Bootloader Design for Microcontrollers in Embedded Systems

By Jacob Beningo
www.beningo.com
http://www.linkedin.com/in/jacobbeningo
twitter : Jacob_Beningo

EDN Blog - Embedded Basics:
http://www.edn.com/electronics-blogs/4375394/Embedded-Basics

Table of Contents

Introduction

Microcontrollers have proliferated into every nook and cranny of our daily lives from simple 8-bit devices that control our toaster ovens to powerful 32-bit DSP's that provide us with the rich media and entertainment that we have all become accustomed to. Without microcontrollers, our lives would not only be less exciting but we would lose a level of control over our world that we can no longer live without. Billions of microcontrollers are sold each year with the number continually climbing.

What happens to these microcontroller based products when millions of units have shipped and a software "enhancement" needs to be made? Does every unit need to be returned to the manufacturer every time the software is updated? Are televisions, blue-ray players and other devices returned to the manufacturer periodically so that the customer can continue to have the latest and greatest software operating on their device? The obvious answer to these questions is absolutely not and the primary reason why is that most systems ship with a boot-loader on-board.

A boot-loader is an application whose primary purpose is to allow a systems software to be updated without the use of specialized hardware such as a JTAG programmer. In certain cases, it can also be the earliest point at which the integrity of an embedded system can be checked. The boot-loader manages the systems images. There are many different sizes and flavors to embedded boot-loaders. They can communicate over a variety of protocols such as USART, CAN, I2C, Ethernet, USB and the list goes on for as many protocols that exist. Systems with boot-loaders have at least two program images coexisting on the same micro-controller and must include branch code that performs a check to see if an attempt to update software is in progress.

During the initial stages of product development it is very common for the boot-loader to be overlooked by the development team. The primary reason for this is that the boot-loader is not the primary end product that is going to be sold to the customer but the boot-loader is potentially the most important part of that product. The boot-loader allows a company to launch their product with software that only fulfills a portion of their final feature set and then add features to their product once it has been launched into the market. It also allows them to fix bugs that are discovered after the system has been released into the wild.

For an embedded software engineer, a boot-loader requires a full understanding of how a processor works, how to utilize its memory and how to work on the processor at the lowest levels. Boot-loader development can be an extremely challenging endeavor to undertake but absolutely a rewarding one. Once a developer has gone through the process, each additional boot-loader becomes easier and easier to implement by following a common and consistent approach to boot-loader design.

The purpose of this paper is to break down the components necessary to develop a boot-loader and present it in an easy to understand methodology that the reader can then use to implement their own boot-loader. While boot-loaders are small programs, they often bring out the worst and nastiest bugs that can be very intimidating to a developer. Not only does the programmer have to dig into flash programming but there is also a need to do a deep dive into memory maps, re-locatable vector tables, copy down functions, flash partitioning, code branching and a number of other tasks which can at first seem monumental and nearly impossible to overcome on a short development schedule.

Boot-loader Requirements

Before any software project is undertaken, it is always recommended that at a minimum some informal process of requirements gathering be performed. Requirements dictate to the developer what it is that they are going to be designing. Requirements can serve as a roadmap in the design process. The IEEE Computer Societies Software Body of Knowledge (SWEBOK) defines a software requirement as "a property which must be exhibited in order to solve a specific problem" (Society, 2004). The requirements tell the software developer key properties that the software must exhibit in order to present an appropriate solution.

There is no such thing as a complete list of project requirements from a customer. Project requirements tend to be fluid and often change throughout the design process. Modern day software processes such as Agile take these changes into effect and break the design process up into iterations of a fixed period of time. At the start of each iteration, the requirements are reviewed with the customer, allowing updates and changes to be made in addition to highlighting the critical path for the iteration. The key components of successful requirements gathering and execution include the following:

- Requirements Elicitation
- Requirements Analysis
- Requirements Specification
- Requirements Validation and Traceability

There are many methods that can be used to track software requirements. Software Requirements Specifications (SRS) are recommended as best practices by the IEEE Computer Society and CMMI. The SRS can be put together at the start of the project and revisited during each iteration of the product development cycle and updated in order maintain the latest requirements. Other formal methods exist such as the use of Rally, Wrike or other online tools to electronically track requirements.

Each boot-loader will have its own unique set of requirements based on the type of application; however, there are still a few general requirements that are common to all boot-loaders. These requirements can be broken down into seven fundamental groups of requirements that are common to the boot-loader. They are

1) Ability to switch or select the operating mode (Application or boot-loader)
2) Communication interface requirements (USB, CAN, I2C, USART, etc)
3) Record parsing requirement (S-Record, hex, intel, toeff, etc)
4) Flash system requirements (erase, write, read, location)
5) EEPROM requirements (partition, erase, read, write)
6) Application checksum (verifying the app is not corrupt)
7) Code Security (Protecting the boot-loader and the application)

Each of these categories can be broken down to generate a list of boot-loader requirements for any application. A broken down list can be found in Appendix A.

