AVR132: Using the Enhanced Watchdog Timer

Features

- **Watchdog System Reset Source**
- **Parameter Backup Prior to Watchdog System Reset**
- **Wakeup Timer from all Sleep Modes**
- **Using the Watchdog for Both Wakeup and System Reset**
- **Handling the Watchdog Reset Flag**
- **Changing the Watchdog Configuration**
- **Flowcharts for Watchdog Operation**
- **Example Source Code**

1 Introduction

“Well designed watchdog timers fire off every day, quietly saving systems and lives without the esteem offered to human heroes.” - Jack Ganssle.

No piece of software, save the very smallest, is free from bugs. The application could get stuck in endless loops. Unexpected error codes could cause serious problems if not handled correctly. Electrical noise or an unusual sequence of external events could put the system in a state not thought of by the designers. All these cases could potentially hang the system forever or cause serious damage to its surroundings. Automatic handling and recovery of such cases is the job of a watchdog timer.

The Enhanced Watchdog Timer (WDT) runs independent of the rest of the system, causing system resets whenever it times out. However, the application software should ensure that the timeout never occurs by resetting the WDT periodically as long as the software is in a known healthy state. If the system hangs or program execution is corrupted, the WDT will not receive its periodic reset, and will eventually time out and cause a system reset.

The WDT in all new AVR devices also has the ability to generate interrupts instead of resetting the device. Since the WDT runs from its own independent clock, it can be used to wake up the AVR from all sleep modes. This makes it an ideal wakeup timer, easily combined with ordinary operation as a system reset source. The interrupt can also be used to get an early warning of an upcoming Watchdog System Reset, so that vital parameters can be backed up to non-volatile memory.

8-bit AVR® Microcontrollers

Application Note

Rev. 2551C-AVR-06/08

2 Theory of operation

When the Enhanced Watchdog Timer (WDT) period has expired, a WDT timeout occurs. The timeout period is adjusted using a configurable prescaler, which divides the WDT oscillator clock by a constant factor. Executing the WDR (Watchdog Reset) instruction resets the timer value. The application software using the WDT must be designed so that it executes the WDR instruction periodically whenever it decides that the system still operates correctly. The timer value is automatically reset on system reset and when disabling the WDT.

The Enhanced Watchdog Timer has three modes of operation. When operating in WDT System Reset Mode, a WDT timeout causes a system reset. If WDT Interrupt Mode and global interrupts are enabled, a WDT timeout sets the WDT Interrupt Flag and executes the WDT Interrupt handler, instead of resetting the system. If both WDT System Reset Mode and WDT Interrupt Mode are enabled, the first WDT timeout is handled as if only WDT Interrupt Mode was enabled. Then WDT Interrupt Mode is disabled automatically and the WDT is back in only WDT System Reset mode.

Figure 2-1 shows what happens when a WDT timeout occurs. The dotted boxes describe actions performed by the system. The solid lined boxes describe actions to be performed by the application

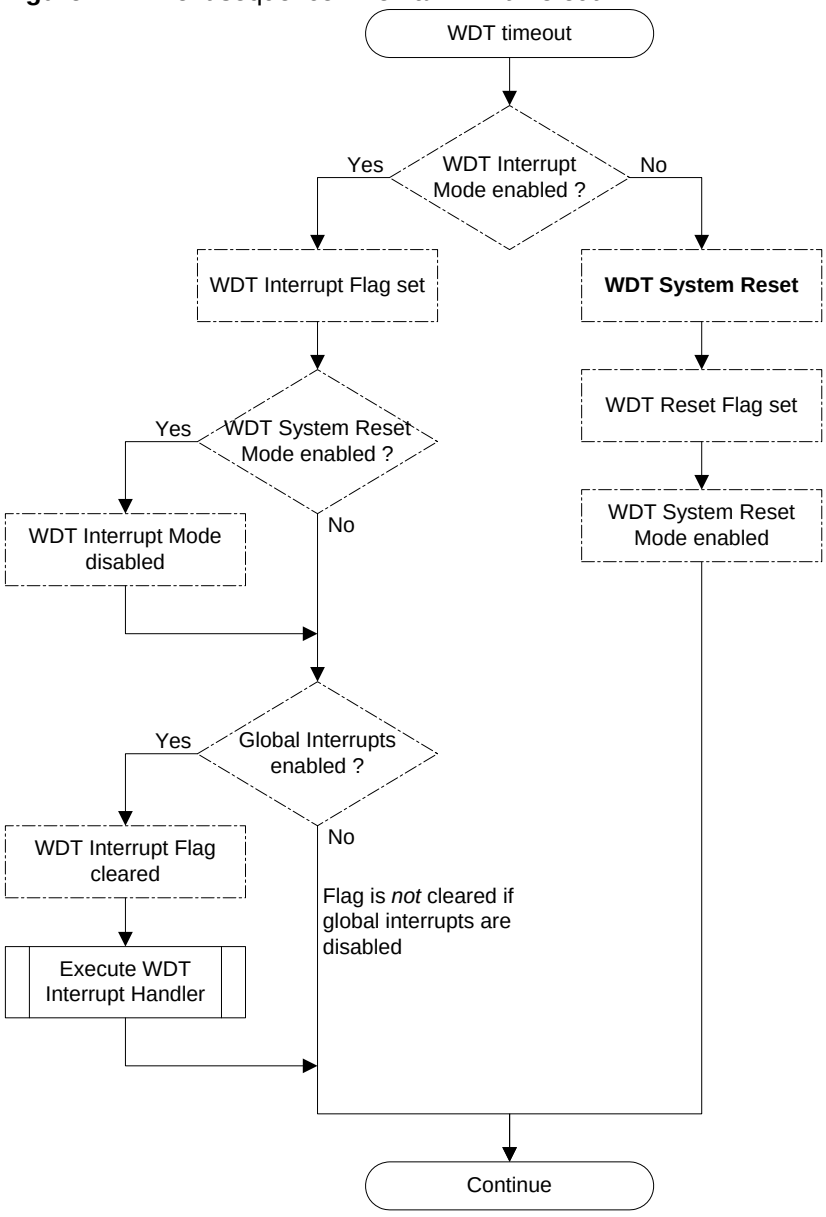
When using the Enhanced Watchdog Timer it is important to know that if the Watchdog Always On (WDTON) fuse is programmed, the only possible operation mode is WDT System Reset Mode. This security feature prevents software from enabling the WDT Interrupt Mode unintentionally, which could disable the WDT System Reset functionality. When the WDTON fuse is unprogrammed, the WDT Interrupt Mode can be used as described in this document.

