Entropy gathering for cryptographic applications in AVR

-

Qualification of WDT as entropy source

Brief

In cryptographic applications, security often depends on the parties' ability to generate unpredictable session keys. Without unpredictable session keys, workhorse algorithms like Diffie-Hellman and RSA achieve strengths far below what would be expected from looking at their implementation, even to the point of being broken in seconds.

Unfortunately, randomness is hard. Microcontrollers and microprocessors are the pinnacle of deterministic machines, and if there's one thing they're really poor at, it's behaving in a non-deterministic fashion.

This problem cannot be solved in software. The needed randomness must come from a physical process – hardware – and it must not be possible for other parties to obtain (listen in on) or influence the data.

In the AVR family there are two possible internal sources of randomness: The A/D converter (for parts that have it), and the on-chip oscillators.

This implementation exploits the jitter in the on-chip oscillators. They have the benefit of being hard to observe from outside the chip.

Obtaining entropy from the watchdog

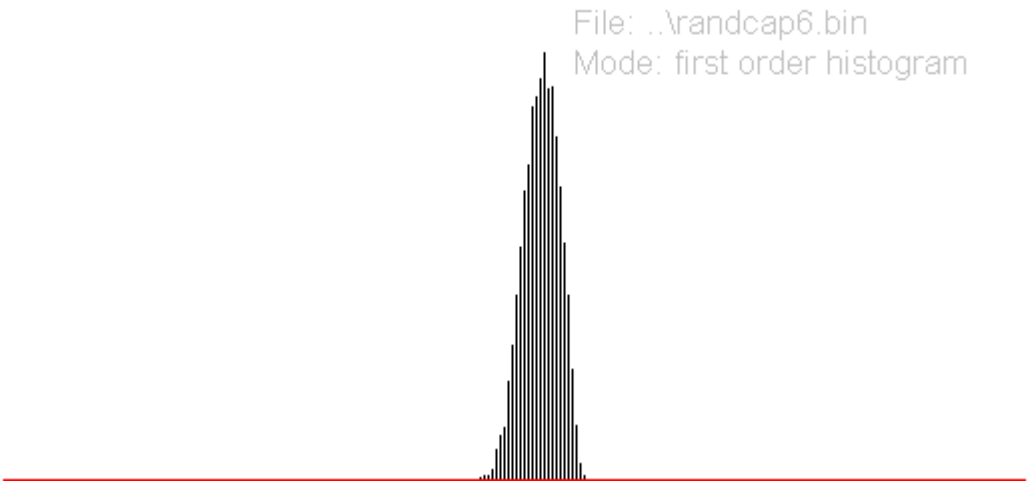
In AVR, the watchdog is clocked from an on-chip RC oscillator, and this oscillator is different from the core clock oscillator.

To see if it was a possible source of randomness, a small program was written, and executed in a Mega168 on an STK500 with 16MHz external clock. Power was supplied from a 12V lead-acid battery to rule out external power supply noise.