The Boot-loader System

Boot-loaders can come in many different sizes and in many different flavors but in general the operation of a system with a boot-loader is relatively standard. There are three major components to these systems that can be seen in Figure 1. They are the branching code (green), the application code (blue) and the boot-loader code (red). For most systems we prefer them to be executing the application path the majority of their operational life. Figure 1 highlights the execution path to get to the application by the dashed red line.
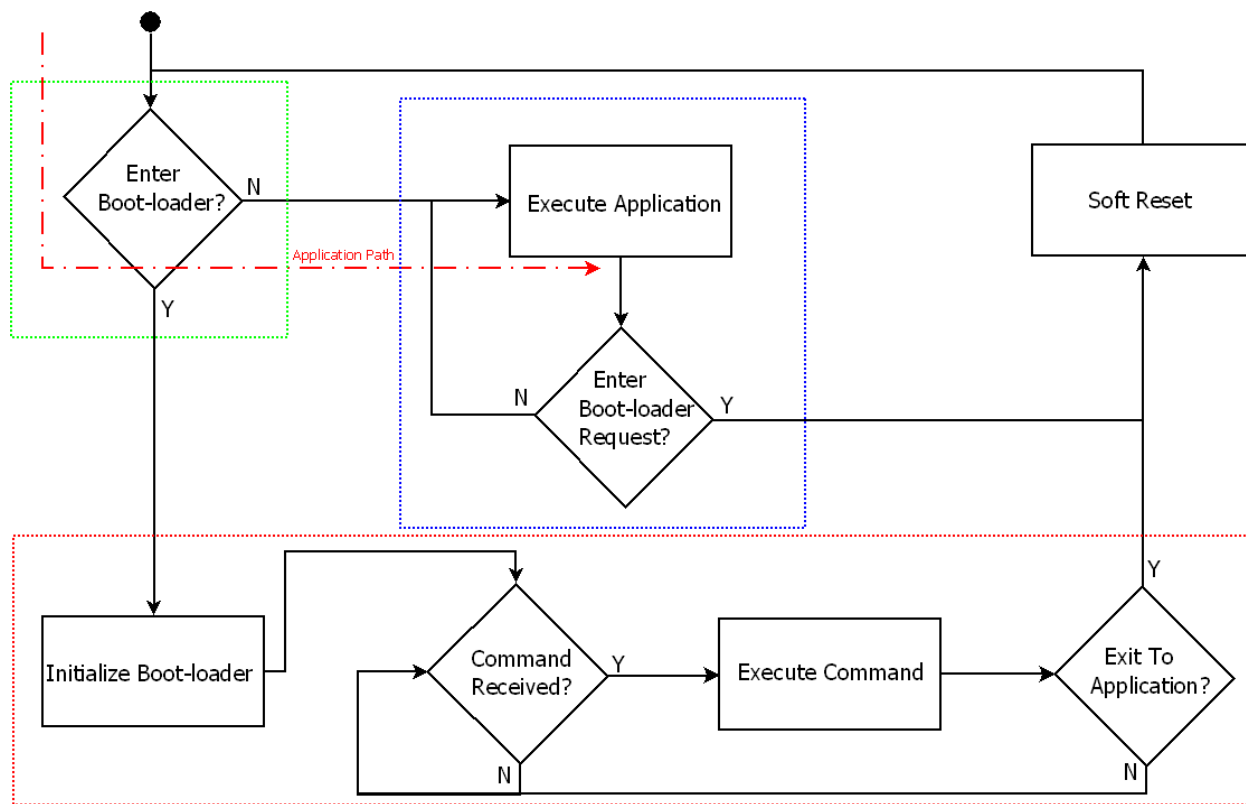
Figure 1 – General Boot-loader Operation

The branching code (green) makes the decision as to whether the boot-loader is loaded or whether the application is loaded. The branching code in a very simple system can be nothing more than checking a GPIO for a certain state to enter the boot-loader; however, such basic systems are usually only used by chip manufacturers in order to demonstrate the functionality of the chip and is usually not recommended if a robust boot-loading solution is required. In a more complex system, the boot-loader may completely load itself into memory and perform some basic system functions for a short period of time before making the branching decision possibly to perform a system integrity check. For this reason, in most cases the branching code is included in the boot-loader. A more detailed discussion of what checks should be performed before branching to the application will be discussed later in this paper.

The application code (blue) is executed only after the branching code has made the decision that there are no requests to enter the boot-loader and that the application code is safe to execute.

Even while the application is loading and performing its duties, it should be designed so that it can accept a command to enter the boot-loader. When the application receives such a request the application will perform any cleanup that needs to be performed and when it is safe to do so the application will perform a soft reset of the system. In most cases this is done by writing incorrect values to the watchdog timer that in turn will then cause a power-on-reset of the system. Additional details of how to make the branching code recognize that there has been a request to enter the boot-loader will be discussed later in the paper.

