Achieving a regular flow of data between main() and an interrupt service routine using a lock-free circular queue with a producer-consumer design pattern

Matthew T. Pandina
artcfox@gmail.com

March 9, 2018

This tutorial aims to illustrate how to take data that might not be produced at a regular interval by a routine inside main(), and smooth out the consumption of that data by an ISR, so it may be consumed at a regular interval using a lock-free circular queue.

The sample code was written using AVR-GCC for an AVR ATmega328P, but it can be easily modified to work with other compilers and/or chips.

Who is the producer and who is the consumer?

In this tutorial, the producer will be code that runs from within main(), and the consumer will be code that runs from within an ISR that is triggered by a timer overflow event.

How do they communicate?

The producer and consumer will coordinate their efforts using a lock-free circular queue. For this example we will be producing and consuming a C struct of RGB triplets (3 bytes of data).

Create a new AVR project using your favorite method. The method I use is to open a Terminal window, change to my avr projects directory, and then use the `avr-project` command to create a new project based on a template. Here is what creating a new project looks like on my terminal:

```
user@debian:~$ cd avr
user@debian:~/avr$ avr-project producer_consumer
Using template: /home/user/.avr-project/templates/TemplateProject
user@debian:~/avr$ cd producer_consumer/
user@debian:~/avr/producer_consumer$ ls
main.c  Makefile
user@debian:~/avr/producer_consumer$
```

The contents of the Makefile might have to be tweaked depending on what chip you are using, but I am going to assume that your Makefile has already been properly configured for your chip and clock speed. Open main.c in your favorite editor, and let's get started!

First include stdint.h so we can use its typedefs for our variable declarations:

```
#include <stdint.h>
```

Next define the structure that will hold an RGB triplet:

```
struct _RGB {
  uint8_t r;
  uint8_t g;
  uint8_t b;
};
typedef struct _RGB RGB;
```

Now create our circular queue:

```
volatile RGB queue[128];
```

Making the length of your circular queue a power of two will ensure that any math that needs to account for the queue length stays fast, since the compiler will be able to turn the modulo operation into a bitwise AND operation.

Next create the variables that keep track of the head and tail of our queue:

```
volatile uint8_t head = 0;
volatile uint8_t tail = 0;
```

The volatile qualifiers above are necessary, since we will be reading and writing to these variables from inside main() and an ISR. In this tutorial we do not need to use an ATOMIC_BLOCK to guarantee atomic access to head and tail, since both are of type uint8_t, we only ever modify head from within the ISR, and we only ever modify tail from within main()[1].

Now define the methods that let us know if the queue is empty or full:

```
// Returns 1 if the queue is empty, 0 otherwise
static inline uint8_t empty() {
  return (head == tail);
}

// Returns 1 if the queue is full, 0 otherwise
static inline uint8_t full() {
  return (head == (tail + 1) % NELEMS(queue));
}
```

These methods are pretty self-explanatory. When the queue is empty, head will be equal to tail, and when the queue is full, head will be one more than tail, accounting for wrap. NELEMS is a macro that computes the number of elements in a statically defined array, so add its definition near the top of our file below the include line:

```
#define NELEMS(x) (sizeof(x)/sizeof((x)[0]))
```

This is all well and good, except now we need a way to enqueue and dequeue the RGB triplets:

```
// enqueue() should never be called on a full queue,
// and should only be called by main()
static inline void enqueue(RGB *rgb) {
  queue[tail] = *rgb;
  tail = (tail + 1) % NELEMS(queue);
}
```

[1] Just because you make a variable volatile, it does not mean that it is safe to read and write to it from both main() and an ISR. For instance if head and tail were of type uint16_t instead of uint8_t, we would need to ensure that any reads and writes to them are wrapped inside an ATOMIC_BLOCK so the ISR cannot interrupt main between updating the high and low bytes. Also if we were writing to head or tail from both main() and an ISR, we would also want to wrap their access with an ATOMIC_BLOCK, so multiple writes always happen sequentially, rather than potentially clobbering each other during a read followed by a write. For more information on how to use ATOMIC_BLOCK, see the avr-libc reference page: http://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html