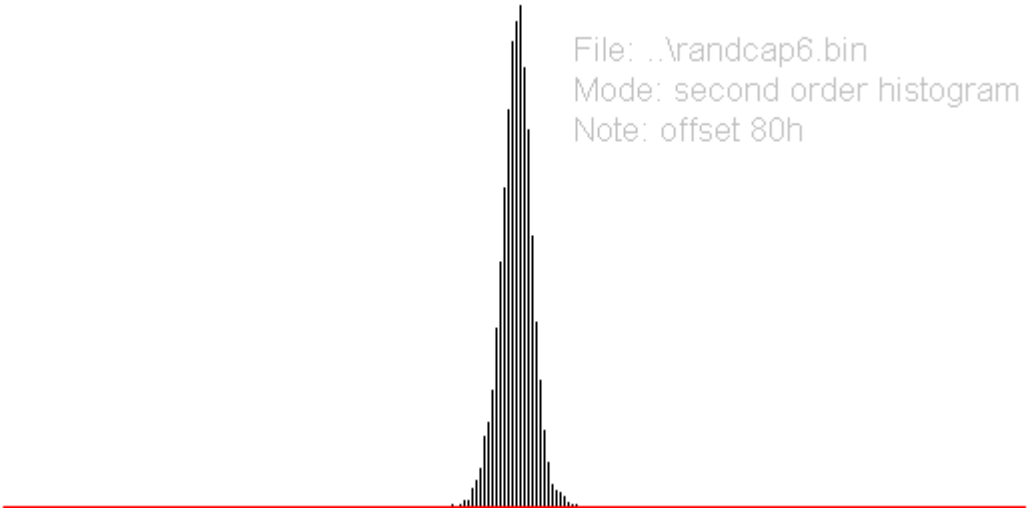
write r24 to the serial port
clear r24
arm the watchdog for 16 ms timeout
count like mad in r24 (3 clocks/iteration) until the watchdog strikes

The program can be found in appendix 1

And the result is promising. 10.000 values were gathered and analyzed. What we get is a probability graph of 'when does the watchdog fire'. If the oscillator was mathematically perfect, we'd get a single narrow line. In reality the on-chip oscillator drifts and jitters: If you were to look at its clock edge on an analog scope, it would be 'thick and fuzzy' (jitter) and move around (drift).



Each bin is 3 clocks wide, and it looks like there are at least 8 well-populated bins, so we might be able to extract 3 bits per sample. Life isn't that good to us, however. The differential value (sample[n] – sample[n-1]) is a little worse:



With goodwill this is 6 bins wide, so it's more like 2, or maybe 2.5, bits. The third order differential is going wide again, so this is probably realistic. But: There IS entropy to be obtained. So, let's distill it.

A practical implementation

The first implementation – just collecting samples – goes a long way to qualifying the hardware, but actually making a seed buffer from it is somewhat more cumbersome. The basic algorithm goes:

On power-on or external hardware reset, start the watchdog
On watchdog reset, get the count value, add it to an entropy buffer, and count like mad.
And if it was an application watchdog, do not catch it.

This implementation has a simple circular buffer, and adds samples into the buffer. To improve the entropy gathering, it includes a tiny (and totally insecure) 8-bit substitution cipher to mix up the bits slightly. It's 8 bits wide because the analysis tool looks at 8 bit bytes. MD5 would be a lot better, but hiding a poor distribution from the analysis tool is not the goal, and so we can't use MD5 for the analysis phase.
The code can be found in appendix 2.

The entropygather entrypoint is to be called early, before any start up code destroys R24.

Observed performance

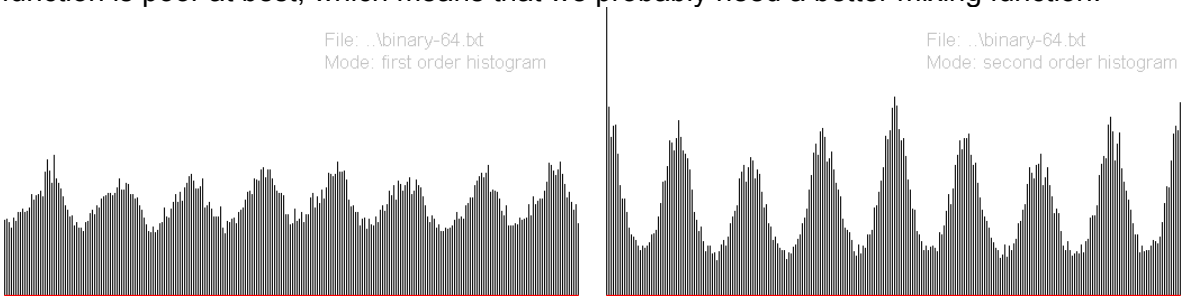
With a single pass of the buffer the result isn't good. This was predicted.

```
- B9 D2 B7 B8 B8 B5 B4 B6 B6 B5 B4 B5 B4 B7 B3 B5
- B6 D9 BB B6 B1 B1 B2 B4 B3 B3 B5 B7 B7 B5 B4 B6
- B5 D5 B3 B4 B1 B3 B4 B4 B6 B2 B5 B3 B3 B9 B6 B6
- B3 D4 B8 B5 B3 B3 B7 B7 B5 B4 B4 B4 B4 B5 B6 B4
- B6 D4 B7 B7 B6 B5 B3 B2 B4 B1 B0 B1 B0 B0 B2 B5
- B5 D7 BC B6 B7 B3 B4 B4 B6 B8 B7 B6 B5 B4 B4 B3
- BA D1 B2 B4 B8 B6 B8 B7 B2 B1 B0 B9 B9 BB BC BB
- BB D8 B7 B8 B8 B5 B6 B7 B2 B6 B7 BB B7 B9 BC BD
- B3 D7 BC B7 B9 B8 B7 B8 B7 BA B8 BC BB B9 B5 B3
- B2 D1 BB B9 B8 B9 B8 B1 B5 B6 B1 B3 B1 AF AF B4
```