As mentioned above, the WDT is independent from the rest of the system. It has its own internal 128 kHz oscillator, which runs as long as one of the WDT operating modes is enabled. This ensures safe operation even if the main CPU oscillator fails.

Note that, even if the software designers never intended to use the WDT, it could be enabled unintentionally, e.g. by a runaway pointer or brown-out condition. If the WDT is enabled unintentionally it will remain enabled until the firmware disables it since a System Reset caused by the WDT will *not* disable the WDR. System Resets not caused by the WDT will reinitialize the WDT to the default configuration according to the fuse settings. The automatic “re-enabling” of the WDT in case of a WDT System Reset is a safety feature to ensure reliable Watchdog functionality. Therefore the startup code should *always* check the Reset Flags and take appropriate action if a WDT System Reset has occurred, even if the application does not use the WDT.

The various settings and functions can be combined to use the WDT for different purposes. The most important setups are described in the following sections.

Figure 2-1. Event sequence when a WDT time out.



Configuring the WDT to work as a system reset source only, is straightforward. Enable the WDT System Reset Mode, set a reasonable timeout delay and off you go. If your initialization routines take longer than the WDT timeout period, they should execute the WDR instruction at appropriate checkpoints during execution. If not, the code will never reach its main loop before the WDT resets the system.

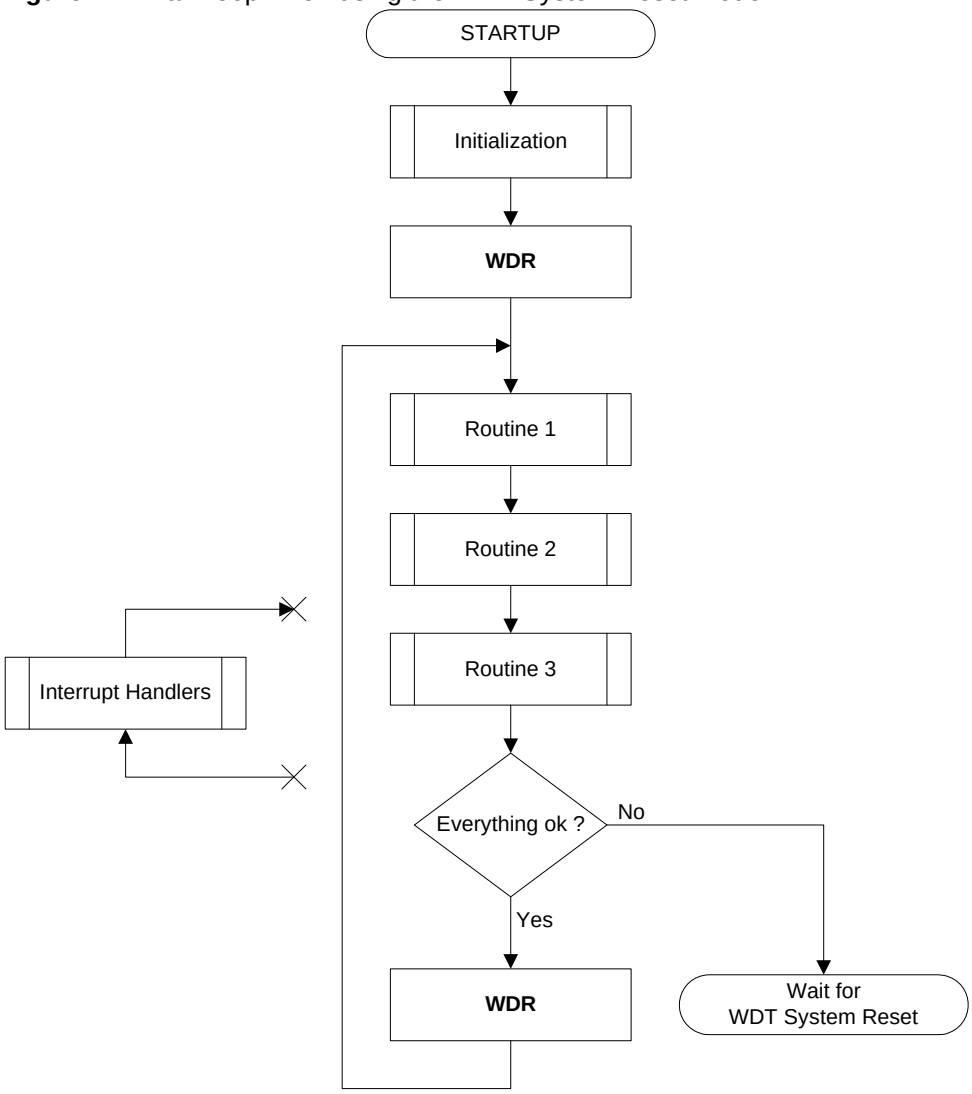
The timeout period must be chosen so that it is longer than the longest possible execution path through the main loop of your application. This includes expected interrupt handlers as well. If your main loop is very large, several checkpoints could be inserted inside the loop to allow a shorter timeout period.

