Katiana Bootloader for Arduino

Project Description

This bootloader enumerates on USB as a CDC Class device (virtual serial port). It implements a subset of the AVR109 protocol, allowing applications such as avrdude to program flash and EEPROM on the MCU. It also provides startup logic for running Arduino sketches.

Out of the box this bootloader builds for the ATmega32U4 with a 4KB bootloader section size If you wish to alter this size and/or change the AVR model, you will need to edit the MCU, FLASH_SIZE_KB and BOOT_SECTION_SIZE_KB values in the accompanying makefile. It should be compatible with ATmega MCUs with flash sizes up to 128kB, but has not been tested with other than the ATmega32U4.

When the bootloader is running, the boards RX and TX LEDs will flash in accordance with data being received from and set to the CDC interface. Optionally, the "L" LED can also be configured to flash a fixed number of times at bootloader startup.

Benefits

The Katiana bootloader offers some benefits compare to previous USB-based designs:

- Sketches can determine the cause of an MCU reset.
- Custom USB serial numbers may be assigned to each physical board on which the bootloader is installed.
- Communications between bootloader and sketch can no longer be accidentally interfered with by the sketch.
- Arduino IDE uploads can now program both flash and EEPROM.
- Address validation protects against accidental or malicious uploads which would corrupt the bootloader.

Compatibility

This bootloader is specifically intended for the ATmega32U4 MCU as used on Arduino boards such as the Leonardo and LilyPad USB. It implements Arduino-specific behaviors and supports uploads by the Arduino IDE.

USB Support

USB Mode:	Device

USB Class:	Communications Device Class (CDC)
USB Subclass:	Abstract Control Model (ACM)
Relevant Standards:	USBIF CDC Class Standard
Supported USB Speeds:	Full Speed Mode

Bootloader Logic

At startup, the bootloader executes a sequence of logical tests and delays as depicted in the figure below. These provide multiple paths by which to enter the bootloader while at the same time providing the fastest possible sketch startup time when the bootloader is not required.

From the point of view of a sketch, the bootloader *interferes* with reset events. In some cases, the interference is minimal and the sketch is started with no delay or a small delay.

In other cases, the bootloader intercepts the reset event and provides the capability to upload a new sketch. When AVR910 commands are being accepted, there is a timeout period. If eight seconds elapses with no valid AVR910 commands being received, the sketch will be started (if there is one). If there's no sketch, there is no timeout period.

Power-on and Brown-out Resets

As long as there's a sketch loaded, the bootloader immediately starts it for these reset events. If there's no sketch, the bootloader begins accepting AVR910 commands until a sketch has been loaded. Note that earlier bootloader versions treated brown-out resets differently (see the section below on Other Reset Events).

WDT Resets

This is the mechanism through which the Arduino IDE uploads new sketches. Arduino core libraries (linked into every sketch) purposely generate a WDT reset to get into the bootloader when the IDE requests an upload. In essence, this takes over control of WDT reset events and would make it more difficult for sketches to make use of WDT resets for their own purposes. The bootloader's startup sequence is designed to make the Arduino core's take-over of WDT resets transparent to the overlying sketch.

The bootloader works in cooperation with the Arduino core to make WDT reset events caused by the sketch behave as expected. By storing a prearranged value at a pre-defined location in SRAM, the Arduino core can signal to the bootloader that a WDT reset was caused by the core. In this case, the bootloader can begin accepting AVR910 commands instead of immediately starting the sketch. The pre-defined location is SRAM is calle the *BootKey*, and when it contain the pre-defined value the BootKey is said to be *active*.