The boot-loader code (red) performs all of the functions of the boot-loader. When the boot-loader is loaded into memory it will begin initializing a basic subset of peripherals that are required in order for the boot-loader to perform all of its functions. These peripherals are typically the system clock, interrupt service routines and tables, a communication peripheral and perhaps a state-machine or basic task scheduler. This allows the boot-loader to communicate with the outside world and accept commands to perform flashing instructions. A detailed examination of how the boot-loader does this will be examined in the rest of this paper.

Boot-loader Behavior

A boot-loader in itself is not that different from a standard application; in fact, it is a standard application. What makes a boot-loader special is that it is sharing flash space with another application and has the capability to erase and program a new application in its place. Like every other application, one of the first priorities of the boot-loader is to initialize the processor and the minimum number of peripherals required to carry out boot-loading functions. It's best to keep the use of peripherals to as few as possible in the boot-loader in order to attempt to maximize the flash space that will be available for the application code.

There are two behavioral models that describe how a boot-loader can behave. In the first model, the boot-loading process is completely automated and self-contained within the system. An example of this would be an SD card boot-loader. The boot-loader would automatically detect the new firmware and manage its own flashing process. Commands from an external source would not be required to successfully carry out the boot-loading process.

In the second model, the system does not automatically handle the boot-loading process itself. Instead, the boot-loader initializes into an idle state and awaits instructions from an outside source. This source would typically be a pc based software application that commands the boot-loader into the different states necessary to flash a new image onto the system. The primary reason for the external software application to command the process is that in most applications without an SD card, there is not sufficient space to retrieve the whole software image. Instead, an external source with the image acts as the master of the boot-loading process. This paper will focus on the later model.

Each boot-loader will be unique in the number and type of commands that are supported. A sophisticated boot-loader will have more commands than a simple boot-loader. Flash usage for the boot-loader increases with each command that is supported potentially resulting in less space for the application image. At a minimum the boot-loader should support three commands.

1) Erase the flash – Removal of the application image from memory
2) Write flash – Addition of a new application image to memory
3) Exit / Restart – reboot with the intention of entering the application code

With these three commands all basic functions of writing an image to flash can be accomplished. There are more sophisticated commands that can be implemented that would greatly enhance the capabilities of the boot-loader. A few examples are

1) Unlock the unit – enter a security key to allow flash to be erased and then written
2) Erase EEPROM – erase configuration data stored on the system
3) Write EEPROM – write configuration data to the system
4) Read EEPROM - verify configuration data from the system
5) Read flash – verify that the application image is correct
6) Image Checksum – calculate an image checksum that can be checked for corrupted application
7) Lock the unit – enter security mode that prevents flash from being written

Locking and unlocking the system provides added robustness and security to the system. It prevents reading the application image out of the system in addition to accidental erasing or writing of the flash. EEPROM functions can be useful in the boot-loader particularly during the end-of-line of the manufacturing process. The boot-loader can act as a special mode in which to program the systems configuration data during the initial start-up of the system. The image checksum adds robustness to the system by allowing the system to verify that nothing has happened to the application image and that it is still completely intact.

The typical sequence for programming a device can be found below:
1) Open image flashing tool (if supported)
2) Start the boot-loader
3) Erase the flash
4) Send binary file information to the boot-loader (to be discussed later)
5) Generate Checksum
6) Quit the boot-loader and enter the application

The final programming tool used to flash a new application image with the boot-loader would be completely automated; however, for the software engineers that develop a boot-loader, a tool that allows each step in the process to be monitored and tested is required.

Application Behavior

The behavior of the application image is of for the most part not of any interest to the boot-loader designer except in one aspect; the application needs to be capable of receiving a command to enter the boot-loader. This means that the application needs to have two boot-loader like capabilities:

1) Set a piece of information that the boot-loader can detect to enter boot-load mode
2) Reset the system to initiate a branching decision

The best place for the application to store a value that can be detected by the branching code is an EEPROM configuration value. EEPROM in most cases exists in a memory space that can be shared by both the boot-loader and the application. When the application receives a request to enter the boot-loader, the application can write a value to EEPROM and then perform the second function which is to reset the system.

The reset of the system is performed by entering an infinite loop, allowing the watchdog timer to reset the system or by purposefully writing incorrect values to the watchdog register to force a software reset. The preferred method is to write incorrect values to the watchdog register because it guarantees that the system will be restarted immediately and minimizes delays.

Start-Up Branching

When the system starts up, there are at least two different software images that can be loaded and executed by the micro-controller, the boot-loader, the application and possibly a back-up application image. It is therefore necessary that as part of the boot-loader image code a branching algorithm be included that can handle the decision making process of which image to load.