Choosing the correct timeout period requires detailed knowledge of the timing characteristics of your main loop. In many applications, the most robust approach

could be to choose a timeout period of several seconds. This will at least reset the system if it is stuck in an infinite loop.

Most embedded systems consist of some initialization code and a main loop. This construction is also the most effective setup for use with a watchdog. An example for using the WDT with such systems is shown in Figure 2-2.

Figure 2-2. Main loop when using the WDT System Reset mode.



Note that if the timeout period is chosen very tight, an unusual number of interrupts could cause a WDT System Reset. This must be taken into consideration when choosing the timeout period.

The ‘Everything ok ?’ check at the end of the loop is the part of the loop deciding whether the application is operating correctly or not. One solution is to use flags that are set in different parts of the main loop to indicate ‘good health’, or that vital parts of the code have been visited. The final check tests all flags and resets the WDT and the flags if everything is ok. If not, a timeout will eventually occur.

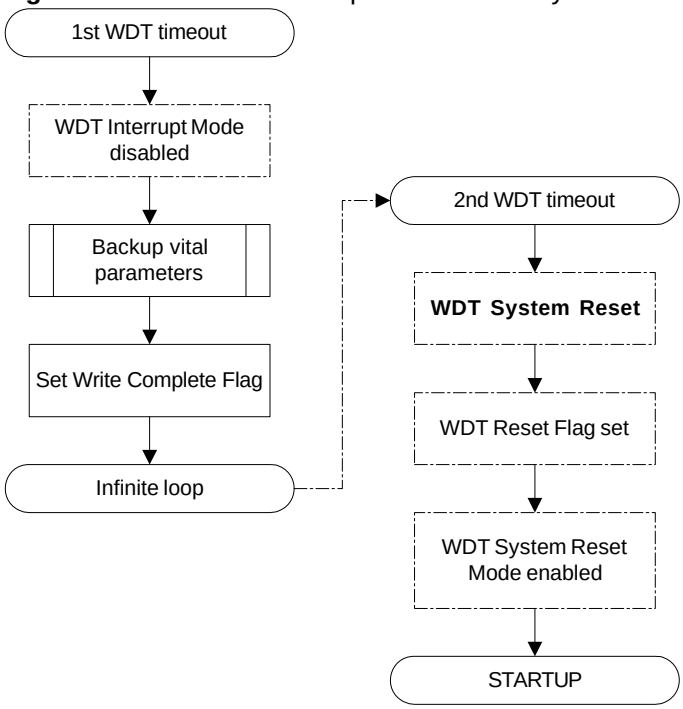
The initialization code should check the WDT Reset Flag and take appropriate actions. This is covered in more detail in section 2.4.

2.1 Parameter Backup Prior to WDT System Reset

The method described in the previous section does not give any warning of a coming WDT System Reset. The application has no means of handling a timeout in software before the system reset occurs. However, by using the WDT Interrupt Mode, the application can use the WDT Interrupt handler for backing up vital parameters before the actual reset.

By enabling both WDT System Reset Mode and WDT Interrupt Mode, the first timeout will disable the WDT Interrupt Mode and run the interrupt handler. The second timeout then causes a system reset. The interrupt handler then has one timeout period for backing up parameters, for example, to EEPROM. The sequence of events is shown in Figure 2-3. The dotted boxes describe actions performed by the system. The solid lined boxes describe actions to be performed by the application.

Figure 2-3. Parameter Backup Prior to WDT System Reset.



The Write Complete Flag could be a byte in EEPROM indicating whether the backup operation was finished before the system reset. This flag is checked in the startup code if the WDT Reset Flag is set, and the backed up parameters can be used for restoring system state or debugging purposes. The flag should be cleared during initialization to invalidate the parameters if other types of resets occur.

Note that there is no guarantee that the interrupt handler is executed prior to a WDT System Reset. If interrupts are disabled too long, the interrupt handler will never execute before the second timeout. Runaway pointers or electrical noise could also unintentionally disable the WDT Interrupt Mode. Therefore the Write Complete Flag is our means of knowing if the stored parameters are valid or not.

