Designing and implementing a high precision clock for real time networks

Your application needs a synchronised real-time clock. There surely must be a correct, or even preferred way of doing so? Wrong! No such implementation presently exists off the shelf. Creating a real-time function has to be done from first principles. Having decided on the precision required and the resources available to implement the design, only then is it possible to figure out the right sort of solution. **Svein Johannessen**

What kind of accuracy does your application? This is the first decision to be made. The decision then comes on allocating resources to the real time clock: such resources relate to CPU, network and hardware. Remember that a small piece of hardware, even if it represents an added cost, will usually save a large amount of CPU resource. Examples of requirements are:

- Time stamping a sparse set of events
- Time stamping a dense set of events
- Controlling a set of events

Examples of accuracy classes are the T1-T5 classes in IEC 61850[1]:

- Class T1: 1ms
- Class T2: 0.1ms
- Class T3: ±25µs
- Class T4: ±4µs
- Class T5:±1µs

For example, a low resource requirement might involve time-stamping a sparse set of events at 1ms accuracy. A medium resource requirements might be time-stamping a medium dense set of events at 0.1ms accuracy while controlling a set of events at 4µs accuracy would generate a high resource requirements.

While trying to pin down the amount of resource to allocate, take the time to look through the paper given in Reference 2. Doing so will impart a greater understanding of what you are trying to do and what steps needed for success.

Representing time values

Throughout IT history, there have been several attempts to represent date and time values. For industrial network purposes, most of them are either too coarse (precision in the millisecond range) or too complicated. Fortunately, the dominant time synchronisation protocol (NTP/SNTP) defines a practical time stamp format. The following offers a suitable definition[2]:

NTP timestamps are represented as a 64-bit unsigned fixed-point number, in seconds relative to 0h on 1 January 1900. The integer part is in the first 32 bits and the fraction part in the last 32 bits. This format allows convenient multiple-precision arithmetic and conversion to Time Protocol representation (seconds), but does complicate the conversion to ICMP Timestamp message representation (milliseconds). The precision of this representation is about 200ps, which should be adequate for even the most exotic requirements.

Note that since some time in 1968 the most significant bit (bit 0 of the integer part) has been set and that the 64-bit field will overflow during the year 2036. Should NTP be in use in 2036, some external means will be necessary to qualify time relative to 1900 and time relative to 2036 (and other multiples of 136 years).

In IT terms, the time value is represented in seconds as a 64-bit fix-point value in 32:32 format. The representation is very convenient, except for two small points. The precision is too high for practical use. A full-resolution clock frequency would be 4.3GHz. Even if this were practical, the synchronisation accuracy is limited to about 1µs[4]. Also an implementation would have to keep track of the rollover in 2036.

The IEEE1588 time synchronisation standard uses a slightly different time representation. The defintion[3] says that the Time Representation type shall be used to specify both Timestamps (time with respect to an epoch) and Time increments. The upper 32 bits represent seconds while the lower 32 bits represent nanoseconds. The sign of the nanoseconds member is interpreted as the sign of the entire representation. A negative timestamp indicates time prior to an epoch.

A negative increment results for example from subtracting a timestamp A from a timestamp B, where A is an instant later in time than B.

The advantage of this representation is that it is possible to express negative time. The disadvantage is that specially coded subroutines are necessary for all arithmetic operations. The same 136-year rollover rule holds here, but since IEEE1588 chose 0h on 1 January 1970 as the origin, the first rollover will occur in 2106.

Adjusting the clock speed

A standard grade crystal oscillator has a worst-case error of 100ppm over its temperature range. Expressed in another way, this means that if the real time clock starts in exact synchronisation, it may be one bit wrong after a count of 10,000. Thus, if the real time clock is to be synchronised to a master real time clock with a maximum allowable discrepancy of 0.5µs, it will require resynchronisation every 5ms. This is no problem if the resynchronisation is done in hardware by a dedicated connection, but may place an unacceptable load on the system if resynchronisation is done over a communications network.

