booty Documentation

Jason R. Jones

Dec 28, 2018

Contents

CHAPTER 1

the easy bootloader

`booty` is a protocol.

`booty` is an implementation.

add a little booty to your project and make upgrading your firmware a breeze.

1.1 Introduction

1.1.1 What is a Bootloader?

A bootloader is a term that is generally applied to microcontrollers. It is a special type of application that will run under certain conditions that allows the actual application to be erased, updated, verified, or flashed as needed.

There are more comprehensive explanations available with a quick search.

1.1.2 What is `booty`

The `booty` protocol focuses on the relatively simple bootloaders required for microcontroller applications. If your microcontroller is in the PIC, dsPIC, STM32F, Atmel, or similar families, then `booty` will likely fulfill your requirements.

The protocol

`booty`, at its highest level, describes the basic operations of a bootloader implementation and is not an implementation itself. Protocol features include:

- serial-device based (UART, RS-232, RS-485, etc)
- device and protocol identification
- device erasure

- loading
- verification
- self protection
- open source!

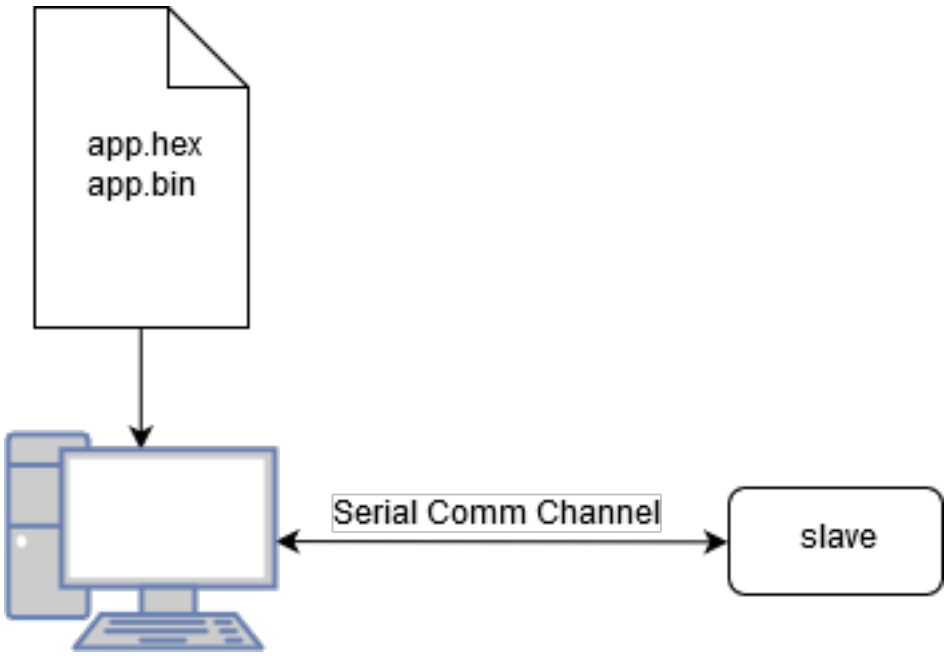
The implementation(s)

On the other hand, since `booty` is a protocol, then there are any number of possible implementations and workflows possible. For instance, the author has implemented a server using Python (described by this documentation) along with a client implentation in C for the dsPIC series of microcontrollers. The C implementation is small, simple, uses no interrupts, and has been successfully tested.

1.2 The Big Ideas

1.2.1 Master/Slave Model

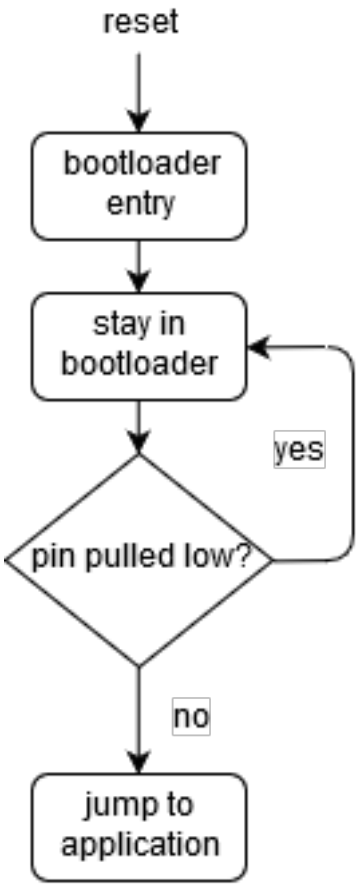
The typical implementation of the bootloader involves having a master - which directs all of the action - and a slave device which simply executes those commands. In this way, it is possible to implement several different workflows on the master while the slave device blindly executes those instructions. The master will be able to command the slave device to execute any command supported by the protocol.