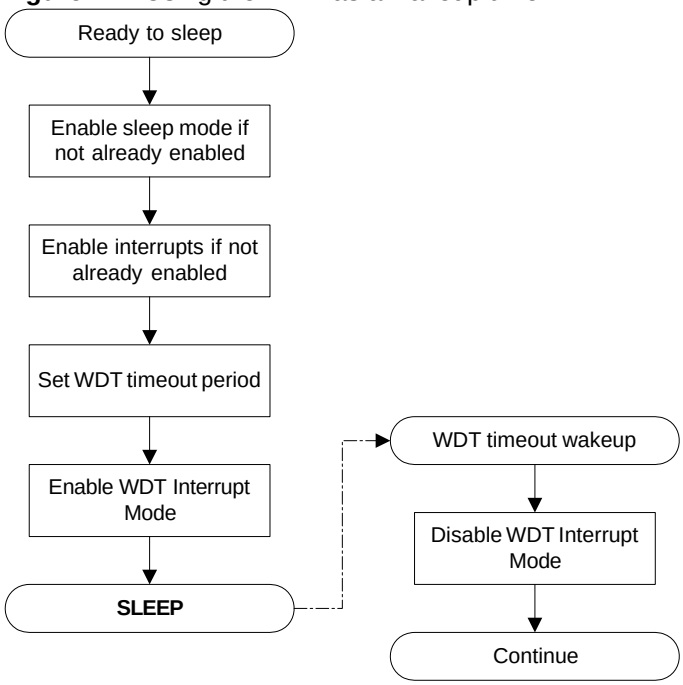
The infinite loop at the end of the interrupt handler prevents the main code from potentially causing more damage.

2.2 Using the WDT Interrupt Mode

As described above, the WDT has its own internal oscillator running independently from the main CPU clock. This makes it possible to use the WDT Interrupt as a wakeup source from all sleep modes. By enabling only the WDT Interrupt Mode, a timeout will generate an interrupt request, but not cause any system resets on further timeouts.

Having a wakeup source without running the main CPU clock is an excellent way of saving power. Using power-down sleep mode with the WDT as a wakeup source draws approx 3µA when running at 3V supply voltage. An example on how to use the WDT as a wakeup source is shown in Figure 2-4.

Figure 2-4. Using the WDT as a wakeup timer.



If periodic wakeups are preferred, the disabling of the WDT Interrupt Mode can be left out. The WDT will then generate an interrupt on every timeout, waking up the CPU if it is in sleep mode.

Note that the WDT System Reset Mode must not be enabled when using the WDT solely as a wakeup timer. If it is enabled, a system reset will occur on the next timeout. Using the WDT both as a wakeup timer and system reset source is described in the following section.

2.3 Using the WDT in Combined Operation

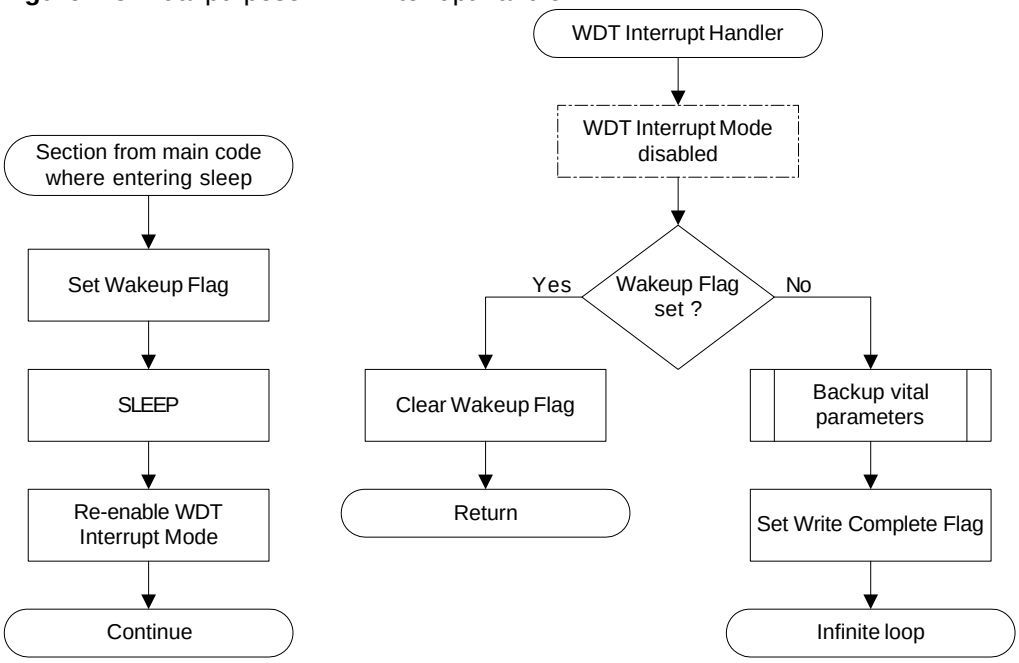
It is also possible to set up the WDT to work as a wakeup timer when entering sleep mode, and switch to WDT System Reset operation when back in active mode. With this setup there is no need for disabling the WDT Interrupt Mode, as it is automatically disabled by the hardware. To use the WDT as a periodic wakeup source, the application therefore has to enable the WDT Interrupt Mode prior to entering sleep mode every time.

Re-enabling the WDT Interrupt Mode inside the interrupt handler is not recommended, as it could cause the WDT to get stuck in WDT Interrupt Mode, if some parts of the code fail.

