AN_653X_003 **APPLICATION NOTE**

AUGUST 2008

RTC Compensation

Introduction

This document describes methods to create an accurate real time clock (RTC) using the 653X-family Teridian Energy Meter ICs. The sample code discussed is from the released Demo Code for the 71M6533, but similar techniques may be used with the 71M6531 and 71M6534 or other Teridian Meter products from the 653X family.

Theory of Operation

There are two basic ways to make the RTC accurate:

- Correct the RTC for all expected errors, using hardware and/or software techniques
- Slave the RTC to the line frequency.

The correction method uses measurements taken at calibration time as well as crystal temperature coefficients to compensate for expected RTC errors. It can be used when the line frequency varies too much to be used as a standard. This method is unavoidably complex, because there are several sources of error in a crystal clock.

The line frequency method adjusts the RTC to the line frequency, and automatically absorbs absolute errors when they exceed ½ second.

With both methods, the RTC of the 653X runs at high adjusted accuracy during power failures. The 653X cannot measure and adjust for temperature during power failures because its ADC is off-line in battery modes.

Achieving High RTC Accuracy with Hardware and Software Error Correction

This solution uses an instantaneous factory adjustment of crystal drift via capacitance and a digital drift adjustment. When mains power is available (mission mode), the digital adjustment is also adjusted for temperature changes. Since the digital and analog adjustments are maintained by the RTC backup battery, constant drift corrections continue when AC mains are off.

Crystal frequencies have a characteristic negative parabolic drift with temperature, and software corrects this with an opposite positive parabola.

The temperature response of a typical crystal may be good enough for some applications. However, for highest accuracies, the temperature curves of individual crystal oscillators need to be measured and corrected. The frequency deviation from nominal is a function of the thermal accuracy of the crystal and, to a minor degree, of the external capacitors (COG/NPO ceramic capacitors typically used at the XIN and XOUT pins of the IC have very good thermal stability). The current source and capacitance of the internal oscillator circuit is carefully designed for a consistent thermal response.

Example: Let us assume a crystal that is characterized by the measurements shown in Table 1.

Table 1: Frequency over Temperature

Deviation from Nominal Temperature [°C]	Measured Frequency [Hz]	Deviation from Nominal Frequency [PPM]
-35	32766.87	-34.55
-30	32767.27	-22.20
-25	32767.61	-11.75
-20	32767.90	-3.20
-15	32768.11	3.45
-10	32768.27	8.20
-5	32768.36	11.05
0	32768.39	12.00
5	32768.36	11.05
10	32768.27	8.20
15	32768.11	3.45
20	32767.90	-3.20
25	32767.61	-11.75
30	32767.27	-22.20
35	32766.87	-34.55

The values show that even at nominal temperature (the temperature at which the chip was calibrated for energy), the deviation from the ideal crystal frequency is 12 PPM, resulting in about one second inaccuracy per day, i.e. more than some standards for electricity meters allow. As Figure 1 shows, even a constant compensation would not bring much improvement, since the crystal will introduce more errors at low and high temperatures.