We will attempt to design a real time clock which shall be able to deliver time values with an accuracy of 0.5µs even when resynchronisation events occur at 5ms or much longer intervals.

It follows that the hardware part of the real time clock cannot deliver sufficient accuracy on its own; it needs further hardware or software encapsulation to correct for accuracy loss. There are several ways of doing this.

Correction entirely in software. If you want to time stamp a few events every second and the required precision is in or slightly below the millisecond range, you will need a hardware timer of some sort. Let this timer run continuously and synchronise it as discussed in [4]. After synchronising, the true time depends on the hardware timer as shown in the following time formula:

$$T = k * (t - t_0) + T_0$$

In this formula, k is the 'error slope', the ratio between a true timer ▶

and the local hardware timer. T_0 is the true time when the clock was synchronised last and t_0 is the corresponding value for the local hardware timer. T is the 'true' time and t is the current value of the local hardware timer.

This approach uses some hardware resources, a moderate amount of software and CPU resources, and a standard amount of network resources. It has the advantage in that it is inexpensive in the sense that no dedicated hardware is used but it requires a moderate amount of CPU cycles

Mixed hardware/software approach. If an extra timer is available and it is possible to have the timer create an interrupt, another solution is possible. It is based on the observation that a commercial low-cost crystal oscillator has a precision of better than 100ppm, and the ratio between a 'true' timer and the local hardware timer is therefore very close to 1. If we let $k=1+a$, we can rewrite the time formula to:

$$T = (1+\alpha)*(t-t_0)+T_0 = t+(T_0-t_0)+\alpha*(t-t_0) = t+T_1+\alpha*(t-t_0)$$

Suppose we want to calculate the absolute time to the nearest microsecond. In this case the algorithm works like this:

1. Determine the time formula constants k, T_0 and t_0. Calculate α. Calculate $\tau = 1\mu s/|\alpha|$.
2. Set the time constant in the interrupt timer to τ , and start the interrupt timer.
3. Let the interrupt service routine for this interrupt timer increase or decrease T_1 by 1 (depending on the sign of α.
4. Whenever we want to calculate T, we only have to read t and add T_1.

The formula for deriving true time is very simple and only requires a moderate amount of hardware but it does use an interrupt service routine with associated overhead.

Doing it the analogue way

A varactor (variable capacitance diode) exhibits a capacitance that depends on the voltage across its terminals. Inserting such a varactor in a crystal oscillator circuit, **Fig. 1**, makes it possible to adjust the oscillator frequency over about 350ppm – more than enough for our purposes. Adding an A to D converter and some signal conditioning circuitry to the hardware allows adjustment of clock speed.

One should note that the required control voltage range exceeds that of a standard A/D converter, necessitating some signal conditioning circuitry. Also the relationship between the control voltage and the oscillator frequency is non-linear.

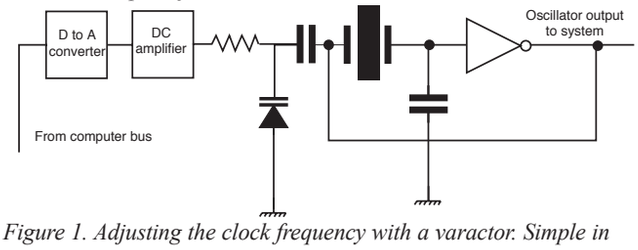

Figure 1. Adjusting the clock frequency with a varactor. Simple in theory, but complex in actual application

Clock adjustment by frequency division. An alternative method for clock adjustment uses variable division ratio on the basic clock frequency so removing the need for analogue circuitry – apart from the clock oscillator itself. Changing the divisor applied to the high frequency oscillator changes the clock rate. This system is essentially the same as that used with dual modulus (pulse swallowing) digital frequency synthesisers found in wireless comms equipment.

