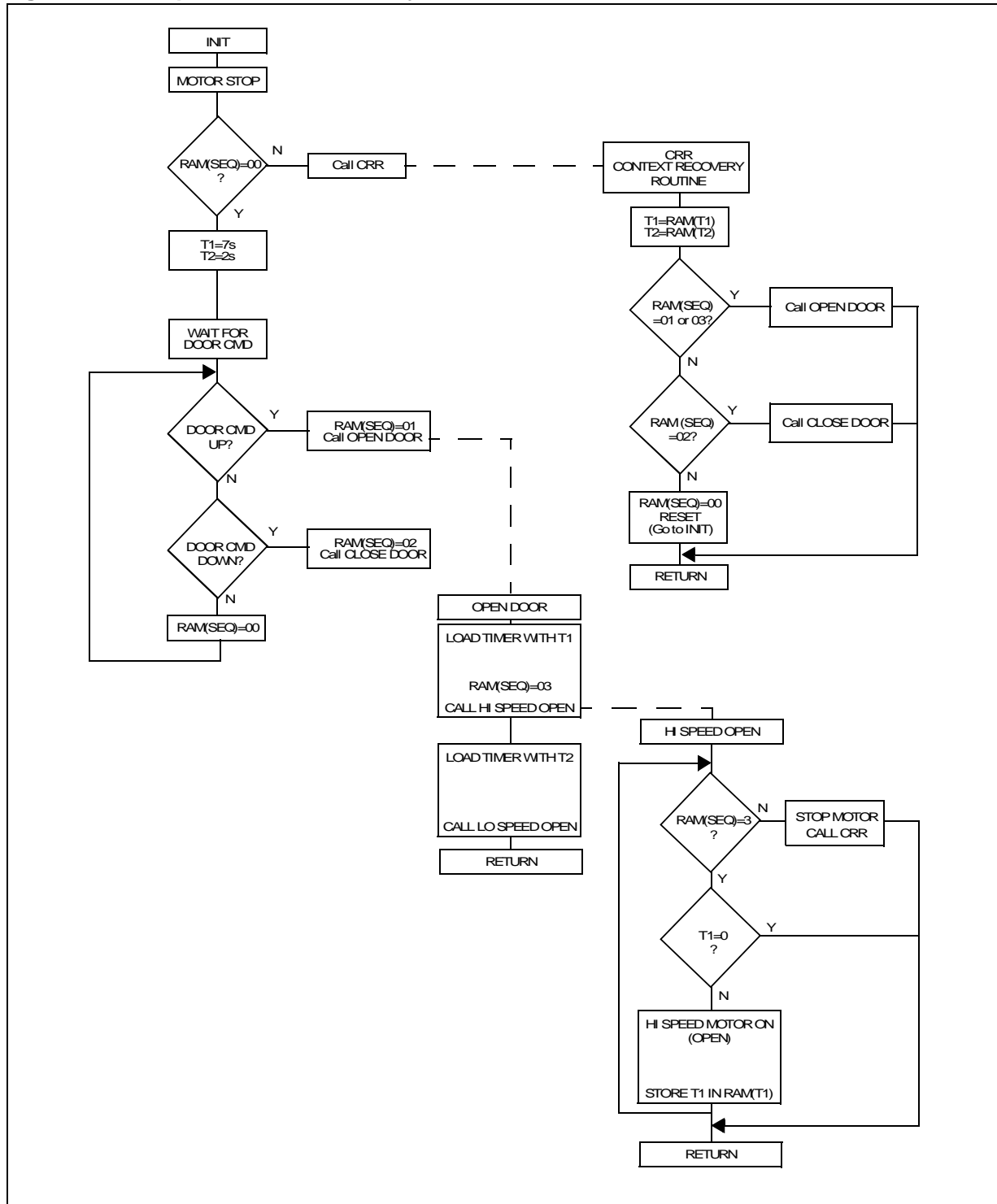
In Figure 3 we can see that the critical software sequences (door OPEN or CLOSE commands, high speed motor controls) are memorized in a RAM byte ("RAM(SEQ)").

This allows us on the one hand to recover the context if an EMC event leads to an MCU reset, and on the other hand we can check the source before a executing critical subroutine. In this case the high speed motor activation is allowed only if RAM(SEQ)=03).

The application parameters (T1&T2 timing values) are also stored in RAM when they are changed.

This means if a software runaway event occurred or the MCU is reset (by the LVD or the watchdog), the recovery routine (CRR) will restore the last door command, reload the timing parameters and resume the program execution without any external intervention.