The entropy is there, and there's about two or three bits' worth. With 4 passes (as in the code given), it looks usable:

```
- C0 31 E9 E6 06 4B EF 2D DF DC 00 7B 79 8E 78 33
- E6 D5 06 E5 27 EF 30 78 72 EA 24 79 AD 35 DA 07
- 42 CF BF D7 E0 C6 35 8D 91 2E 8F C5 68 06 47 82
- 49 04 BB 81 27 8A 25 05 10 3C 69 B9 DF 65 E8 A2
- 83 DF B0 D8 E5 01 6E 96 5C 97 3D 27 90 49 49 42
- 58 23 2D 5A 5D 35 82 58 E5 9E 8F B4 5E 33 CA 3E
- 98 DA 6B 47 EE 94 80 8F F8 07 A3 AA 8B 49 E8 E0
- 59 77 C2 41 00 C0 21 C3 5E D0 FB 76 CD A4 1A 01
- C0 D9 0D F0 F9 1B E6 96 4F 6A 55 6F 6C CC 05 48
- 6B 24 32 FB AA F9 5F CD 09 27 E7 08 34 FB 7C 78
```

And at this point encodings like Lempel-Ziv (zip) aren't able to compress it.
It does shows some funny statistical anomalies though, the reason being that the mixing function is poor at best, which means that we probably need a better mixing function.



Not pretty at all.

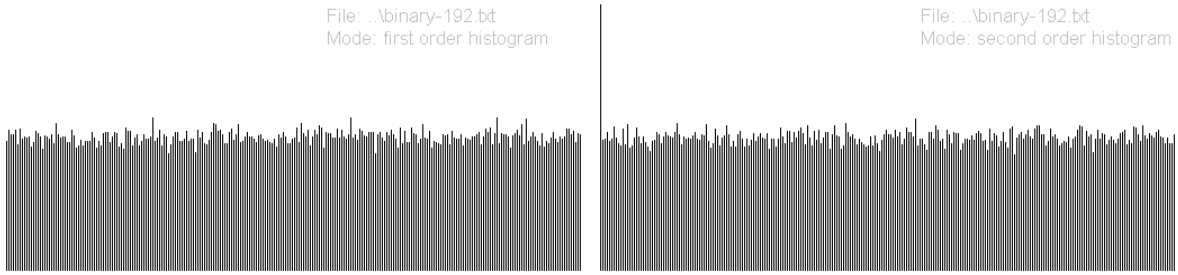
Third implementation

A single multiply-by-123 is added to the mixing function; This is lossless, and doesn't conceal things from the histogram plots. *The code can be found in appendix 3.*