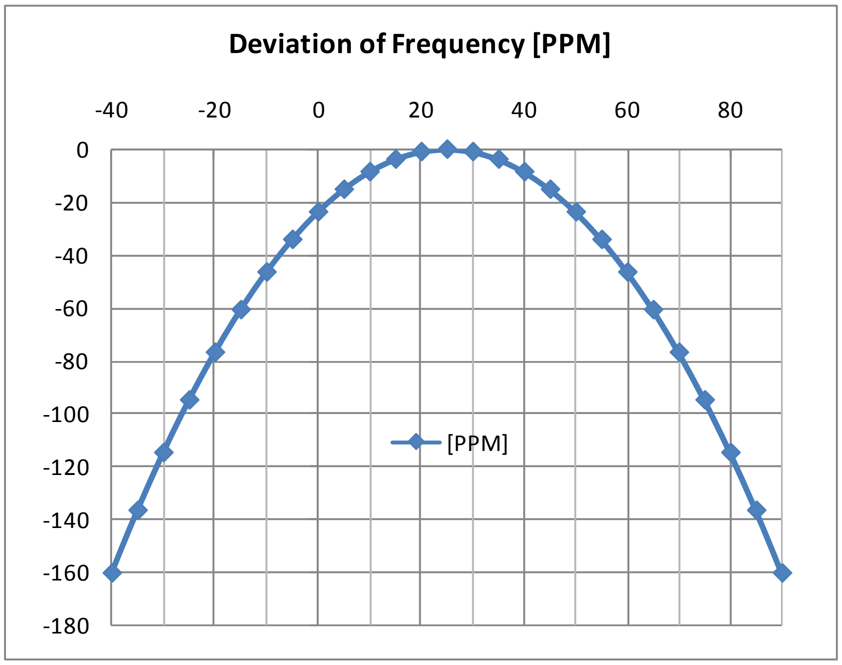

Figure 1: Crystal Frequency over Temperature (PPM Deviation from Nominal)

One method to correct the temperature characteristics of the crystal is to obtain coefficients from the curve in Figure 1 by curve-fitting the PPM deviations. A fairly close curve fit is achieved with the coefficients a = 12, b = 0, and c = –0.0038.

$$f = f_{nom} * (1 + a/10^6 + T * b/10^6 + T^{2} * c/10^6)$$

When applying the inverted coefficients, a curve will result that effectively neutralizes the original crystal characteristics. The coefficient c is available in many cases from the data sheet of the crystal (sometimes referred to as "K").

Advantages and Disadvantages of the Compensation Methods

The instantaneous factory adjustment permits fast clock calibration of the basic crystal drift to <5 PPM. Hardware performs this drift adjustment and maintains clock accuracy when mains power is off.

A valid clock calibration adds to the complexity of a meter's calibration.

Since temperature is sensed with the ADCs, temperature adjustments of the digital drift occur only when the meter is measuring power.

Using the Analog Adjustment (*RTCA_ADJ*[6:0])

The 71M653X ICs have mechanisms to absorb component tolerances in the oscillator clock circuit by adding a controllable capacitance value to the external capacitor at the XIN pin. The I/O RAM register *RTCA_ADJ[6:0]* controls the internal capacitive loading of the crystal. Setting this register to 0 minimizes the capacitive load, maximizing the frequency so that the clock runs fast. Setting *RTCA_ADJ[6:0]* to 0x7F maximizes the capacitive load, minimizing the frequency so that the clock runs slow. The combined capacitance of the oscillator is shown in Figure 2.

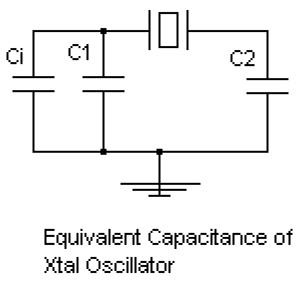

Figure 2: Crystal Oscillator

The combined serial capacitance C is:

$$C = \frac{(C_i + C_1) \cdot C_2}{C_i + C_1 + C_2}$$

The overall capacitance for the combined capacitors C_i and C_1 equals $C_i + C_1$. C_i can be controlled with the *RTCA_ADJ[6:0]* register in I/O RAM; and the formula for capacitance control of C_i is:

$$Ci = 16.5 pF \cdot \frac{RTCA\_ADJ[6:0]}{128}$$

The nominal value of Ci = 8.25pf when *RTCA_ADJ* is hex 0x40. With a standard value 15 pF capacitor as C1, which is easily available as an SMT part, the typical adjustment range is ± 30 PPM, as shown in Figure 3 (the frequency shown is the MPU clock, which is 150 * 32.768 Hz).