It is very important that we don't call `enqueue()` without first calling `full()` to be sure the queue is not full and that inside `enqueue()` we make sure to copy the data into `queue` before modifying `tail`. Otherwise we could end up with a corrupt queue.

```
// dequeue() should never be called on an empty queue,
// and should only be called by the ISR
static inline void dequeue(RGB *rgb) {
  *rgb = queue[head];
  head = (head + 1) % NELEMS(queue);
}
```

Likewise, it is just as important that we don't call `dequeue()` without first calling `empty()` to be sure that the queue is not empty and that inside `dequeue()` we make sure to copy the data out of `queue` before modifying `head`. A corrupt queue would not be very useful to us.

I like declaring these methods `static inline`, since they are so short, and it avoids the overhead of a function call.

Phew! That takes care of our lock-free circular queue which can hold RGB triplets. If this were a real program, you might want to make the `empty()`, `full()`, `enqueue()` and `dequeue()` methods a bit more generic, but I wanted to keep this tutorial as easy to understand as possible. Let's look at what we have written so far:

```
#include <stdint.h>

#define NELEMS(x) (sizeof(x)/sizeof((x)[0]))

struct _RGB {
  uint8_t r;
  uint8_t g;
  uint8_t b;
};
typedef struct _RGB RGB;

volatile RGB queue[128];
volatile uint8_t head = 0;
volatile uint8_t tail = 0;

// Returns 1 if the queue is empty, 0 otherwise
static inline uint8_t empty() {
  return (head == tail);
}

// Returns 1 if the queue is full, 0 otherwise
```

```
static inline uint8_t full() {
  return (head == (tail + 1) % NELEMS(queue));
}

// enqueue() should never be called on a full queue,
// and should only be called by main()
static inline void enqueue(RGB *rgb) {
  queue[tail] = *rgb;
  tail = (tail + 1) % NELEMS(queue);
}

// dequeue() should never be called on an empty queue,
// and should only be called by the ISR
static inline void dequeue(RGB *rgb) {
  *rgb = queue[head];
  head = (head + 1) % NELEMS(queue);
}
```

Creating the consumer

We want to design the consumer to consume RGB triplets at a regular interval, no matter what else is happening inside main(). An easy way to do this is to use a timer with an ISR set up to be called each time the timer overflows. I happen to be using an ATmega328P, so your timer registers may be a bit different if you are using a different AVR, but you can just search in your data sheet for `TCCR` and you should be able to figure out what the differences are. (If you are not using an AVR, the datasheet for your microcontroller should describe how to setup and use its timers.) For this tutorial, I will be using Timer0, which is an 8-bit timer. I'm not going to go too in-depth about how to use a timer, since Dean Camera already wrote a great timer tutorial[2].

First we need to include some header files. At the top of your file, below the first include add:

```
#include <avr/io.h>
#include <avr/interrupt.h>
```

These header files will allow us to use the registers and interrupts specific to the chip specified in your Makefile.

[2] https://www.avrfreaks.net/forum/tut-c-newbies-guide-avr-timers

Next we will create a method that will configure an overflow interrupt to fire every $\mathrm{CLK_{io}}$ / 256 / 256 cycles, and enable global interrupts.

```
// This sets up our interrupt to be called every CLK_io / 256 /
// 256 cycles. The second / 256 is there because the interrupt
// is only called on 8-bit overflow.
static void Timer0_Init(void) {
  // Normal port operation, OC0A disconnected; Normal
  TCCR0A = 0;

  // Set prescaler to CLK_io / 256 (3.556 ms period at 18.432MHz)
  TCCR0B |= (1 << CS02);

  // Enable overflow interrupt
  TIMSK0 |= (1 << TOIE0);

  // Enable global interrupts
  sei();
}
```

Now let's start filling in the ISR that will be called when our timer overflows:

```
ISR(TIMER0_OVF_vect) {
  if (!empty()) {     // Is there something on the queue?
    RGB rgb;
    dequeue(&rgb);    // Hooray, let's see what it is!

    // Do something interesting with it...

  } else {
    // If we get here it means that the queue was empty.
    // Since we want to consume RGB triplets at a regular
    // interval, having an empty queue is bad. Maybe our
    // timer interval is too short...
  }
}
```

Once `Timer0_Init()` has been called, and the timer is configured, this ISR will be called every $\mathrm{CLK_{io}}$ / 256 / 256 clock cycles. We might not know how many clock cycles the producer will take to produce a new RGB triplet, as this could depend on different code paths, or external sensors, or we might just be lazy and don't want to figure out how many clock cycles our complicated routines in main() might take. That's ok, because we can add some smarts to the consumer!

