Software-Based Real Time Clock (RTC)

What is a Real Time Clock? (RTC)

A Real-Time-Clock (RTC) is, as the name suggests, a clock which keeps track of time in a "real mode." While there are a number of 8051-compatible microcontrollers that have built-in, accurate real-time clocks (especially from Dallas Semiconductor), some simple applications may benefit from a software RTC solution that uses the built-in capabilitites of an 8051 microcontroller.

This page will go through the development of a simple software-base RTC solution using 8051 Timer 1 (T1). Thus, your software application will have the benefit of an RTC without requiring any additional hardware.

What are the drawbacks of a software-based RTC?

The drawback to this or any other similar software-based RTC is accuracy: This software RTC is based on the 8051 Timer. The 8051 Timer, in turn, is based on the crystal speed used in your application. Thus there are two potential (and real) issues that you need to take into consideration:

- Your software will require a known crystal speed. If you change the crystal speed connected to your 8051, you will have to modify the software accordingly.
- The accuracy of our RTC will only be as accurate as the crystal you use.

The variation of an RTC compared to the current time is called "drift" and is often measured in "seconds of drift per month." A specification may indicate that a given hardware RTC is accurate "+/- 10 seconds per month." If you are going to use a software-based RTC, such as this one, be sure your crystal is rated with minimal variation.

Step 1: Our Variables

Before we start developing the code, lets get a few variables established. These variables will be used frequently within interrupts, so it is a good idea to put

8052 FAQ
News
Code Library
Chips
Links
8052 Books
Product Reviews
Projects Online
8052 TCP/IP
Products Online
Products
Services Consulting

Before we start developing the code, lets get a few variables established. These variables will be used frequently within interrupts, so it is a good idea to put them in Internal RAM. To make this code as non-instrusive as possible, we'll locate our variables at the end of Internal RAM (07Ch-07Fh).

```
HOURS       EQU  07Ch
MINUTES     EQU  07Dh
SECONDS     EQU  07Eh
TICKS       EQU  07Fh
```

Our interrupt will use these four variables to keep track of time. Additionally, your main program may access these variables whenver it wishes to determine the "current time" from the RTC.

Step 2: The Crystal Frequency

The next thing we need to take into account is the speed of the crystal being used. Keep in mind that with a crystal of 11.059Mhz, Timer 1 will increment 11,059,000/12=921,583 times per second.

> **NOTE:** The standard 8051 Timer increments every 12 crystal cycles. However, some derivative chips increment their timers after a different number of crystal cycles: For example, Dallas microcontrollers can be programmed to increment every 4 cycles. If you are using a derivative that uses some value other than 12, you will have to make the appropriate changes to this code.

Let's establish some more equates to make our code more portable:

```
CRYSTAL   EQU 11059000          ;The crystal speed
TMRCYCLE  EQU 12                ;The number of crystal cycles per timer increment
TMR_SEC   EQU CRYSTAL/TMRCYCLE  ;The # of timer increments per second
```

Thus, should our crystal frequency change or should we move our code to a derivative microcontroller that uses some other value than 12, we simply need to modify our constants.

Step 3:Calculating the Timer 1 Overflow Frequency

Remember, a 16-bit timer will count from 0 to 65,535 before resetting. This is important when you consider that Timer 1 will be incremented 921,583 times per second. Obviously it will overflow it's 65,535 maximum value a number of times in the course of one second-to be exact, it will overflow 921583/65536=14 times per second. If we were to use the timer in 8-bit or auto-reload mode, the timer would end up overflowing 3599 times per second, which is a lot harder to keep track of.

So we will have timer 1 running in 16-bit mode. However, we have a problem: Timer 1 will actually overflow 921583/65536=14.0622 times per second. Obviously it's not possible for it to overflow .0622 times. This means we can't simply count the number of overflows from it counting from 0 to 65536. We'll be introducing even more inaccuracy. In the case of an 11.059Mhz crystal, this inaccuracy will only be about 0.44% which is probably acceptable. But if we were to use this same program with a 12.000Mhz crystal, the inaccuracy would be 1.70% which isn't as acceptable-especially if we consider we are already working with a less then accurate crystal. And other crystal frequencies could result in even less accuracy. Generally, the slower the crystal, the are already working with a less then accurate crystal. And other crystal frequencies could result in even less accuracy. Generally, the slower the crystal, the more pronounced the error will be-and the error can easily become significant.

That being the case, we're going to need to have timer 1 overflow at some frequency that adds up nicely to 1 second intervals. For example, 65536 timer 1 cycles is 65536/921583 = .071 seconds. In other words, for timer 1 to start counting at 0, count up to 65,535, and overflow back to 0 will take .071 seconds. The problem is that 1.00 seconds divided by .071 seconds does not produce an integer result, thus we have inaccuracy. Our goal is to have timer 1 overflow at a frequency that can be multiplied by an integer to arrive at 1.00 seconds.

