How to design and implement firmware for embedded systems

Last changes: 17.06.2010

Author: Rico Möckel

BIOROB
EPFL Biorobotics Laboratory

The very beginning: What should I avoid when implementing firmware for embedded systems?

- Writing code for embedded systems is not the same as writing the user code for a PC (personal computer). It is more like writing a driver for a PC. If no special embedded operation system (OS) like TinyOS is used the firmware engineer has to take care of all basic things like setting up hardware registers himself. The code should be minimalistic, efficient, real-time, stable, easy to read etc.
- Your system is embedded meaning that it is typically connected to real hardware like motors, sensors, communication busses etc. The means that software errors can have dramatic and expensive results. Pins can get shorted in software leading to high currents and damage of the electronics. Batteries that should be monitored can be under- and overcharged and explode. Sensors can get destroyed when not being operated correctly. Gear boxes of motors can break without proper control. So test you system first in a safe environment using power supplies with selectable current limit and make sure that the pins of your microcontroller are correctly configured as inputs and outputs.
- Never ever do busy waiting with wile loops. Always use timers! Why: your busy waiting will keep the processor waiting in a busy state. It will not be able to do anything useful during that time. Remember: Embedded systems have to run in real time. There is typically no scheduler like when writing software for an operating system that automatically switches between different processes.
- Make yourself familiar with design strategies like interrupts and DMA (direct memory access). Interrupts will help you to avoid actively waiting for a condition like a change of an input signal but will instead allow you to react only when the change happens. DMA will free your processor from the dump work of just waiting to be ready to transfer some data from one memory to another.
- Never ever put a lot of code into an interrupt service routine. These call back functions are only there as signals and should be as short as possible. So if you need to do a lot of work after an interrupt: set a flag and get the work done in the main loop.
- Be aware of the fact that embedded systems have to face unknown time conditions. E.g. you never know when and in which precise order a user will use a switch or push button or you will receive some data through a UART interface. So be prepared for this.
- To make your code readable and reusable take care of coding standards and be consistent.
- Separate your code into different files and functions in a meaningful manner. Do not write spaghetti code. Very good C-code looks like C++. Make use of structs where useful. Do not make extensive use of functions where it is not necessary. Each call requires the current variables to be copied on the stack and after the call to be copied back. This is time consuming and can seriously decrease the speed of your code.
- Use pointers and buffers to generate efficient code. Avoid copying data.
- Make use of macros to design efficient and readable code. You can even implement simple functions with macros since macros take parameters.
- Document your code. Write the documentation is a way that you can use automatic extraction tools like Doxygen.
- It is always a good idea to use a subversion (SVN) system to keep track of the different versions of your code.

- Microcontrollers and their development environments typically come with debugging functionalities. Use them!
- If useful think about using some highly efficient ASEMBLER code where useful.

First step: design your firmware

Before even starting to implement any code think, plan and write a documentation of your code. Writing firmware for embedded systems is not a matter of trial and error but of thinking and coding. Without a good implementation strategy and a properly set up firmware architecture your code will be unreadable, difficult to update and debug and full of bugs and design errors. Nobody will be willing to reuse your code. And this simply means that your firmware was a waste of time.

So this is what you need to write down carefully:

- What is roughly the purpose of your firmware? E.g. I want to write the firmware that allows an RS232 interface to talk to an RS485 interface and vice versa.
- What are the challenges you are facing with regard to the purpose of your firmware? E.g. RS485 is a half-duplex point-to-multipoint communication interface where only one communication partner is allowed to send data at a time while RS232 is a full-duplex point-to-point communication interface. Problems I need to handle are:
  (1) How to ensure that only one communication partner is sending data at a time?
  (2) What to do with data that I received over RS232 that cannot be forwarded to the RS485 since the bus is currently used by someone else?
  (3) How to avoid deadlocks – meaning that the RS485 bus is blocked forever?
- How do you plan to deal with the challenges you are facing? Which strategies do you plan to follow? E.g.
  (1) I will implement a TDMA (time division multiple access) protocol with one master where only one communication partner can send data at a time. The master will ensure that only one communication partner will send data at a time by asking for the data. No slave is allowed to send data without being asked for it.
  (2) I will implement a ring buffer that can store the data that cannot be forwarded. Once the bus is free I will empty the buffer by sending the stored data.
  (3) I will use a timer that ensures that the RS485 bus is used again if a timeout occurs. If the answer of a slave is not received in time I will assume that there is a problem with this device and I will send an error message to the host.
- Make a detailed plan for the firmware architecture that you will be using. Which files do I need? Which peripheries of the microcontroller are used? How do I set up these peripheries correctly? Which functions need to be implemented? E.g. I need one source and header file for each periphery I am using. Furthermore, I need a main file as well as a configuration file where I am storing general configurations that I will need to change more often. Since I want to make my code as reusable as possible I will implement a special header and source file as well where I place configurations that are specific for my particular microcontroller like the configuration of IO ports. There will be initialization functions for each of the peripheries that are called from the main init function in the main file. I will have to set up the system clock correctly. I plan to use the full speed of 40MIPS (mega instructions per second). That is why I need to set up the PLL (phase locked loop). I will need to UARTs (universal asynchronous receiver transmitter). I do not plan to use the ADC. So I will ensure that this is switched off. I will need some GPIOs (general purpose input output) for debugging and to talk to the LEDs. And so on. Make drawings!

After and only after all these things are planned and carefully written down we start writing the implementation.

Second step: implementation of your firmware

After your documentation is ready you may start the implementation. Remember: think first and do not just load your code into the microcontroller and try some stuff. You are working with an embedded system.

1. Set up your framework of (empty) header and source files. Commit your files to the SVN system.
2. Set up the clock of your microcontroller as well as the other basic configuration registers.
3. Create the main() and the main_init() functions.
4. Properly set up the pins of your microcontroller. Make sure all pins are correctly configured as inputs and outputs. Create macros to talk to your pins. It is much easier to set a GPIO port to one by writing

   ```
   PORT_LED = 1;
   ```

   then by writing

   ```
   (PORTA << 9) & 1;
   ```
5. You will need a timer. Set this up correctly.
6. Now it would be a good idea to test your timer and clock by driving a GPIO port with that timer and carefully measuring the port with an oscilloscope. Make sure that the period is correct and that there is not jitter. Once this is working it is a good time to commit your files to the SVN system.
7. You may now want to set up a UART to have some extra debugging possibility. Try to send and receive data with the UART.
8. After all these things are working correctly you may start implementing your specific firmware. Do not forget to commit your files to the SVN system.