Programming an Arduino In a Realtime Environment, Version 1.0

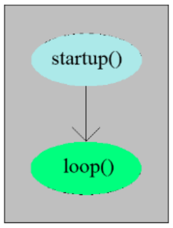
By R. G. Sparber

Disclaimer: Although I have worked with many talented professional programmers and Computer Scientists over the years, I have only a little formal training in software. This article presents how I deal with real-time and is certainly not the only or best way to do it. I welcome comments from people with more knowledge, like Dave Kellogg.

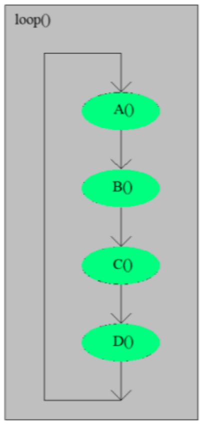
Conclusion

The software that runs on an Arduino compatible processor can handle real-time events without resorting to using interrupts.

Background



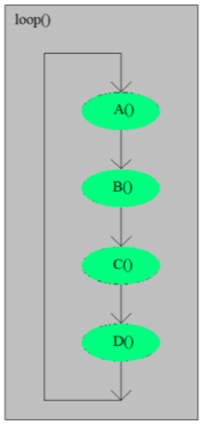
Arduino software is structured into two subroutines. If code is to run only at startup, it is placed within startup(). If the code is to run one or more times after startup, it goes into loop().



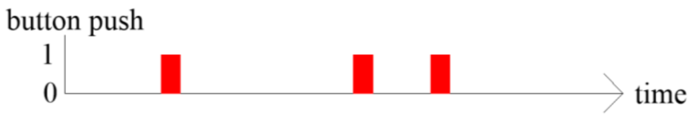
The subroutines within loop() are called in sequential order. When the last one completes, we begin the first one again.

How often a given subroutine is called depends on how much time is consumed by the other subroutines within loop(). If the code is only doing computations, time is usually not important.

The Problem

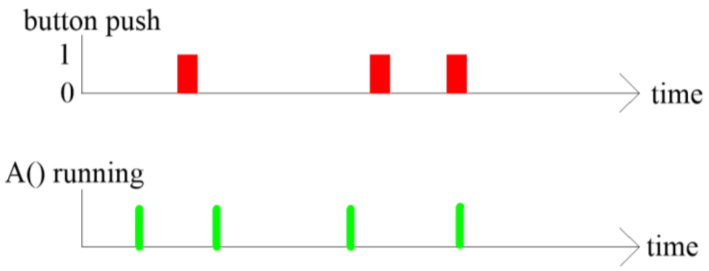


D() is interacting with the outside world.



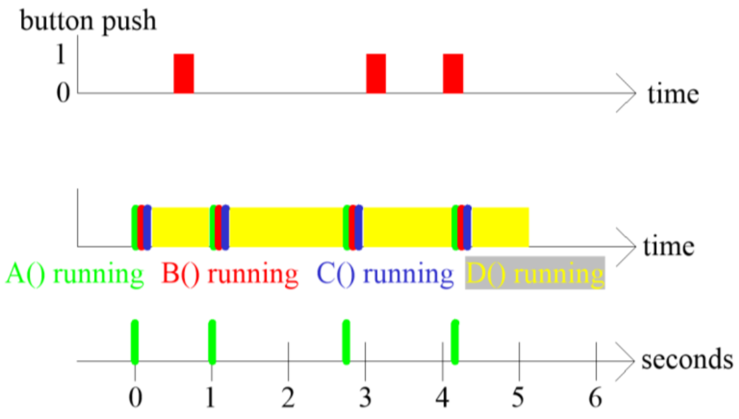
It must look for button pushes.

If loop() cycles very fast, A() will see all three button pushes. The user will experience a reliable system.



What happens if A() only runs periodically? The user pushes the button for the first time. A() ran before and after the event, so it misses it. The user gets no response, so she pushes it again. A() ran just before the push, so again misses it. Out of frustration, the user pushes the button a third time. By luck, A() was running while the button was being pressed, so sees the event.

As far as the user is concerned, this is one flaky system. As far as A() is concerned, all is fine. The user only pushed the button once.



What is going on? A(), B(), and C() all take 1 millisecond to complete but D() takes 500 to 2000 milliseconds. As viewed from the outside world, A's run time is sliding around, and it isn't looking at the button often enough. It is the victim of D() being a real-time hog.

The button push is called an asynchronous event relative to A(). When the button push lines up with A() running, the event is detected. Otherwise, it never happened as far as A() is concerned.

```
A()
    Line 1
    Line 2
B()
    Line 1
    Line 2
    Line 3
C()
    Line 1
    →interrupt
        Line a
        Line b
        ←
    Line 2
D()
    Line 1
    Line 2
```

A standard way to deal with such asynchronous events is to set up the Arduino, so the button push triggers an interrupt. At that moment, we stop executing the "base level" code and jump to interrupt level code. A flag can be set to say the button had been pushed, and then we jump back to the baseline code.