The problem with this method is that it is inherently non-linear. Since we are only able to use integers for divisors, the smallest change in frequency is dependent on the current divisor. If our nominal divisor is n, the smallest change is to change the divisor to either $n+1$ or $n-1$, resulting in a frequency change of either:

$1/n-1/(n+1) = 1/n*(n+1)$ or $1/(n-1)-1/n = 1/(n-1)*n$

If we only want to change the clock rate by 1ppm, this means that the start value for n must be 1000. With a basic clock rate of 1MHz, this means that our crystal oscillator must run at 1GHz!

A related method is to use a VCO frequency synthesiser phase-locked loop. The output of the frequency synthesiser is divided by a suitable amount and compared with a reference frequency using a phase discriminator. The discriminator output is filtered and used as a control voltage input to the frequency synthesiser, closing the loop.

Unfortunately, there are problems with this method as well. Leaving aside the fact that a phase-locked loop is hard to design and implement, the output of the frequency synthesiser must be an integer multiple of the reference frequency. If we only want to change the clock rate by 1ppm, this means that the reference frequency must be 1ppm of the basic clock rate. With a basic clock rate of 1MHz, this implies a reference frequency of 1Hz. Since the phase-locked loop needs a number of periods to stabilise and with each period being 1s, we are looking at a response time of several minutes from the clock rate divisor is changed until the clock rate stabilises at its new value.

Adjustment by pulse control logic

The traditional digital approach to frequency adjustment is thus impractical for very fine clock rate tuning. It is, however, possible to approach the problem in another way.

Figure 2 shows one way of implementing both a real time clock and a fine-tunable clock rate generator in one compact solution.

The clock logic block normally divides the input frequency by 2. If the *Enable* bit is set and the *Add/Subtract* bit indicates *Add*, the block inserts an extra pulse whenever the output of the Binary Rate Multiplier goes high. Conversely, if the *Enable* bit is set and the *Add/Subtract* bit indicates *Subtract*, the block suppresses one output pulse whenever the output of the Binary Rate Multiplier goes high.

The OFL bit is set whenever the MSB in the second counter rolls around from 1 to 0. One implementation is to show this bit in the LSB position in the time counter and clear it whenever it is read. The clock interface software should test this bit whenever the clock value is read and increment a software counter whenever it is read.

The Binary Rate Multiplier is a clever circuit. If its control input has the binary value m and the number of stages is n, the ratio between the

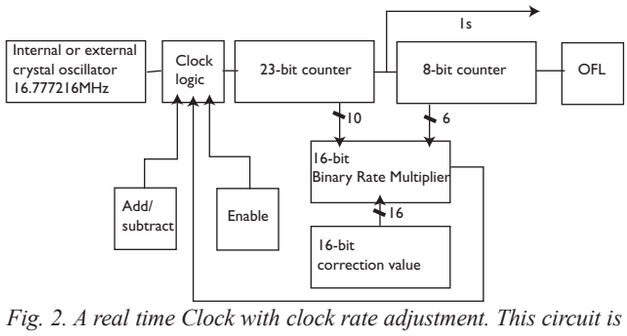

Fig. 2. A real time Clock with clock rate adjustment. This circuit is able to handle very small corrections to the clock speed.

number of output and input pulses is $m/2^n$. This Binary Rate Multiplier is connected in such a way that its LSB adds or subtracts one pulse every 64 seconds. With a length of 16 bits, it can add or subtract a maximum of slightly below 1024 pulses every second. In addition, those extra or missing output pulses are distributed as evenly as possible.

Example. If the specified crystal oscillator has a tolerance of 100ppm and the pulse count logic is disabled, the maximum error count is 838 with a resolution of 0.12µs. This is entirely within our control range. After maximum correction this will give us a real time clock with a resolution of 0.12µs and a maximum drift of one count (0.12µs) every 64s. Put another way, when the crystal oscillator has stabilised and the synchronisation is finished, the resulting clock will drift less than 0.5µs every 256s.