Figure 3. Example of Auto-Recovery Software

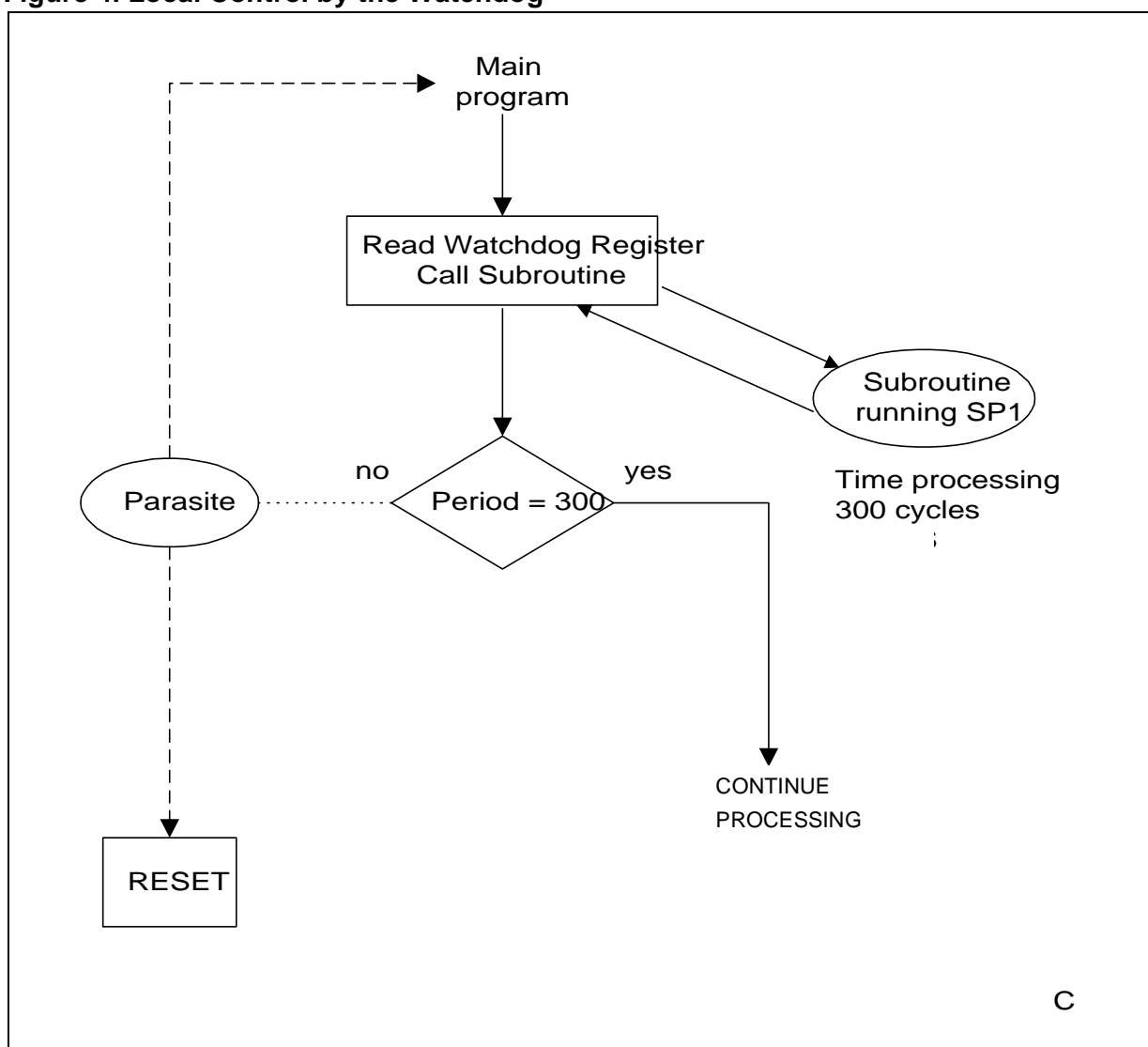


3.2 USING THE WATCHDOG FOR LOCAL CONTROL

Very often programmers consider the Watchdog Timer more or less just as a time bomb and only refresh it to its maximum value to have the widest possible margin without any direct relation to the expected program execution time.

This is a poor approach, a far better method is to use the watchdog timer register to check the execution times of individual software routines and in case of an abnormality to react promptly before the Watchdog end of count and either perform an immediate reset or go into a software recovery routine.

Figure 4. Local Control by the Watchdog



3.3 USING THE RESET FLAGS TO IDENTIFY THE RESET SOURCE

There are several possible internal reset sources: LVD (Low Voltage detector) or Watchdog reset, POR (Power On Reset), hot reset (parasitic or external reset following a low state of the Reset pin).