When the CPU is back in active mode, the WDR instruction is used for resetting the WDT inside the main loop as described earlier. With WDT Interrupt Mode disabled, the WDT functions just as it did without the wakeup functionality.

If timeout warning prior to system reset is needed for parameter backup etc., the WDT Interrupt handler needs some slight changes. The interrupt handler must use a flag to decide whether it should serve a wakeup interrupt or a timeout warning interrupt. An example interrupt handler is shown in Figure 2-5.

Figure 2-5. Dual purpose WDT interrupt handler.



Note that the wakeup flag must be set manually prior to entering sleep mode to ensure that the correct part of the handler is executed on wakeup. The WDT Interrupt Mode must be re-enabled outside the interrupt handler after serving the wakeup interrupt.

The right branch of the flowchart is described in section 2.1.

2.4 Startup Considerations

When designing for devices having the Enhanced Watchdog Timer, it is important to evaluate the WDT Reset Flag in the startup code. This applies even if the application never intends to use the WDT. If the WDT System Reset Mode should unintentionally be enabled and cause a system reset, the WDT Reset Flag will be set and the WDT System Reset Mode is kept enabled after the system reset. Therefore the startup code should check the WDT Reset Flag and disable the WDT System Reset Mode if it is enabled but never used. These considerations apply when the WDTON fuse is unprogrammed only. If the WDTON fuse is programmed, the WDT System Reset Mode is always enabled. How to change the fuse settings is described in the device datasheets.

If the WDT is intentionally used in the application and a system reset occurs, the startup code should have a scheme for handling the WDT Reset Flag. The easiest solution is to just ignore the flag and continue as usual. This approach saves the system from bugs appearing occasionally, but has no way of handling repeated or persistent errors.

A possible extension is to keep a WDT system reset counter in non-volatile memory. The startup code should then shut down the system safely and notify the operator if this counter exceeds a predefined limit. Using some sort of system clock tick (backed up in non-volatile memory), the startup code can also try to detect repeated resets over a fixed period of time.

If parameter backup is used, the startup code should check the Write Complete flag described in the Parameter Backup section and try to restore the system to a safe state, or at least be able to supply some debugging information to the operator.

2.5 Changing the WDT Configuration

To prevent accidental changes to the WDT configuration, special timed sequences are needed to disable WDT System Reset Mode or change the timeout period.

To disable the WDT System Reset Mode, the Watchdog Change Enable bit must be set within four CPU clock cycles prior to the disabling. If not, the WDT System Reset Mode will stay enabled. If the WDTON fuse is programmed the WDT System Reset Mode is always enabled.

To change the timeout period, the Watchdog Change Enable bit must be set within four CPU clock cycles prior to changing the timeout value. It is however not recommended to change the timeout period during normal operation. This should be done once in the initialization code.

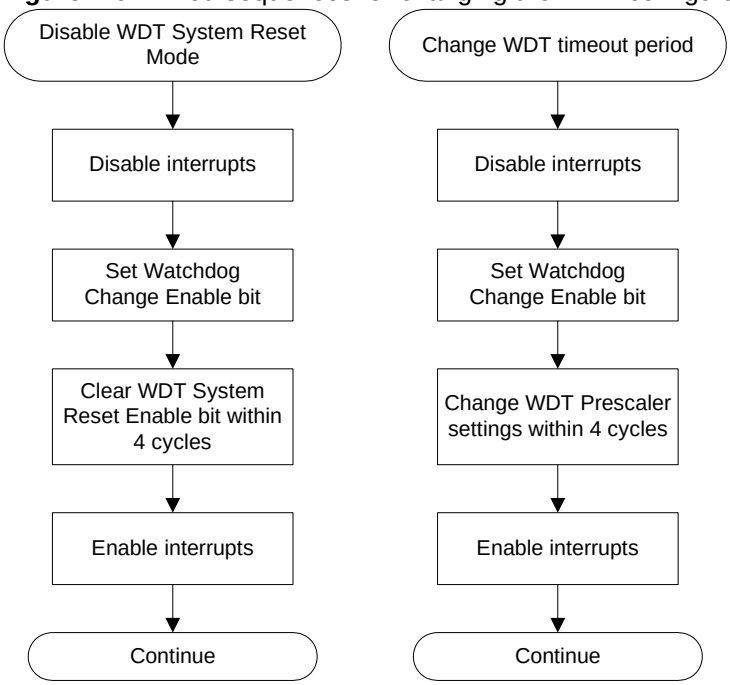
If the WDTON fuse is unprogrammed on ATtiny13 and ATtiny2313, it is possible to change the WDT timeout period without following the timed sequence.

Changing the WDT Interrupt Mode setting or enabling the WDT System Reset Mode needs no special considerations.

Interrupts should be disabled when changing the configuration. This ensures that no interrupts occur, causing the 4-cycle limit to expire.

Flowcharts for changing the WDT configuration are shown in Figure 2-6.