There are many different methods that can be used to decide which image to load. The simplest method that can be used and is most often used in example code from chip manufacturers is the use of a GPIO line to make the decision. For example, if the GPIO signal is high, load the application; if it is low then load the boot-loader. This method is very simple and is often implemented using assembly language in order to quickly make the decision and jump to the image. Listing 1 shows a simple example for the Freescale S12X. In this example, if the logic signal checked is grounded, the top of the stack is loaded and program flow jumps to the boot-loader main function. If the logic signal is high, the application reset vector is loaded and then executed.

Closer examination of Listing 1 will reveal that there is a potential problem. What happens if the boot-loader has been loaded onto the micro-controller but an application has not yet been loaded and there is a request to enter the application? The code in Listing 1 will jump to the application anyways and jump right off into the weeds. This leads to one of the most important code branching checks; verification that the application reset vector exists.

```
; ------------------------------------------------------------------------

  brclr $0259, $01, GoBoot   ; if PP0 == 0 then start the boot-loader
                             ; if PP0 == 1 then start the application

  ldd   AppResetVect         ; Load the Application Reset Vector
  ldx   AppResetVect
  jmp   0,x                  ; jump to the application

GoBoot:
  lds  #StackTop
  jmp  main                  ; jump to the boot-loader

; ------------------------------------------------------------------------
                               Listing 1
```

When a microcontroller flash section has been erased it will be erased to all 1's. This means that if the reset vector is all 1's, we know that this reset vector is not valid and that the application has not yet been programed. By performing an extra check on the application reset vector the programmer can prevent the code from branching to a non-existent application. Instead the boot-loader would execute and wait for programming instructions. Listing 2 is an updated version of Listing 1 that includes the extra check of the application reset vector.

```
; ------------------------------------------------------------------------

  brclr $0259, $01, GoBoot   ; if PP0 == 0 then start the boot-loader
                             ; if PP0 == 1 then start the application

  ldd   AppResetVect         ; Load the Application Reset Vector
  cpd   #$ffff               ; Compare it to 0xFFFF
  beq   _GoBoot              ; if the application reset vector is not
                             ; available then start the bootloader
  ldx   AppResetVect
  jmp   0,x                  ; jump to the application

_GoBoot:
  lds  #StackTop
  jmp  main                  ; Continue Boot-loader startup

; ------------------------------------------------------------------------
                               Listing 2
```

A single I/O line being used to branch to the boot-loader is not a very robust solution. A user could accidentally push the boot-load button or a noise event could cause the system to enter the boot-loader. If the boot-loader is entered, there is no guarantee that an image loading tool is attached. In most cases you will not find a GPIO triggered boot-loader implemented on a production product. Instead, a common method used to detect a request to enter the boot-loader is to change an EEPROM value.

Nearly every embedded system stores configuration values in some type of EEPROM whether its on-chip, off-chip or part of some emulated EEPROM in flash. Storing a byte or a word configuration value for boot status is then a natural place to store which mode the system should

boot into. If the micro-controller supports on-chip EEPROM, the branching code from Listing 2 can evolve into Listing 3. In Listing 3, the GPIO check has been removed and instead the branch first checks that an application reset vector exists and if it does then it checks the EEPROM for which mode to jump to. If an ASCII character 'B' is stored then load the boot-loader. If it is any other value than the application code will be loaded.

Every microcontroller does not necessarily have internal EEPROM. Sometimes a system has EEPROM externally such as on a Serial Peripheral Interface (SPI) or Inter-Integrated Circuit (I2C). In these types of systems, the software developer won't be able to quickly check the EEPROM byte but will instead need to further initialize the system. In this case, there is an advantage to building the branching code into the boot-loader.

By integrating the branching code into the boot-loader there are a number of more sophisticated checks that can be performed by the branching code. The first of which is an application checksum. As part of the boot-loading process, the boot-loader can perform an image checksum on the application when it is written to flash. Then when the system starts, the branching code can calculate the flash image in order to verify that the application is complete and has not been corrupted before jumping to the application and running code.

```
; ------------------------------------------------------------------------

  ldd   AppResetVect         ; Load the Application Reset Vector
  cpd   #$ffff               ; Compare it to 0xFFFF
  beq   _GoBoot              ; if the application reset vector is not
                             ; available then start the bootloader

  ldd   EepromProgStatus     ; Read the programmed status byte from eeprom
  cpd   #'B'                 ; Compare it to 'B' for boot-load
  beq   _GoBoot              ; if Status == 'B' for Boot-loader then jump to
                             ; boot-loader, otherwise continue to the
                             ; application

  ldx   AppResetVect
  jmp   0,x                  ; jump to the application

_GoBoot:
  lds  #StackTop
  jmp  main                  ; Continue Boot-loader startup

; ------------------------------------------------------------------
                    Listing 3 – EEPROM Branching Code
```

A final check that can be performed by the branching code is a back-door access of sorts. The branching code can look for a programming tool that is attached. The branching code sends out a message to the programming tool that if the programming tool is present will respond with an acknowledgement. If the acknowledgement is received then the system will load the boot-loader instead of loading the application code.

