January 8, 2015 at 20:51

1. ASED. With an emergency generator connected through an interlocked load-center, it's hard to tell when the *Ancillary Service* has been restored. Switching back to Main requiers shutting everything down for a moment. It would be good to know if main is live before swicking back. The obvious method is to measure the voltage at the main-breaker's input. The safety concern is that it's not breaker-protected making for a massive fault-current, should insulation be breached or the circuit shorted. Also, installation of a simple meter is somewhat involved, having to tap into live lines and, idealy, providing some form of isolation.

The obvious solution is to have a high-impedance connection very near to the source. A small capacitance would do. Simple capacitive coupling can be had with a "gimmick"; a technique used since the 1920s. This may be several turns of THHN around the large-gage insulated incomming line. Since the voltage is with respect to neutral, and neutral is bonded to ground, just the one wire is needed. No need to mess with live conductors, just coupling to the electric field through the insulation already present. Installation still has some risk, but much less.

With this signal, a circuit and be built to detect the difference between having AC and having no AC, and provide a signal to indicate that state. The signal provided to the generator-operator could be a lamp or buzzer.

The line-voltage is ± 170 V peak, with respect to ground. The peaks will be about 8.3 ms apart. The signal will be much less, depending on the circuit's input impedance and capacitance.

The circuit would need a high input impedance, so as to see a strong enough signal. The circuit would need a reference to ground to compare to. The input may need a bit of protection from line transients, which could pass trough the gimmick.

2. I had seven Ada Fruit Trinkets just laying around. They use the Atmel ATTINY85 processor. The analog inputs are about 100 MΩ. Not great, but I think it should be good enough. If we can muster 1 pf of gimmick, we will have $\frac{1}{2\pi f_c}$ of X_c. Ohms law indicates $100e6\frac{170}{(2pi*60*1e-12)^{-1}+100e6)} = 6.16$ volts peak, ignoring input pin capacitance. The steering diodes will keep the analog innards safe since the current is so low. Supply voltage at "BAT" is 5.5 to 16 V and it has a red LED on-board.

3. Here's the code

⟨ Header files to include 4 ⟩

4. hmm

/************************************ ASED Version 1.1.0r Ancillary Service Electric Detector 2015-01-01 Bjorn Burton

Just for fun. ************************************/

⟨ Header files to include 4 ⟩ ≡

```
#include "ased.h"
```

This code is used in section 3.

5. At the beginning the I/O is configured.

```
int main(void){      /* set the led port direction */
    DDRB |= (1 << LED_RED_DD);      /* set the siren port direction */
    DDRB |= (1 << SIREN_DD);      /* enable pin change interrupt for clear-button */
    PCMSK |= (1 << PCINT3);      /* General interrupt Mask register for clear-button */
    GIMSK |= (1 << PCIE);
```

6. The LED is set "assuming" that there will be an AC signal.

```
    /* turn the led on */
ledcntl(ON);
```

7. Here the timer and comparator are setup.

```
    /* set up the nowave timer */
initnowavetimer( );
```

8. The Trinket runs at a speedy 8 MHz so the slow 60 Hz signal is no issue. One could use the ADC but that doesn't make too much sense as the input may spend a lot of time cliped. The inbuilt comparator seems like the right choice, for now. /* set up the wave-event comparator */ initwavedetector();

9. Of course, any interrupt function requires that bit I "Global Interrupt Enable" is set; usualy done through calling sei().

```
    /* Global Int Enable */
sei( );
```

10. Rather than burning loops, waiting for something to happen for 16 ms, the sleep mode can be used. Interrupts are used to wake it.

```
    /* configure sleep_mode() to go to "idle". Idle allows the counters and comparator to continue
    during sleep. */
MCUCR &= ~(1 << SM1);
MCUCR &= ~(1 << SM0);
```

11. This is the loop that does the work.

```
for ( ; ; )      /* forever */
{
  static unsigned char nowaves = WAVETHRESHOLD;
  static unsigned int armwait = ARMTHRESHOLD;      /* hold-off to minimise noise susceptibilty */
  waveholdoff( );      /* now we wait in idle for any interrupt event */
  sleep_mode( );      /* some interrupt was detected! Let's see which one */
  if (f_state & (1 << WAVES)) {
    nowaves = (nowaves) ? nowaves - 1 : 0;
    if (¬nowaves)      /* ancilliary electric service restored */
    {
      ledcntl(ON);
      if (f_state & (1 << ARM)) chirp(ON);
      TCNT1 = TIMESTART;      /* reset the timer */
    }
    f_state &= ~(1 << WAVES);      /*reset int flag */
  }
  else if (f_state & (1 << NOWAVES)) {
    ledcntl(OFF);
    nowaves = WAVETHRESHOLD;
    chirp(OFF);      /* ASE dropped, stop alarm chirp */
    armwait = (armwait) ? armwait - 1 : 0;
    if (¬armwait ∧ ~f_state & (1 << ARM)) f_state |= (1 << ARM);
        /* at this time the only way to disarm is a power cycle */
    f_state &= ~(1 << NOWAVES);      /*reset int flag */
  }
}
return 0;      /* it's the right thing to do! */
}
```