For example, if instead of overflowing every .071 seconds, timer 1 were to overflow every .05 seconds, we would know that after 20 overflows exactly one second had passed. How long is .05 seconds in terms of timer cycles? Simple: 921583 * .05 = 46079.15. In other words, after timer 1 has been incremented 46079 times, 1/20th of a second (.05 seconds) have passed. The missing 0.15 is truly trivial. If you multiply the missing 0.15 by 20, we are effectively missing 3 timer cycles per second, which is an error of only 0.0003%. In other words, that's 1467 times less error than that .44% error we had above! So this is probably accurate enough.

> **NOTE:** If this ISN'T accurate enough, then you need a hardware-based RTC. Remember, your crystal will probably vary between 5 and 20% from its stated speed. So, really, if .0003% error is not acceptable, the very accuracy of your crystal is going to produce perhaps 15000 to 65000 times more error than this minor programming deviation.

So the trick is to have our timer overflow every .05 seconds instead of every .071 seconds. Remember that the timer overflows when it reaches 65,535 and is incremented to 0. We calculated above that we want the timer to overflow every 46,079 cycles. To do that, we need to have the counter start counting at some value other than 0. In fact, we need to have timer 0 start counting at 65536-46079=19457. In other words, if we initialize timer 1 to 19457, it will then take 46,079 cycles for it to reset to 0. When it resets to 0, we need to once again reset it to 19457.

Again, we want our code to be portable, so first let's define an equate that indicates how many timer cycles will pass in .05 seconds. We already have an equate TMR_SEC which indicates how many timer cycles pass in a second, so to determine how many cycles make up 1/20th of a second is just a matter of multiplying the first value by .05.

```
F20TH_OF_SECOND EQU TMR_SEC * .05
```

Thus, F20TH_OF_SECOND indicates how many cycles our timer will count in 1/20th of a second. However, we need an "initialization value" for our timer. The initialization value, as we discussed above, is actually the number 65536 less the constant we just calculated:

```
RESET_VALUE EQU 65536-F20TH_OF_SECOND
```

Now, armed with these equates, we can really start coding.

Step 4: Starting Timer 1

We will use Timer 1 in 16-bit mode as our basic underlying timer. You could also choose to use Timer 0 by making the necessary changes to the program.

First, we need to initialize timer 1 to the reset value that we calculated in our equates in the last section. We do that with the following instructions:

First, we need to initialize timer 1 to the reset value that we calculated in our equates in the last section. We do that with the following instructions:

```
MOV TH1,#HIGH RESET_VALUE  ;Initialize timer high-byte
MOV TL1,#LOW RESET_VALUE   ;Initialize timer low-byte
```

Now that timer 1 has been initialized with a reset value, we need to configure timer 1 for 16-bit mode and get it running:

```
MOV TMOD,#10h ;Set timer 1 to 16-bit mode
SETB TR1      ;Start timer 1 running
```

We're set: The timer will now overflow in 46,079 timer cycles. But then what? We need to use the 8051 interrupt facility so that whenever timer 1 overflows, our special RTC clock code will be executed.

One other thing: We should initialize our clock variables. We do this with the following instructions:

```
MOV HOURS,#00   ;Initialize to 0 hours
MOV MINUTES,#00 ;Initialize to 0 minutes
MOV SECONDS,#00 ;Initialize to 0 seconds
MOV TICKS,#20   ;Initialize countdown tick counter to 20
```

This initializes our clock to 0 hours 0 minutes 0 seconds. The tick counter is initialized to 20; more on that later.

Step 5:Configuring the Timer 1 Interrupt

Configuring the Timer 1 interrupt is very easy. We just need to enable interrupt (set the EA bit) and enable timer 1 interrupt (set the ET1) bit. We do that with the following code:

```
SETB EA   ;Initialize interrupts
SETB ET1  ;Initialize Timer 1 interrupt
```

That done, whenever timer 1 overflows (i.e., is incremented from 65536 to 0), an interrupt will be immediately triggered and the interrupt service routine (ISR) at 001Bh will be executed. So our task is to write the ISR that will be executed each time 1/20th of a second has passed.

Step 6: Writing the Timer 1 Interrupt Service Routine (ISR)

Before we write our code, let's consider what we need to do every 20th of a second:

- We need to reset timer 1 to the reset value of 19457.
- We need to increment our variable TICKS.
- If TICKS is equal to 20, it means a second has passed and we need to increment the SECONDS variable.
- If SECONDS is equal to 60, it means an entire minute has passed and we need to increment the MINUTES variable.
- If MINUTES is equal to 60, it means an entire hour has passed and we need to incrmenet the HOURS variable.
- Exit the interrupt routine.