Listing 4 shows a robust solution to the branching code. The code will calculate the application image checksum. When it is completed, the branching code will perform each of the following checks:

1) Does the application reset vector exist?
2) Is the Mode set to 'B'?
3) Is a programming tool present?
4) Is the application checksum valid?

```
; --------------------------------------------------------------------------
   //  When the checksum has completed and the timer has expired for waiting   //
   for a programming tool to respond, perform the branch checks.
   if((Checksum_Complete == TRUE) && (StartUpTmr == EXPIRED))
   {
        if((*ResetVector != 0xFFFF)     &&  // Does app reset vector exist?
           (Mode != 'B')                &&  // EEPROM byte set?
           (Boot_ToolPresent != TRUE)   &&  // Tool present?
           (Checksum_Valid != FALSE))       // Checksum valid?
        {
            App_LoadImage();
        }
        else
        {
            Boot_LoadImage();
        }
   }
; --------------------------------------------------------------------------
                    Listing 4 – EEPROM Branching Code
```

Memory Partitioning

Every microcontroller has some form of non-volatile memory that is used to store the program. The most commonly used type of memory is flash. Flash is broken up into divisible sections. The smallest section of flash is often referred to as a page. Pages are organized into larger structures known as sectors. Sectors are in turn organized into larger structures known as blocks. Each microprocessor is different as to how these sections of flash can be manipulated. Most will allow you to write a single byte to flash at a time.   Others may require that 8 bytes or 256 bytes be written at one time. In most cases, the smallest section of flash that can be erased at a time is a single sector that often consists of 4kB.

It is important that before a designer gets too far in their boot-loader design that they pull out the microcontroller datasheet and read through the flash and memory organization chapters. In fact, it is recommend that the designer read it, go to lunch and then read it again. It would be a good idea to read it the next day as well. The primary reason for studying the memory sections of the manual in such detail is that while memory mapping seems straightforward and easy to understand for some reason it always becomes a struggle when implementing the details. It is good for the software developer to get an expert understanding of how the microcontroller memory is organized and how it functions. It won't be long before the developer will need to manipulate the flash memory in order to burn a new application image to it.

The primary purpose for detailed examination of the memory map is to determine what sections of flash are available and best used for the boot-loader. A software developer need to examine the memory map and decide where in flash the boot-loader is going to be located and where in flash the application is located. These two flash areas need to be completely separate from

each other. The boot-loader must understand the existence of the flash memory map so that it can write an application to flash. On the other hand, the application should never know that the boot-loader section exists. That way, there is not the possibility of the application trying to over-write the boot-loaders program area. This section of flash can usually be protected in a number of ways but in general the two code areas need to be kept separate.

Each microcontroller manufacturer organizes the memory map in a different manner in addition to offering different features that can be utilized by the boot-loader. This can be seen in Figures 2 and 3. Figure 2 shows part of the memory map from a TI TMS320F28035 with 64k of flash broken up into 8 sectors of 8k of flash each. Figure 3 shows part of the memory map from a Freescale S12X with up to 1 MB of flash broken up into 4K sectors.
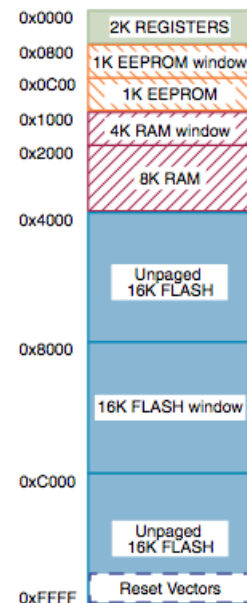

Figure 2

Figure 3

There are a number of factors that should be taken into account when selecting where to locate the boot-loader.

1) Size of the boot-loader
2) Location of the vector tables
3) Write protection and code security flash sections

Program space is always at a premium. From a manufacturing standpoint the preference is usually to keep the flash size as small as possible in order to save a few cents per unit. This means that we want to keep the boot-loader as small as possible in order to provide the maximum amount of flash space to the application. After all, the developer wants the application to be running and doing all those cool and nifty things not the boot-loader. So how much space should be allocated for the boot-loader?

At a minimum, allocate the smallest erasable block of flash that the processor can handle. This may be 1k, 4k or even 8k! A good starting rule of thumb is to allocate 4kB of flash space for the boot-loader. Odds are the developer will not come close to filling it but for the first pass allocate 4kB and once everything is working the memory map can be tweaked. The developer should not forget that there is a minimum size that can be erased. For the S12X the minimum flash size that can be erased at a time is 256 bytes. That is easily manageable. However, for the C2000, the minimum size of flash is an entire 8kB sector!