On most of the ST MCUs the reset source is flagged in a "reset register" and this information is kept as long as the MCU's power supply is on.

Figure 5 shows how you can test the reset register at the beginning of your program and then branch to a context recovery routine (depending to the detected reset source) instead of re-starting the "Power On Reset" initialization routine which is often complex and time consuming.

It is very important to detect and manage parasitic resets as they are the most usual cause of microcontroller EMC failures.

Figure 5. Identify Reset Sources

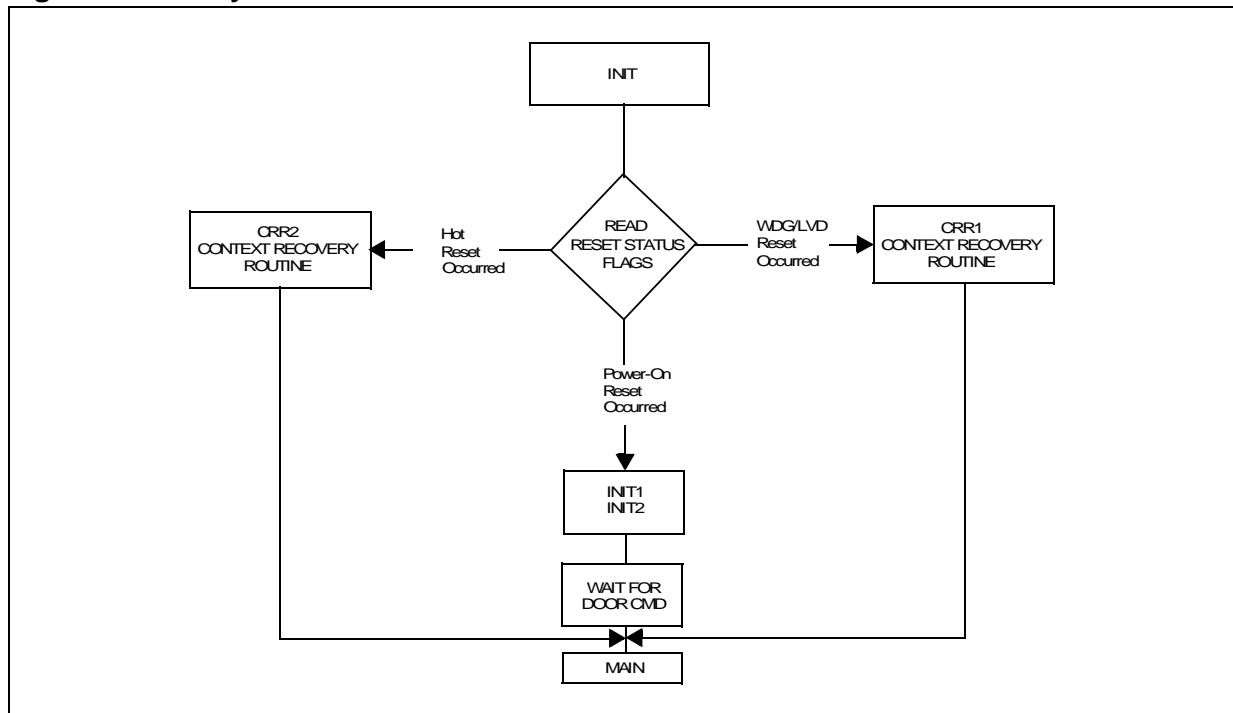


Table 2. Summary of Auto-Recovery Techniques

Software Quality Auto-recovery Methods	Advantage	Disadvantage	Implementing
Local Control by the Watchdog	Process control of critical sequential blocks	Need a calculation of a accurate time window	Check the sequence execution time using the WDG timer register
Identify Reset Sources	Fast recovery from unexpected reset failures	None	Use the MCU "reset register" or the RAM to detect various reset sources.
Application context save in RAM or FLASH	Save application parameters, ensure critical task execution resume in case of MCU failures.	Uses MCU resources	Store software critical phases and parameters in RAM or FLASH. Use data in RAM or FLASH to recover the last context before failure.