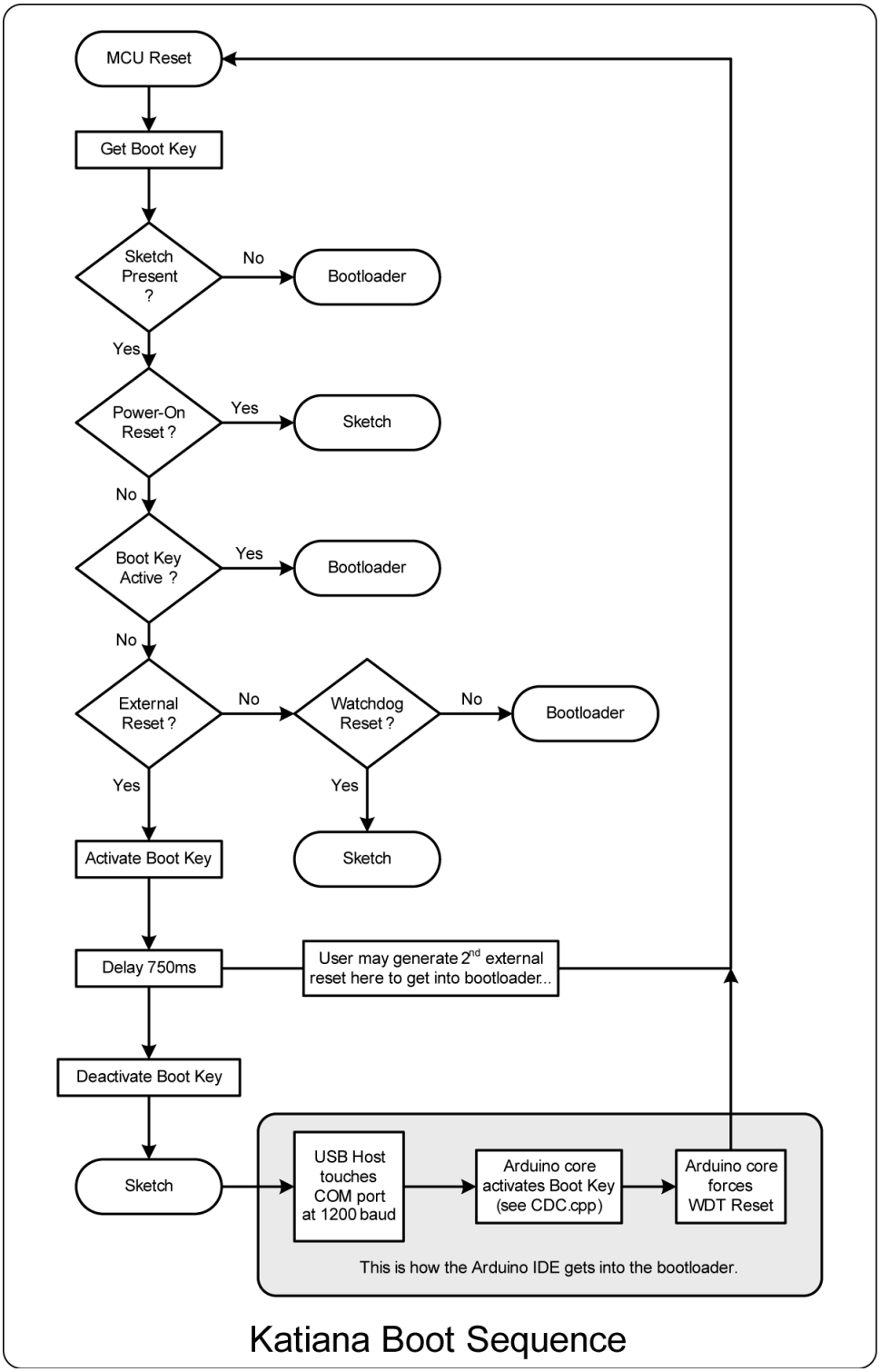
Beginning with IDE versions 1.6, the BootKey location is `(RAMEND-1)`. When a sketch is running this normally contains the return address for the **`main()`** function. Since **`main()`** never returns this is a safe place to pass a value to the bootloader. The bootloader has some code in its `.init0` section to read this value, before it is overwritten by the bootloader's own startup code.

External Resets

The bootloader will normally start the sketch after a small delay (750ms) when an external reset occurs. However, if a second external reset occurs during the delay period the bootloader will instead begin accepting AVR910 commands. This is a failsafe mechanism which allows the user to activate the bootloader if a sketch has been loaded which breaks the normal CDC serial port provided by the Arduino core.

Other Reset Events

All reset events not mentioned above cause the bootloader to begin accepting AVR910 commands with an 8-second timeout. On most (or all?) MCUs, these are the JTAG and USB reset events.



Arduino IDE Upload Process

Many Arduino boards include a separate hardware component which implements the USB interface. On these boards it is possible for the host computer (e.g. Windows or Linux) to force a hardware reset on the Atmel MCU – and this provides an entry into the bootloader to upload a new sketch. This is not the case with boards such as the Leonardo or LilyPad USB – the USB interface is built into the Atmel MCU here. As a result, there is no way for the host computer to force a hardware reset on the Atmel MCU. A different

technique must be used to get into the bootloader for sketch uploads.

To understand this process, one should know that the CDC serial port exposed by the bootloader has a differnt USB vendor ID (aka VID) than does the serial port exposed by the sketch. For example, when the bootloader is active the board might appear on Windows as the serial port `COM7`, but when the sketch is running the board enumerates as `COM8`.

