Robust MCU firmware design

Last update: 26.03.2018

Ibrahim KAMAL
IKALOGIC S.A.S.
19 Rue Columbia
87000 Limoges
FRANCE

WARNING

Table of contents

Intro

We're in 2018 and the number of persons who think they know how to write micro-controller firmware is constantly growing. Let me get this straight: Writing a firmware that will work on a prototype is one problem, regardless of what this firmware is achieving. Ensuring this firmware will work on thousands of circuit boards and never endure sporadic system failure is a whole other story, and this is what this eBook is all about.

In this document, I won't be explaining the basics of programming, I'll even assume you already know how to code in C, and that you have already dug your way through several embedded firmwares.

This eBook will help you to organize your code to be stable and robust. It will show you how to design the firmware in such a way that allows it to grow without breaking up at some point. A firmware is something that is often meant to evolve and needs to be easily maintained.

It's also worth noting that I am presenting a proven method of designing robust firmware according to my personal experience during the past 15 years. I am not saying it's the best method nor the only one.

Finally, please note that I won't be using any real time OS in this eBook. Only "bare metal" programming. It's a choice that can be discussed, but I've always preferred to have full understanding and control over the code I'm writing in a microcontroller. I consider that being so close to the hardware is a privilege you don't have when working on non-embedded software.

Expect the unexpected

The first rule I'd like to present is this one: Expect the unexpected. One of the main reasons why embedded code (or code in general) fails at some point, is that the system may run with input values it was never designed for in the first place. Ok, I know that may seem vague, but bare with me, it will get clearer. Let's start with a simple function that does a simple mathematical division:

```
int my_function(int a, int b)
{
   return a/b;
}
```

Although this function is quite simple (returns the result of ‘a’ divided by ‘b’), it has major flaw. It assumes that b will never be equal to 0. Although this may never happen, especially in well defined test conditions, there is absolutely no guarantee that the input ‘b’ will never ever be equal to zero and lead to the famous “division by zero” crash.

The solution here is obvious, ensure ‘b’ is never zero, or prevent the division by zero in some other way:

```
int my_function(int a, int b)
{
   if (b ==0) return 0;
   return a/b;
}
```

Now, this example is quite simple, but you won’t believe how many times a software or a firmware will fail because of such simple flaws in the code.

The lesson here is to always ask yourself: “What if…?”. Never assume the inputs will always have normal, expected values. This is true with almost any function, and is even more dramatic when dealing with outside stimuli, like reading ADC values.

The exact solution is different for each every particular case, but always try to ensure that there are some limits on the value the input can take.

For example, if you’re regulating the speed of a DC motor and getting a speed reading of more than 10 000 RPM, you may have something dramatically wrong in your system, and you may want to stop regulation and enter a “fail safe” mode. Again, never assume you’ll never get a reading of 10 000 RPM because you know your motor can never reach this speed. There are plenty of reasons why you could get such a very high reading even if your motor is not moving: don’t forget that the speed is measured with a sensor of some kind, and like any sensor, intermittent failure, or noisy reading is something that may (and will) happen. I could go on and on about all the reasons why you could get a totally wrong (and physically impossible) readings, but that’s not the point. The point is: you can write intelligent code that can detect such situations and cope with it. That’s what makes a code “robust”.

Design for reuse

Although this is not directly related to the firmware being robust, a code that is written to be easily reused, will help you reach that goal eventually. Even better, if your code is well organized enough so you can easily integrate *proven* code snippets, functions or modules written by others, it will help you move faster and cut debugging time. Remember: a good

programmer will never reinvent the wheel and write code that already exists and is fully debugged (unless you want to achieve better performance for example)

Now, when you're writing a firmware, start by carefully grouping functions in a way that makes them independent from the rest of the program. In C programming, this translates to having a *.h (header) file and *.c file where all functions are implemented.