Adding smarts to the consumer

As written above, the consumer does not wait for the queue to fill up before it tries to consume the RGB triplets; as soon as Timer0 is enabled, it just starts trying to pull colors off the queue. Our goal was to pull things off the queue at a regular interval, so we should probably wait for the queue to fill up before we start trying to remove things. After all, what good is making a buffer if we aren't going to use it?

What if we decide to go into a mode where the producer shouldn't be producing anything? We would want a way for it to tell the consumer "Hey! I'm going to stop producing things now, so you don't need to be looking for anything quite yet."

Let's define a new variable called `enable_consumer` that controls whether the consumer is enabled or not, and put it just below our definition of `tail`:

```
volatile uint8_t enable_consumer = 0;
```

Note that `enable_consumer` defaults to zero, since we want our queue to fill up before anything gets consumed. It will be the job of the producer to enable the consumer as soon as it notices that the queue is full.

Let's modify the ISR to respect this new variable:

```
ISR(TIMER0_OVF_vect) {
  if (enable_consumer) {
    if (!empty()) {     // Is there something on the queue?
      RGB rgb;
      dequeue(&rgb);    // Hooray, let's see what it is!

      // Do something interesting with it...

    } else {
      // If we get here it means that the queue was empty.
      // Since we want to consume RGB triplets at a regular
      // interval, having an empty queue is bad. Maybe our
      // timer interval is too short...
    }
  }
}
```

Now our consumer can be enabled or disabled, though it hasn't really bought

us anything yet[3].

Assuming `enable_consumer` is true, and the queue is not empty, everything is fine, but what happens if `enable_consumer` is true, and the queue is empty. This doesn't quite fit with our goal of consuming colors at a regular interval. Furthermore, if the queue gets emptied, and we don't allow it to fill back up again, we will probably keep emptying it at an irregular interval.

Here's an idea! Why not automatically increase our timer interval if we find that our queue is empty?

To further this idea, let's add a couple of variables, and some more smarts to our ISR as follows:

```
ISR(TIMER0_OVF_vect) {
  // Cycle counter for knowing which cycle we are on
  static uint8_t cycle = 0;

  // Timer cycles that need to pass before we attempt a dequeue.
  static uint8_t consume_every = 1;

  if (enable_consumer && ++cycle >= consume_every) {
    cycle = 0;          // reset the cycle counter

    if (!empty()) {     // Is there something on the queue?
      RGB rgb;
      dequeue(&rgb);    // Hooray, let's see what it is!

      // Do something interesting with it...

    } else {
      // If we get here it means that the queue was empty.
      // Since we want to consume RGB triplets at a regular
      // interval, having an empty queue is bad. Maybe our
      // timer interval is too short...
    }
  }
}
```

Note that `cycle` and `consume_every` are declared `static`.

Now each time the ISR is called when the consumer is enabled, we will increment `cycle` and compare it to our `consume_every` variable, which has

[3]Note that if the consumer is the only code in our ISR, we could simply disable/enable the timer rather than creating the `enable_consumer` variable, but I usually use this ISR to de-bounce my buttons as well, and if I disabled the timer it would also disable the button de-bouncing code.

been initialized to one. We are using >= instead of == for reasons that will become clear later. Once `cycle` reaches `consume_every`, we reset `cycle` back to zero. Still nothing has changed, but the astute reader will see where we are headed with this.

Inside that `else` clause, we now have a way to increase our timer interval if we found an empty queue when we were expecting to find a RGB triplet!