To get into the bootloader, the Arduino IDE (running on the host computer) opens the CDC serial port at 1200 baud and then immediately closes it. With these boards, all sketchs include an Arduino core file (CDC.cpp) which detects the port being opened at 1200 baud; when it is closed the core forces a WDT reset. Before doing this however, it activates the BootKey so the booloader knows that an upload is desired.

The IDE then waits for the sketch serial port to disappear, and for the bootloader serial port to appear. Once this happens it begins sending AVR910 commands to the bootloader.

Custom USB Serial Numbers

There is an option to have the bootloader enumerate on USB with a custom hardware serial number. Without this on Windows at least, every time the bootloader is plugged into a different USB port, it will get a different serial port COM number. That's rather annoying. Having the bootloader report a serial number fixes this – the same board will always enumerate with the same COM port number.

If this is enabled, you will need to be careful not to assign the same serial number to more than one physical board. Not sure what happens if you don't obey this guideline, but it might not be good. The serial number needs to be placed in the file **`UsbHdwrSerial.h`** prior to building the bootloader. If more than one board is being loaded, a different build with different serial number is required for every board. The serial number header file should look like this:

```
//
#pragma once
#define USB_HDWR_SERIAL L"12345678901234567890"
//
```

The string must be exactly twenty characters long. Be sure to include the "L" string prefix so that a Unicode string is generated.

Generating Custom Serial Numbers

An example of generating semi-random custom serial numbers is provided by the shell script `NewUsbHdwrSerial.sh`. This generates a serial by taking the MD5 hash of current date and time strings, then extracting the first twenty hexadecimal characters. There are many other ways to do this, of course.

Custom Sketch Serial Numbers

Above, it was pointed out that the Arduino sketch enumerates with a different PID on USB than does the

bootloader. This makes the sketch appear as a different serial port but the sketch will not report a USB serial number and will still have the annoying property of changing COM port numbers as it is moved between USB ports.

With a modified Arduino core file (`USBCore.cpp`) it is possible to have the sketch report the same serial as the bootloader. Because the VID for bootloader and sketch are different, it's okay to have them report identical S/Ns. This solves the changing port number problem and does not require a serial number to be specified with each sketch – it is kept by the booloader instead.

One way to do this is by defining a global function to obtain a custom serial number. A **"weak"** version of this function in the core returns a zero length result, indicating that there is no custom S/N available. In this case, the core behavior is unchanged.

The sketch may override this function, providing a copy of the serial number in bootloader flash. This alters the core's behavior – causing it to enumerate on USB with the desired S/N. Thus, the core's behavior only changes if the overridden function returns a non-zero length result. This scheme does not require any special compile time flags.

See the section on bootloader API for information on how to extract the serial number from bootloader flash. Here's an example of what the modifications to USBCore.cpp might look like to implement this behavior:

```
//
// this is the new weak function to obtain a custom USB serial number
// sketches should override it if they wish to provide a serial number
//
__attribute__((weak)) uint8_t USB_GetCustomSerialNumber(uint8_t **AsciiString)
{
    if (AsciiString) *AsciiString = 0;
    return 0;
}
//
// this is an existing core function with modifications
//
bool SendDescriptor(USBSetup& setup)
{
    ...existing code omitted...
    //
    else if (setup.wValueL == ISERIAL) {
            uint8_t *serial;
            uint8_t len = USB_GetCustomSerialNumber( &serial );
            if (len)
            {
                    return USB_SendStringDescriptor(serial, len, 0);
            }
#ifdef PLUGGABLE_USB_ENABLED
            else
            {
                    char name[ISERIAL_MAX_LEN];
                    PluggableUSB().getShortName(name);
                    return USB_SendStringDescriptor((uint8_t*)name, strlen(name),
      0);
            }
#endif
    }
    //
    ...remaining code omitted...
//
```

Building Katiana

1. Obtain a copy of the LUFA source code; version 170418 is known to work; later versions may work as well.
2. Create a subdirectory named `Katiana` within the `Projects` directory of the LUFA source.
3. Place the files for this bootloader inside the new Katiana subdirectory.
4. Edit makefile, specifying correct sizes for flash and boot sector.
5. Edit `Config/AppConfig` and select desired options.
6. Normally, `Config/LUFAConfig` does not require any changes.
7. If desired, the bootloader's USB PID/VID can be changed in **`Descriptors.c`**. Beware however that changing these typically requires a custom Boards.txt file in the Arduino installation – that is where USB VID/PID for the sketch are specified.
8. Run `make`.
9. To get HTML in the `Documentation` folder, run `make doxygen`.