For example, if you're writing a firmware for a microcontroller and you need to access GPIO pins, a good idea would be to start by creating what i would call a "GPIO module". That's a *.h and *.c file that contains all the functions related to GPIOs, like:

- Initializing GPIO as inputs our outputs
- Setting up alternate function for a pin (like timer output, UART Rx line, etc)
- Setting up internal pull-up resistor
- Writing to an output pin
- Toggling an output pin
- Reading from an input pin

That GPIO module will take some time to be written, but as soon as it has proven to be functional, you'll be saving a lot of time in this project and next ones.

The same goes for other interfaces (like UART, SPI, I2C, etc), or functions (like FIFO blocks, or sorting algorithms).

Time critical VS non time critical code

When I intend to write a firmware, I always try to see the functions being executed in two categories: time critical and non time critical. This helps me to carefully design the architecture of my firmware and prioritize some routines over others, and some interrupts over other, less critical interrupts.

I know that many modern microcontrollers have sometimes more than 10 different priority levels, and this may be very useful in some cases, but to make things simpler, at least try to make use of two interrupt priority levels. In case you don't know, interrupt priority is what prevents a less important interrupt from stopping the execution of a more important (and time critical) interrupt.

Let's take this real life example: a microcontroller is reading some audio content from a flash memory, and streaming it to an audio amplifier. A time critical interrupt sends a new audio sample at a frequency of exactly 44.1 KHz, as per the requirement of the audio amplifier. Another interrupt is used to receive user commands sent via a UART interface. Now, do you want that slow, random, human solicited commands causes audible pops and click in the beautiful audio stream being played? Of course not! That's why in that particular scenario, we

would attribute a quite low priority to the UART receive interrupt and a high priority to the audio transfer interrupt. Actually, the audio interrupt may delay the UART from receiving a byte, but only by a few microseconds, which is frankly, not a problem at all.
Now, while we're speaking of interrupt, there is more to say about interrupt than priorities, and that's what we will discuss in the next chapter.

Interrupts and the danger zone

Once you start using interrupt, you feel that interrupts are pretty cool: they can allow your code to run much faster and more efficiently. The processor no longer needs to keep polling a UART interface to check if a new byte was received for example. That's good, but be careful, interrupts are double-edged swords!

By definition, interrupts will.. *Interrupt..* your code. Anytime, anywhere, and that's where the problem lies. Not knowing where your code will be interrupted requires careful design techniques to ensure that you never get unexpected behavior. And apart from being expected, it's very difficult to debug, because the bug may appear anytime, anywhere, and never in the same conditions. In other words, it's a nightmare to try to find this kind of flaw in a sophisticated firmware.

Now, the solution I adopt to avoid this problem is simple: only do the absolute minimum in an interrupt routine, and above all, follow those rules that apply to the code executed inside the interrupt routine:

- Never call a function from inside an interrupt
- Never access a hardware module (UART, SPI, etc) that may be accessed by any other part of the code.
- Try to limit the code in an interrupt to a very short and simple list of tasks, like raising a flag or incrementing a variable.

For example, when I write a timer interrupt service routine (ISR), it usually looks like this:

```
void timer_isr(void) //10 ms timer
{
  time++; //a global variable
  //add code here to rearm the interrupt
}
```

That code simply increments the global variable "time" every 10 ms. Then, I can use this variable to calculate different delays in the code. The beauty of this is that it can be used many times in many different functions, with different precision levels depending on your needs. For example, if you need to wait for 1 second between two event, you could write:

```
unsigned int time1 = time;
while ((time - time1) < 100); //wait for exactly one second, 100*10ms
//continue code execution.
```

The same thing applies when writing UART receive interrupt routine. Too many times I've seen a UART ISR that handles all high level decoding of the protocol being implemented, and even takes care of sending responses back whatever is listening on the other side. (I may have even done that in the past to be honest!). The right way to write a UART receiver ISR is something like this:

```
void uart_isr(void)
{
  if (new_uart_data == true)
  {
    return; //previous byte was not processed yet!
  }
  uart_data = UART_D0; //read data from hardware register
  new_uart_data = true;
}
```

Then in your main program loop you would write:

```
while (1)
{
  ... //other code
  If (new_uart_data)
  {
    ... // write the code here to interpret this data correctly
    ... // You may also want to add integrity check routines
    //when data is not needed any more, allow new bytes in:
    new_uart_data = false;
  }
  ... //other code
}
```

As you may notice, the ISR (interrupt service routine) is as simple as it can get. A "new_uart_data" flag prevents new bytes from overwriting previous unread ones.