4 WHAT RESULTS CAN BE ACHIEVED?

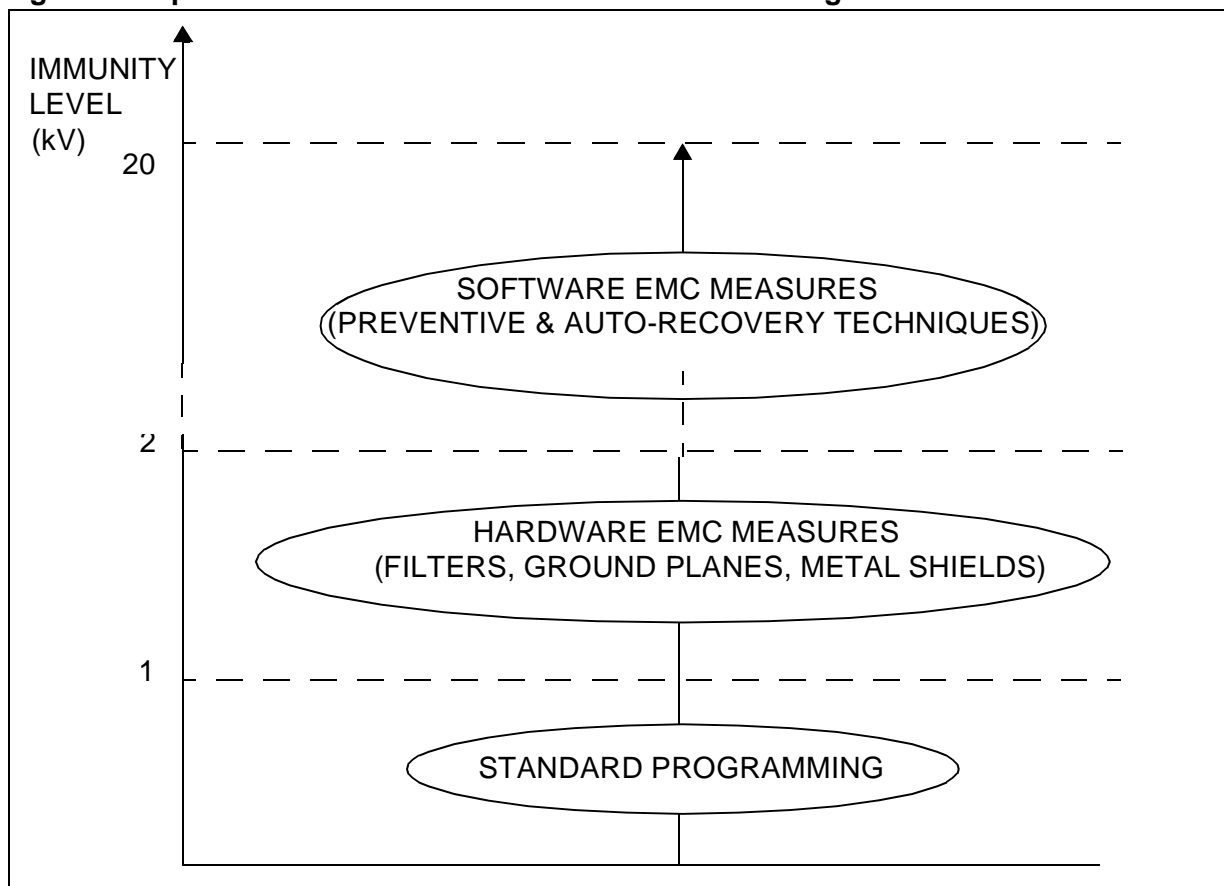
ST microcontrollers are designed tested and optimized to remain fully functional with +/-1kV ESD voltages (according EN1000-4-2 standard) directly applied on any pin. Although this performance is acceptable in most cases, it does not guarantee that the MCU will be fully robust at application level which is sometimes above 4kV.

Such voltages cannot be withstood by any microcontroller using standard programming techniques. This means that the EMC hardening techniques described in this document are absolutely necessary to improve application robustness in many cases.

Figure 6 quantifies the typical EMC robustness limits you can expect from hardware or software EMC-hardening measures (This data is sourced from qualification reports provided by ST customers).

You can see that good EMC-hardened software can bring the application immunity to a very high level, limited only by the physical silicon resistance.

Figure 6. Impact of Hardware and Software EMC Hardening



SOFTWARE TECHNIQUES FOR IMPROVING EMC PERFORMANCE

"THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNEXION WITH THEIR PRODUCTS."

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without the express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics

©2001 STMicroelectronics - All Rights Reserved.

Purchase of I²C Components by STMicroelectronics conveys a license under the Philips I²C Patent. Rights to use these components in an I²C system is granted provided that the system conforms to the I²C Standard Specification as defined by Philips.

STMicroelectronics Group of Companies

Australia - Brazil - China - Finland - France - Germany - Hong Kong - India - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain
Sweden - Switzerland - United Kingdom - U.S.A.

<http://www.st.com>