January 11, 2015 at 16:34

1. ASED Version 1.1.0r Ancillary Service Electric Detector Bjorn Burton

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.

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.

In use, the AC signal goes to the pin marked "#2" on the Trinket. The LED port is marked "#1". The Siren port is marked "#0". Clear, should it be implemented, is on "#3".

```
#define F_CPU 8000000UL
#define ARMCLEAR PORTB3 /* Trinket's Clear is pin #3 */
#define ARMCLEAR_DD DDB3 /* Boolean */
#define ON 1
#define OFF 0
#define SET 1
#define CLEAR 0 /* Flags for f_state */
#define NOWAVES 2 /* no ASE detected for some time */
#define WAVES 1 /* ASE detected */
#define ARM 0 /* ARM for Alarm */ /* Parameters */
#define WAVETHRESHOLD 15 /* Number of wave before considering the service on. Range to 255 */
 /* About 250 ms. Don't take too long or time will run out */
#define TIMESTART 12 /* preset for the timer counter. Range to 255 */ */ /* Prescaler is set to
    clk/16484. 0.5 seconds *(8e6/16384) is 244.14. 256-244 = 12, leaving 500 ms to time-out */
#define WAVEHOLDOFFTIME 100 /* Range to 255 */ /* hold-off time in us for wave detection */
#define ARMTHRESHOLD 1200 /* Range to 65535 */ /* alarm arm delay in nowave counts*/
 /* chirp parameters for alarm */
#define CHIRPLENGTH 7 /* number of waves long */
#define CHIRPPERIOD 200 /* number of waves long */
#define LED_RED PORTB1 /* Trinket's LED and pin #1 */
#define LED_RED_DD DDB1 /* siren port */
#define SIREN PORTB0 /* Trinket's Siren and pin #0 */
```

```
#define SIREN_DD DDB0
#include <avr/io.h>     /* need some port access */
#include <util/delay.h>     /* need to delay */
#include <avr/interrupt.h>     /* have need of an interrupt */
#include <avr/sleep.h>     /* have need of sleep */
#include <stdlib.h>
```

2. ⟨ Prototypes 2 ⟩ ≡

```
void ledcntl(char state);     /* LED ON and LED OFF */
void sirencntl(char state);     /* alarm siren control */
void chirp(char state);     /* alarm siren modulation */
```

This code is used in section 4.

3. The f_state global variable needs the type qualifier 'volatile' or optimization may eliminate it. f_state is just a simple bit-flag that keeps track what has been handled.

⟨ Global variables 3 ⟩ ≡

```
volatile unsigned char f_state = 0;
```

This code is used in section 4.

4. Atmel pins default as simple inputs so they need to be configured to use them for output. Additionaly, we need the clear button to wake the device through an interrupt.

```
⟨ Prototypes 2 ⟩
⟨ Global variables 3 ⟩
int main(void){ ⟨ Initialize pin outputs and inputs 13 ⟩
```

5. The LED is set, meaning 'on', assuming that there is an AC signal. The thought is that it's better to say that there is AC, when there isn't, as opposed to the converse.

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

6. Here is the timer and comparator are setup.

```
    /* set up the nowave timer */
⟨ Initialize the no-wave timer 15 ⟩
```

7. The Trinket runs at relativly 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. We just need to know when the signal changes. The inbuilt comparator seems like the right choice, for now.

```
    /* set up the wave-event comparator */
⟨ Initialize the wave detection 16 ⟩
```

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

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

9. Rather than burning loops, waiting for something to happen for 16 ms, the sleep mode is used. The specific type of 'sleep' is 'idle'. Interrupts are used to wake it.

```
⟨ Configure to wake upon interrupt 17 ⟩
```

10. This is the loop that does the work. It should spend most of its time in *sleep_mode*, comming out at each interrupt event. The ISRs alter the bits in *f_state*.

```
for ( ; ; )      /* forever */
{ static unsigned char nowaves = WAVETHRESHOLD;
static unsigned int armwait = ARMTHRESHOLD;
```