1.2.2 Entry into the Bootloader

The bootloader is automatically entered upon a device reset. Depending on the configuration and hardware, there are various conditions that may be present that will allow the bootloader to simply "fall through" to the application or to initialize the bootloader itself.

One of the most common methods for staying in a bootloader is to simply pull a pre-defined pin low within the application. In this case, the device stays in the bootloader for as long as the pin is pulled low.



Another method supported by booty is to stay in the bootloader for a time after reset or after the most recent valid message has been received. This method has the advantage of requiring no additional hardware, but the disadvantage of slow application startup time.

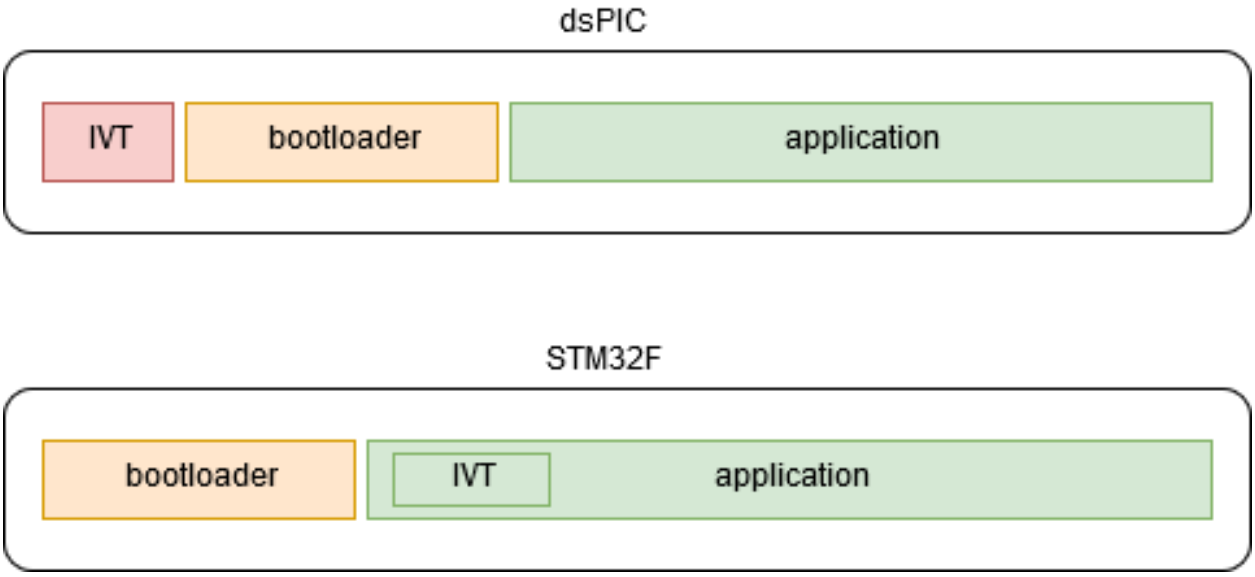
1.2.3 Device Memory Map

There are two primary areas of concern for applications which require a bootloader. The first and foremost is the bootloader itself. It is expected that `booty` will reside somewhere in the beginning memory of the device. The exact location of the bootloader should be stored within the bootloader itself so that it may protect itself from erasure.

The application must have some offset applied to its starting address in order to allow the bootloader room to reside. This offset is device-specific, but is made directly available through the communications protocol using the READ_APP_START_ADDRESS command.

Different device architectures require different memory maps and strategies. For instance, in the dsPIC series, the interrupt vector table is always located at or near the beginning of the device memory. As a result, the bootloader must be located such that it is after the hardware-defined interrupt vector table (IVT) but before the application. It is also very advantageous that the bootloader itself does not utilize interrupts so that the application may fully utilize the hardware-defined IVT.

The ARM architecture, on the other hand, supports offsetting the entire IVT within the application, making it relatively easy to locate the bootloader at the beginning of the memory and the application a defined offset from that point.