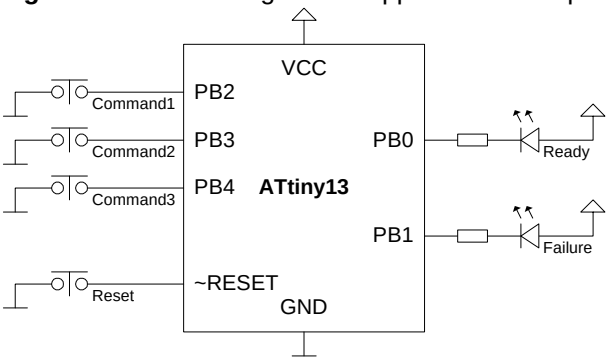
Figure 2-6. Timed sequences for changing the WDT configuration.



3 Implementation

This application note provides three code examples written in C. They are all designed for the ATtiny13 device placed on the STK®500 development board or similar. The ports PB0 and PB1 are connected to a ready-LED and a failure-LED respectively, and PB2, PB3 and PB4 are connected to three of the STK500 switches. Note that driving an output low turns on a LED, and pressing a button drives the corresponding input low. The setup is shown in Figure 3-1.

Figure 3-1. Circuit diagram for application example.



The examples demonstrate the following concepts:

- Using the WDT as a system reset source (section 3.1).
- Using the WDT as a Wakeup Timer (section 3.2).
- Using the WDT as a combined Wakeup Timer and system reset source with parameter backup (section 3.3).

Note: The WDTON fuse must be unprogrammed when running the examples using the WDT Interrupt Mode.

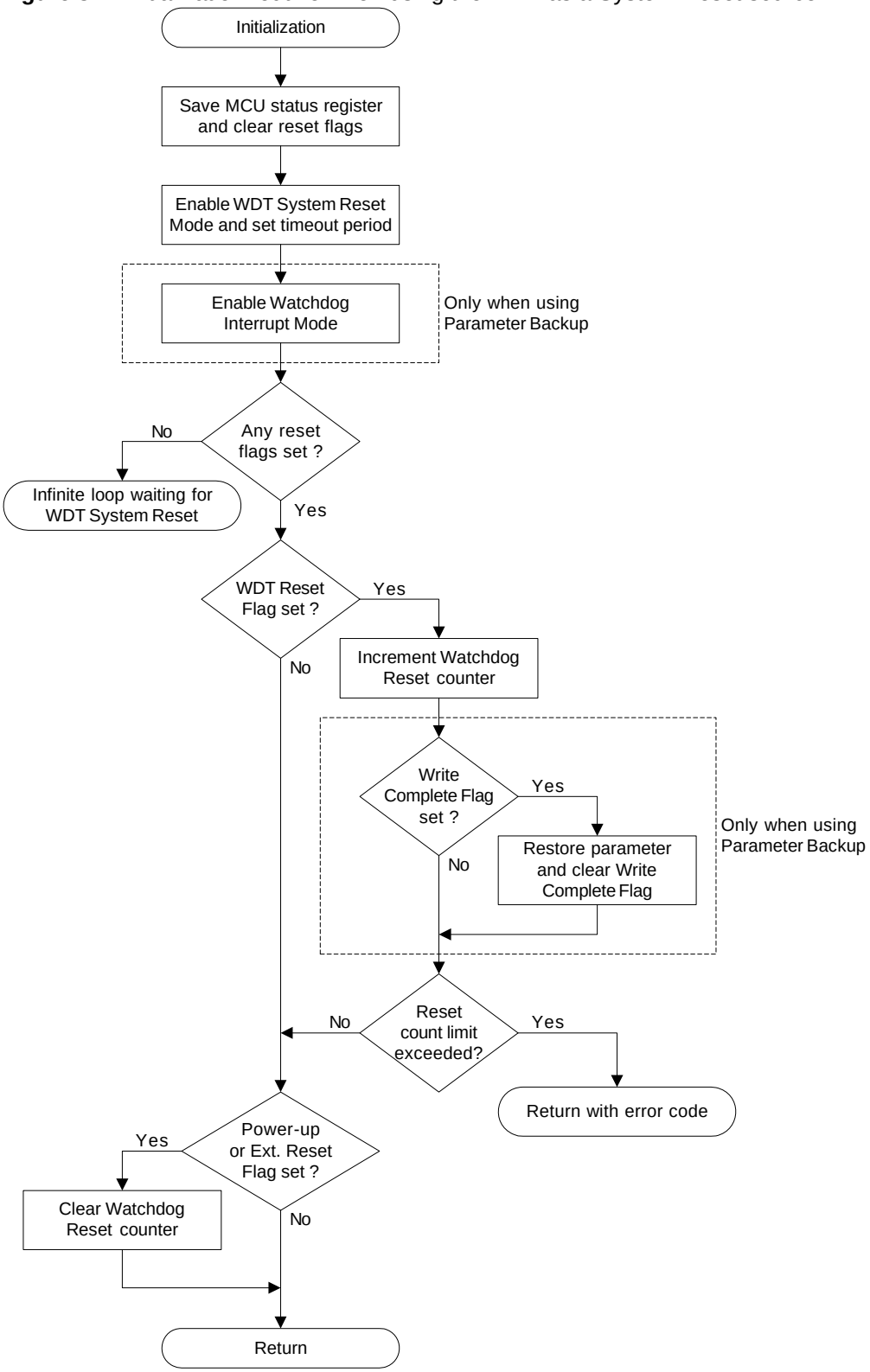
3.1 Using the WDT as a System Reset Source

This example implements the structure described in Figure 2-2, with an initialization routine and a main loop with three routines and a health check at the end. Each routine has its own health flag to indicate that everything is ok. The three routines get a command, parse and execute it, respectively.