The same safety principles applie to all interrupt routines you may write. The smaller the interrupt function, the safest your program is.

Designing the main loop

I don't want to be overdramatic but, I really see the main loop in an embedded software as the "beating heart" of the system. Each pass in the loop is like a heart beat, and you should try to keep it looping (beating) at a steady rate. That means that you should never use "delays" in the main loop (at least not the kind of delays that just freeze the execution of the rest of the code)

Also the main loop should be designed in a way that easily allows your code to grow, without becoming a total mess.

A clean main loop - at least, the way I design them - looks like this:

```
while (1)
{
  task_1();
  task_2();
  task_3();
  task_4();
  // etc...
}
```

Each "task" function represents a different function of the firmware. For example, if we're talking about a system with an LCD, a motor and keyboard, the main look would literally look like this:

```
while (1)
{
  motor_task();
  lcd_task();
  keyboard_task();
}
```

Each one of those task functions should be designed to do whatever needs to be done and return as quickly as possible. Never use delays (we'll get into implementing delays the right way later in the document).

Now the beauty of this is that it can be easily read and understood by anyone who has never seen the code before. It's also easy to maintain, because this architecture can grow at will. For example, if we need to add a task that takes care of controlling some LEDs, you would simply add one line to the main loop:

```
while (1)
```

```
{
  motor_task();
  lcd_task();
  keyboard_task();
  leds_task();   //New task added here
}
```

Following this rule makes your code clear, clean, readable and maintainable.

Now, what about delays?

Simple, let's see how it's done! Let's assume you have a LEDs task, that needs to execute once per second (you don't want the LEDs state to be updated too frequently).

Let's also assume you have a time interrupt that increments a variable called "led_task_time" as shown below (of course, this code is oversimplified and would probably need more hardware-dependent initialization):

```
void timer_isr(void) //10 ms timer
{
  If (led_task_time < 100) led_task_time++; //a global variable
}
```

Now, the LEDs task function would simply loop like this:

```
void leds_task(void)
{
  If (led_task_time >= 100)  // after 1 second (10ms x 100)
  {
    Led_task_time = 0;
    //Do what need to be done to the LEDs here
  }
}
```

So what? What makes this delay better? The answer is quite simple: this delay never ever "lock" the processor in a loop, waiting for the delay to elapse. Instead, if required delay didn't elapse, it will simply return, and let the main loop execute other tasks.

Actually, this way of coding is very simple kind of multi tasking. One where all tasks have the same priority.

Conclusion

I hope one or two of those advices will help you design better code. I am sure you also have nice tricks you have learnt over the years, so don't hesitate to mix and match to get the most out of your experience and mine.

One final advice, is to take your time. Always start by drawing (yes, drawing with boxes circles and arrows) your code. Try to have a clear view of your code as independent blocks, where each block can be tested, proven and reused independently.

About the author

Ibrahim KAMAL, CEO at Ikalogic.

Growing up in Egypt and France gave Ibrahim a deep sense of multiculturalism. After obtaining his Bsc of Mechatronics Engineering in Egypt (2007) and master degree in France (2009), he worked as CTO of an IOT Start-up before founding Ikalogic in 2010.

Ibrahim has always been fond of self-education. In his young age, his passion to programming and electronics drove him to learn as much programming languages as he could.

By creating Ikalogic, Ibrahim wants to empower engineers around the globe with tools that unleash their creativity.

Contact the author: i.kamal@ikalogic.com