Generally, a customer linker script will be utilized to impose image location requirements on the bootloader and the application.

1.3 Protocol

This document describes the communications protocol for the bootypic repository

There are two primary layers that this document is concerned with, framing and command. The framing layer is primarily concerned with delimiting the packet boundaries along with ensuring data integrity of the packet using a fletcher16 checksum. The command layer is concerned with interpreting the payload of the frame so that the software layer may respond accordingly.

1.3.1 Framing

All frames have a Start of Frame (SOF) byte at the beginning and an End of Frame (EOF) byte at the end. The bytes between the SOF and EOF comprise the data being transmitted. Any byte that corresponds to the SOF, EOF, or Escape (ESC) characters will be escaped and XORed so that they do not interfere with the overall transmission process. The last two bytes of the data bytes will be the fletcher16 checksum of the payload.

Escaping

Any byte between the SOF and EOF that corresponds to SOF, EOF, or ESC will be replaced by two bytes, the first will be the ESC byte and the second will be the data byte XORed with the XOR value.

Structure

The structure, without escaping, is as follows:

```
[SOF] [dat0] [data1] [data2] [...] [dataX] [F16(0:7)] [F16(15:8)] [EOF]
```

Note that if F16 bytes correspond to special characters, they will be properly escaped.

If, for instance, data4 corresponded to the SOF byte, the stream would be modified as follows:

```
[data2] [data3] [ESC] [ESC_XOR ^ data4] [data5]
```

In this way, it is possible to give up a small amount of transmission efficiency in order to be able to transmit the entire range of data.

Special Characters

The special characters are the SOF, EOF, ESC, and ESC_XOR. These values are ONLY to be used as part of the framing protocol. All payload bytes that correspond to these values are to be escaped:

```
[SOF] 0xf7
[EOF] 0x7f
[ESC] 0xf6
[ESC_XOR] 0x20
```

Behavior

If a F16 value is correct, then the packet is forwarded up to the next software layer. If it is incorrect, then the packet is discarded.

Fletcher 16 Verification

Each frame is verified using a fletcher16 algorithm on all data bytes. The Python implementation of the fletcher16 algorithm is as follows:

```
sum1 = 0
sum2 = 0

for i, b in enumerate(data):
    sum1 += b
    sum1 &= 0xff  # Results wrapped at 16 bits
    sum2 += sum1
    sum2 &= 0xff

return sum1, sum2
```

A similar implementation in C is as follows:

```
uint16_t fletcher16(uint8_t* data, uint16_t length){
    uint16_t sum1 = 0, sum2 = 0, checksum;

    uint16_t i = 0;
    while(i < length){
        sum1 = (sum1 + (uint16_t)data[i]) & 0xff;
        sum2 = (sum2 + sum1) & 0xff;
        i++;
    }

    checksum = (sum2 << 8) | sum1;

    return checksum;
}
```

1.3.2 Commands

The command layer may have multiple versions, which will be saved to the microcontroller at compile time.

Command Structure

A typical command packet, stripped of framing information:

```
[reserve0] [reserve1] [CMD] [payload0] [payload1] [...] [payloadX]
```

The first two bytes are reserved for future use. They may contain any type of data the user prefers. For the remainder of this document, these reserved bytes will be ignored.

CMD refers to a 1-byte command. The command will determine how the remainder of the payload is interpreted. In some cases, there may be no additional bytes after the command, such as in the CMD_START_APP command.

Behavior

The PC will typically be the server and the microcontroller will simply respond to the commands as a directed client. Many commands have no required response, such as CMD_ERASE_PAGE. Others that require a response will simply embed the same command into the structure of the response.

When strings are being passed, they will be passed as ASCII bytes and transmitted with the null string terminator `\0`. These are represented in the commands as a string with quotes.:

```
representation: "my str\0"
transmitted: [0x6d] [0x79] [0x20] [0x73] [0x74] [0x72] [0x00]
```

1.3.3 Command Sets

In order to increase flexibility across devices, command version sets are created which will allow the server software to determine the command set that may be utilized with the particular device and bootloader variant. At this time, there is only one command set called `0.1`, which includes all commands as listed below. All of the below commands will be included as a subset of future command sets for backward compatibility.