11. Now we wait in idle for any interrupt event

```
sleep_mode( );
```

12. Some interrupt has been detected! Let's see which one

```
if (f_state & (1 ≪ WAVES)) {
  nowaves = (nowaves) ? nowaves − 1 : 0;      /* countdown to 0, but not lower */
  if (¬nowaves)      /* ancillary electric service restored */
  {
    ledcntl(ON);
    if (f_state & (1 ≪ ARM))      /* now we annunciate this fact */
      chirp(ON);
    TCNT1 = TIMESTART;      /* reset the timer */
  }
  f_state &= ∼(1 ≪ WAVES);      /*reset int flag since actions are complete */
}
else if (f_state & (1 ≪ NOWAVES)) {
  ledcntl(OFF);
  nowaves = WAVETHRESHOLD;
  chirp(OFF);      /* ASE dropped, stop alarm chirp */
  armwait = (armwait) ? armwait − 1 : 0;      /* countdown to 0, but not lower */
  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 */
}
⟨ Hold-off all interrupts 18 ⟩} return 0;      /* it's the right thing to do! */
}
```

13. ⟨ Initialize pin outputs and inputs 13 ⟩ ≡

```
{     /* 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);
}
```

This code is used in section 4.

14. 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. In fact it is called with each AC cycle interrupt so that `CHIRPLENGTH` and `CHIRPPERIOD` are defined a multiples of $\frac{1}{Hz}$. It may be disarmed, stopping the chirp, by pressing a button or power-cycle.

```
void chirp(char state)
{
  static unsigned char count = CHIRPLENGTH;
  count = (count) ? count - 1 : CHIRPPERIOD;
  if (count > CHIRPLENGTH ∨ state ≡ OFF) sirencntl(OFF);
  else sirencntl(ON);
}
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);
}
```

15. 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.

⟨ Initialize the no-wave timer 15 ⟩ ≡

```
{    /*set a very long prescale of 16384 counts */
  TCCR1 = ((1 ≪ CS10) | (1 ≪ CS11) | (1 ≪ CS12) | (1 ≪ CS13));
    /* Timer/counter 1 f_overflow interrupt enable */
  TIMSK |= (1 ≪ TOIE1);
}
```

This code is used in section 6.

16. 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

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

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.

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.

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

⟨ Initialize the wave detection 16 ⟩ ≡

```
{    /* 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);    /* Disable digital inputs to save power */
  DIDR0 |= ((1 ≪ AIN1D) | (1 ≪ AIN0D));    /* Connect the + input to the band-gap reference */
  ACSR |= (1 ≪ ACBG);    /* Trigger on falling edge only */
  ACSR |= (1 ≪ ACIS1);    /* Enable the analog comparator interrupt */
  ACSR |= (1 ≪ ACIE);
}
```

This code is used in section 7.

17. Setting these bits configure sleep_mode() to go to "idle". Idle allows the counters and comparator to continue during sleep.

⟨ Configure to wake upon interrupt 17 ⟩ ≡

```
{
  MCUCR &= ∼(1 ≪ SM1);
  MCUCR &= ∼(1 ≪ SM0);
}
```

This code is used in section 9.

18. ⟨ Hold-off all interrupts 18 ⟩ ≡

```
{    /* Disable the analog comparator interrupt */
  ACSR &= ∼(1 ≪ ACIE);
  _delay_us(WAVEHOLDOFFTIME);    /* Enable the analog comparator interrupt */
  ACSR |= (1 ≪ ACIE);
}
```

This code is used in section 12.

19. This is the ISR for the main timer. When this overflows it generaly means the ASE has been off for a while.

```
    /* Timer ISR */
ISR(TIMER1_OVF_vect)
{
  f_state |= (1 ≪ NOWAVES);
}
```

20. 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);
}
```

21. This ISR responds to the Clear button.

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

22. Done

⟨ Configure to wake upon interrupt 17 ⟩ Used in section 9.
⟨ Global variables 3 ⟩ Used in section 4.
⟨ Hold-off all interrupts 18 ⟩ Used in section 12.
⟨ Initialize pin outputs and inputs 13 ⟩ Used in section 4.
⟨ Initialize the no-wave timer 15 ⟩ Used in section 6.
⟨ Initialize the wave detection 16 ⟩ Used in section 7.
⟨ Prototypes 2 ⟩ Used in section 4.