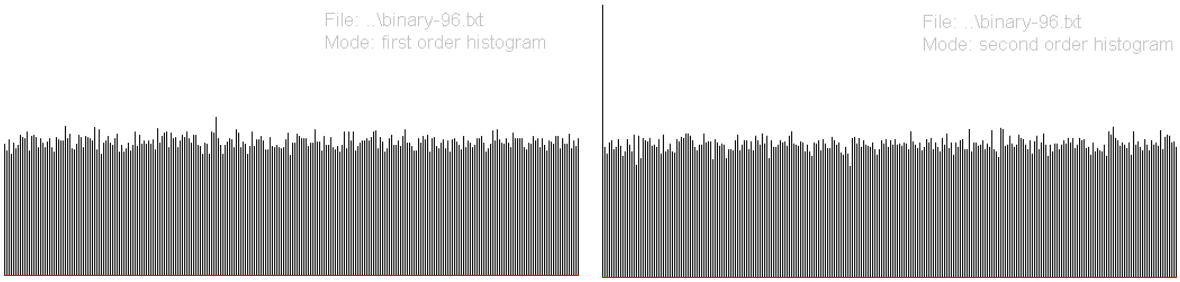
We rerun the experiment with the improved mixing function, using 192, 96, 64, 48, and 32 samples per 128 output bits. This is 0.7, 1.3, 2.0, 2.7, and 4.0 bits of entropy required from each sample. We already know that there isn't 4 bits of entropy available, so the last run should show structure. The 2.7-bits-of-entropy run is an unknown.

The following plots were all generated from 122000 bytes of output (7625 data sets) each, and scaling is fixed.

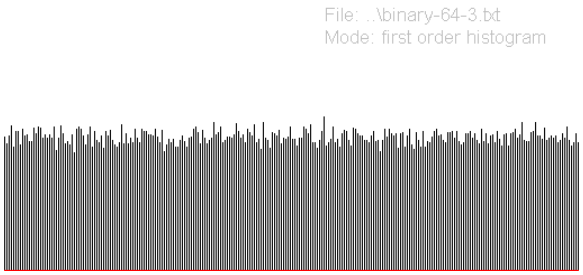
With 192 samples forming a 128 bit output:

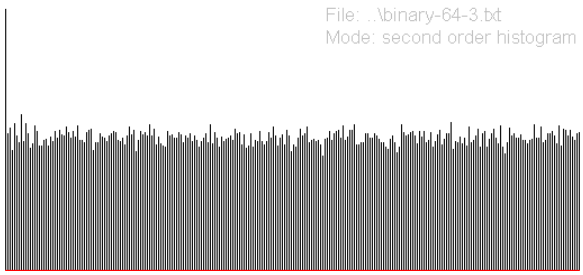


With 96 samples forming a 128 bit output:



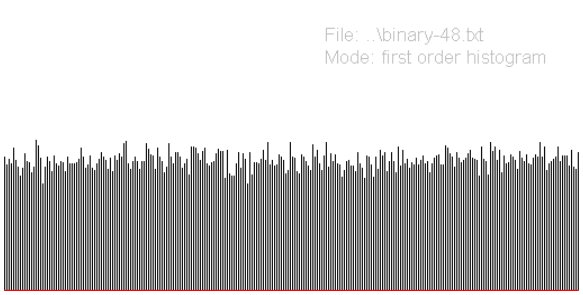
with 64 samples forming a 128 bit output:

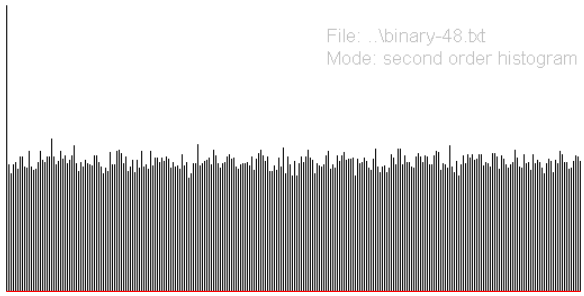




All of these share one common characteristic: The probability that you'll get two identical bytes in a row is twice that of other combinations; For practical purposes it doesn't matter, and I suspect it's a bug in either the analysis tool or the data collection software.
And all of them would be suitable for cryptographic applications.