Let's add that code now:

```
ISR(TIMER0_OVF_vect) {
  // Cycle counter for knowing which cycle we are on
  static uint8_t cycle = 0;

  // Timer cycles that need to pass before we attempt a dequeue.
  static uint8_t consume_every = 1;

  if (enable_consumer && ++cycle >= consume_every) {
    cycle = 0;          // reset the cycle counter

    if (!empty()) {     // Is there something on the queue?
      RGB rgb;
      dequeue(&rgb);    // Hooray, let's see what it is!

      // Do something interesting with it...

    } else {
      // If we get here it means that the queue was empty.
      // Since we want to consume RGB triplets at a regular
      // interval, having an empty queue is bad. Maybe our
      // timer interval is too short...

      consume_every++;      // wait an additional cycle next time
      enable_consumer = 0;  // wait for the queue to fill up
                            // before we try again
    }
  }
}
```

Now if we find the queue is empty when we were expecting data, we can actually do something about it. First we increment `consume_every`, so that next time we will wait one more timer cycle before we look at the queue again. Then we set `enable_consumer` back to zero to ensure that the queue gets filled up again before we start pulling RGB triplets off. Again, the producer is responsible for re-enabling the consumer when it sees that the queue has become full.

Note that we could be clever and instead use a 16-bit timer, and modify TOP and/or the prescaler to change the interval, but I decided to go with something a bit easier to understand and implement for this tutorial.

Also note that, instead of incrementing `consume_every` by one each time, you can use whatever algorithm you want here, such as doubling it or using a lookup table. Just watch you don't overflow your variables! If you find that your `consume_every` variable is getting anywhere close to overflowing, you should probably choose a different prescaler and/or TOP value for your timer, otherwise you are just wasting cycles calling the ISR way too often for nothing.

We will revisit this ISR once more to add a tiny enhancement, but for now let's move on to our producer.

Creating the producer

To keep this part simple, we are going to create a producer that just loops over and over producing sequential RGB triplets and putting them onto the queue at an irregular interval. Most likely in a real application your producer will be much more complicated than this, but for the sake of example, this will do just fine.

First create the skeleton for main(), which for now will just call our Timer0 initialization function:

```
int main(void) {
  // Configure and start the timer
  Timer0_Init();

  for (;;) {
  }

  return 0;
}
```

Next add three nested loops which will produce our RGB triplets:

```
int main(void) {
  // Configure and start the timer
  Timer0_Init();

  for (;;) {
```

```
    RGB rgb = {0};

    do {
      do {
        do {
          // Inside here we have our RGB triplet
        } while (++rgb.b != 0);
      } while (++rgb.g != 0);
    } while (++rgb.r != 0);
  }

  return 0;
}
```

This just creates an RGB triplet, with all of its members initialized to zero, and then loops over them, eventually hitting all 2^{24} colors before repeating.

Now what we need to add is code that will pause the producer and enable the consumer if the queue is full, and if the queue was not full, put our RGB triplet onto the queue and continue looping:

```
int main(void) {
  // Configure and start the timer
  Timer0_Init();

  for (;;) {
    RGB rgb = {0};

    do {
      do {
        do {
          // Inside here we have our RGB triplet
          while (full())          // stop producing if queue full
            enable_consumer = 1;  // and enable the consumer

          enqueue(&rgb);          // copy our color onto the queue

        } while (++rgb.b != 0);
      } while (++rgb.g != 0);
    } while (++rgb.r != 0);
  }

  return 0;
}
```

First we check to see if the queue is full, and if so, we enable the consumer

and keep spinning until the queue is no longer full. This effectively stops main() until the ISR calls `dequeue()`.

As written, our producer will always be faster than our consumer, so let's slow it down and ensure that it produces RGB triplets at an irregular interval, by delaying for random amounts between colors.

For this we need to include a couple more header files, so at the top of the file below the other includes add the following lines:

```
#include <util/delay.h>
#include <stdlib.h>
```

These includes will allow us to use the `_delay_ms()` and `random()` functions respectively.

