ATmegaxx8PA-15 RC Oscillator Frequency Drift Compensation

Robin Walsh

The new Atmel® AVR® ATmegaxx8PA-15 family with UART and 32-pin package is perfectly suited for multiple automotive applications with local interconnect network (LIN) or pulse width modulation (PWM) communications. Design engineers can include a small software routine to compensate the RC oscillator temperature drift over temperature. This article describes a method to achieve this by using the on-chip temperature sensor.

The ATmegaxx8PA's internal RC oscillator (RCO) is useful in applications where an external quartz crystal or resonant element cannot be used for cost reasons. The RCO is capable of providing a reasonably accurate 8MHz clock source for the AVR microcontroller where the high precision of an external crystal resonator is not required for the application. The RCO frequency is, however, sensitive to temperature and voltage change as are many semiconductor elements. The degree of sensitivity to voltage and temperature change varies considerably from device to device. Therefore, no general compensation rule can be applied; instead, it must be determined for each part empirically.

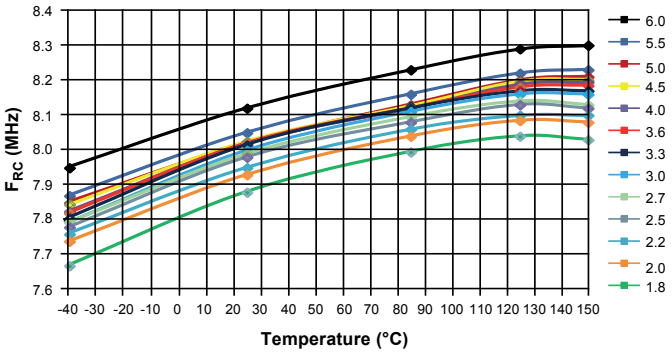

Figure 1. Typical RCO Frequency Drift Characteristic (Calibrated 8MHz RC Oscillator Frequency vs. Temperature)

Compensation Measures and Parameters

With the introduction of the new automotive ATmegaxx8PA-15 devices, it is now possible to determine with reasonable accuracy the current operating temperature of the microcontroller using the integrated temperature sensor peripheral. It is also possible during automotive part production test procedures to determine the tendency of the RCO on each individual device to change in frequency as the temperature and voltage is changed. This specific device-dependent characteristic is then stored in a read-only non-volatile ATmegaxx8PA memory space accessible to the user's application program, the signature row.

With this characteristic value and specified base reference conditions, it is now possible to compensate the RCO frequency drift to good effect in an application program with the addition of a small supplementary software routine which is called regularly from the main application. To function, this software routine needs, as parameters, to know the degree to which

Signature Byte	Z-Pointer Address
Device Signature Byte 1	0x0000
Device Signature Byte 2	0x0002
Device Signature Byte 3	0x0004
RC Oscillator Calibration Byte 3V	0x0001
TS_ADC_25_L-Temp Sensor Value at 25˚C - Low Byte	0x0005
TS_ADC_25_H-Temp Sensor Value at 25˚C - High Byte	0x0007
RC Oscillator Calibration Byte 5V	0x0009

Note: All other addresses are reserved for future use

Table 1. Signature Row Addressing

the RCO will drift with the change of temperature for the actual device, that is to say: Δ_RCO_Frequency per Δ_ Temperature_Sensor_Reading. This value can be determined during production testing and is stored as a signed byte value in the signature row at address 0x0003. It corresponds to the calculation:

$$S_{(sensitivity)} = \frac{(TS\_ADC\_Hot - TS\_ADC\_25C)}{(Osccal\_Hot - Osccal\_25C)}$$

Where:
TS_ADC_Hot is the result obtained from the temperature sensor ADC reading of the microcontroller when it is being subjected to high temperature operational testing.

TS_ADC_25C is the result obtained from the temperature sensor ADC reading of the microcontroller when it is being subjected to ambient 25°C temperature operational testing.

Osccal_Hot is the best case RCO frequency adjustment register value (OSCCAL) to obtain 8MHz when the device is being subjected to high temperature operational testing.

Osccal_25C is the best case RCO frequency adjustment register value (OSCCAL) to obtain 8MHz when the device is being subjected to ambient 25°C temperature operational testing.

Typical values obtained during this testing would be:

TS_ADC_Hot	=	437
TS_ADC_25C	=	297
Osccal_Hot	=	143
Osccal_25C	=	154

Giving an example result of: (437-297) / (143-154) = -13 (rounded).