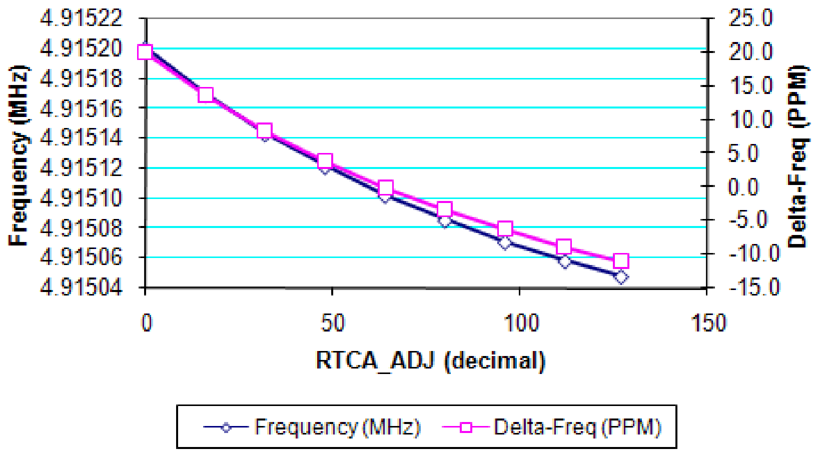

Figure 3: Crystal Oscillator Adjustment with *RTCA_ADJ*[6:0]

C_i has less than 5% of variation due to temperature, which amounts to less than 2 PPM over the designed temperature range. C1, C2 and the crystal are all more significant sources of temperature-based error. The most significant source is the crystal, which has a quadratic temperature variation.

The Digital Adjustment Register

The purpose of the digital adjustment is to absorb large variations in oscillator behavior. This adjustment is done automatically by the hardware and is based on the values in the *PREG* & *QREG* registers. Involvement of the MPU other than setting the *PREG* & *QREG* registers is not required. It is important to note that the *PREG* & *QREG* registers do not change the oscillator frequency but rather control the digital correction of the RTC register contents.

The register is a single 19-bit register documented as two parts, *PREG*[16:0] * 4 + *QREG*[1:0].

If Δ is the required change in RTC rate measured in PPM, the register values can be calculated as:

$$4 \cdot PREG + QREG = floor\left\{\frac{32768 \cdot 8}{1 + \Delta \cdot 10^{-6}} + 0.5\right\}$$

The digital logic does this adjustment every four seconds, and therefore requires a four-second test interval. For zero adjustment, the register values are *PREG* = 0x10000 and *QREG* = 0; Combined, the registers for a zero adjustment are set to 262144 (decimal).

For an adjustment of –988 PPM, the value is 262403, and for +988 PPM the value is 261885. These values can be used to test calculations.

Adjustments with *PREG* & *QREG* outside the range of ±988 PPM should be avoided. For example, if *PREG* & *QREG* are initialized to zero, the RTC counts two real seconds as one, apparently running at half speed. This is not a defect, because in this case, *PREG* & *QREG* are outside the documented adjustment range.

Adjusting and Measuring the Performance of the RTC

To adjust *RTCA_ADJ[6:0]*, it is recommended to measure the crystal frequency using the test multiplexer output pin (TMUXOUT) while the output of the 32-kHz main oscillator is selected with a value of 0x1D in the I/O RAM register *TMUX*[4:0]. Measuring frequency directly at the crystal pins is impractical because it would load the crystal and unpredictably change the frequency.

Measurements should occur with the meter soaked to the crystal's documented center frequency temperature (usually 25C; check the crystal documentation). After that, *RTCA_ADJ[6:0]* is calculated, set, and verified. Firmware must set and record the value for *RTCA_ADJ[6:0]*. Since components and PCB layouts differ, the calculations should be verified and adjusted for each particular model of meter.

After the capacitance is adjusted, *PREG & QREG* are set to unity (0 PPM, or 262144 decimal). Next, a four-second interval is measured on TMUXOUT pin (selection 0x11 for *TMUX*[4:0]). This provides the initial RTC accuracy that has not been corrected by the capacitance controlled by *RTCA_ADJ*.

Throughout the calibration, all measurements should use a frequency counter with a highly accurate reference oscillator. The frequency counter should be warmed-up according to its instructions. The frequency counter should be periodically calibrated, and the calibration should be traceable to a national standard.