We’ll take it one step at a time.

Step 6.1: Reset Timer 1

The first thing we need to do is reset timer 1 to our reset value. If we don’t, timer 1 will take the necessary .05 seconds to overflow the first time, but subsequent overflows will occur every .071 seconds as the timer counts from 0 up to 65,535.

Thus whenever our interrupt is triggered, we need to reset the timer to RESET_VALUE that we calculated earlier. Also remember that we need to make sure our interrupt leaves the main working variables in the same state they were in when the interrupt started, so we start by pusing the registers we will change onto the stack so we can restore them when we finish the interrupt.

Our interrupt service routine starts with:

```
ORG 001Bh                 ;This is where Timer 1 Interrupt Routine starts
PUSH ACC                  ;We’ll use the accumulator, so we need to protect it
PUSH PSW                  ;We may modify PSW flags, so we need to protect it
CLR TR1                   ;Turn off timer 1 as we reset the value
MOV TH1,#HIGH RESET_VALUE ;Set the high byte of the reset value
MOV TL1,#LOW RESET_VALUE  ;Set the low byte of the reset value
SETB TR1                  ;Restart timer 1 now that it has been initialized
```

Step 6.2: Countdown TICKS variable

Now that we’ve reset timer 1, we need to "do what needs to be done." We need to count this interrupt as a "tick." When 20 ticks have passed, we know that a second has passed. If 20 ticks have not yet passed, we need not do anything else: we simply exit the interrupt service routine. We can do this with the following code:

```
DJNZ TICKS,EXIT_RTC ;Decrement TICKS, if not yet zero we exit immediately
```

This will decrement the TICKS countdown-timer and, if it hasn’t reached zero yet, will exit. You will recall from above (Step #4) that we initialized the TICKS variable to 20. Thus each time our interrupt is triggered, TICKS will be decremented. If it hasn’t reached 20, a second has not yet passed and we simply exit to EXIT_RTC.

Step 6.3: One Second has Passed

Once TICKS is decremented to 0, the DJNZ instruction above will fail and execution will continue with this section of code meaning that a full second has passed.

We must first reset TICKS to 20 so that the countdown is ready for another second to pass, and we must increment the number of seconds. We do that with the following code:

```
MOV TICKS,#20 ;Reset the ticks variable
INC SECONDS   ;Increment the second varaiable
```

Step 6.4: Have 60 seconds passed?

After we increment SECONDS, we must obviously make sure that seconds has not overflowed. It would not make sense to indicate 85 seconds. Rather, we wish to indicate 1 minute and 25 seconds. Thus we must check to see if the value of SECONDS is equal to 60. If it isn't, that means we have not yet counted 60 seconds and we may simply exit the interrupt routine.

```
MOV A,SECONDS       ;Move the seconds variable into the accumulator
CJNE A,#60,EXIT_RTC ;If we haven't counted 60 seconds, we're done.
```

Step 6.5: Have 60 minutes passed?

If the above test fails, it means we've counted 60 seconds. Thus we need to reset the SECONDS variable to 0, increment MINUTES, and if 60 MINUTES have passed we need to reset MINUTES to 0 and increment the HOURS variable.

```
MOV SECONDS,#0      ;Reset the seconds varaible
INC MINUTES         ;Increment the number of minutes
MOV A,MINUTES       ;Move the minutes variable into the accumulator
CJNE A,#60,EXIT_RTC ;If we haven't counted 60 minutes, we're done
MOV MINUTES,#0      ;Reset the minutes variable
INC HOURS           ;Increment the hour variable
```

Step 6.6: Exit the Interrupt Routine

Finally, we need to do the standard housekeeping of any interrupt service routine: we need to restore the values that we protected on the stack in step 6.1. Then we simply finish the interrupt routine with a RETI instruction.

```
EXIT_RTC:
         POP PSW ;Restore the PSW register
         POP ACC ;Restore the accumulator
         RETI    ;Exit the interrupt routine
```