1.3.4 Supported Commands

Read Platform

Character: 0x00 Command Sets: 0.1

The `CMD_READ_PLATFORM` command instructs the microcontroller to return a string containing the platform, which usually corresponds to a microcontroller part number:

```
master:   [CMD_READ_PLATFORM]
response: [CMD_READ_PLATFORM] "dspic33ep32mc204\0"
```

Read Version

Character: 0x01 Command Sets: 0.1

The CMD_READ_VERSION command instructs the microcontroller to return a string containing the instruction set that it supports:

```
master:   [CMD_READ_VERSION]
response: [CMD_READ_VERSION] "0.1\0"
```

Read Row Length

Character: 0x02 Command Sets: 0.1

The CMD_READ_ROW_LENGTH command instructs the microcontroller to return the smallest row length that can be programmed at one time:

```
master:   [CMD_READ_ROW_LENGTH]
response: [CMD_READ_ROW_LENGTH] [length(7:0)] [length(15:8)]
```

Read Page Length

Character: 0x03 Command Sets: 0.1

The CMD_READ_PAGE_LENGTH command instructs the microcontroller to return the page erasure size in instructions:

```
master:   [CMD_READ_PAGE_LENGTH]
response: [CMD_READ_PAGE_LENGTH] [length(7:0)] [length(15:8)]
```

Read Max Program Memory Length

Character: 0x04 Command Sets: 0.1

The CMD_READ_PROG_LENGTH command instructs the microcontroller to return the program length, which is the maximum address that may be programmed to:

```
master:   [CMD_READ_PROG_LENGTH]
response: [CMD_READ_PROG_LENGTH] [length(7:0)] [length(15:8)] [length(23:16)]␣
↪[length(31:24)]
```

Read Max Program Size

Character: 0x05 Command Sets: 0.1

The CMD_READ_MAX_PROG_SIZE command instructs the microcontroller to return the maximum programming size that it will support in instructions:

```
master:   [CMD_READ_MAX_PROG_SIZE]
response: [CMD_READ_MAX_PROG_SIZE] [length(7:0)] [length(15:8)]
```

Read App Start Address

Character: 0x06 Command Sets: 0.1

The CMD_READ_APP_START_ADDRESS command instructs the microcontroller to return the starting address of the application. This will usually be 0x1000. This will be utilized for checking application integrity during the verification stage.:

```
master:   [CMD_READ_MAX_PROG_SIZE]
response: [CMD_READ_MAX_PROG_SIZE] [address(7:0)] [address(15:8)]
```

Erase Page

Character: 0x10 Command Sets: 0.1

The CMD_ERASE_PAGE command instructs the microcontroller erase a page of flash memore starting at the provided address.:

```
master:   [CMD_ERASE_PAGE] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
response: -
```

Read Address

Character: 0x20 Command Sets: 0.1

The CMD_READ_ADDRESS command instructs the microcontroller read a single value from flash memory and to return that value.:

```
master:   [CMD_READ_ADDRESS] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
response: [CMD_READ_ADDRESS] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
                             [value(7:0)] [value(15:8)] [value(23:16)] [value(31:24)]
```

Read Max

Character: 0x21 Command Sets: 0.1

The CMD_READ_MAX command instructs the microcontroller read the maximum number of values from flash memory and return them as an array of values. This allows for much more efficient reading of memory:

```
master:   [CMD_READ_ADDRESS] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
response: [CMD_READ_ADDRESS] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
                             [value0(7:0)] [value0(15:8)] [value0(23:16)]␣
↪[value0(31:24)]
                             [value1(7:0)] [value1(15:8)] [value1(23:16)]␣
↪[value1(31:24)]
                             [...]
                             [valueX(7:0)] [valueX(15:8)] [valueX(23:16)]␣
↪[valueX(31:24)]
```

Write Row

Character: 0x30 Command Sets: 0.1

The CMD_WRITE_ROW command instructs the microcontroller to write an entire row of data, as defined by the microcontroller datasheet, starting at the address. In many cases, a row consists of only 2 instructions, so it may not be very efficient.:

```
master:   [CMD_WRITE_ROW] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
                          [value0(7:0)] [value0(15:8)] [value0(23:16)]␣
↪[value0(31:24)]
                          [value1(7:0)] [value1(15:8)] [value1(23:16)]␣
↪[value1(31:24)]
                          [...]
                          [valueX(7:0)] [valueX(15:8)] [valueX(23:16)]␣
↪[valueX(31:24)]

response: -
```

