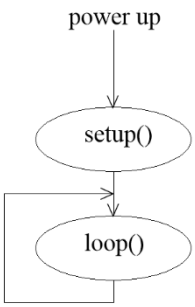
A Technique for Sharing Realtime in the Arduino Software Environment, Version 1.1

By R. G. Sparber

Disclaimer: I am mostly a self-taught programmer who has been around professional programmers for decades. I'm sure the pros have a far better way to do this. Their constructive criticism is welcome and will improve this article. One caveat – all solutions must be easy to understand. Concepts like pointers lose a lot of novice programmers. Sure, this could be done with interrupts but that can make debugging difficult.

The Problem



Arduino subroutines can either be called from setup() or loop(). Subroutines within Setup() execute once so realtime typically doesn't matter. Subroutines within loop() execute on each cycle. If one subroutine within loop() decides to wait for a while, loop() freezes until this time is over. If other code in loop() must deal with external stimuli, events can be missed.

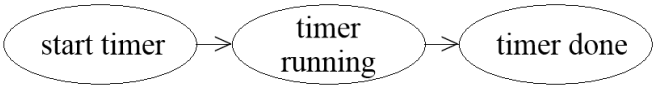
The Solution

Rather than just waiting, give control back until the desired delay time has passed. This can be accomplished by having a separate subroutine which contains a timer that is checked on each loop() cycle.

Details

Timers

I define a subroutine, Time(), that executes at the start of loop(). It contains all realtime timers which are used by subroutines.

Each timer has 3 states: 

For example, I will define RTC_TimerStateByte which will contain the current timer state. The states are:

RTC_TimerStartByte which equals 1
RTC_TimerRunningByte which equals 2
RTC_TimerDoneByte which equals 3.

When RTC_TimerStateByte is one of these states, the timer subroutine will return RTC_TimerNoErrorByte which equals 0. Otherwise, it will return the illegal value.

```
byte RTC_Timer(){
     if RTC_TimerStateByte  equals RTC_TimerStartByte{
          set the start time to the current time
          set RTC_TimerStateByte to RTC_TimerRunningByte
          return RTC_TimerNoErrorByte
     }
     if RTC_TimerStateByte equals RTC_TimerRunningByte{
               if the current time - the start time is < the duration{
                    return RTC_TimerNoErrorByte
               }else{
                    set RTC_TimerStateByte  to RTC_TimerDoneByte
                    return RTC_TimerNoErrorByte
               }
     Return RTC_TimerStateByte
}
```

The `RTC Timer` subroutine is state-driven. If the state equals `Timer Start`, it sets the start time equal to the current time so it can later calculate the lapse time. It then advances the state from `Timer Start` to `Timer Running` and returns with no error.

```
byte RTC_Timer(){
     if RTC_TimerStateByte  equals RTC_TimerStartByte{
          set the start time to the current time
          set RTC_TimerStateByte to RTC_TimerRunningByte
          return RTC_TimerNoErrorByte
     }
     if RTC_TimerStateByte equals RTC_TimerRunningByte{
               if the current time – the start time is < the duration{
                    return RTC_TimerNoErrorByte
               }else{
                    set RTC_TimerStateByte  to RTC_TimerDoneByte
                    return RTC_TimerNoErrorByte
               }
     Return RTC_TimerStateByte
}
```

The next time the timer is called, it sees that the state is not `Timer Start` so skips to the second test. There it finds that the state does equal `Timer Running` so it sees if the elapsed time is less than the defined duration. If more time is needed, it returns with no error. If enough time has elapsed, it moves to the `Timer Done` state and returns no error. If the timer state is none of the above, it returns with the illegal state value. Note that RTC_Timer is called from Time() which must deal with this returned error code.

Calling Subroutines

If realtime delays are not a problem, I can write a subroutine like this:

```
void RealTimeHog(){
     code block A
     delay for 2 seconds
     code block B
}
```

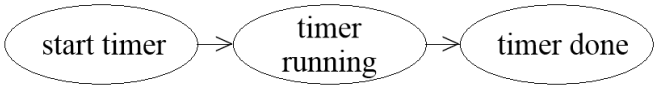
Each time I call `RealTimeHog()` it will execute `code block A`, stop all execution of code for 2 seconds, and then execute `code block B`.

But what if I had another subroutine called `Scan()` that had to read an input pin every 100 milliseconds? `Scan()` would be disabled each time `RealTimeHog()` ran.

This conflict can be resolved by defining a new subroutine that is realtime considerate:

```
  void RealTimeConsiderate(){
     if RTC_TimerStateByte equals RTC_TimerDoneByte {
          code block A
          set RTC_TimerStateByte to RTC_TimerStartByte
          return
     }
     if RTC_TimerStateByte equals RTC_TimerRunningByte, return
     if RTC_TimerStateByte equals RTC_TimerDoneByte{
        code block B
        return
     }
}
```

This subroutine depends on the RTC timer to know what to do next.



When `RealTimeConsiderate()` first runs, the timer is idle so is in the `Timer Done` state. This means it should execute `code block A`. When finished, it changes the RTC timer's state to `Timer Start`. We then return from `RealTimeConsiderate()`.

Each time `RealTimeConsiderate()` runs, it tests the timer state to see if it is still running. If so, we just return. If the timer is done, it executes `code block B` and then returns having completed its tasks.

I welcome your comments and questions.

If you wish to be contacted each time I publish an article, email me with "Subscribe" in the subject line. In the body of the email please tell me if you are interested in metalworking, software, and/or electronics so I can put you on the best distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org