3.1.1 Initialization

The initialization routine has two main purposes: initializing peripherals and handling reset flags. Its flowchart is shown in Figure 3-2. The parts inside the dashed frames are only used in the Combined Operation code example and are described later.

Figure 3-2. Initialization routine when using the WDT as a System Reset source.



The first conditional branch handles the case where no reset flags are set upon startup. Since the reset flags are always cleared in the initialization routine, this only happens when runaway code wraps back to address 0 and runs the startup code once again without a reset. This clearly indicates a bug or fault in software and is handled like a WDT System Reset. The initialization routine just enters an infinite loop and waits for the WDT to reset the device properly.

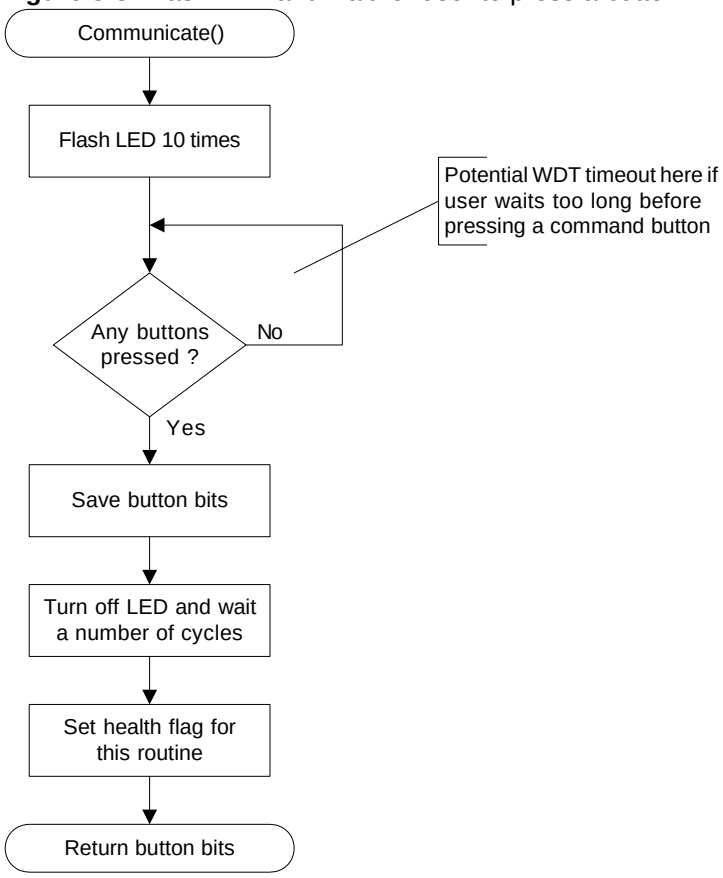
The code then checks the WDT Reset Flag. If it is set, the routine increments the WDT Reset counter and checks it against a predefined limit. If this limit is exceeded, the application assumes that there is a permanent repeating error and indicates this by turning on the failure indicator LED and halting execution. By entering an infinite loop with a WDR instruction inside, execution is effectively halted until an external reset occurs.

Power-up or external reset events are considered to be manual intervention and the WDT Reset counter is cleared. This makes it possible for a human operator to manually reset an application that has been halted by too many WDT System Resets. The operator must of course try to find the source of the WDT System Resets before resetting. Blindly resetting and hoping for things to fix themselves is not a recommended solution. The rest of the flowchart should be self-explanatory.

3.1.2 Communicate Command

The routine that gets a command is an example of a poorly designed communication routine. It flashes a LED 10 times and then waits for any button to be pressed. The problem arises when the user waits too long. A robust design should implement some sort of timeout check and return with an error code if the communication times out. However, this routine does not, and the WDT will reset the device if no button is pressed within the WDT timeout period. The flowchart for the communication routine is shown in Figure 3-3.

Figure 3-3. Flash LED and wait for user to press a button.



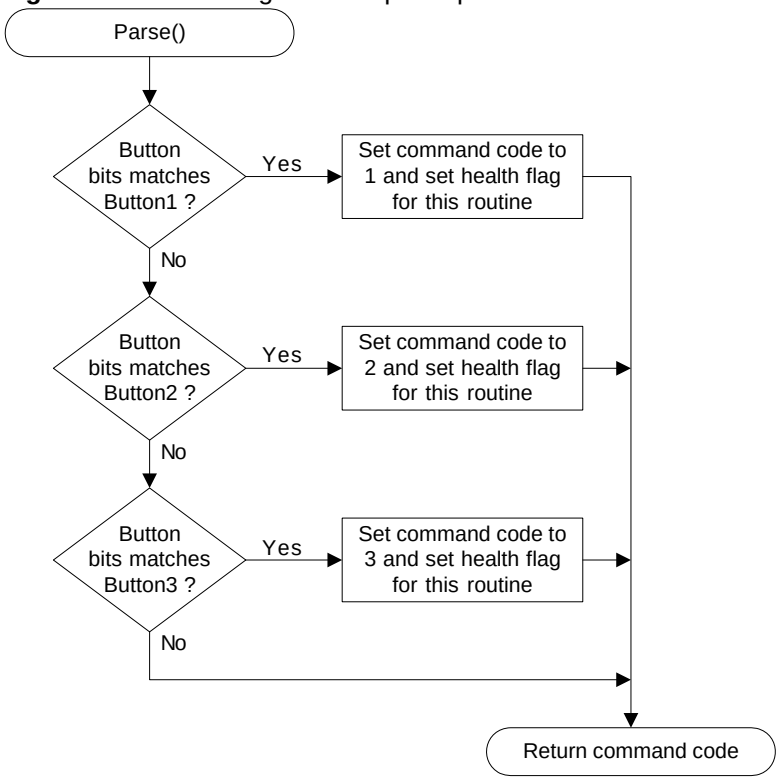
If a command button is pressed in time, the routine sets its health flag and returns the button press bit pattern.