Another important aspect of the flash to look at is any section specifically designed for a boot-loader? Many processors today have space allocated for the sole purpose of putting a boot-loader there. Special flash protection can often be enabled in these sections so that you can guarantee that the boot-loader won't be erased or over-written by accident.   Read that chapter again on memory organization and while you are at it read the one on flash memory. Note anything special about the memory map. For example, the C2000 has One Time Programmable sections and secure code sections while the S12X has only secure code sections. The C2000 OTP section is only 1kB so if the boot-loader is small enough you will want to take advantage of this section.   You will not be able to use OTP during development but once the boot-loader works, is tested and validated the linker can be adjusted to use OTP.

Once the flash area(s) has been selected, it is important to update the linker for the application to exclude this area of memory. It is critical to add really good comments on this area of memory and why this range was selected. When maintenance is being performed a year later and someone is looking for more flash space, if the code is documented well the flash space that the boot-loader resides in won't be in jeopardy of being pillaged!

Reset and Interrupt Vectors

Reset vectors can become a confusing concept when they become mixed in with the development of a boot-loader. When developing application code software developers typically don't need to think about where the reset vector is stored, where it's pointing or really what it is doing. The application libraries just handle everything for us and we end up in main() when we start up our system and start debugging.

The reset vector is the location in memory where the first instruction for the application is located. When a processor is first started up it begins program execution at the address stored in the reset vector. For a system with a boot-loader, this address will be the branching code or entry into the boot-loader itself. So if the processor reset vector is already used by the boot-loader, how on earth does code branch to the application code? Where is the application reset vector stored?

In simple terms, anywhere the developer wants it to be. One of the most common approaches to handling the reset vector in an application is to leave it in the default location. While in the end this doesn't help, it does for the engineers that are developing the application. If the reset vector is moved to a unique location issues with power-up and resets can result when the boot-loader is not used. Basically the reset vector location that the processor is looking for won't be valid.

The heavy thinking as to where to locate an applications reset vector and how it gets there is squarely placed on the boot-loader designer. The common approach to take is when the binary file is read into the flashing software, it looks through the records and when the application reset vector

is found it relocates it to the address specified by the boot-loader. This has the advantage of allowing the application engineer to develop their software without needing to worry about what is going on with the boot-loader (except for flash usage). Another method is to have the boot-loader itself monitor for the reset vector and put it in the location of its choosing. By doing this, the boot-loader designer has complete control over the reset vector and does not need to rely on another application properly translating the reset vector.

It is still up to the boot-loader designer to decide where to place the reset vector. It can really go just about anywhere provided that location is carved out in flash for that purpose. However, the designer will most likely want to select either the first address in the flash space or the last. By doing this contiguous memory space is not interrupted. For example, in the C2000, in the boot-loader you can define a reset vector location in the linker as follows:

APP_RESET   : origin = 0x3F5FFB, length = 0x000002   /* Application Reset Vector */

This assumes that sector A is allocated to the boot-loader and the next two addresses after sector A are allocated for the application reset vector. The application would then start after that.

The reset vector is not the only complication associated with the boot-loader. The boot-loader like any application will need a vector table. The application that it jumps to will also need a vector table. This means that that is an extra vector table that needs to be placed somewhere. In some processors this is not an issue. For example, the C2000 more or less decides where to place it in flash and there is not issue. However, the S12X has a vector table register that needs to be setup to tell the processor where the vectors are located. For the boot-loader it needs to be set to one location while for the application it needs to be set to another.

Typically, the vector table for the application is stored in the flash sector directly following the boot-loader. This allows the application vector table to be at the "top" of flash, where it would typically be found if a boot-loader was not present. The vector table for the boot-loader is stored with the boot-loader in protected flash.

Application File Format Selection

Each manufacturer, compiler vendor and tool manufacturer have their own file format that they support to put code onto the microprocessor. Over the years instead of a common standard being developed that everyone could agree on, everyone more or less developed their own standard binary file format. That said, there are still a number of formats which are commonly used or have been somewhat adopted and have some degree of support by most compilers. These include Motorola S-Records, Intel Hex file format, binary and COFF(Common Object File Format) files.

The greatest recommendation when selecting a file format to use is to select the format which is easiest for you to understand and that your compiler can compile to without too much extra effort. After all, whether you like it or not at some point you are going to have to look through every line of one of these files looking for mishaps or issues that are causing your processor to go crazy. Therefore you are going to want a format that you can understand.

We select a format based on its ease of use with our processor in addition to ease of developing and interpreting the file. The file will need to be sent to the boot-loader which will then need to understand the command for the file, be able to parse and reassemble the file, determine where in flash the data needs to be written and then write that data to flash. In order to prevent

bugs from creeping into the design keep things simple and select a format you can understand and are familiar with.