Step 7: Puting it All Together

That's really about all there is to it. We've written all the code fragments, so let's put it all together in a single program:

```
HOURS           EQU 07Ch            ;Our HOURS variable
MINUTES         EQU 07Dh            ;Our MINUTES variable
SECONDS         EQU 07Eh            ;Our SECONDS variable
TICKS           EQU 07Fh            ;Our 20th of a second countdown timer
CRYSTAL         EQU 11059000        ;The crystal speed
```

```
TMRCYCLE        EQU 12               ;The number of crystal cycles per timer increment
TMR_SEC         EQU CRYSTAL/TMRCYCLE ;The # of timer increments per second
F20TH_OF_SECOND EQU TMR_SEC * .05
RESET_VALUE     EQU 65536-F20TH_OF_SECOND

ORG 0000h                  ;Start assembly at 0000h
LJMP MAIN                  ;Jump to the main routine

ORG 001Bh                  ;This is where Timer 1 Interrupt Routine starts
PUSH ACC                   ;We'll use the accumulator, so we need to protect it
PUSH PSW                   ;Protect PSW flags
CLR TR1                    ;Turn off timer 1 as we reset the value
MOV TH1,#HIGH RESET_VALUE ;Set the high byte of the reset value
MOV TL1,#LOW RESET_VALUE  ;Set the low byte of the reset value
SETB TR1                   ;Restart timer 1 now that it has been initialized
DJNZ TICKS,EXIT_RTC        ;Decrement TICKS, if not yet zero we exit immediately
MOV TICKS,#20              ;Reset the ticks variable
INC SECONDS                ;Increment the second varaiable
MOV A,SECONDS              ;Move the seconds variable into the accumulator
CJNE A,#60,EXIT_RTC        ;If we haven't counted 60 seconds, we're done.
MOV SECONDS,#0             ;Reset the seconds varaible
INC MINUTES                ;Increment the number of minutes
MOV A,MINUTES              ;Move the minutes variable into the accumulator
CJNE A,#60,EXIT_RTC        ;If we haven't counted 60 minutes, we're done
MOV MINUTES,#0             ;Reset the minutes variable
INC HOURS                  ;Increment the hour variable
EXIT_RTC:
POP PSW                    ;Restore the PSW register
POP ACC                    ;Restore the accumulator
RETI                       ;Exit the interrupt routine

MAIN:
MOV TH1,#HIGH RESET_VALUE ;Initialize timer high-byte
MOV TL1,#LOW RESET_VALUE  ;Initialize timer low-byte
MOV TMOD,#10h              ;Set timer 1 to 16-bit mode
SETB TR1                   ;Start timer 1 running
MOV HOURS,#00              ;Initialize to 0 hours
MOV MINUTES,#00            ;Initialize to 0 minutes
MOV SECONDS,#00            ;Initialize to 0 seconds
MOV TICKS,#20              ;Initialize countdown tick counter to 20
SETB EA                    ;Initialize interrupts
SETB ET1                   ;Initialize Timer 1 interrupt
       .... Your main program continues here ...
```

Step 8: Using The RTC

Once you've included the above code in your program, you may simply add your "main" program to the end. Your program can set the RTC by setting the HOUR, MINUTE, and SECONDS variables, or may obtain the current time by reading them. Other than that, you can pretty much forget about the RTC because it will be running all by itself in the background using timer 1 interrupt.

Some Additional Comments

It's probably a good idea to point out a few shortcomings and observations about the above solution-because if I don't, I'll receive lots of email! First, there is a slight error introduced in the ISR. As you can see in the code, the ISR turns off timer 1 while it resets TH1 and TL1. In all, the timer is turned off for three instructions: It is turned off for the two MOV instructions, and it is turned off until the end of the SETB instruction. On a standard 8051, each MOV instruction requires 2 clock cycles to operate, and the SETB instruction requires 1. Thus the clock effectively loses 5 cycles due to the ISR implementation. If you wish to take this into account, you may simply replace the ISR code with the following. which will take into account these 5 "lost" cycles.

```
CLR TR1
MOV TH1,#HIGH (RESET_VALUE-5)
MOV TL1,#LOW (RESET_VALUE-5)
SETB TR1
```

Second, this solution is based on interrupts. If you use other interrupts in your program, the timer 1 interrupt may not necessarily execute right away. If another interrupt of the same priority is executing when timer 1 overflows, our RTC interrupt will not execute into the other interrupt has finished. This will introduce inaccuracy. The only way to guarantee that our RTC interrupt will always execute immediately is to give it an interrupt priority of "1" and give all other interrupts a priority of "0". This can be done with the following instruction:

```
MOV IP,#8 ;Timer 1 Priority=1, all others = 0
```

Finally, another disadvantage is the fact that the solution requires dedicated use of timer 1. Your main program isn't allowed to change the value of the timer-doing so will cause the RTC to become completely inaccurate. Your program can read the timer, but it may never change it.

Conclusion

As mentioned at the beginning, a software-based RTC is a simple solution that can be implemented instead of using RTC hardware in your design. This is a reasonable solution if you don't require tremendous accuracy or if you already have hardware in the field that doesn't have RTC hardware, but new requirements include some kind of clock. This is a neat way to avoid recalling or replacing all the hardware.