This is a good solution if there aren't too many external events, and the interrupt level code doesn't take too long to run.

Life takes a decided turn for the worse when bugs are caused by the interaction of the base level code and the interrupt level code. These bugs are driven by the button push, so they occur at random places in the base level code. This makes them extremely hard to find.

By avoiding interrupt level code, the software behaves predictably. We just need each subroutine within loop() to behave themselves and not be a real-time hog. This is easier said than done.

Although delays due to real-time hogs tend to vary, I will illustrate the problem with a fixed delay.

```
A()
    Line 1
    Line 2
B()
    Line 1
    Line 2
    Line 3
C()
    Line 1
    Line 2
D()
    Line 1
    delay(1000)
    Line 2
```

Within D(), I need to run some code, wait 1000 milliseconds, and then run more code. This delay stops the execution of loop(), which causes all of the subroutines to shift in time.

A Solution

High-Level View

```
Time()
A()
B()
C()
D()
```

The first piece of the puzzle is to create a new subroutine that I call Time(). It keeps track of real-time and communicates with the rest of the code via flags.

Time() can contain as many timers as needed. They all work the same way:

1. When a client subroutine wants to start their timer, they set a Timer Running flag.
2. This timer sees the flag is true and starts a timer based on the Arduino's hardware real-time clock.
3. Every time the timer code runs, it checks the time.
4. When the specified interval completes, it clear the flag.

The client subroutine checks this flag each time it runs, so it knows when the time is up.

```
A()
    Line 1
    Line 2
B()
    Line 1
    Line 2
    Line 3
C()
    Line 1
    Line 2
D()
    Line 1
    delay(1000)
    Line 2
```

versus

```
Time()
A()
B()
C()
D()
    If Timer Running true,
return.
    If flow control flag is
    false, set Timer Running true
    and then execute Line 1.
    If flow control flag is true,
    set flow control flag  false
    and then execute Line 2.
```

Let's walk through this logic in a little more detail.

```
Time()
A()
B()
C()
D()
    If Timer Running true,
     return.
    If flow control flag is
    false, execute Line 1, set
    the Timer Running true, set
    the flow control flag to
    true, and return.
    If flow control flag is true,
    set flow control flag  false
    and then execute Line 2.
```

In setup(), we set the `Timer Running` and flow control flag to false.

When we call D() for the first time, the `Timer Running` and flow control flag are false. We execute **Line 1,** which was before the delay(1000) function. Rather than being a real-time hog, we set `Timer Running` to true, which will tell the timer within Time() to record the current real-time and start looking for when the timed interval is up. It then sets the flow control flag to true. This will signal to D() that we should start execution after **Line 1**. We then return from D().

While the timer is running, `Timer Running` is true. Each time we enter D(), we just return.

When the timed interval has concluded, the timer sets the `Timer Running` to false.

The next time D() runs, it sees that the `Timer Running` is false so it checks the flow control flag. It was set true, so we skip **Line 1**. Then we set the flow control flag to false in preparation for the next time we run D(). And finally, we execute **Line 2**.

The Code

For the timer function, I define TimerRunning as false, BeginTimer as true, and TimeLimit as 1000.

In Time() I will have

```
if(TimerRunning == false){
     return;
}else{ //timer is running
     if(BeginTimer){//at beginning of interval
          StartTime = millis();//initialize start time
          BeginTimer = false;//stop reinitializing
          Return;
     }
     if((millis() - StartTime) >= TimeLimit){//interval over?
          TimerRunning = false;//signal calling subroutine
          BeginTimer = true;//prepare for next start
          Return;
     }
}
```

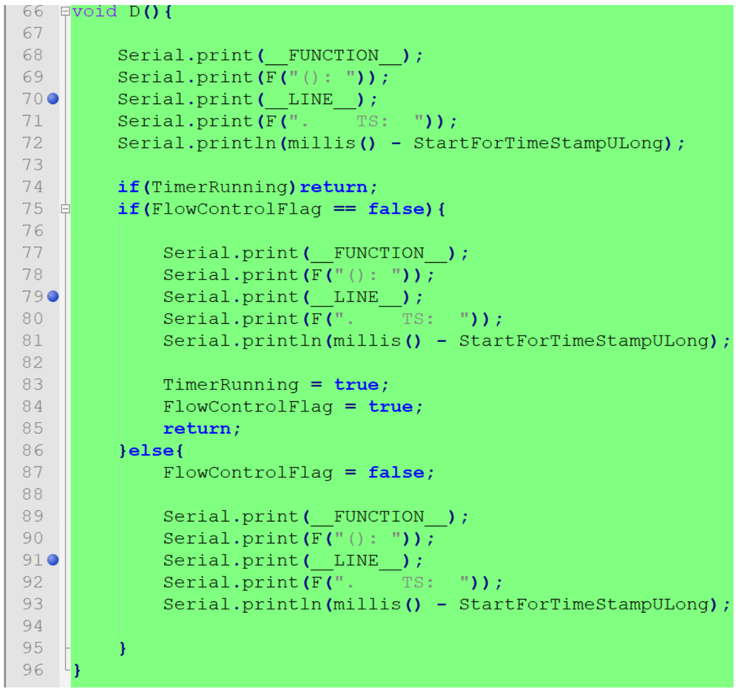
To recap: when D() is run for the first time, it executes **Line 1** and then returns. After the specified interval, it executes **Line 2**. Then it clears all flags in preparation for executing **Line 1** again the next time D() is called.

```
void D(){
     if(TimerRunning)return;
     if(FlowControlFlag == false){
          Line 1;
          TimerRunning = true;
          FlowControlFlag = true;
          Return;
     }else{
          FlowControlFlag = false;
          Line 2;
     }
}
```