AVRDude as used by the Arduino IDE uses only AVR910 block read/write operations for upload, so it is only necessary to enable block support when building Katiana. Flash and EEPROM byte support are not required.

Building Arduino Sketches

It may be necessary to add a custom board version which specifies some extra build flags. Try building sketches without this at first. Sketch uploads should always work the first time after flashing the bootloader and they will be uploaded to the bootloader's COM port. After the first upload, the sketch will appear on a different port; if uploads to this port are failing, then try adding the custom board with build flags as shown below.

```
<BoardName>.build.extra_flags={build.usb_flags} -DMAGIC_KEY_POS=(RAMEND-1)
```

The Arduino core libraries should detect the new bootloader based on its signature at (FLASHEND-1) and automatically use the correct location for the boot key. If that for some reason doesn't happen, the above build flags should force the issue.

Driver Installation

After running this bootloader for the first time on a new computer, you will need to supply the .INF file located in this bootloader project's directory as the device's driver when running under Windows. This will enable Windows to use its inbuilt CDC drivers, negating the need for custom drivers for the device. Other Operating Systems should automatically use their own inbuilt CDC-ACM drivers.

Host Controller Application

This bootloader is compatible with the open source application AVRDUDE, Atmel's AVRPROG, or other applications implementing the AVR109 protocol, which is documented on the Atmel website as an application note.

AVRDUDE (Windows, Mac, Linux)

AVRDude is a free, cross-platform and open source command line programmer for Atmel and third party AVR programmers. It is available on the the Windows platform as part of the "WinAVR" package, or on other systems either from a build from the official source code, or in many distributions as a precompiled binary package. This is what the Arduino IDE uses under the hood to upload sketches.

To load a new Intel HEX file directly with AVRDude, locate the temporary directory where the sketch is built. Do not close the IDE after building as it will delete the temporary directory when closed. On Windows this directory is typically located here:

```
C:\Users\<username>\AppData\Local\Temp\arduino_build_<random number>
```

For these examples, it is assumed that the command window is in the temporary build directory and that the file name of the sketch is `Sketch.ino`

It is necessary to specify "AVR109" as the programmer, with the allocated COM port. On Windows platforms this will be a standard COM port name. For example:

```
avrdude -c AVR109 -p atmega32u4 -P COM7 -U flash:w:Sketch.ino.hex
```

On Linux systems, this will typically be a device file in the `/dev/ttyACMx` directory. Like this:

```
avrdude -c AVR109 -p atmega32u4 -P /dev/ttyACM7 -U flash:w:Sketch.ino.hex
```

It also possible to upload both flash and EEPROM in the same operation. Arduino IDE builds use a `.eep` suffix for the EEPROM hex files. Don't specify the `.eep` file if there's no EEPROM data declared in the sketch; that may cause AVRDude to generate an error. Here's an example of uploading both flash and EEPROM for a sketch:

```
avrdude -c AVR109 -p at90usb1287 -P COM0 -U flash:w:Sketch.ino.hex -U
       eeprom:w:Sketch.ino.eep
```

Run AVRDude with no command line options, or refer to the AVRDude project documentation for additional usage instructions.

API for Sketches

Information about the bootloader is available to sketches and the Arduino core libraries at pre-defined locations in flash. The arduino core libraries (CDC.cpp) use this to determine the bootloader version. Without this, the Arduino core won't know where the BootKey is located.

Here's an example of macros that could be used to access Katiana's API for sketches:

```
//

#define KATIANA_TABLE_SIZE       8
#define KATIANA_TABLE_START      (FLASHEND - KATIANA_TABLE_SIZE + 1)

#define KATIANA_SIGNATURE        (uint16_t *)(KATIANA_TABLE_START +
       KATIANA_TABLE_SIZE - 2)
#define KATIANA_SIGNATURE_VALUE  0xDCFB

#define KATIANA_CLASS_SIGNATURE  (uint16_t *)(KATIANA_TABLE_START +
       KATIANA_TABLE_SIZE - 4)
#define KATIANA_CDC_SIGNATURE    0xDF00

#define KATIANA_UNUSED_WORD      (uint16_t *)(KATIANA_TABLE_START +
       KATIANA_TABLE_SIZE - 6)

#define KATIANA_USB_SERIAL       (uint8_t *)(KATIANA_TABLE_START +
       KATIANA_TABLE_SIZE - 8)

#define BOOTKEY                  (* (uint16_t *)(RAMEND - 1) )
#define BOOTKEY_BOOT_REQUEST     0x7777
//
```