Write Max

Character: 0x31 Command Sets: 0.1

The CMD_WRITE_ROW command instructs the microcontroller to write an entire row of data, as defined by the return value of READ_MAX_PROG_SIZE, starting at the address. This is usually a much more efficient method of writing.:

```
master:   [CMD_WRITE_ROW] [address(7:0)] [address(15:8)] [address(23:16)]␣
↪[address(31:24)]
                          [value0(7:0)] [value0(15:8)] [value0(23:16)] [value0(31:24)]
                          [value1(7:0)] [value1(15:8)] [value1(23:16)] [value1(31:24)]
                          [...]
                          [valueX(7:0)] [valueX(15:8)] [valueX(23:16)] [valueX(31:24)]

response: -
```

Start Application

Character: 0x40 Command Sets: 0.1

The CMD_WRITE_ROW command instructs the microcontroller to start the application. Note that the bootloader will no longer respond after the application is started.:

```
master:   [CMD_START_APP]
response: -
```

1.4 Master

1.4.1 Language

booty itself is language-agnostic; however, this booty master is written using Python. The intent is to have a command-line utility that may be easily utilized in a development or production environment.

1.4.2 Installation

Installation into your environment should be as easy as pip install booty.

1.4.3 Usage

Assuming that this is installed in your root python environment, it will create a command-line utility which can be directly invoked:

```
Usage: __main__.py [OPTIONS]

Options:
  -h, --hexfile PATH      The path to the hex file
  -p, --port TEXT         Serial port (COMx on Windows devices, ttyXX on Unix-
                          like devices)  [required]
  -b, --baudrate INTEGER  Baud rate in bits/s (defaults to 115200)
  -e, --erase             Erase the application space of the device
  -l, --load              Load the device with the hex file
  -v, --verify            Verify device
  -V, --version           Show software version
  --help                  Show this message and exit.
```

Of course, to use the package, there are some options that need to be specified. The two most necessary options are the *–hexfile* and *–port* options. Additionally, either the *–erase*, *–load*, or *–verify* should be specified or no action will take place. This is, after all, a loading and/or verification utility.

Regardless of the order of the input parameters, the order of execution will be erase, load, then verify.

A common command to load and verify a device might look like this:

```
user ~$ booty -p COM20 --load --verify -hexfile "C:/path/to/my/hex.hex"
```

The utility will execute a series of commands and result in an output similar to this:

```
user ~$ booty -p COM20 --load --verify -hexfile "C:/path/to/my/hex.hex"
INFO:booty:Using provided hex file at "C:/path/to/my/hex.hex" to load and verify␣
↪device
INFO:booty.comm_thread:platform set: dspic33ep32mc204
INFO:booty.comm_thread:version set: 0.1
INFO:booty.comm_thread:row length set: 2
INFO:booty.comm_thread:page length set: 512
INFO:booty.comm_thread:program length set: 21996
INFO:booty.comm_thread:max programming size set: 128
INFO:booty.comm_thread:application start address set: 4096
INFO:booty.comm_thread:device identification complete
INFO:booty:loading...
INFO:booty:device successfully loaded!
INFO:booty:verifying...
INFO:booty:device verified!
```

1.5 How it Works

1.5.1 Programming Sequence

All relevant information is stored on the microcontroller, meaning that the relevant data is stored at compile-time.

The programming takes place in three stages:

1. device identification - determines what the device is, the command set available, and the page erase and write sizes

2. erasure - erasure of all application-programmable memory
3. loading - a series of write cycles which write to the program memory of the microcontroller
4. verify - a series of read cycles and final verification of the user memory

Shown is a complete id/erase/load/verify of a 16k device at 57600bits/s and operating at 12MIPS, which takes 15.8s. Each section is delimited by the green markers. This load time could obviously be reduced by running at a faster baud rate.

1.5.2 Threaded Execution

At the lowest level, there is a thread which takes commands from higher level software and creates an internal queue which is executed in sequence. This layer will execute simple commands, such as "read flash", "write flash", etc. while also ensuring that the protocols, required sizes, and timing constraints are enforced.

At the higher level, the hex file is read and a command set is created for the low-level software. At various places, there are "waits" put in place. For instance, the high level software might request that the low level software do all of the write operations before it moves on to a verification stage. This is more clear in the source code.

The high-level operations may be found in `/booty/__main__.py` and `/booty/util.py` while the low-level thread may be found in `/booty/comm_thread.py`.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search