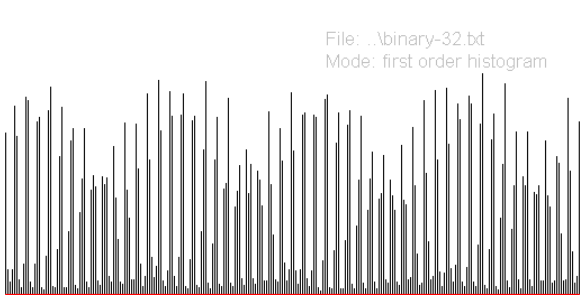
With 48 samples forming a 128 bit output:

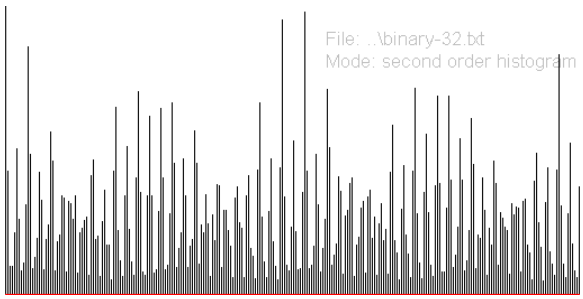




This is pretty good, and means that there is at least 2.66 bits of entropy to be obtained from each sample.

The 32-samples-in-128-bits version should, and does, fail:





These two are scaled to 50% vertically, as they would otherwise exceed the scale.

When there isn't enough entropy, because the mixing function's block size (one byte) is the same as the analysis' tool's, the histograms become sparse.

Using the entropy buffer

A note to people new to using entropy buffers:

While it might seem like the thing to do, do not use the entropy in the buffer directly. The entropy is valuable in that it takes time to obtain (as in resetting the microcontroller 64 times), and there really isn't any reason for sending it to third parties.

Instead, use it as initialization for a pseudo-random number generator. If you are doing cryptography, chances are that you have a block cipher or hash. Block ciphers make excellent cryptographically secure pseudo-random number generators, look up CTR mode. The same goes for hashes (hash the buffer and a counter, but do search the literature to see if there are known vulnerabilities in your chosen hash).

Conclusion

The on-chip watchdog timer provides more than 2 bits per sample when compared against an external 16MHz clock every 16 ms (the fastest watchdog timeout available). If the clock frequency is only 8MHz, expect 1 bit of entropy per sample.

- With external 16MHz+ clock, 125 bits per second of entropy
- With external 8-16MHz clock, 63 bits per second of entropy
- The internal oscillator was not tested; Most likely it will work as well, but test!

The entropy mixing function used in the third implementation works, but for real life applications, replacing it with something else would be good from a trust perspective.

Second, the mixing function used here is fragile. Trying to optimize it by throwing out operations gave poor results, so if you modify it, and don't have a rock solid understanding of what it needs to do, do analyze it afterwards.

Appendix 1: Feasibility test code

```
;
; Does the watchdog fire with a good random spread?
;
.INCLUDE "m168def.inc"
.org 0
rjmp mainp

.org 0x40

mainp:
      ldi r16,0xFF
      ldi r17,0x04
      out SPL,r16
      out SPH,r17

      ;set up uart
      ;16M/19200/16 = 52

      ldi   r16,52-1
      sts UBRR0L,r16 ;reload
      ldi r16,0x06
      sts UCSR0C,r16 ;8 bit
      ldi r16,0x18
      sts UCSR0B,r16 ;enable

      in   r16,MCUSR
      andi r16,0x0F
      cpi  r16,0x08
      breq acquiremode

      ;power on, not acquisiton. so.
      clr  r16
      out  MCUSR,r16
      wdr
      ldi  r16,0x08
      sts  WDTCSR,r16

stopp:  rjmp stopp;

acquiremode:
      clr  r16
      out  MCUCR,r16

      mov  r16,r24
      sub  r16,r25     ;to count increment distance
      mov  r25,r24
      rcall sendc
      ldi  r16,0x08
      sts  WDTCSR,r16
acqloop:
      inc   r24
      rjmp  acqloop
```

```
sendc:
     lds  r17,UCSR0A
     sbrs r17,5
     rjmp sendc
     sts  UDR0,r16
     ret
```

