January 11, 2015 at 22:33

1. Introduction. This is the firmware portion of an Ancillary Service Electric Detector or ASED, written by Bjorn Burton.

With my emergency generator connected through an interlocked load-center, it's hard to tell when the *Ancillary Service* has been restored. The neighbor's lights offer a clue at night, but aren't reliable. Switching back to Main, from the genny, requiers shutting everything down for a moment. It would be good to know if main, or Ancillary Service, is live before switching back to it.

The obvious method is to measure the voltage at the main-breaker's input, using a meter. One 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 to get a "sample". 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.

This is still not Double insulated, so whatever gizmo is connected should provide an additional level of protection, but it's better than a copper connection.

With access to this signal, a circuit and be built to detect the difference between having AC and not having AC, providing a signal to indicate that state. The signal provided to the generator-operator could be a lamp or buzzer.

Looking at some numbers, the line-voltage is ± 170 V peak, with respect to ground. The peaks will be about 8.3 ms apart, in North America. The signal we will see will be much less, depending on the circuit's input impedance and capacitance, but should be good..

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 Adafruit Industry 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, PB2 on the chip, and in the Atmel datasheet. The LED port is marked #1, which is PB1. The Siren port is marked #0, and is PB0. Clear is on #3, PB3 on the chip.

2. Implementation and Specification. Extensive use was made of the datasheet, Atmel, Atmel ATtiny25, ATtiny45, ATtiny85 Datasheet Rev. 2586QAVR08/2013 (Tue 06 Aug 2013 03:19:12 PM EDT).

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.

3. `"F_CPU"` is used only to convey the Trinket clock to delay.h.

```
#define F_CPU 8000000UL
```

4. Here are some basic Boolean definitions that are used.

```
#define ON 1
#define OFF 0
#define SET 1
#define CLEAR 0
```

5. Bit flags within f_state.

```
#define NOWAVES 2 /* no ASE detected for some time */
#define WAVES 1 /* ASE detected */
#define ARM 0 /* ARM for Alarm */
```

6. `"WAVETHRESHOLD"` is maybe about 250 ms. Don't take too long or timer will overflow. This is the number of waves before considering the ASE 'on'. Range to 255.

```
#define WAVETHRESHOLD 15
```

7. The prescaler is set to clk/16484 at ⟨ Initialize the no-wave timer 25 ⟩. `"TIMESTART"` is the timer preset so that overflow happens in about 500 ms. The math goes: $0.5 seconds * (8e6/over16384) = 244.14$. Then to overflow, $256 - 244 = 12$, thus leaving 500 ms until time-out, unless it is reset.

```
#define TIMESTART 12 /* preset for the timer counter. Range to 255 */ */
```

8. This is the hold-off time in us for wave detection. This value is used by the `"\_delay\_us()"` function here ⟨ Hold-off all interrupts 28 ⟩.

```
#define WAVEHOLDOFFTIME 100 /* Range to 255 */
```

9. Alarm arm delay in "nowave" counts of whose size is defined by `"TIMESTART"`.

```
#define ARMTHRESHOLD 1200 /* Range to 65535 */
```

10. Chirp parameters for alarm. These unit are of period $\frac{1}{f}$.

```
#define CHIRPLENGTH 7 /* number of waves long */
#define CHIRPPERIOD 200 /* number of waves long */
```

11. ⟨ Include 11 ⟩ ≡

```
#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>
```

This code is used in section 14.

12. ⟨ Prototypes 12 ⟩ ≡

```
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 14.

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

⟨ Global variables 13 ⟩ ≡

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

This code is used in section 14.

14. Here is *main*. Atmel pins default as simple inputs so the first thing is to configure to use LED, Siren pins as outputs. Additionaly, we need the clear button to wake the device through an interrupt.

```
⟨ Include 11 ⟩
⟨ Prototypes 12 ⟩
⟨ Global variables 13 ⟩
int main(void){ ⟨ Initialize pin outputs and inputs 23 ⟩
```

15. 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);
```

16. Here the timer and comparator are setup.

```
⟨ Initialize the no-wave timer 25 ⟩
```

17. 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.

```
⟨ Initialize the wave detection 26 ⟩
```

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

```
sei( );
```

19. 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 27 ⟩
```

20. This is the loop that does the work. It should spend most of its time in *sleep_mode*, comming out at each interrupt event.

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

21. Now we wait in "idle" for any interrupt event.

```
sleep_mode( );
```

22. If execution arrives here, some interrupt has been detected!

The ISR would have left a flag set in f_state. Let's see which one by testing each possibility and acting on it.

```
if (f_state & (1 ≪ WAVES)) {
  waveless = (waveless) ? waveless − 1 : 0;      /* countdown to 0, but not lower */
  if (¬waveless)      /* ancillary electric service restored */
  {
    ledcntl(ON);
    if (f_state & (1 ≪ ARM)) chirp(ON);      /* annunciate */
    TCNT1 = TIMESTART;      /* reset the timer */
  }      /* end if waveless */
  f_state &= ∼(1 ≪ WAVES);      /*reset int flag since actions are complete */
}      /* end if WAVES */
else if (f_state & (1 ≪ NOWAVES)) {
  ledcntl(OFF);
  waveless = 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 */
}      /* end if NOWAVES */
⟨ Hold-off all interrupts 28 ⟩}      /* end for */
return 0;      /* it's the right thing to do! */
}      /* end main */
```

23. ⟨ Initialize pin outputs and inputs 23 ⟩ ≡

```
{      /* set the led port direction; This is pin #1 */
  DDRB |= (1 ≪ DDB1);      /* set the siren port direction */
  DDRB |= (1 ≪ DDB0);      /* 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 14.

24. 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 ≪ PORTB1) : PORTB & ∼(1 ≪ PORTB1);
}    /* simple siren control */
void sirencntl(char state)
{
  PORTB = state ? PORTB | (1 ≪ PORTB0) : PORTB & ∼(1 ≪ PORTB0);
}
```

25. 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 $\frac{1}{2}$ 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 25 ⟩ ≡
  {    /*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 cited in section 7.

This code is used in section 16.

26. 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 26 ⟩ ≡

```
{     /* 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 17.

27. 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 27 ⟩ ≡

```
{
  MCUCR &= ~(1 << SM1);
  MCUCR &= ~(1 << SM0);
}
```

This code is used in section 19.

28. ⟨ Hold-off all interrupts 28 ⟩ ≡

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

This code is cited in section 8.

This code is used in section 22.

29. 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);
}
```

30. 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);
}
```

31. This ISR responds to the Clear button at pin #3 or PB3.

```
    /* Clear Button ISR */
ISR(PCINT0_vect)
{
  if (PORTB & (1 ≪ PORTB3)) f_state &= ∼(1 ≪ ARM);
}
```

32. Done

⟨ Configure to wake upon interrupt 27 ⟩ Used in section 19.
⟨ Global variables 13 ⟩ Used in section 14.
⟨ Hold-off all interrupts 28 ⟩ Cited in section 8. Used in section 22.
⟨ Include 11 ⟩ Used in section 14.
⟨ Initialize pin outputs and inputs 23 ⟩ Used in section 14.
⟨ Initialize the no-wave timer 25 ⟩ Cited in section 7. Used in section 16.
⟨ Initialize the wave detection 26 ⟩ Used in section 17.
⟨ Prototypes 12 ⟩ Used in section 14.