What this -13 example "S" parameter effectively means to the microcontroller is that for every change in its temperature sensor ADC result of -13 (counts) from its base reference value TS_ADC_25C, we should adjust the RCO Osccal register upwards (increment) by one (count) from its base starting point Osccal_25C to compensate for the temperature-induced drift.

Naturally there are sanity checks on the values of these parameters generated during production testing to ensure that only reasonable values are accepted to be written to the signature row memory. For instance, the minimum S parameter values considered acceptable are S > 7 or S < -7, and the difference between the temperature sensor 25°C reading and the temperature sensor hot reading corresponds closely to the expected temperature excursion. Frequency adjustment guard bands are also verified to ensure that the device possesses sufficient oscillator adjustment range capability to compensate for worst case adjustment situations.

The ATmegaxx8PA microcontroller by default starts operation with the RCO adjusted to the RC Oscillator Calibration Byte 3V value, which is also stored in the AVR read-only signature row at address 0x0001. During production testing of the automotive ATmegaxx8PA family, Atmel also stores the TS_ADC_25C ADC reading as an unsigned 16-bit word value, as two individual byte values in the signature row (0x0007 and 0x0005) as well as the signed 8-bit S parameter (0x0003).

The Application Algorithm

To adjust the ATmegaxx8PA-15 RCO calibration register OSCCAL to a temperature-compensated value, the factory-supplied calibration parameters (OSCCAL_25C$_{Vcc*}$, TS_ADC_25C, S) as well as the on-chip actual temperature sensor reading (TS_Actual) are used as follows:

OSCCAL_Compensated =
OSCCAL_25C$_{Vcc*}$ + ((TS_Actual – TS_25C) / S)

V_{cc*} is respectively OSCCAL_25C 5V (0x0009) or OSCCAL_25C 3V (0x0001), whichever is closer to the application operational voltage.

Note if application space is constrained, the S division in the above equation can be usefully replaced by successive subtractions in a loop to avoid voluminous library division functions as is shown in the chapter "Sample Code Routines".

The algorithm is designed to make good use of the dynamic range of the signed byte S parameter (very little of the numerical range is unused) while being simple and code-efficient to implement on even resource-limited devices. The slowest part of the algorithm is the reading of the current temperature sensor value get_temperature() due to the time it takes to configure the ADC and possibly allow the internal references to stabilize. This operation could be usefully removed from the recalibrate() function and integrated into an application ADC task such that the current temperature sensor value is passed to the recalibrate() function only when a significant temperature change (±10°C) has been recognized, for optimal performance.

Hardware Constraints

The only specific hardware constraints for correct operation of the procedure is that the ADC-internal reference supply 1.1V be available for selection, as this is essential for the calibrated reading of the on-chip temperature sensor. Therefore, the ATmegaxx8PA-15 ARef pin should not be tied to an external reference supply but connected to a 10nF smoothing capacitor to ground as recommended in the datasheet. This still allows selection of the AVCC supply as an alternate reference to the on-chip 1.1V reference via the ADMUX register but precludes the use of any other external ADC reference.

Performance

Analysis of a large number of ATmega48PA-15 parts has shown that the use of software temperature compensation is capable of holding the RC oscillator frequency stability to better than ±2% over the temperature range +125°C to -20°C and better than ±3%, normal performance over the full operational temperature range of +125°C to -40°C. This is in comparison to the ±10% performance for the unadjusted device. This stability is normally sufficient to allow usage of PWM signaling as well as UART communication.