Appendix 2: CRNG 2 source (practical RNG test 1, poor mixing function)

UART code not included, identical to the code in appendix 3

```
.dseg
eg_seedbuf:   .byte 16
eg_seedptr:   .byte 1
eg_seedstate: .byte 1

.cseg
entropygather:
     ;Look at status register to decide what the situation is.
     in   r30,MCUSR
     ;is this an external reset or power on?
     andi r30,(1<<BORF)|(1<<EXTRF)|(1<<PORF)
     brne eg_init ;if so, we want to gather.
     ;it could also be watchdog
     in   r30,MCUSR
     andi r30,(1<<WDRF)
     brne eg_feed   ;watchdog fired.
     ;and if not, someone called the reset vector
     ret

eg_init:
     clr  r30                        ;Clear all the reset flags,
     sts  eg_seedptr,r30             ;null the feed counter,
     sts  eg_seedstate,r30           ;and state
     out  MCUSR,r30                  ;and power-on flags.
     rcall eg_clear                  ;only for testing
eg_wait:
     clr  r24 ;not needed in an actual implementation
     ;enable the watchdog and wait for it to fire.
     wdr
     ldi  r30,0x08
     sts  WDTCSR,r30
eg_wait2:
     inc  r24                       ;spinning around like mad.
     rjmp eg_wait2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
eg_feed:
     ;was it an application watchdog, and not a sampling wdog?
     lds  r30,eg_seedstate
     or   r30,r30
     brne eg_done

     ;watchdog fired. accumulate data.
     rcall eg_stir        ;takes r24 as argument
     ;now we have ~15 ms until the next kick
     ;have enough in buffer?
     lds  r30,eg_seedptr
     cpi  r30,64                    ;four passes, please
     brne eg_wait ;no..
```

```
eg_done:
      sts  eg_seedstate,r30 ;all done.
      wdr
      ldi  r30, (1<<WDCE) | (1<<WDE)    ;remove watchdog
      ldi  r31, 0
      out  MCUSR, r31 ;M168:cannot disable wdog if WDRF is set.
      sts  WDTCSR,r30
      sts  WDTCSR,r31
      ret

;;;;;;;;;;;;;;;;;;;
; simple entropy gatherer.
; Yes, it's not much, but it's low-cost, so it can
; feed continuously from other places too..
eg_stir:
      lds  r30,eg_seedptr
      inc  r30
      sts  eg_seedptr,r30
      andi r30,15           ;wrap to buffer
      clr  r31
      subi r30,low(-eg_seedbuf)  ;base of buffer
      sbci r31,high(-eg_seedbuf)

      ld   r16,Z                ;fetch
      swap r16                  ;stir some
      add  r16,r16              ;rotate
      sbci r16,0                ;and feed in inverted carry.
      add  r16,r24              ;add in data
      st   Z,r16                ;store
      ret

      ;this is purely to make it less random - TEST ONLY
eg_clear:
      ldi r30,low(eg_seedbuf)  ;base of buffer
      ldi r31,high(eg_seedbuf)
      clr r24
eg_cloop:
      st  Z+,r24
      cpi r30,low(eg_seedbuf+16)  ;done?
      brne eg_cloop
      ret
```