Most compilers come with a hex conversion utility which allows for the compiler output to be converted to a number of common formats. For example, the TI assembly tools allow for conversion to the following commonly used formats:

- ASCII-Hex, supporting 16-bit addresses
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit, 24-bit, and 32-bit addresses

Freescale Code Warrior automatically generates S-Record files with every compile making a conversion utility unnecessary. S-Records are a commonly used format. They support 16-bit, 24-bit and 32-bit addressing. There are 3 different types of records: header, code/data and termination records. This can be seen in Figure 4.

Record Type	Description
S0	Header record
S1	Code/data record for 16-bit addresses (S1 format)
S2	Code/data record for 24-bit addresses (S2 format)
S3	Code/data record for 32-bit addresses (S3 format)
S7	Termination record for 32-bit addresses (S3 format)
S8	Termination record for 24-bit addresses (S2 format)
S9	Termination record for 16-bit addresses (S1 format)

Figure 4 (Instruments, 2011)

Each software image is made of multiple records with the S-Records file. Figure 5 is an example of a couple of records that could be found in an S-Record file. As can be seen, the file starts with the S0 header record followed by repeated S3 records before a termination S7 record.
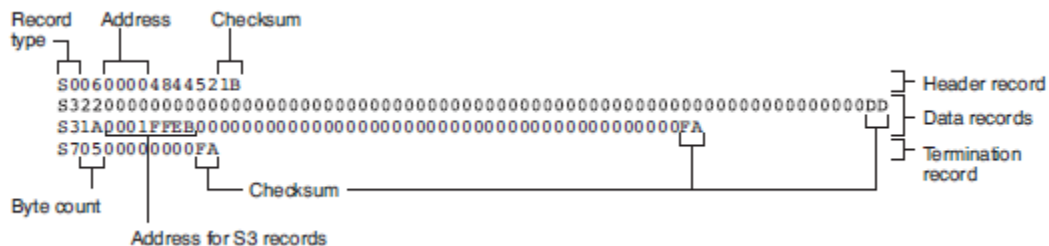

Figure 5 (Instruments, 2011)

Each record consists of a number fields. The first, is the record type consisting of the S type. Next the record contains the number of bytes that are contained within the record followed by the address that the data should be placed in within flash. Finally the data followed by a record checksum. The checksum is calculated using the least significant byte of the 1s complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields. (Instruments, 2011).

Embedded Application Setup

In order for the embedded application to function properly there is an important change that needs to be made so that it plays nice with the boot-loader. The linker file needs to be updated so that it does not try to locate code or data within the boot-loader memory space.

The linker file is used by the compiler to decide where to place functions in RAM and flash. While the boot-loader memory space would be protected within flash, if the application tries to locate itself in the flash space the boot-loader would ignore those memory locations. The application would never get written correctly to flash. This could lead to a unit that is completely non-functional. It is therefore imperative that the linker be modified so that it does not occupy the same space.

There is one final adjustment that needs to be made to the application. In the event that a request to update the firmware is made while the application is running the application needs to accept a command to enter the boot-loader. The application therefore needs a command that can write an EEPROM byte to notify the boot-loader that it should run and then perform a soft reset of the system.

A common method used for restarting the system is to cause the watchdog timer to timeout or restart the system. Most watchdog timers will immediately reset if incorrect values are written to it. The other option would be to place the system into an infinite loop and allow the watchdog to time-out at its own pace.

Troubleshooting

Unfortunately every software project at some point or another runs into problems and the development of a boot-loader is no different. Undoubtedly the software engineer will run into a number of challenges whose solution is not evident. There are a number of techniques which can be used to debug a boot-loader and application and a number of other common problems that have been encountered by thousands of other software developers. There are a number of techniques and issues to keep in mind when debugging the boot-loader.

One of the most common issues is how to debug an application that has been loaded through the boot-loader. It works by itself but when loaded through the boot-loader instead of seeing the brilliance of blinking, flashing LED's there is nothing. It is possible to send out status messages through a UART or other interface to check the status of the device but stepping through code in a debugger would be best.

Nearly every IDE has a feature that is for the most part ignored by programmers 99% of the time. It is a feature that allows an engineer to add additional symbols to the debugger. When developing a single application there is rarely a time when you would want to add additional symbols; however, when debugging a boot-loaded system this feature is extremely helpful.

While debugging the boot-loader, an engineer can select add symbols and add in the symbols for the application that is going to be debugged. An example of this is shown in Figure 6. Now instead of jumping off into a hardly understandable disassembly window when the boot-loader jumps to the application it will instead jump to readable C code and allow debugging of the application as usual.
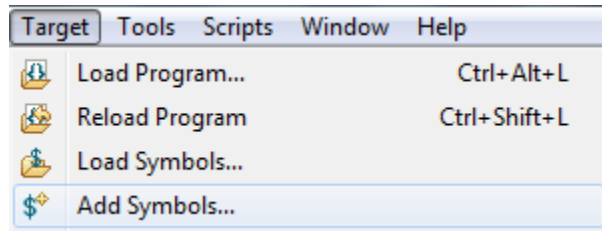
Figure 6

It might be that when the software developer adds the symbols they discover that the code never jumps to the application. In this case, the best place to start is going to be the boot-loader safety checks. Verification that the reset vector for the application exists is a good place to start. Verify that each safety check is indeed getting passed.