Correcting Temperature Drift

The MPU Demo Code supplied with the TERIDIAN Demo Kits has constant, linear, and quadratic correction coefficients and it directly controls the *QREG* and *PREG* registers. These coefficients are the MPU variables *Y_CAL*, *Y_CALC*, and *Y_CALC2* (MPU addresses 0x04, 0x05, 0x06). For the Demo Code, the coefficients have to be entered in the form:

$$CORRECTION(ppm) = \frac{Y\_CAL}{10} + T \cdot \frac{Y\_CALC}{100} + T^2 \cdot \frac{Y\_CALC2}{1000}$$

Note that the coefficients are scaled by 10, 100, and 1000 to provide more resolution. For our example case discussed above (a = 12, c = -0.038), the coefficients would then become (after rounding):

Y_CAL = -120, *Y_CALC* = 0, *Y_CALC2* = 38

There are several choices for RTC compensation:

a) One can use *RTCA_ADJ*[6:0] to defeat the room temperature frequency deviation from nominal and then use *Y_CAL* and *Y_CAL2* only to compensate for the linear and quadratic temperature effect of the crystal. This method can also be used if some external function on the meter PCB depends on the room-temperature accuracy of the 32-kHz oscillator frequency, supplied by the TMUXOUT pin.

b) One can leave *RTCA_ADJ*[6:0] at its default value (for no compensation) and use only the digital function controlled by *Y_CAL* , *Y_CALC*, and *Y_CALC2* for compensation.

c) One can compensate a part of the oscillator frequency, using the *RTCA_ADJ*[6:0] register, then use *Y_CAL* to compensate the remaining constant part, and use *Y_CALC* and *Y_CALC2* to compensate the linear and quadratic temperature effects of the crystal. For example, if there is a large deviation at room temperature (100 PPM), one can use *RTCA_ADJ*[6:0] to compensate 30 PPM but one will then have to use *Y_CAL* to compensate the remaining 70 PPM.

Code for Temperature Compensation

Code samples can be found in the source file `rtc_30.c`.
The variable `value` calculated in the `RTC_Compensation()` routine listed below is in PPB (parts per billion).

$$value = \frac{Y\_CAL}{10} + T \cdot \frac{Y\_CALC}{100} + T^2 \cdot \frac{Y\_CALC2}{1000}$$

It represents the expected deviation of RTC time in PPB. It is scaled in the `RTC_Adjust_Trim()` routine and then used to control the RTC via the *QREG*/*PREG* registers.

The actual correction code for the 653X Demo Code (from file rtc_30.c) is:

```
// This is the RTC's temperature compensation.  It's updated in run_meter()
// in meter.c, running each time the meter measures the temperature.
// It figures the clock's compensation in parts per billion.
// Y_FREQ = XTAL_FREQ * {[Y_DEG2 * deltaT + Y_DEG1] * deltaT + Y_CAL}.
//
// Y_Cal_Deg2 is   1 ppb =>   1 Y_Cal_Deg2 is 1 ppb.
// Y_Cal_Deg1 is  10 ppb =>  10 Y_Cal_Deg1 is 1 ppb.
// Y_Cal      is 100 ppb => 100 Y_Cal      is 1 ppb.
//
// deltaT     is 0.1 degrees => deltaT / 10 is degrees C.
//
void RTC_Compensation (void)            // Do temperature compensation of RTC.
{
    float dt;
    long value;

    dt = ((float)deltaT);

    value = lroundf(((( ((float) Y_Cal_Deg2) * dt) / 100.0
        + ((float) Y_Cal_Deg1)) * dt)
        + ((float) Y_Cal_Deg0) * 100.0));

    // defer the set-up to the next adjustment interval
    RTC_Adjust_Trim (FALSE, value);
}
```

```
   #define RTC_PNQ_NOM 262144L // nominal, center value
void RTC_Adjust_Trim (bool clr_cnt, int32_t value)
{
    // RTC_PNQ_NOM = 1.000000 for clock hardware adjustment
    value = lroundf(((float)RTC_PNQ_NOM) / (1.0 + (1.0e-9 * (float)value)));
    if (value > RTC_PNQ_MAX)
        value = RTC_PNQ_MAX;
    if (value < RTC_PNQ_MIN)
        value = RTC_PNQ_MIN;

    if (clr_cnt)
    {
        // if the flag is set, the interrupt writes the PnQ register
        rtc_adj_flag = FALSE;  // begin critical section (disable writes)

        bgnd_trim_value = value;

        WE = 0;        // enable writes
        RTC_CTRL = 1;  // clear subsecond register
        RTC_PNQREG[0] = (bgnd_trim_value >> 16) & 0x07;
        RTC_PNQREG[1] = (bgnd_trim_value >> 8) & 0xFF;
        RTC_PNQREG[2] = bgnd_trim_value & 0xFF;

        #if RTC_LINE_LOCKED
        trim_count = 1; // reset the count of ok cycles
        #endif

        // critical section ends;
        // The writes are done, so no need to request them
    }
else
    {
        // have the RTC's 1-second interrupt update the register
        if (value != bgnd_trim_value)
        {
            // if the flag is set, the interrupt writes the PnQ register
            rtc_adj_flag = FALSE;  // begin critical section (disable writes)

            bgnd_trim_value = value;

            rtc_adj_flag = TRUE;  // end critical section (request writes)
        }
    }
}
```