Sample Code Routines: ATmegaxx8PA-15 Compact Using Loop Subtract

IAR C

recalibrate.c

```
#include <iom48pa.h>
#include <intrinsics.h>

#define OSCCAL_3V_ROOM 0x01 // Location of Factory 25C 3V osccal value
#define OSCCAL_5V_ROOM 0x09 // Location of Factory 25C 5V osccal value
#define SENSITIVITY 0x03    // Location of S parameter
#define TS_ADC_ROOM_HI 0x07 // Temp Sensor Factory 25C ADC value high byte
#define TS_ADC_ROOM_LOW 0x05 // Temp Sensor Factory 25C ADC value low byte
#define SIGRD 5             // Signature row read activation bit in MCUCR
#define FCPU 8000000         // 8MHz CPU
#define TEMP_SENSOR ((3<<REFS0)|(8<<MUX0))
#define millisecond_delay FCPU/1000

unsigned int get_temperature(void);
unsigned char read_sig_mem(unsigned int addr)
{
  return (__AddrToZByteToSPMCR_LPM((void __flash*)(addr),((1<<SIGRD)|(1<<SELFPRGEN))));
}

void recalibrate(void)
{
  unsigned int temperature_factory_25C,current_temperature; // TS worker variables
  unsigned char new_osccal; // Temporary holding variable for result OSCCAL
  signed char step_size; // Temporary holding for osccal temperature sensitivity
  signed char pos_neg_sensitivity; // Increment or decrement variable
  signed int temp_diff; // Temp sensor

  current_temperature=get_temperature(); // First we get the actual device temperature

  /* new_osccal is the (unsigned char) best-case value for OSCCAL to give nearest 8MHz RC oscillator reading at the 25˚C test point
     at 3V and 5V. These values are determined during production test and stored directly in the signature row at addresses 0x0001
     3V or 0x0009 for the 5V setting.
  */

  new_osccal=read_sig_mem( OSCCAL_5V_ROOM ); // Get ambient 5V osccal reading from sigrow

  /* step_size is the (signed char) value which gives the change required in the temp sensor ADC value to warrant an incremental
     change in the OSCCAL register to compensate. This value can be derived by the formula :

     ((ADC_Temp_Hot - ADC_Temp_Ambient)/(Best_OSCCAL_Hot - Best_OSCCAL_Ambient)) = step_size

     This value can be either positive or negative depending on the device.
  */

  step_size=read_sig_mem(SENSITIVITY);   // Get osccal sensitivity reading from signature ram
```

```
  /* Optional safety check, valid sensitivity values are (abs)S > 7 */

  if((step_size >= -7) && (step_size <= 7)) return;   // If invalid S exit without recalibration

  if(step_size<0)
  {
    pos_neg_sensitivity=-1;
    // Negative sensitivity means reduce OSCCAL on increasing temperature
    step_size=((~step_size)+1);
    // Now that sensitivity has been determined make step_size absolute
  }
  else pos_neg_sensitivity=1;
  // Positive sensitivity means increase OSCCAL on increasing temperature

  /* room_temp is the (unsigned integer) raw value which the temp sensor returned via the ADC when the device was calibrated
    at room temperature.
  */

  temperature_factory_25C=((unsigned int)((read_sig_mem(TS_ADC_ROOM_HI))<<8)+read_sig_mem(TS_ADC_ROOM_LOW));

  /* For optimal code size here it needs to be ensured that all calculations are adjusted to positive operations
  */
  if((temp_diff=(current_temperature-temperature_factory_25C))<0)
  {
    pos_neg_sensitivity=(~pos_neg_sensitivity)+1;
    // Invert the sensitivity if we are dealing with below reference temperatures
    temp_diff=(~temp_diff)+1; // And make temperature difference absolute
  }


  /* Now the parameters for the temp sensor value at room, the sensitivity Osccal/Temp and the OSCCAL at room are available. With
    these parameters and the current temperature sensor reading, the OSCCAL register can be adjusted.
  */

  // Here we perform the calibration operation itself based on the given parameters
  // We do a repetitive subtraction instead of a division for code size efficiency

  while(temp_diff > step_size) // While an adjustment to OSCCAL is necessary
  {
    new_osccal+=pos_neg_sensitivity; // Adjust OSCCAL in appropriate direction
    temp_diff-=step_size; // Reduce temperature difference by step amount
  }

  OSCCAL=new_osccal; // Calibrate Osccal

}


unsigned int get_temperature(void)
{
  // Wait for conversion to complete just in case ADC is already in use
```

```
  while(ADCSRA&(1<<ADSC));
  ADCSRB=0;

  // Be careful of the order here, if ADC is not enabled ADMUX doesn't update
  ADCSRA = ((1<<ADEN)|(1<<ADIF)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0));
  // Enable ADC,  clear ADC flag, 8MHz / 64 = 125kHz

  /* If the internal reference is not already on, turn it on and wait about 1ms for it to settle */

  if((ADMUX&(3<<REFS0))!=(3<<REFS0)) // Is 1.1V reference already on ?
  {
    ADMUX|=(3<<REFS0); // Activate internal 1.1V reference
    __delay_cycles(millisecond_delay);
    // Wait 1ms for reference to stabilize on AREF capacitor
  }

  ADMUX=TEMP_SENSOR; // Configure for temperature measurement

  ADCSRA|=(1<<ADSC); // Start dummy conversion
  while(ADCSRA&(1<<ADSC)); // Wait for conversion to complete
  ADCSRA|=(1<<ADSC); // Start proper conversion
  while(ADCSRA&(1<<ADSC)); // Wait for conversion to complete

  return (ADC);
}
```

Atmel Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441-0311 **F:** (+1)(408) 487-2600 | **www.atmel.com**