Now add the code to delay a random amount after we enqueue a color:

```
int main(void) {
  // Configure and start the timer
  Timer0_Init();

  for (;;) {
    RGB rgb = {0};

    do {
      do {
        do {
          // Inside here we have our RGB triplet
          while (full())          // stop producing if queue full
            enable_consumer = 1;  // and enable the consumer

          enqueue(&rgb);          // copy our color onto the queue

          uint8_t delay_in_ms = random() % 16;
          while (delay_in_ms--)
            _delay_ms(1); // avoid pulling in floating point code

        } while (++rgb.b != 0);
      } while (++rgb.g != 0);
    } while (++rgb.r != 0);
  }

  return 0;
}
```

This just picks a random integer between zero and fifteen, and delays for that many milliseconds. Many of you are probably wondering why I chose

to put a `while` loop around the call to `_delay_ms(1)`, instead of just calling `_delay_ms(delay_in_ms)` directly. If you look at the avr-libc documentation for `_delay_ms()`, you will see that it accepts a `double`. If we end up calling it with an argument that the compiler doesn't recognize as constant, you will end up pulling in a lot of slow floating point code and wasting space.

A step back, looking at the big picture

We've come a long way so far!

We have a consumer built using a timer-based ISR that will get called at a regular interval, which will pull RGB triplets off a lock-free, array-based circular queue. The consumer has enough smarts in it to automatically increase the interval it consumes at if it finds an empty queue, and it even allows the queue to fill back up again before resuming consumption.

Note that if you don't want the consumer to stop and re-buffer before it finds the optimal timer interval, you can monitor how high `consume_every` gets during normal operation, and initialize it to that value instead of initializing it to one.

As written, the consumer will discover the optimal number of timer intervals that need to occur between dequeues. This works great if we want the consumer to consume at the fastest regular interval that is possible, but you might want to allow the user to increase that delay so the consumer consumes at an slower, possibly adjustable rate.

Near the top of the file, below the declaration for `enable_consumer`, add a new variable declaration:

```
// This allows us to manually increase the time between consumption
volatile uint8_t consume_every_modifier = 0;
```

Then go back to the ISR and change the `if` condition from:

```
if (enable_consumer && ++cycle >= consume_every)
```

to:

```
if (enable_consumer && ++cycle >= (consume_every +
                                   consume_every_modifier))
```

Don't forget to keep the curly brace at the end of that `if` statement!

Now from inside main(), or an ISR that might be de-bouncing your buttons, you can increase and decrease `consume_every_modifier` to manually increase or decrease the number of timer intervals that need to occur before the consumer tries to dequeue an RGB triplet from the queue.

If you are trying to determine the maximum value that `consume_every` reaches so you can hard code it to avoid re-buffering, it is best to leave `consume_every_modifier` set to zero. For determining the maximum value `consume_every` reaches, I am leaving that as an exercise for the reader. Methods that I've used are sending the value of `consume_every` out over the USART right after it is incremented in the `else` clause of the ISR, or turning on an LED if I've hard coded it to a value other than one and it ends up getting increased higher than the value I thought was the maximum. How you choose to communicate this value is up to you, or you could just leave it set to one, and let it pause and re-buffer until the interval stabilizes. It totally depends on your application.

For the sake of clarity, let's see all the code we have written so far:

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>

#define NELEMS(x) (sizeof(x)/sizeof((x)[0]))

struct _RGB {
  uint8_t r;
  uint8_t g;
  uint8_t b;
};
typedef struct _RGB RGB;

volatile RGB queue[128];
volatile uint8_t head = 0;
volatile uint8_t tail = 0;

volatile uint8_t enable_consumer = 0;

// This allows us to manually increase the time between consumption
volatile uint8_t consume_every_modifier = 0;

// Returns 1 if the queue is empty, 0 otherwise
static inline uint8_t empty() {
```

```
  return (head == tail);
}

// Returns 1 if the queue is full, 0 otherwise
static inline uint8_t full() {
  return (head == (tail + 1) % NELEMS(queue));
}

// enqueue() should never be called on a full queue,
// and should only be called by main()
static inline void enqueue(RGB *rgb) {
  queue[tail] = *rgb;
  tail = (tail + 1) % NELEMS(queue);
}

// dequeue() should never be called on an empty queue,
// and should only be called by the ISR
static inline void dequeue(RGB *rgb) {
  *rgb = queue[head];
  head = (head + 1) % NELEMS(queue);
}

