MCU Programming: Bootloader

Khanh J. Phan

Last Updated: December 11, 2014

A bootloader is a program that runs in the microcontroller to be programmed. It receives new program information externally via some communication means and writes that information to the program memory of the processor.

This is in contrast with the normal way of getting the program into the microcontroller, which is via special hardware built into the micro for that purpose. On PICs, this is a SPI-like interface. If I remember right, AVRs use Jtag, or at least some of them do. Either way, this requires some external hardware that wiggles the programming pins just right to write the information into the program memory. The HEX file describing the program memory contents originates on a general purpose computer, so this hardware connects to the computer on one side and the special programming pins of the micro on the other. My company makes PIC programmers among other things as a sideline, so I am quite familiar with this process on PICs.

The important point of external programming via specialized hardware is that it works regardless of the existing contents of program memory. Microcontrollers start out with program memory erased or in a unknown state, so external programming is the only means to get the first program into a micro.

If you are sure about the program you want to load into your product and your volumes are high enough, you can have the manufacturer or a distributor program chips for you. The chip gets soldered to the board like any other chip, and the unit is ready to go. This can be appropriate for something like a toy, for example. Once the firmware is done, it's pretty much done, and it will be produced in large volumes.

If your volumes are lower, or more importantly, you expect ongoing firmware development and bug fixes, you don't want to buy pre-programmed chips. In this case blank chips are mounted on the board and the firmware has to get loaded onto the chip as part of the production process. In that case the hardware programming lines have to be made available somehow. This can be via a explicit connector, or pogo pin pads if you're willing to create a production test fixture. Often such products have to be tested and maybe calibrated anyway, so the additional cost of writing the program to the processor is usually minimal. Sometimes when small processors are used a special production test firmware is first loaded into the processor. This is used to facilitate testing and calibrating the unit, then the real firmware is loaded after the hardware is known to be good. In this case there are some circuit design considerations to allow access to the programming lines sufficiently for the programming process to work but to also not inconvenience the circuit too much. For more details on this, see my in-circuit programming writeup.

So far so good, and no bootloader is needed. However, consider a product with relatively complex firmware that you want field upgradable or even allow the end customer to upgrade. You can't expect the end customer to have a programmer gadget, or know how to use one properly even if you provided one. Actually one of my customers does this. If you buy their special field customizing option, you get one of my programmers with the product.

However, in most cases you just want the customer to run a program on a PC and have the firmware magically updated. This is where a bootloader comes in, especially if your product already has a communications port that can easily interface with a PC, like USB, RS-232, or ethernet. The customer runs a PC program which talks to the bootloader already in the micro. This sends the new binary to the bootloader, which writes it to program memory and then causes the new code to be run.

Sounds simple, but it's not, at least not if you want this process to be robust. What if a communication error happens and the new firmware is corrupt by the time it arrives at the bootloader? What if power gets interrupted during the boot process? What if the bootloader has a bug and craps on itself?

A simplistic scenario is that the bootloader always runs from reset. It tries to communicate with the host. If the host responds, then it either tells the bootloader it has nothing new, or sends it new code. As the new code arrives, the old code is overwritten. You always include a checksum with uploaded code, so the bootloader can tell if the new app is intact. If not, it stays in the bootloader constantly requesting a upload until something with a valid checksum gets loaded into memory. This might be acceptable for a device that is always connected and possibly where a background task is run on the host that responds to bootloader requests. This scheme is no good for units that are largely autonomous and only occasionally connect to a host computer.

Usually the simple bootloader as described above is not acceptable since there is no fail safe. If a new app image is not received intact, you want the device to continue on running the old image, not to be dead until a successful upload is performed. For this reason, usually there are actually two special modules in the firmware, a uploader and a bootloader. The uploader is part of the main app. As part of regular communications with the host, a new app image can be uploaded. This requires separate memory from the main app image, like a external EEPROM or use a larger processor so half the program memory space can be allocated to storing the new app image. The uploader just writes the received new app image somewhere, but does not run it. When the processor is reset, which could happen on command from the host after a upload, the bootloader runs. This is now a totally self-contained program that does not need external communication capability. It compares the current and uploaded app versions, checks their checksums, and copies the new image onto the app area if the versions differ and the new image checksum checks. If the new image is corrupt, it simply runs the old app as before.

I've done a lot of bootloaders, and no two are the same. There is no general purpose bootloader, despite what some of the microcontroller companies want you to believe. Every device has its own requirements and special circumstances in dealing with the host. Here are just some of the bootloader and sometimes uploader configurations I've used:

1. Basic bootloader. This device had a serial line and would be connected to

a host and turned on as needed. The bootloader ran from reset and sent a few upload request responses to the host. If the upload program was running, it would respond and send a new app image. If it didn't respond within 500 ms, the bootloader would give up and run the existing app. To update firmware therefore, you had to run the updater app on the host first, then connect and power on the device.

2. Program memory uploader. Here we used the next size up PIC that had twice as much program memory. The program memory was roughly divided into 49% main app, 49% new app image, and 2% bootloader. The bootloader would run from reset and copy the new app image onto the current app image under the right conditions.

3. External EEPROM image. Like #2 except that a external EEPROM was used to store the new app image. In this case the processor with more memory would have also been physically bigger and in a different sub-family that didn't have the mix of peripherals we needed.

4. TCP bootloader. This was the most complex of them all. A large PIC 18F was used. The last 1/4 of memory or so held the bootloader, which had its own complete copy of a TCP network stack. The bootloader ran from reset and tried to connect to a special upload server at a known port at a previously configured IP address. This was for large installations where there was always a dedicated server machine for the whole system. Each small device would check in with the upload server after reset and would be given a new app copy as appropriate. The bootloader would overwrite the existing app with the new copy, but only run it if the checksum checked. If not, it would go back to the upload server and try again.

Since the bootloader was itself a complicated piece of code containing a full TCP network stack, it had to be field upgradeable too. They way we did that was to have the upload server feed it a special app whose only purpose was to overwrite the bootloader once it got executed, then reset the machine so that the new bootloader would run, which would cause the upload server to send the latest main app image. Technically a power glitch during the few milliseconds it took the special app to copy a new image over the bootloader would be a unrecoverable failure. In practise this never happened. We were OK with the very unlikely chance of that since these devices were parts of large installations where there already were people who would do maintainance on the system, which occasionally meant replacing the embedded devices for other reasons anyway.

Hopefully you can see that there are a number of other possibilities, each with its own tradeoffs of risk, speed, cost, ease of use, downtime, etc.