Test Code

I placed diagnostic print statements[2] in all subroutines. Each line displays the subroutine, line number, and a time stamp.

Subroutines A, B, and C only contain diagnostic print statements. D has three of them:



```
66  void D(){
67
68      Serial.print(__FUNCTION__);
69      Serial.print(F("(): "));
70      Serial.print(__LINE__);
71      Serial.print(F(".     TS:  "));
72      Serial.println(millis() - StartForTimeStampULong);
73
74      if(TimerRunning)return;
75      if(FlowControlFlag == false){
76
77          Serial.print(__FUNCTION__);
78          Serial.print(F("(): "));
79          Serial.print(__LINE__);
80          Serial.print(F(".     TS:  "));
81          Serial.println(millis() - StartForTimeStampULong);
82
83          TimerRunning = true;
84          FlowControlFlag = true;
85          return;
86      }else{
87          FlowControlFlag = false;
88
89          Serial.print(__FUNCTION__);
90          Serial.print(F("(): "));
91          Serial.print(__LINE__);
92          Serial.print(F(".     TS:  "));
93          Serial.println(millis() - StartForTimeStampULong);
94
95      }
96  }
```

D(): 70 is start of D()

D(): 79 executed code before 1000 ms delay runs

D(): 91 executed when code after delay runs

[2] DTFA.pdf (sparber.org)

Test run 1

A 200 ms delay was added to loop() to reduce the number of diagnostic prints.
Note: the line numbers are off from the above listing.

```
A(): 44.   TS:  0
B(): 52.   TS:  0
C(): 60.   TS:  1
D(): 69.   TS:  1  first time D() runs
D(): 76.   TS:  1 executed code before 1000 ms delay
A(): 44.   TS:  102  subtract out the delay(100) and A() runs every 2 ms
B(): 52.   TS:  103
C(): 60.   TS:  103
D(): 69.   TS:  104 D() doesn't run because timer is running
A(): 44.   TS:  204
B(): 52.   TS:  204
C(): 60.   TS:  205
D(): 69.   TS:  205  D() doesn't run because timer is running
…
A(): 44.   TS:  1019
B(): 52.   TS:  1019
C(): 60.   TS:  1020
D(): 69.   TS:  1020  D() doesn't run because timer is running
A(): 44.   TS:  1121
B(): 52.   TS:  1122
C(): 60.   TS:  1122
D(): 69.   TS:  1123 start of D()  Time() clears 1000 ms flag
D(): 86.   TS:  1123 executed code after 1000 ms delay
A(): 44.   TS:  1223
B(): 52.   TS:  1224
C(): 60.   TS:  1224
D(): 69.   TS:  1225 start of D()
D(): 76.   TS:  1225  executed code before 1000 ms delay
…
D(): 69.   TS:  2245
A(): 44.   TS:  2345
B(): 52.   TS:  2346
C(): 60.   TS:  2346
D(): 69.   TS:  2347
D(): 86.   TS:  2347  executed code after 1000 ms delay
```

Conclusion

A(), B(), and C() each run every 1 to 2 ms. D() runs every 1000 ms plus the 200 ms added to slow down loop().

Test Run 2

200 ms delay removed; only print from D() before and after 1000 ms delay.

```
D(): 78.    TS:  0       code before 1000 ms delay runs
D(): 88.    TS:  1001 code after 1000 ms delay runs
D(): 78.    TS:  1001  code before 1000 ms delay runs
D(): 88.    TS:  2001 code after 1000 ms delay runs
D(): 78.    TS:  2001 code before 1000 ms delay runs
D(): 88.    TS:  3001 code after 1000 ms delay runs
D(): 78.    TS:  3001 code before 1000 ms delay runs
D(): 88.    TS:  4001 code after 1000 ms delay runs
```

Conclusion

The 1000 ms delay is evident.

I welcome your comments and questions.

If you want me to contact you 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 plus electronics, kayaking, and/or the Lectric XP eBike so I can put you on the right distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line. No hard feelings.

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