// This sets up our interrupt to be called every CLK_io / 256 /
// 256 cycles. The second / 256 is there because the interrupt
// is only called on 8-bit overflow.
static void Timer0_Init(void) {
  // Normal port operation, OC0A disconnected; Normal
  TCCR0A = 0;

  // Set prescaler to CLK_io / 256 (3.556 ms period at 18.432MHz)
  TCCR0B |= (1 << CS02);

  // Enable overflow interrupt
  TIMSK0 |= (1 << TOIE0);

  // Enable global interrupts
  sei();
}

ISR(TIMER0_OVF_vect) {
  // Cycle counter for knowing which cycle we are on
  static uint8_t cycle = 0;

  // Timer cycles that need to pass before we attempt a dequeue.
  static uint8_t consume_every = 1;

  if (enable_consumer && ++cycle >= (consume_every +
                                     consume_every_modifier)) {
    cycle = 0;        // reset the cycle counter
```

```
    if (!empty()) {      // Is there something on the queue?
      RGB rgb;
      dequeue(&rgb);    // Hooray, let's see what it is!

      // Do something interesting with it...

    } else {
      // If we get here it means that the queue was empty
      // Since we want to consume RGB triplets at a regular
      // interval, having an empty queue is bad. Maybe our
      // timer interval is too short...

      consume_every++;      // wait an additional cycle next time
      enable_consumer = 0;  // wait for the queue to fill up
                            // before we try again
    }
  }
}

int main(void) {
  // Configure and start the timer
  Timer0_Init();

  // Uncomment the following line to force the consumer to consume
  // slower than its max rate.
  //consume_every_modifier = 10;

  for (;;) {
    RGB rgb = {0};

    do {
      do {
        do {
          // Inside here we have our RGB triplet
          while (full())          // stop producing if queue full
            enable_consumer = 1;  // and enable the consumer

          enqueue(&rgb);        // copy our color onto the queue

          uint8_t delay_in_ms = random() % 16;
          while (delay_in_ms--)
            _delay_ms(1); // avoid pulling in floating point code

        } while (++rgb.b != 0);
      } while (++rgb.g != 0);
    } while (++rgb.r != 0);
  }
```

```
  return 0;
}
```

Note that I added a commented out line of code inside main() after the call to `Timer0_Init()`, which you can uncomment to force the consumer to delay longer between consuming RGB triplets inside the ISR. In this example I just hard coded it to ten, but feel free to experiment.

As written, the ISR does not do anything with the RGB triplet it removes from the queue. You'd want to replace the comment:

```
// Do something interesting with it...
```

with whatever you need for your particular application.

Conclusion

I hope you enjoyed this tutorial!

A heavily commented, full featured example which uses USART0 to output text so the user can trace what happens and see the output of the consumer and/or producer is provided as an attachment to this PDF file (you may have to double-click to get it to open). Please note that your clock speed is important when using the USART. If your clock speed is not within the generally accepted 2% margin of error, you will get a warning when compiling. Setting the variable `CLOCK` in your Makefile is not enough, you have to actually attach a crystal, and set the fuses so your chip uses your crystal. Failure to heed this advice will most likely result in garbled output. If all else fails, set the variable `CLOCK` in the Makefile to your actual clock speed, and try setting the BAUD rate really low in the code, like 1200.

How can this be lock-free and still work?

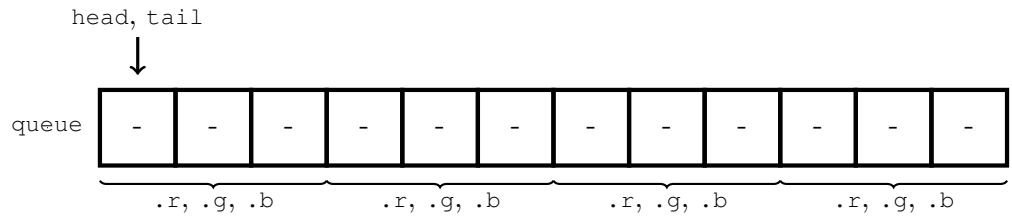
I kind of glossed over why I don't need to use an `ATOMIC_BLOCK` in my tutorial, but I will attempt to explain it better here.

At the basic level, the code boils down to this:

```
volatile RGB queue[4];  // Changed to 4 so I can draw diagrams
volatile uint8_t head = 0;
```

```
volatile uint8_t tail = 0;

ISR(TIMER0_OVF_vect) {
  if (head != tail) {
    RGB rgb = queue[head];
    head = (head + 1) % NELEMS(queue);
    // Do something with rgb
  }
}

int main(void) {
  Timer0_Init();

  for (;;) {
    RGB rgb;
    getNextColor(&rgb); // Something that sets the next color

    while (head == (tail + 1) % NELEMS(queue));
    queue[tail] = rgb;
    tail = (tail + 1) % NELEMS(queue);
  }
}
```

When we start, our state looks like this:

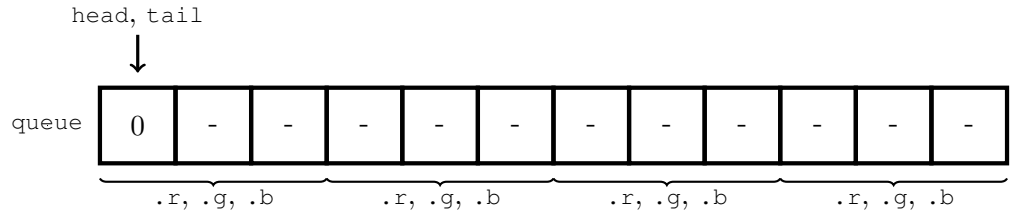


ISR sees `head == tail` and does nothing. Since both `head` and `tail` are `volatile` and `uint8_t`, there are no atomicity problems with this.

main() sees `head == tail`, so it skips the `while` loop. Again, since both `head` and `tail` are `volatile` and `uint8_t`, there are no atomicity problems.

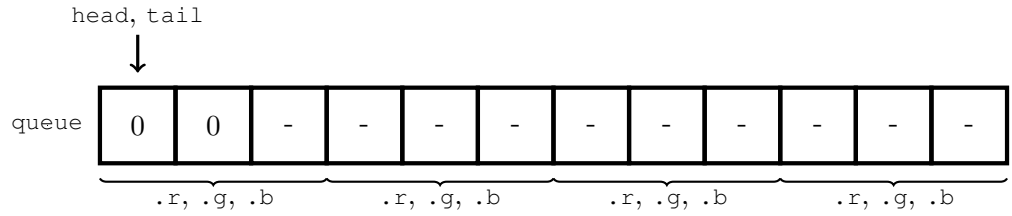
ISR sees `head == tail`, and does nothing. No problem.

main() copies `rgb.r` into `queue[tail].r`. The only place that `tail` will ever be modified is from within main(), so this is safe.



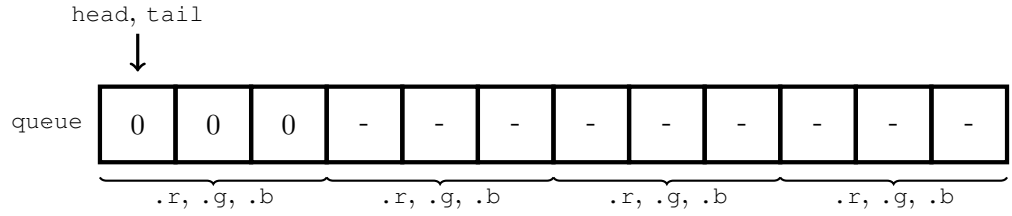
ISR sees `head == tail`, and does nothing. No matter what, the ISR does not modify `tail`. No problem.

main() copies `rgb.g` into `queue[tail].g`. Again, `tail` will not have changed, so this is safe.



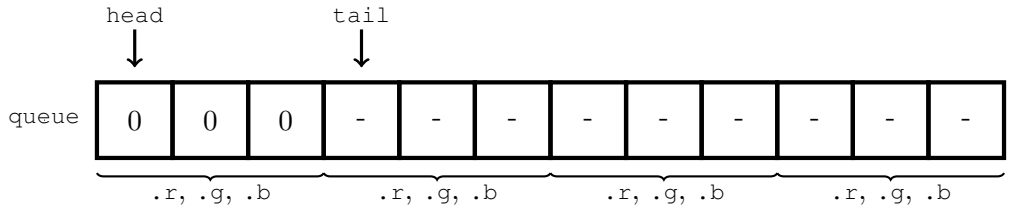
ISR sees `head == tail`, and does nothing. No problem.

main() copies `rgb.b` into `queue[tail].b`. Still no problems.



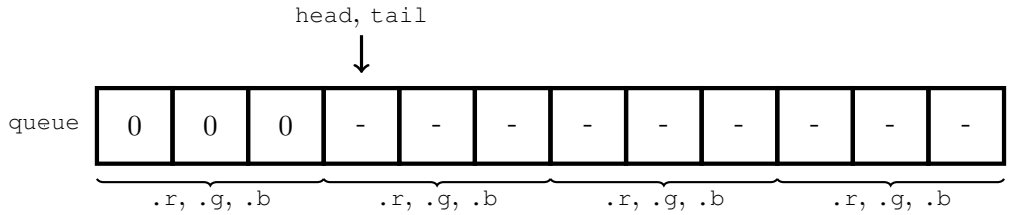
ISR still sees `head == tail`, and does nothing. No problem.

main() adds one to `tail`, accounting for wrap. Since `tail` is a `uint8_t`, this will be atomic. There is no read-modify-write problem because main() is the only place we ever modify `tail`, so it could not have changed between the time we read, modify, and write it.



ISR sees `head != tail` so it reads all three bytes from `queue[head]`. Interrupts are disabled, so this always works.