Appendix 3: Real Use version complete with UART test/demo code

```
.INCLUDE "m168def.inc"
.dseg

eg_seedbuf:   .byte 16
eg_seedptr:   .byte 1
eg_seedstate: .byte 1

.cseg

.org 0
      rjmp mainp

mainp:
```

```
      ldi r16,0xFF
      ldi r17,0x04
      out SPL,r16
      out SPH,r17
      rcall   entropygather ;called early, once stack is okay.


      ;set up uart and report the frames
      ;16M/19200/16 = 52
      ldi   r16,52-1
      sts UBRR0L,r16 ;reload
      ldi r16,0x06
      sts UCSR0C,r16 ;8 bit
      ldi r16,0x18
      sts UCSR0B,r16 ;enable

      ldi r30,low(eg_seedbuf)  ;base of buffer
      ldi r31,high(eg_seedbuf)
      clr r24
cloop:
      ld  r16,Z+
      rcall sendc
      cpi r30,low(eg_seedbuf+16)  ;done?
      brne cloop

      ;rjmp eg_init   obtain-forever-option for analysis
stopp:  rjmp stopp;

sendc:
      lds  r17,UCSR0A
      sbrs r17,5
      rjmp sendc
      sts  UDR0,r16
      ret


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The entropy gatherer
;
entropygather:
      ;Look at watchdog to decide what the situation is.
      in   r30,MCUSR
      ;is this an external reset or power on?
      andi r30,(1<<BORF)|(1<<EXTRF)|(1<<PORF)
      brne eg_init ;if so, we want to gather.
      ;it could also be watchdog
      in   r30,MCUSR
      andi r30,(1<<WDRF)
      brne eg_feed   ;watchdog fired.
      ;and if not, someone called the reset vector, or it's watchdog.
      ret

eg_init:
      ;right here you null out the buffer if you're analyzing
      ;the mixing function. We aren't any longer, so no code.
      clr  r30                    ;Clear all the reset flags,
      sts  eg_seedstate,r30       ;and state
      out  MCUSR,r30              ;no more power-on flags.
      ;fall through to wait-for-data.
eg_wait:
      inc  r30
```

```
      sts  eg_seedptr,r30                ;feed count: will be 1.
      ;enable the watchdog and wait for it to fire.
      ldi  r30,0x08
      sts  WDTCSR,r30
      wdr
eg_wait2:
      inc  r24
      rjmp eg_wait2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
eg_feed:
      ;was it an application watchdog, and not a sampling wdog?
      lds  r30,eg_seedstate
      or   r30,r30
      brne eg_done
      ;watchdog fired. accumulate data.
      rcall eg_stir        ;takes r24 as argument
      ;have enough in buffer?
      lds  r30,eg_seedptr
      cpi  r30,64
      brne eg_wait ;no. still hungry.
eg_done:
      sts  eg_seedstate,r30 ;all done.
      wdr
      ldi  r30, (1<<WDCE) | (1<<WDE)    ;remove watchdog
      ldi  r31, 0
      out  MCUSR, r31 ;M168: you cannot disable wdog if WDRF is set.
      sts  WDTCSR,r30
      sts  WDTCSR,r31
      ret               ;and back to main for business as usual

;;;;;;;;;;;;;;;;;;;;
; simple entropy gatherer/compressor
; Tweak at your own peril
eg_stir:
      lds  r30,eg_seedptr
      inc  r30
      sts  eg_seedptr,r30
      andi r30,15           ;wrap to buffer
      clr  r31
      subi r30,low(-eg_seedbuf)  ;base of buffer
      sbci r31,high(-eg_seedbuf)
      ld   r16,Z                ;fetch
      swap r16                  ;stir some
      add  r16,r16              ;rotate
      sbci r16,0                ;and feed in inverted carry.
      add  r16,r24              ;add in data
      mov  r0,r16               ;the hastily added multiply
      ldi  r24,123
      mul  r0,r24
      st   Z,r0
      ret
```