3.1.3 Parse Command

The command parser uses the switch keyword in C to convert the button press bit pattern to a command code. The pattern is compared against the bit masks for each of the command buttons. When a match is found, the command code is set accordingly and the health flag for this routine is set.

The flowchart for the parser is shown in Figure 3-4.

Figure 3-4. Converting a button press pattern to a command code.



Note that if two or more command buttons are pressed simultaneously, the parser will never find a match and its health flag is never set. Because if this, the health check in the main loop will not reset the WDT, and a system reset could occur if the main loop is not executed successfully and quickly enough a second time. This is an example showing that unexpected inputs may cause problems if not handled by a default case in the switch block.

3.1.4 Execute Command

In this routine, the command code decides which action to perform.

Command 1 has no particular action, but it keeps the main loop running healthy by being a valid command. The other commands demonstrate various bugs that could occur in real life applications.

Command 2 enables the EEPROM Ready Interrupt. This interrupt is executed continuously as long as the EEPROM module is ready, which means always in this case, since the EEPROM is never used after the initialization routine. The EEPROM Ready Interrupt executing over and over slows down the main loop considerably, and a WDT System Reset will eventually occur. Command 2 is therefore an example showing how too many or poorly configured interrupts may slow down the main loop too much.

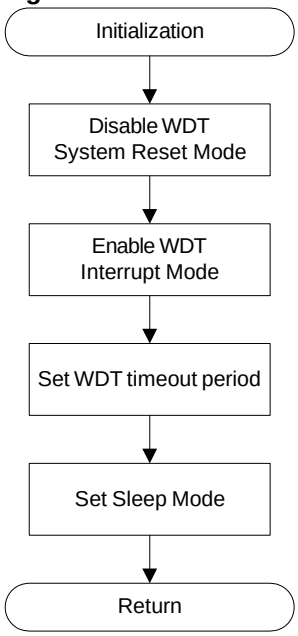
Command 3 gives an example of runaway code. This example just calls a function at an unused address. The program counter runs to the end of program memory and wraps back to address 0. No reset flags will be set and the fault is caught safely in the initialization routine.

To simulate the bad function call, the following code fragment is used: "`((void(*)()) 0x1FF)();`" The integer 0x1FF is converted to a pointer-to-a-function, and the function is called. Refer to the ANSI C standard for more details on function pointers and type conversions.

3.2 Using the WDT as a Wakeup Timer

This example only uses the WDT Interrupt Mode, and the initialization routine is thus quite reduced. As described earlier it is important to disable the WDT System Reset Mode upon startup even if the WDT System Reset Mode is never used. The initialization routine is shown in Figure 3-5.

Figure 3-5. Initialization routine when using the WDT as Wakeup Timer.



The main loop of this example flashes the LED connected to PB0 10 times to show that it is awake. It then resets the WDT, enables the WDT Interrupt Mode and enters sleep mode. When the WDT times out, it wakes up the CPU again. The interrupt handler disables WDT Interrupt Mode, so that no unnecessary interrupts are generated if the main loop runs long before entering sleep mode once again.

3.3 Combined Operation

The third example shows how to use the WDT both as a Wakeup Timer and system reset source with parameter backup. It is an extended version of the first code example, now using Command 1 to enter sleep mode.

In this example, the initialization routine includes the parts shown in dashed frames in the flowchart. This means that the WDT Interrupt Mode is enabled and backed up parameters are restored if the Write Complete flag is set upon startup.

The parameter to be backed up is the value of the Timer/Counter1. It has no particular function in this application, but serves as an example of a parameter that is cleared on reset and needs to be restored.

The WDT Interrupt handler is implemented as described in Dual purpose WDT Interrupt Handler. The Sleep Enable bit is used as a Wakeup flag. When Command 1 is executed, the application resets the WDT, sets the Sleep Enable bit and then enters sleep mode. The interrupt handler is executed when the WDT timeout wakes up the CPU, and the Wakeup flag decides what action to take. If it is already cleared, an error has occurred and the failure LED is lit. The rest of the interrupt handler implementation complies with the flowchart.

The rest of the code is the same as described in the first example.

4 Literature References

- Michael Barr – Introduction to Watchdog Timers
  http://www.embedded.com/story/OEG20010920S0064
- Niall Murphy – Watchdog Timers
  http://www.embedded.com/2000/0011/0011feat4.htm
- Jack Ganssle – Born to Fail
  http://www.embedded.com/design_library/OEG20021211S0032
- Kernighan & Ritchie – “The C Programming Language”, 2nd edition.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature

2551C-AVR-06/08