Here's how to get a more accurate RTC clock set from an NTP time server

Phil-D
lut '18post #1
Hi

Having started a small project with an Arduino which in turn connects to a RTC Dallas 3231 and to an ESP8266 to retrieve weather information and accurate time from an NTP server, I was disappointed to find I couldn't get the time more accurate than around 1 to 2 seconds using example code for NTP timestamps.

It seemed a shame to accept that amount of inaccuracy after going to all the effort, and especially when clocks based on radio time receivers for little money can be as good as bang on, and the Dallas chip is accurate to a handful of seconds a year, that I couldn't at least get the time more accurate to set it, say within ~100ms.

Generally, example code for NPT via the ESP8266 has a disclaimer that accuracy was only within 1 or 2 seconds, so I did some tests and looked at the code to find out why that should be.

RTCs like the Dallas chips do not allow setting the clock down to the millisecond, only to the second, so this is the first problem. Say for example the time stamp from the NTP packet is 09:05:20 and 900ms. All the example code for NTP on the ESP8266 disregards the fractional seconds as they can't be set on most RTCs, so the RTC gets set as 09:05:20, which then starts from that second, i.e. 20 seconds and 0 milliseconds exactly, so right away we have set the RTC at almost a second behind. It's also inconsistent as another attempt might be more accurate as the time stamp might have a lesser fraction of a second. Overall there is a 50% chance that the RTC will be set at half a second or more slow.

So that explains up to a second out, why two seconds? Well we get the same problem when we come to read the time back from the RTC clock, as again most RTCs will not provide milliseconds. So when we read the RTC time we might get back 09:10:05, but in reality it might be 09:10:05 and 900ms, so we are almost a second out when we've read the time, add that to the potential we might have set the RTC up to almost a second slow, and we now know where the often stated 1 to 2 second accuracy comes from.

We can do better than this can't we? Here's what I'm doing which gets accuracy down to sub 100ms.

First step, lets get the fractional part of the NTP time stamp in milliseconds, this is in packets 45 to 48, we can add this under the code that gets the epoch time in seconds, easily found in any example NTP code, so we now have something like this, getting the seconds then the fractional part:

```
// Combine the 4 timestamp bytes into one 32-bit number
uint32_t NTPTime = (NtpBuffer[40] << 24) | (NtpBuffer[41] << 16) |
(NtpBuffer[42] << 8) | NtpBuffer[43];

// Now get the fractional part
uint32_t NTPmillis = (NtpBuffer[44] << 24) | (NtpBuffer[45] << 16) |
(NtpBuffer[46] << 8) | NtpBuffer[47];

// Get the fractional part as milliseconds
int32_t fractionalPart = (int32_t)(((float)NTPmillis / UINT32_MAX) *
1000);
```

So knowing we can only set the RTC using seconds where it always starts counting from 0ms, and we know the fractional part of the timestamp, we simply need to wait until the time is exactly the next second and 0ms and not a fractional part, so we just delay until the next full second, so lets add that next under that code:

```
// Increment the seconds as we are waiting for the next one
NTPTime++;

// Burn off the remaining fractional part of the existing second
delay(1000 - fractionalPart);
```

We can now set the RTC using the epoch time knowing it is the very start of a new second at 0 milliseconds. Typically, the NTPTime is converted to seconds since 1-1-2000 to set Dallas clocks, but that code remains the same as in any examples.

Of course we haven't made any adjustments for network round trip delays etc, but for now, assuming a close NTP server source, we should be down to around sub 50ms accuracy.

What about reading the RTC time back and removing fractional errors of a second. This takes a bit more ingenuity but actually sets things up quite nicely for a clock, as we are going to use the 1Hz square wave from the Dallas RTC. It needs a change to any wiring, and the INT/SQW (Square Wave) pin broken out from the RTC, this usually is on most break out boards, I've tested this with a Dallas DS3231.

We need to connect the INT/SQW to a true interrupt on the Arduino (or other), and the Dallas DS3231 SQW pin is 5 volt tolerant so a direct connection is fine but if yours is different do check the datasheet. A true interrupt is one that can detect a rising/falling edge, these are typically called INT something on Arduinos, if described as PCINTs, don't use these. For example, on the Arduino Uno the real INTs are on pins 5 and 4.

This square wave is pretty useful to us, as the falling edge indicates the start of a second precisely, even more useful is when we set the time on the RTC which resets the second fraction back to 0ms, is the square wave is stretched or shortened at the same time, so the edge is always the start of a new second, think of it like a 0ms indicator.