The function of the temperature compensation is illustrated in Figure 4: Column 1 and 2 show the temperature and temperature deviation from nominal (dT). Columns 3, 4, and 5 contain the expected frequency, the deviation from nominal in PPM, and the resulting uncompensated deviation in seconds per day. Column 6 contains the expected frequency deviation in PPB, and column 7 contains the calculated contents for the correction register (4**PREG* + *QREG*). Columns 8, 9, and 10 contain the bytes that are to be written to the array PNQREG[ ], resulting in the value for 4**PREG* + *QREG* shown in column 11 that implement the compensation shown in column 12.

Note that due to rounding effects, the effective compensation in column P is not exactly the same as the calculated frequency deviation in column E.

TERIDIAN Semiconductor provides a spreadsheet for RTC-related calculations with every 653X Demo Kit that is shipped to customers. The file name is “65XX Calibration Spreadsheets REV 5-1.XLS”.

Temp. [°C]	dT [0.1 °C]	Frequency [Hz]	Frequency [PPM]	Deviation [s/day]	"value" in rtc_30.c [PPB]	"value" re-calculated in rtc_30.c	Resulting RTC_ PNQREG[0]	Resulting RTC_ PNQREG[1]	Resulting RTC_ PNQREG[2]	Calculated 4*PREG =+QREG	Effective correction [PPM]
-40	-650	32763.39	-140.553	-12.1438	136325	262108	3	255	220	262108	137
-35	-600	32764.17	-116.803	-10.0918	113200	262114	3	255	226	262114	114
-30	-550	32764.89	-94.9523	-8.20388	91925	262119	3	255	231	262119	95
-25	-500	32765.54	-75.0019	-6.48016	72500	262124	3	255	236	262124	76
-20	-450	32766.13	-56.9515	-4.92061	54925	262129	3	255	241	262129	57
-15	-400	32766.66	-40.8012	-3.52523	39200	262133	3	255	245	262133	42
-10	-350	32767.13	-26.5509	-2.294	25325	262137	3	255	249	262137	27
-5	-300	32767.53	-14.2007	-1.22694	13300	262140	3	255	252	262140	15
0	-250	32767.88	-3.75048	-0.32404	3125	262143	3	255	255	262143	4
5	-200	32768.16	4.799696	0.414694	-5200	262145	4	0	1	262145	-4
10	-150	32768.38	11.44983	0.989265	-11675	262147	4	0	3	262147	-11
15	-100	32768.53	16.19992	1.399673	-16300	262148	4	0	4	262148	-15
20	-50	32768.62	19.04998	1.645918	-19075	262149	4	0	5	262149	-19

Figure 4: RTC Adjustment over Temperature

Correcting Long-Term Drift in the Field

Typical crystals change by up to 3 PPM/year for the first year. Capacitors also drift.

These drifts can be corrected in the meter firmware by measuring and correcting the long-term average drift each time the clock is set.

A common method is to recall the last time and date that the clock was set. When the clock is set again, the long-term drift correction is calculated and added to the previous digital drift correction.

The major practical problem in applying this logic in meter firmware is finding the number of seconds between the two times. TSC's Demo Code contains validated routines that convert times and dates to and from Julian seconds. The Julian second describes a time and date as a 32-bit count of seconds since January 1, 2000.

