7 tips for designing a robust Bootloader

Jacob Beningo  - April 23, 2015

Embedded software is never finished. The software always requires some tweak, adjustment, a feature addition or bug fix. It is just the way the world works. A product could be sent back to the manufacturer for updates or a technician could be sent to the product. Either way it is expensive! Some products are built in such a way that accessing the microcontroller's programming port would require a complete tear down of the system. A bootloader is the perfect solution for either situation.

Bootloaders have become a necessity in nearly every embedded system but unfortunately they are given little to no attention. The ability to update firmware is an afterthought rather than forethought for most developers. The result of this oversight is a last minute ditch effort to build any update functionality into the system. Bugs creep into the system and due to a lack of testing undoubtedly find their way into the field. There are 7 tips that can be followed to ensure that a bootloader doesn’t just update firmware but does so correctly every time.

Tip 1 - Add a bootloader early

Start on the bootloader early in the design, preferably after the proof of concept stage when the project becomes officially funded. Integrating the bootloader into the system early not only allows thorough testing but also puts a limit on the changes to the application that may be necessary during integration.

Tip 2 - Check the reset vector

One of the purposes of a bootloader is to get the system to a known state and make a decision to either jump to the application or wait for programming instructions. In the event that the bootloader decides that the application should be started, the jump to the application vector should not be a blind one. The system flash could have previously been erased, leaving the reset vector with a unique location of 0xFFFFFFFF. What will happen to the system if this reset vector is executed?

The simplest check that can be performed on an application prior to starting it is to make sure the reset vector has been programmed to a value, i.e., it's not erased. A programmed reset vector gives a minimum level of confidence that the program won’t jump off into the weeds. An example can be seen below in Figure 1. A developer may even want to get fancier and provide a bounds check on possible valid reset vectors. A boundary check would help ensure that not only has the reset vector been programmed but it has been pointed to a valid location.
Tip 3 - Perform a CRC check

A reset vector check is a great start toward validating that an application is present, but the reset vector alone doesn't tell the whole story. What would happen if the firmware update were aborted part way through? The reset vector could have been written to flash but the application itself corrupted. Adding a CRC check to the application is a great way to validate the application. The bootloader can perform a quick CRC check of the application space at start-up and if the check succeeds the bootloader can assume that the application space is intact.

Tip 4 - Lock the bootloader

Updating the bootloader in the field is an extremely dangerous endeavor. A power glitch, stray cosmic ray, or a dirty look could cause the bootloader section of memory to become corrupted and unusable. The result is an embedded system that is bricked until it can be rescued by a more sophisticated programming tool. A bootloader should be a small, simple, and robust application that doesn’t require feature upgrades or bug fixes. The bootloader should also be placed in protected memory sections in order to ensure that it cannot be accidentally be erased or overwritten in any way, shape, or form.

Tip 5 – Break flash into multiple application slots

Embedded systems sometimes have complex requirements such as updating the firmware while the application is still running or being able to revert to a previous version in the event of a failed update. These requirements would be difficult to implement without breaking the flash space up into multiple application segments. For example, flash space could be broken up to include the bootloader and the original firmware in protected memory and then two spare application sections for updated firmware. The bootloader would program one of the spare slots and if success could then switch to using that application. In the event of a failure the bootloader could always fall back to the original or the application stored in the backup slot.

Tip 6 - Use a hand shaking protocol

The compiler will generate a representation of the application that is broken down into records. These records can be generated in a variety of formats but one common feature is that each record represents a smaller chunk of the application and contains a checksum. The developer can create a protocol that sends one record at a time, verifies the record, writes it to flash, and then responds to the update tool with a confirmation. Once the tool receives the confirmation it can send the next record. Using a handshake breaks the flashing process up into manageable chunks and in the event of an error the system can handle the error immediately rather than at the end of the update cycle.
Tip 7 - Use a delta generator

A traditional microcontroller bootloader will completely erase the application flash space during the update process. This approach can waste time updating firmware if there were only minor changes to the code. An approach that is often taken in larger computer systems instead is to determine the delta or difference between application files and then only update those files or memory sections that have changed. Delta generators can also be used to minimize the time required to update a microcontroller application.

These seven tips are a good start, but to learn more about bootloaders attend my "Bootloader Design Techniques for MCUs" on May 7th at ESC Boston. You can read the associated white paper at http://bit.ly/1yv6gnI.

Jacob Beningo is a Certified Software Development Professional (CSDP) whose expertise is in embedded software. He works with companies to decrease costs and time to market while maintaining a quality and robust product. Feel free to contact him at jacob@beningo.com, at his website www.beningo.com, and sign-up for his monthly Embedded Bytes Newsletter here.

Join over 2,000 technical professionals and embedded systems hardware, software, and firmware developers at ESC Boston May 6-7, 2015 and learn about the latest techniques and tips for reducing time, cost, and complexity in the embedded development process.

Passes for the ESC Boston 2015 Technical Conference are available at the conference’s official site with discounted advance pricing until May 1, 2015. The Embedded Systems Conference and EDN are owned by UBM Canon.