So how to code to use this. First, in setup we need to tell the RTC to output a 1Hz square wave, I'm using the JeeLabs RTC library which has that option. We then attach to the interrupt. So setup will include code like this (of course define the pin to the one in use). Note I am using the Ardunio pull-up here, this has worked perfectly, however you may wish to add a 10K pull up to 5 volt.

```
void setup
{
     // Set up 1Hz square wave
     rtc.writeSqwPinMode(DS3231_SquareWave1Hz);

     // Attach the interrupt and use the internal pull up
     attachInterrupt(digitalPinToInterrupt(INT_SECOND_TICK_PIN),
secondInterupt, FALLING);
     digitalWrite(INT_SECOND_TICK_PIN, HIGH);  /
}
```

Lets add two global variables, one to keep count of the seconds epoch since 2000 that the RTC uses, and another to indicate a new second interrupt has been received.

```
volatile int32_t _rtcTotalSecondsSince2000 = 0;
volatile boolean _newSecond = false;
```

Next we add the interrupt routine, which is very simple, it will simply toggle the _newSecond flag to true and increment the epoch by one second. For debugging, we toggle the led.

```
void secondInterupt()
{
     _rtcTotalSecondsSince2000++;
     _newSecond = true;
     digitalWrite(LED_PIN, !digitalRead(LED_PIN));
}
```

In our loop, we will look for the new second flag, then, if the epoch seconds haven't been set yet (we simply look for a silly low number), we will set these from the RTC, and because this is in the interrupt from the square wave, we know there are no fractions of a second, it is exactly at the start of the new second.

```
void loop
{
    if(_newSecond)
    {
         _newSecond = false;

        if(__rtcTotalSecondsSince2000 < 1000)
```

```
        {
              _rtcTotalSecondsSince2000 = rtc.now().secondstime();
        }

         // Display the  time
         int32_t dayTotalSecondsNow = _rtcTotalSecondsSince2000;
         uint8_t hour = dayTotalSecondsNow / 3600 % 24;
         uint8_t minute = dayTotalSecondsNow / 60 % 60;
         uint8_t second = dayTotalSecondsNow % 60;

         Serial.print(hour);
         Serial.print(":");
         Serial.print(minute);
         Serial.print(":");
         Serial.println(second);
    }
}
```

This approach gives a couple of benefits, one is we only need to read from the RTC once on power up/reset, and regardless of the accuracy of the oscillator used on the Arduino, the global _rtcTotalSecondsSince2000 is always good, i.e. it will not drift off from the actual RTC time no matter how long it runs for, as it's the RTC interrupt and it's own crystal oscillator advancing the time, and not some millis check in our code based on the Arduino clock.

Second, no pun intended, we have a pulse for each new second, so in place of the code that debug prints the time, if that becomes for example an update to an LED matrix to show the time and seconds, that time is exactly updated on each new second, and we've removed completely the issue of fractions of a second causing us to be up to one second off when reading from the RTC.

If a more granular timestamp is needed and fractions of second are required, then that would need the Arduino counting millis between each second, or better yet use the 1KHz square wave and adjust the code accordingly.

By doing the above, I now have a clock that shows times and seconds that ticks exactly the same moment as my radio-controlled clock, certainly the same as far as the eye can see. It could be make more accurate by taking into account the NTP round trip times when initially setting the RTC (or refreshing it) but for me, my goal was to have something that visually matched all other accurate time displays, and this it does.

I hope that helps others.

Regards

Phil

[solved] Highly accurate Arduino time via RTC pulse to update seconds?1

- utworzono

alto777
lut '18post #2
Nice! Thank you.

6v6gt
lut '18post #3
It looks good. To take it any further and squeeze even more accuracy out of it, the next step would be to build a full NTP client instead of simply using SNTP to grab a timestamp on demand.

Phil-D
lut '18post #4
Hi

Thanks for the feedback.

We could use a crude way of accounting for the round trip on the NTP packet by storing the millis() just before the packet is sent, then on receiving back calculating the round time trip by taking millis() - millisWhenSent.

Because the time stamp is only outdated by the trip from the time server back to us we can divide by two, then use that to adjust the received time stamp. For example:

```
fractionalPart += tripDelay;  // The result of the round trip / 2

// Check if going over to a new second
if (fractionalPart > 999)
{
     NTPTime++;
     fractionalPart = 1000 - fractionalPart;
}

// We return the result at the next second deadon
NTPTime++;
delay(1000 - fractionalPart);
```

Debugging this I'm getting a trip delay of around 16ms, so it's not making any visible difference, but for longer delays in a round trip to the NTP server that might help get nearer the mark.

I've seen on the Wiki how to calculate more accurately the round trip times so may have a look at writing something but for me it's pretty close enough, and certainly better than seeing the time out by 1 or 2 seconds.

n6rob
maj '18post #5
@Phil-D Thank you very much for posting this. Now I understand - at least conceptually - why my NTP clock was never "right-on" to WWV. Although the error was negligible, I couldn't understand why it was happening. I had incorrectly assumed it was all network delay from the NTP server. I'm not an experienced coder so your detailed explanation helped me understand the more subtle issues. Thank you for taking the time to post the explanation.