The drift can be easily calculated in Julian seconds, using floating point calculations:

Clt = (Tc – Ts2)*1000000/(Ts2 – Ts1)

Clt = Correction, long term, in PPM

Tc = time and date of the clock as read when the time is to be set to Ts2

Ts2 = Now, the time to which the clock is going to be set.

Ts1 = Then, the time of the previous set of the clock.

Clt*10 can be added to the *Y_CAL* value of the previous section. The changed *Y_CAL* value should be stored to the meter's permanent calibration data, along with the date and time that the meter's clock was set.

Example: To calculate an example, let us assume that the meter's RTC was last set at 15:38:23 on July 1, 2008. The Julian seconds, (the time since 00:00:00 on January 1, 2000) for this time are decimal 268241903. Now, an automatic meter reading unit with an on-board clock, traceable to national standards, says that the current time is 11:22:13 on October 3, 2008. This is equivalent to a Julian time of 276348133. The meter's internal clock reads 11:22:29 (Julian 276348149), which means that the clock has drifted by +16 seconds from standard. The time between settings is 276348133 – 268241903, or 8106230 seconds, which makes the drift rate in PPM 10^6*(16/8106230) or 1.973 PPM. Rounding to the nearest 0.1 PPM, we obtain 2.0 PPM. *Y_CAL* units are 0.1 PPM. Since the clock ran slightly too fast, the value 20 has to be subtracted from *Y_CAL*.

The 653X demo code does not correct long-term drift, but it does have validated calculations for Julian time.

```
// seconds since midnight, january 1, 2000
int32_t Julian (struct RTC_t xdata *prtc)
{
    int32_t a, y, m, j;
    RESET_WD();  // in brownout mode
    a = (14 - prtc->Month) / 12L;
    y = (uint32_t)prtc->Year + 6800L - a;
    m = (uint32_t)prtc->Month + (12 * a) - 3;
    RESET_WD();  // in brownout mode
    // julian days since Jan 1, 2000; 5.8e6 years range in 32-bit signed int
    j = (uint32_t)prtc->Date
        + (((153 * m) + 2)/5)
        + (365 * y)
        + (y / 4)
        - (y / 100)
        + (y / 400)
        - 2483590;
    RESET_WD();  // in brownout mode
    j = (j * 86400)
         + ((((uint32_t)prtc->Hour) - 12L) * 3600)
         + ((uint32_t) prtc->Min * 60L)
         + (uint32_t) prtc->Sec
         + 43200L;
    RESET_WD();  // in brownout mode
    return j;
}

// Get time and date from a julian count of seconds
void Unjulian (struct RTC_t xdata *prtc, int32_t j)
{
    uint32_t jd, w, x, a, b, c, d, e, f, Month;
    RESET_WD();  // in brownout mode
    prtc->Sec = j % 60;
    prtc->Min = (j / 60) % 60;
    prtc->Hour = (j / 3600) % 24;
    jd = (j / 86400L) + 2451545L; // jd = julian days
    RESET_WD();  // in brownout mode
    // Calculate day of week: 1 = Sunday
    prtc->Day = ((jd + 1) % 7) + 1; // January 1, 2000 was saturday
    // In standard calculation w = int((jd - 1867216.25)/36524.25);
    w = ((4L * jd) - 7468865L) / 146097L;
    x = w / 4L;
    a = (jd + w + 1L) - x;
    b = a + 1524L;
    RESET_WD();  // in brownout mode
    // In standard calculation c = int((b - 122.1) / 365.25);
    c = ((20L * b) - 2442L) / 7305L;
    // In standard calculation d = int(c * 365.25);
    d = (c * 1461L) / 4;
    // In standard calculation e = int((b - d)/30.6001);
    e = ((b - d) * 10000L)/306001L; // b - d is a few hundred, so no overflow
    f = (306001L * e)/10000L; // e is less than 25, so no overflow
    RESET_WD();  // in brownout mode
    prtc->Date = (b - d) - f;
    prtc->Month = Month = (e < 14) ? (e - 1) : (e - 13);
    prtc->Year = ((Month < 3) ? (c - 4715) : (c - 4716)) - 2000L;
    RESET_WD();  // in brownout mode
}
```

Using the Mains Frequency for RTC Correction

Alternatively, the mains frequency may be used to stabilize or check the function of the RTC. For this purpose, the CE provides a count of the zero crossings detected for the selected line voltage in the *MAIN_EDGE_X* address. This count is equivalent to twice the line frequency, and can be used to synchronize and/or correct the RTC.