12. Siren function will arm after a 10 minute power-loss; that is, the Trinket is running for a full 10 minutes without seeing AC at pin #2. Once armed, siren will chirp for 100 ms at a 5 second interval, only while AC is present. It may be disarmed, stopping the chirp, by pressing a button or power-cycle.

```
    /* Alarm Pulsing function */
  void chirp(char state)
  {
    static unsigned char count = CHIRPLENGTH;
    count = (count) ? count − 1 : CHIRPPERIOD;
    if (count > CHIRPLENGTH ∨ state ≡ OFF) sirencntl(OFF);
    else sirencntl(ON);
  }     /* simple led control */
  void ledcntl(char state)
  {
    PORTB = state ? PORTB | (1 ≪ LED_RED) : PORTB & ∼(1 ≪ LED_RED);
  }     /* simple siren control */
  void sirencntl(char state)
  {
    PORTB = state ? PORTB | (1 ≪ SIREN) : PORTB & ∼(1 ≪ SIREN);
  }     /* configure the no-wave timer */
  void initnowavetimer(void)
  {     /* set a very long prescal of 16384 counts */
    TCCR1 = ((1 ≪ CS10) | (1 ≪ CS11) | (1 ≪ CS12) | (1 ≪ CS13));
      /* Timer/counter 1 f_overflow interupt enable */
    TIMSK |= (1 ≪ TOIE1);
  }     /* configure the the wave detection comparator */
  void initwavedetector(void){
```

13. The ideal input AN1, PB1, is connected to the LED in the Trinket! That's not a big issue since the ADC's MUX may be used. That MUX may address PB2, PB3, PB4 or PB5. Of those, PB2, PB3 and PB4 are available. Since PB3 and PB4 are use for USB, PB2 makes sense here. This is marked "#2" on the Trinket. PB2 connects the the MUX's ADC1. Use of the MUX is selected by setting bit ACME of port ADCSRB. ADC1 is set by setting bit MUX0 of register ADMUX

```
    /* Setting bit ACME of port ADCSRB to enable the MUX input ADC1 */
  ADCSRB |= (1 ≪ ACME);     /* ADC1 is set by setting bit MUX0 of register ADMUX */
  ADMUX |= (1 ≪ MUX0);
```

14. Disable digital input buffers at AIN[1:0] to save power. This is done by setting AIN1D and AIN0D in register DIDR0.

```
    /* Disable digital inputs to save power */
  DIDR0 |= ((1 ≪ AIN1D) | (1 ≪ AIN0D));
```

15. Both comparator inputs have pins but AIN0 can be connected to a reference of 1.1 VDC, leaving the negative input to the signal. The ref is selected by setting bit ACBG of register ACSR.

```
    /* Connect the + input to the band-gap reference */
  ACSR |= (1 ≪ ACBG);
```

16. It can be configured to trigger on rising, falling or toggle (default) by clearing/setting bits ACIS[1:0] also on register ACSR. There is no need for toggle, and falling is selected by simply setting ACIS1.

```
    /* Trigger on falling edge only */
  ACSR |= (1 ≪ ACIS1);
```

17. To enable this interrupt, set the ACIE bit of register ACSR.

```
    /* Enable the analog comparator interrupt */
  ACSR |= (1 << ACIE); }
```

18. A timer is needed to to encompase some number of waves so it can clearly discern on from off. The timer is also interrupt based. The timer is set to interrupt at overflow. It could overflow within about 1/2 second. Over the course of that time, 25 to 30 comparator interrupts are expected. When the timer interrupt does occour, the LED is switched off. Comparator Interrupts are counted and at 15 the timer is reset and the LED is switched on.

```
    /* Wave detection Hold-Off or debounce */
  void waveholdoff()
  {    /* Disable the analog comparator interrupt */
    ACSR &= ~(1 << ACIE);
    _delay_us(WAVEHOLDOFFTIME);      /* Enable the analog comparator interrupt */
    ACSR |= (1 << ACIE);
  }    /* Timer ISR */
  ISR(TIMER1_OVF_vect)
  {
    f_state |= (1 << NOWAVES);
  }
```

19. The event can be checked by inspecting (then clearing) the ACI bit of the ACSR register but the vector ANA_COMP_vect is the simpler way.

```
    /* Comparator ISR */
  ISR(ANA_COMP_vect)
  {
    f_state |= (1 << WAVES);
  }    /* Clear Button ISR */
  ISR(PCINT0_vect)
  {
    if (PORTB & (1 << ARMCLEAR)) f_state &= ~(1 << ARM);
  }
```

⟨ Header files to include 4 ⟩ Used in section 3.