1 miesiąc później

PumpkinEater
cze '18post #6
Hi Phil,
thank you very much for the description - very helpful for me.

One questions (for my understanding):

if(__rtcTotalSecondsSince2000 < 1000)

What is the reason to use the arbitrary value 1000 here and not just '0'?

Thanks.
Peter

odometer
cze '18post #7

Phil-D:
Next we add the interrupt routine, which is very simple, it will simply toggle the _newSecond flag to true and increment the epoch by one second. For debugging, we toggle the led.

```
void secondInterupt()

{
  _rtcTotalSecondsSince2000++;
  _newSecond = true;
  digitalWrite(LED_PIN, !digitalRead(LED_PIN));
}
```

I've never used interrupts, but I've read that it's not a good idea to try to do too much an in interrupt, and that there are some things (such as Serial.print) that simply should not be used an in interrupt.

I wonder: is it OK to use digitalWrite and/or digitalRead in an interrupt? (This is an honest question; I don't know the answer.)

As for debugging, I would simply have the main loop look at _rtcTotalSecondsSince2000 to see whether it is odd or even, and turn the LED on or off accordingly.

Your project reminds me of one of mine, which keeps track of the time internally to within 0.02 second or so. You might find the code amusing.
https://forum.arduino.cc/index.php?topic=408565.msg2835936#msg2835936 7

6v6gt
cze '18post #8
Most NTP libraries use blocking code. Having sent the request to the NTP server, they either issue a delay of about 1 second or wait in a while loop for the return packet.
I have an application which does not tolerate blocking code, and also needs greater than 1 second accuracy (its a speaking clock) so I have solved the blocking problem by separating the sending of the ntp request and the receiving of the reply. The accuracy problem I solve similar to the OP except I don't use a delay, I simply schedule the update for the next full second. Also for reading the RTC I'm planning to do multiple read operations using a binary chop type technique to get an accuracy of better than on tenth of a second (this part is not implemented yet).

1

sterretjeKarma: 1500+
cze '18post #9

odometer:
I've never used interrupts, but I've read that it's not a good idea to try to do too much an in interrupt, and that there are some things (such as Serial.print) that simply should not be used an in interrupt.

I wonder: is it OK to use digitalWrite and/or digitalRead in an interrupt? (This is an honest question; I don't know the answer.)

Because interrupts are disabled while running an ISR, other functionalities that rely on interrupts will not work. So if you have a long ISR routine, e.g. millis() will not be updated because the timer interrupt is disabled and e.g. a Serial.print will not print all characters because it uses interrupts to place the next byte in the hardware TX buffer once a singe byte is transmitted.

It's absolutely safe to use digitalWrite and digitalRead as they don't use interrupts; you can check in C:\Program Files (x86)\Arduino\hardware\arduino\avr\cores\arduino\wiring_digital.c (or it's equivalents in Linux/Mac) what those functions do.

odometer
cze '18post #10

6v6gt:
Most NTP libraries use blocking code. Having sent the request to the NTP server, they either

issue a delay of about 1 second or wait in a while loop for the return packet.
I have an application which does not tolerate blocking code, and also needs greater than 1 second accuracy (its a speaking clock) so I have solved the blocking problem by separating the sending of the ntp request and the receiving of the reply. The accuracy problem I solve similar to the OP except I don't use a delay, I simply schedule the update for the next full second. Also for reading the RTC I'm planning to do multiple read operations using a binary chop type technique to get an accuracy of better than on tenth of a second (this part is not implemented yet).

From what I understand, you need not use binary chop: just read the seconds again and again until they change. I'm not sure exactly how many times per second it is OK to read the seconds, though.

As for sub-second accuracy, I wanted that for a chiming clock, probably for reasons similar to why you want it for your speaking clock. You might find the timekeeping code for my chiming clock useful:
https://forum.arduino.cc/index.php?topic=408565.msg2835936#msg2835936 10
I admit, though, that the code is not as clean as it could be. This is at least partly intentional, as I wanted to make my code read the **full** date and time from the RTC many times each second in order to verify proper functioning of the RTC.

1

Phil-D
lip '18post #11
Hi

PumpkinEater:
Hi Phil,
thank you very much for the description - very helpful for me.

One questions (for my understanding):
What is the reason to use the arbitrary value 1000 here and not just '0'?

Thanks.
Peter

It's just in case it's not zero when it gets to that point in the code, for example if there are other setup routines going on then it may 1 or 2 or 3 seconds perhaps, so it's just a sure to be sure 🙂

Regards

Phil