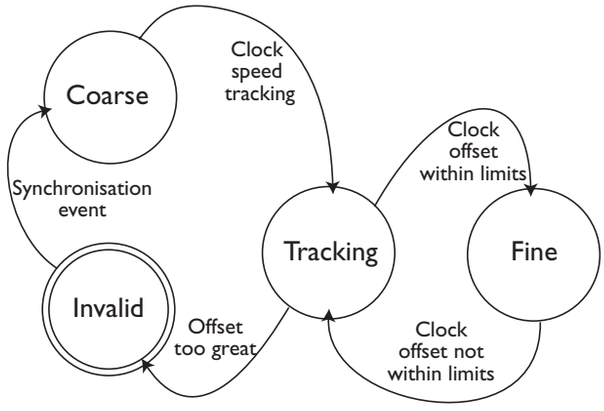

Fig. 3. State diagram for the real time clock. The adjustment algorithms and the timestamp precision are not shown.

Additions and refinements. The real time clock design as advocated has impressive specifications, but there are a few possible additions that would increase the usefulness of the circuit appreciably. If the data bus interface is narrower than 32 bits, an output latch is almost mandatory. This allows us to latch the output of the clock counter and read the contents of the latch in whatever way we want.

Another useful addition would be an externally triggered latch. This would allow a precise timestamp of an event and the clock counter could be extended to handle the full NTP time value. We do not have to use the full precision (24 bits is more than enough for automation purposes), so we could keep our 16.772MHz oscillator and assign a fixed value to the least significant bits.

Adjusting clock offset

Adjusting the clock offset is actually more complex than it looks, since there are several restrictions. The clock should not exhibit sudden jumps forward. This would result in events before and after the correction getting time stamps that are artificially far apart, which again could be interpreted as a sudden disturbance in a data collection or as a missing data set. Neither should the clock ever make any jumps backward. This would result in events before and after the correction getting time stamps that are artificially close together or even being reversed in apparent time.

A real-time clock may be in one of several possible states with respect to synchronisation; these are explained in the following paragraphs with the state type followed by the comment on the state:

Invalid. No synchronisation has been received. Any timestamp request (except for synchronisation purposes) should return a value of 0 (in all 64 bits). At the first time synchronisation, the clock is set according to the synchronisation value and the state is changed to Coarse.

Coarse. In this state, the clock speed has not yet been fully adjusted. Any timestamp request (except for synchronisation purposes) should return the time value with only 10 significant bits below the decimal point. The rest of the bits should be set to zero, except for the most significant of those, which should be set to one. In this state, no direct clock adjustment is allowed, but the clock speed may be adjusted according to the discrepancies between the local clock and the synchronisation values.

Tracking. In this state, the clock speed adjustment has brought the speed of the local clock to within a predetermined amount of the reference clock. There may still be a clock offset, however. From this state, there are two possible outcomes: If the absolute clock offset is greater than a predetermined amount, the clock speed adjustment value is kept and the state changed to Invalid. Otherwise, the clock speed is changed in order to bring the clock offset closer to 0. Whenever the absolute value of the clock offset is lower than a predetermined amount, the state changed to Fine. In this state any timestamp request (except for sync purposes) should return the time value with a number of significant bits that reflects the size of the clock offset.

Fine. In this state, any timestamp request should return with the full precision of the underlying clock. The clock speed should be continuously adjusted to keep within the predetermined bounds. If the clock offset exceeds those bounds, the state should be changed to Tracking.

References

[1] *IEC 61850 Communication Networks and Systems in Substations*, Part 5: Communication Requirements for Functions and Device Models, Part 7-2: Basic Communication Structure for Substations and Feeder Equipment, 1999.
[2] David L. Mills, *RFC 1305: Network Time Protocol (Version 3)*, Specification, Implementation and Analysis, 1992.
[3] *IEEE Standard for a Precision Clock Synchronisation Protocol for Networked Measurement an Control Systems*, IEEE Std 1588-2002.
[4] Svein Johannessen, *Time Synchronization in a Local Area Network*, IEEE Control Systems Magazine, April 2004.

Svein Johannessen is with ABB, Norway