ISR then adds one to `head`, accounting for wrap. There is no read-modify-write problem, interrupts are disabled, and even so the ISR is the only place we ever modify `head`.

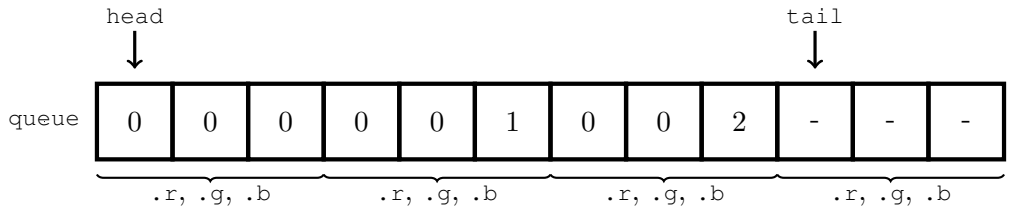


ISR can now do something cool with the RGB triplet it pulled off the queue (perhaps configure three PWM outputs to change the color of an LED).

Now we are back where we started with `head == tail`, except both `head` and `tail` are now one instead of zero. Rinse, lather, repeat.

In the tutorial case, the only real difference is the ISR does not fire as often, so we are still safe. Even so, lets look at what happens.

In the case where we are only producing and waiting for the queue to fill up, the ISR does not modify anything (since `enable_consumer` is zero) and main() will just do its thing, copying RGB triplets into the queue until it is full:



At which point, main() will notice that the queue is now full because, `head == (tail + 1) % NELEMS(queue)`, and it will spin until the queue is no longer full, setting `enable_consumer` to one over and over while it is spinning. Since `head`, `tail`, and `enable_consumer` are all `volatile` and `uint8_t`, this is fine.

Once `enable_consumer` is set to one by main() the ISR will notice, and it will start to consume again. When the ISR finally does fire and returns, it will have incremented `head`, so now the `while` loop inside main() will stop spinning and it will continue to produce.

This will work with any size structure, just make sure your queue length is 256 or less, so `head` and `tail` can be stored in a single byte. As a bonus, if your queue length is exactly 256, you don't even need to mod by `NELEMS(queue)` in `full()`, `enqueue()`, or `dequeue()`, since the wrapping will occur naturally due to overflow (though if you leave it there, the compiler should automatically optimize it out for you).

I hope this fully explains why this method works without the use of an `ATOMIC_BLOCK`. And since we never have to disable interrupts to maintain atomicity, we are guaranteed to have our interrupt fire at a perfectly regular interval.