Watchdog Timers

Klaus-Peter Zauner

COMP2215: Computer Systems II
Fault Recovery
Autonomous Fault Recovery

Typical embedded systems need to operate continuously, but cannot rely on human supervision:

- system not easily accessible
  - implanted device
  - space probe
- human supervision is too costly
- human supervision is too slow

⇒ use a watchdog to supervise the system
Watchdog

• Hardware counter with its own Clock ⇒ Timer
• When timer runs out system receives a hardware reset  “dog bites”
• Software in normal operation restarts the timer before it runs out  “kick the dog”
Software faults

- Endless loops
- Interrupt flood
- Deadlocks from multitasking
Watchdog in the AT90USB1286
AT90USB1286

- Programmable Watchdog Timer\(^1\)
  - with separate on-chip oscillator
- Time-out of the watchdog can cause:
  1. a RESET of the system
  2. an Interrupt (WDT)
  3. an Interrupt, and switch to Reset mode
- Time range: 16ms–8s
  - also: voltage and temperature dependent

\(^1\)Datasheet section 9.8
Acting on the Watchdog

- **Interrupt**
  ⇒ ISR on WDT will take action

- **Reset**
  ⇒ Check Watchdog Reset Flag (WDRF) in MCU status register to recognise if the system was reset due to a watchdog time out
  - Reset peripheral hardware!
Usage scenarios

- Wake up from sleep → **interrupt**
  - other oscillators can be powered down
Usage scenarios

• Wake up from sleep → interrupt
  • other oscillators can be powered down

• Time-out on operations → interrupt
Usage scenarios

• Wake up from sleep → *interrupt*
  • other oscillators can be powered down
• Time-out on operations → *interrupt*
• Exit system hangup → *reset*

1. log the fault
2. enter endless loop
Usage scenarios

- Wake up from sleep → *interrupt*
  - other oscillators can be powered down
- Time-out on operations → *interrupt*
- Exit system hangup → *reset*
- Recover from assertion fault → *reset*
  1. log the fault
  2. enter endless loop
    - watch out for real-time requirements
Usage scenarios

• Wake up from sleep → interrupt
  • other oscillators can be powered down
• Time-out on operations → interrupt
• Exit system hangup → reset
• Recover from assertion fault → reset
  1. log the fault
  2. enter endless loop
    • watch out for real-time requirements
• Recover from task death → interrupt & reset
  • verified task execution as condition for feeding the watchdog
Power Consumption

Note, that if the watchdog is enabled it will run in all sleep modes and always consume power. This may be the dominant power consumption in a deep sleep mode.

The watchdog’s oscillator is much slower than the typical main clock and requires therefore less power—it can be used to wake from sleep instead of a timer running off the main clock.
Protection I

It is important that a run-away program will not accidentally disable the watchdog timer before it times out!

- Watchdog can be set by fuse programming to be always on → fixed in Reset mode
Protection II

Configuration bits for reset mode and time-out are protected by a specific write sequence:

**Watchdog configuration**

1. In one operation write a 1 to:
   - the Watchdog change enable bit (WDCE)
   - the System Reset mode bit (WDE)

2. Within the next four clock cycles:
   - write desired value to WDE
   - write Prescaler bits (WDP0–WDP3)
   - write 0 to WDCE
Protection III

It is important that the watchdog timer is not accidentally disabling a system!

• If the watchdog is not used always clear its enabling bits at the start of program initialization
Protection III

*It is important that the watchdog timer is not accidentally disabling a system!*

- If the watchdog is *not* used always clear its enabling bits at the start of program initialization

**Why?**
avr-libc support

```
#include <avr/wdt.h>

Enable the watchdog with a given timeout period:

```
wdt_enable(WDTO_1S);
```

Restart the timer:

```
wdt_reset();
```

Disable the watchdog:

```
MCUSR = 0;
wdt_disable();
```

WDTO: 15 ms, 30 ms, 60 ms, 120 ms, 250 ms, 500 ms, 1 s, 2 s, 4 s, and 8 s
Initialization Period

• After the Watchdog has triggered it will stay enabled with the fastest prescale value²
• If program initialization takes longer than then the time out period the device will lock up in an endless reset cycle
• See the avr-libc manual for code to handle this

²Not the case on older AVR microcontrollers.
Initialization Period

• After the Watchdog has triggered it will stay enabled with the fastest prescale value\(^2\)
• If program initialization takes longer than then the time out period the device will lock up in an endless reset cycle
• See the avr-libc manual for code to handle this

• In general: count number of watchdog induced resets
  • consider to disable watchdog if this number seems excessive

\(^2\)Not the case on older AVR microcontrollers.
Watchdogs in Practice
Software Reset

Applications for the Watchdog Timer:

• Fault recovery
• Low power oscillator
  • wake from sleep
  • timing not precise

• Clean Software Reset
  1. Firmware update  
     → Bootloader
  2. Enable Watchdog
  3. Enter endless loop
Watchdogs differ from device to device

- Some watchdogs can only be enabled, but not disabled
- Some watchdogs allow an extra delay during the first time out period
  - account for initialization period
- Some watchdogs are disabled after a time-out
  - need to be enabled during initialization

Watchdogs have different modes—know which mode is active.
Stuck Tasks

• Time-outs should protect any task that waits for external events from locking up indefinitely
• Watchdog can be last defense or part of the time-out design
  • critical system?
  • cost critical?
Multi-tasking: Monitor Task

- Monitor task can watch over other tasks
  - This is the only task allowed to restart the watchdog
- All tasks report their operation by acting on a global set of flags
  - E.g., clearing the flag when task completes
- Monitor task only restart the watchdog if all tasks have reported success
Multi-tasking: Time Scales

Often Tasks run at very different time scales.

- A frequent high priority task may be critical
- A less frequent task will report at a much lower frequency

How can this be handled?
Event-driven Tasks

Tasks may arrive at unpredictable intervals

How can the execution of such tasks be monitored with a watchdog?
Robust Monitoring

Assume that random bit-flips can happen.

How can the task monitoring be hardened against not recognizing faults?
What to do when the dog bites?

1. Reset is often a good option
   - start from a clean state
   - keep periphery in mind
2. Try to make it known that there was a Watchdog reset
   - Flash LED
   - Log the event to EEPROM
Acknowledgments

These slides were produced with Teodor Nistor’s beamr:

https://teonistor.github.io/beamr/