Advantages and Disadvantages of the Mains Frequency Method

With good component selection, no calibration step is required during manufacturing.

Long term accuracy is comparable to the clock controlling the grid. In some countries, the grid is controlled with an atomic clock.

The fall-back nonvolatile rate control is automatically adjusted for use in power failures.

This method only works where grid frequencies are reliable.

Analog Adjustment

With good component selection, *RTCA_ADJ[6:0]* can often be set to a nominal constant value. The nominal value is not always the center value and it will depend on component selection. See the section on Capacitive Adjustment.

If tolerances exceed +/- 988 PPM over all temperatures and components, TSC recommends adjusting *RTCA_ADJ*[6:0] in order to absorb component variations and center the oscillator's nominal frequency. This maximizes the available digital adjustment range.

Rate Adjustment

The most reliable way is to use the mains to adjust the digital drift. The demo code has optional code to adjust the drift to 1.9 PPM, the resolution of the PREG & QREG digital adjustment.

The theory is to count zero crossings, and then, each second, subtract a second's worth of zero crossings. When there is an extra or missing zero crossing, this indicates that the real-time clock is slow, or fast, respectively. The ratio correction is thus:

Cr = (Ezc * 262144 + (z/2))/Z

where

Cr = ratio correction to be added to current *PREG* & *QREG*

262144 = a *PREG* & *QREG* value of unity = no correction

Z = number of zero crossings since last correction

Ezc = number of extra or skipped zero crossings (skipped is a negative number)

The equation only runs when there is an error of +/- 1 zero crossing, so Ezc is always either +/- 1. So, this becomes two correction equations, that happen depending on whether there is an extra or missing zero crossing.

When there is an extra zero crossing, then the RTC is slow. So, the correction equation in this case is:

Cr = (262144 + (z/2))/Z

When there is a missing zero crossing, then the RTC is fast. So the correction equation in this case is:

Cr = ((z/2) – 262144)/Z

In this equation, given a reliable count of zero crossings, at the first extra or missing zero crossing, *PREG* & *QREG* are adjusted. Further adjustments occur until Z > 262144. At that point, the further error is less than the rate register can adjust.

Absolute Time Correction

In many countries, the mains frequency has a long-term stability equal to an atomic clock, and is further adjusted to astronomical calculations. This means that no practical rate adjustment can have enough resolution to accurately adjust the time.

In this case, the solution is to sum up uncorrected adjustments, and then update the seconds count of the clock. So, when Cr (above) is calculated, an error of +/- 1/Hz error is also added to an absolute error variable that is stored in nonvolatile memory. When this error count reaches +/- 1 second, the seconds register of the RTC is adjusted at a convenient second within a minute.

Behavior with Electricity Grid Frequency Variation

In real electricity grids, long term frequency accuracy is good, but over the course of a day, the frequency can change by hundreds of PPM. This means that the rate correction of the code presented in this Application Note may oscillate on a daily cycle. However, the absolute correction assures that the clock accuracy remains as good as the mains frequency control.

Behavior with Power Failure

The battery-powered RTC is accurate to the last measured frequency. However, error accumulates while the meter's power is off. This is common to all frequency-counting clocks.