It may be that the boot-loader believes that the application is valid but when it jumps to the application somewhere in the application it fails. The application image may not be correct or the copy down may not be occurring. From most IDE's, you can create an image of the flash space from a starting address to an ending address. Take an image of the program loaded on the target by itself and save this file. Then load the application through the boot-loader and save this file. You can use WinMerge to compare the images. Are they the same? If they are then the application is getting there in one piece!!! If it isn't then there could be issues with writing the data to flash, buffer issues, etc. By using this method the engineer can at least identify if the application has gotten there in one piece.

If the reset vector is good, the application image is good but the application still is not running properly the system could be experiencing a copy down issue. When the boot-loader starts, it runs its copy down functions to setup the C environment and to setup its variables. When it jumps to the application the application needs to do this exact same thing. Now an engineer might think this is done automatically by the processor or the compiler. Unfortunately this is not the case. Some manufacturers do not automatically include the copy down functions into a project. They expect it to be placed there and called manually. Most times the debugger will insert the copy down call itself even if it isn't included in the project. This results in an application that will function correctly when debugged by itself but exhibit strange behavior when loaded onto the system with the boot-loader. The software engineer should check their documentation and make sure that this crucial piece of code is being executed. Some processors have it burned into their ROM and all you need to do is call it from RAM and then the magic happens and the C environment will be setup properly.

One area that can cause extreme pain on the part of the software developer is when a boot-loader loads itself into memory, initializes the system and then decides to jump to the application. Some processors have registers that are only allowed to be written or setup one time. If the boot-loader sets up these registers and then the application tries to set them up another way then misbehavior of the system can result. There are a couple of common registers that should be watched for this type of behavior. They are

1) Watch Dog Timers
2) Processor Mode
3) Memory registers.

Troubleshooting

Boot-loaders are intimidating the first time that an engineer develops one. There are a lot of things to learn and many nuances that need to be kept in mind in order to make the system behave properly. It can be a struggle to get the first boot-loader up and running but once the process has been gone through once, each successive boot-loader becomes easier and easier to write.

Beware of boot-loaders developed by the manufacturer of your processor! The examples that they provide are excellent at providing a quick starting point from which a developer can take over but provided as is they are NOT robust enough to be put into the field.

There is a lot to take into account when developing a boot-loader and this paper has just grazed the top of the process. However, it should serve as a good starting point for the development of the readers' next boot-loader project.

References

Instruments, T. (2011, 05). TMS320C28x Assembly Language Tools. United States.

Society, I. C. (2004). *Guid to the Software Engineering Body of Knowledge.* Los Alamitos: Angela Burgess.

Appendix A

While there are 7 classes of requirements for boot-loader design, each of these classes can be broken down to generate a list of requirements that can be used to guide the development of the boot-loader. Below is each of the 7 classes with additional example requirements broken out below them. Each boot-loader will have its own requirements and will be different in certain aspects from another boot-loader.

1) Ability to switch or select the operating mode (Application or boot-loader)
   a. A GPIO input held for 10 seconds shall cause the system to enter boot-loader mode
   b. An EEPROM byte shall be used to switch modes
   c. An enter boot-loader message received within 100 ms of start-up shall cause the system to enter boot-loader mode
2) Communication interface requirements (USB, CAN, I2C, USART, etc)
   a. Application images shall be transferred to the system via UART.
3) Record parsing requirement (S-Record, hex, intel, toeff, etc)
   a. The application image shall be sent to the system in S-Record format.
4) Flash system requirements (erase, write, read, location)
   a. The boot-loader shall be capable of erasing only the application section of flash
   b. The boot-loader shall write application image records to flash
   c. The boot-loader shall be capable of reading back specified address location in flash upon command
   d. The boot-loader shall be located in the first 4 kB of flash
5) EEPROM requirements (partition, erase, read, write)
   a. The boot-loader shall partition EEPROM when commanded
   b. The boot-loader shall be capable of erasing the EEPROM
   c. The boot-loader shall be capable of reading out a specified EEPROM location
   d. The boot-loader shall be capable of writing data to specified EEPROM locations
6) Application checksum (verifying the app is not corrupt)
   a. The boot-loader shall calculate an image checksum and store it in flash
   b. The boot-loader shall verify that the application checksum is valid before executing the application
7) Code Security (Protecting the boot-loader and the application)
   a. The boot-loader shall be located in protected flash to prevent updating or modification at any time