Source Code for rtc_isr.c

```
void rtc_isr (void) small reentrant
    // This locks the RTC to the line frequency
    // This requires no calibration.
    {
        // get the nominal crossing counts per second, calculated in meter\freq.c
        int8_t cnt = (int8_t)main_edge_cnt_nom;

        // Are mains data available? (demo code is often run without mains)
        if (cnt != 0) // from meter\freq.c
        {
            // how much error has accumulated?
            trim_count += cnt; // count good cycles since last adjustment

            // Adjust the rate as needed
            MainEdgeCount -= (long) cnt;       // remove this second's counts
            if (MainEdgeCount < 0) // clock is too fast
            {
                // This calculation minimizes the clock's rate error.
                // It adjusts by the ratio of mains cycles missed (1) to ok.
                // trim count is the number of OK cycles, and the number
                // missed is assumed to be 1.
                // This gets the delta-error to less than 5 seconds/month.
                // (minimum adjustment is 3.817e-6/count, average
                // error is 1/2 of this, 1.9e-6/count)
                // If the trim count > 262144, then the adjustment
                // is smaller than the adjustment resolution.
                // trim count is nonvolatile, stored in
                // the eeprom on power failure. see Meter\meter.h
                bgnd_trim_value += (RTC_PNQ_NOM + (trim_count/2))/trim_count;
                trim_count = 1;

                if (bgnd_trim_value > RTC_PNQ_MAX) // maximum?
                    bgnd_trim_value = RTC_PNQ_MAX; // maximum.

                RTC_set_PnQ_reg = TRUE;

                // This measures the absolute cycles of error
                // trim value is nonvolatile, stored in
                // the eeprom on power failure. see Meter\meter.h
                trim_value += MainEdgeCount; // count missed cycles of mains
                MainEdgeCount = 0;
            }
            else if (MainEdgeCount > 0) // clock is too slow
            {
                // This calculation minimizes the clock's rate error.
                // It adjusts by the ratio of mains cycles missed (1) to ok.
                // trim_count is the number of OK cycles, and the number
                // missed is assumed to be 1.
                // This gets the delta-error to less than 5 seconds/month.
                // (minimum adjustment is 3.817e-6/count, average
                // error is 1/2 of this, 1.9e-6/count)
                // If the trim count > 262144, then the adjustment
                // is smaller than the adjustment resolution.
                // trim count is nonvolatile, stored in
                // the eeprom on power failure. see Meter\meter.h
                bgnd_trim_value -= (RTC_PNQ_NOM + (trim_count/2))/trim_count;
                trim_count = 1;

                if (bgnd_trim_value < RTC_PNQ_MIN) // minimum?
                    bgnd_trim_value = RTC_PNQ_MIN; // minimum.

                RTC_set_PnQ_reg = TRUE;
                // This measures the absolute cycles of error
                // trim value is nonvolatile, stored in
                // the eeprom on power failure. see Meter\meter.h
                trim_value += MainEdgeCount; // count missed cycles of mains
                MainEdgeCount = 0;
            }
            else if (trim_count > RTC_PNQ_NOM)
            {
                // The error is less than the adjustment resolution
                trim_count = (RTC_PNQ_NOM + 1); // avoid overflow
            }
            // Correct the absolute error. Needed because a perfectly rated
            // clock can still drift up to the adjustment quantization,
            // +/- 1.9ppm = +/- 5 seconds per month.
            if (trim_value > (cnt/2)) // clock is slow by 1/2 second...
            {
                RTClk_Stabilize();
                if (RTC_SEC == 30)
                {
                    // This should execute very infrequently
                    WE = 0;
                    RTC_SEC = 31;       // advance clock by 1 sec
                    trim_value -= cnt;  // average error is zero...
                }
            }
            else if (trim_value < (-cnt/2)) // clock is fast 1/2 second...
            {
                RTClk_Stabilize();
                if (RTC_SEC == 30)
                {
                    // This should execute very infrequently
                    WE = 0;
                    RTC_SEC = 29;       // advance clock by 1 sec
                    trim_value += cnt;  // average error is zero...
                }
            }
        }
    }
```

Revision History

Revision	Date	Description
Rev. 1.0	06/25/2008	First publication.
Rev. 1.1	06/26/2008	Added description of mixed adjustment modes (*RTCA_ADJ*[6:0] and *Y_CAL*) for initial frequency tolerance. Added description of location of related software.
Rev. 1.2	07/28/2008	Updated description and source code for rtc_30.c. Deleted statement that rtc_30.c has not been completely tested. Added graph showing frequency change as a function of *RTCA_ADJ*[6:0].
Rev. 1.3	08/25/2008	Minor corrections, additions, and clarifications, such as graph in Figure 1. Included reference to 65XX Calibration Spreadsheets.XLS file.