Obviously, the signature, class signature and unused values are in flash memory and must be accessed using AVR `pgm_read_...()` functions. For example the siganture can be read like this:

```
uint16_t sig = pgm_read_word( BOOTLOADER_SIGNATURE );
```

BootKey API

The boot key is in SRAM and accessed like any other global variable. Normally, accessing BootKey is only something which is done within the Arduino core (i.e. in CDC.cpp). However, if a sketch wishes to force a reset into the bootloader, it can activate the BootKey and then force a WDT reset. For example,

```
BOOTKEY = BOOTKEY_BOOT_REQUEST;
wdt_enable(0);                   // (presumes that WDT interrupts are not enabled)
while (1);
```

USB Serial Number

If the option is enabled, `KATIANA_USB_SERIAL` contains the address in bootloader flash of the USB string descriptor. This is a struct contianing a header (two unsigned bytes) followed by a Unicode string. The first header byte is the size of the descriptor including header and should always equal 42. The second byte indicates the data type to follow – 0x03 for a Unicode string. The string itself is in lil' endian order and contains no terminating null character.

The section above discussing custom USB serial numbers for the sketch requires the sketch override a weak function in the Arduino core to obtain the bootloader's serial number. Here is an example of that function:

```
//
uint8_t sn[21];

uint8_t USB_GetCustomSerialNumber( uint8_t **Serial )
{
    if (sn[0])
    {
```

```
        // return cached value if available.
        if (Serial) *Serial = sn;
        return 20;
    }

    uint16_t ptr = pgm_read_word( KATIANA_USB_SERIAL ); // same as reading from
       (FLASHEND-7)
    if (ptr == 0x0000 || ptr == 0xffff) return;
    uint8_t cnt = pgm_read_byte(ptr++);
    uint8_t kind = pgm_read_byte(ptr++);

    if (cnt != 42 || kind != 3) return;

    for (int k=0; k<20; k++)
    {
        sn[k] = pgm_read_byte( ptr );
        ptr += 2;   // skip over the Unicode high byte
    }
    sn[20] = 0;     // sn[] now contains the ASCII serial number.

    if (Serial) *Serial = sn;
    return 20;
}
//
```

Valid USB Serial Numbers

This driver only supports 80-bit serial numbers, reported as 20 hexadecimal characters. There seem to be no such restriction on USB S/Ns in general. Feel free to change that at your own risk. Here's what Microsoft says about validating serial numbers in Windows:

> The string is non-NULL.
> The string is not longer than 255 bytes.
> The descriptor type returned in the descriptor is type STRING.
> The number of bytes in the string is an even number, as it is a Unicode string.
> The string does not contain any invalid characters:
>
> - The character must have a value greater than or equal to 0x20.
>   The character must have a value less than or equal to 0x7F.
>   The character must not be a comma (0x2C).

Recovering MCUSR

When the MCU is reset, MCUSR contains bits indicating the cause of the reset. The bootloader clears this register which means the sketch cannot examine it to find the cause of the last reset. The bootloader provides a work-around for this problem by saving the initial value of MCUSR and passing it to the sketch in an MCU register (R2).

When the bootloader enumerates on USB and attempts to read AVR910 commands prior to starting the sketch then the MSB of R2 is turned on. This resolves the ambiguity that can occur if the reset source is an external reset or the watchdog timer.

Reset Source	R2 MSB	Interpretation

WDT	0	Sketch Generated
WDT	1	Arduino Upload
External	0	Normal reset
External	1	Double-tap bootloader run

The code snippet below shows how the value in R2 can be recovered for use in an Arduino sketch. Normally, initializtion code would destroy the value in R2, making it unavailable to the sketch. Functions in the `.init0` section are automatically called prior to any other initialization code being executed and have access to the un-sullied value of R2. This function is called "under the hood", even though it is not referenced elsewhere in the sketch. Because of this, the function must also be declared with the `used` attribute to prevent the optimizing compiler from stripping it from the linked output image.

```
//
static volatile uint8_t bootMCUSR;
//
void
__attribute__ ((naked))                  // std call/return linkage not required
__attribute__ ((section (".init0")))     // runs before any other init code
__attribute__ ((used))                   // prevents optimizer from stripping the
       function
saveMCUSR(void)
{
    __asm__ __volatile__ (
    "sts %0,r2\n"
    : "=m" (bootMCUSR)
    :
    );
}
//
void setup(void)
{
    // we can safely examine the value of bootMCUSR here.
    if (bootMCUSR & _BV(PORF)) ... etc ...
}
//
```

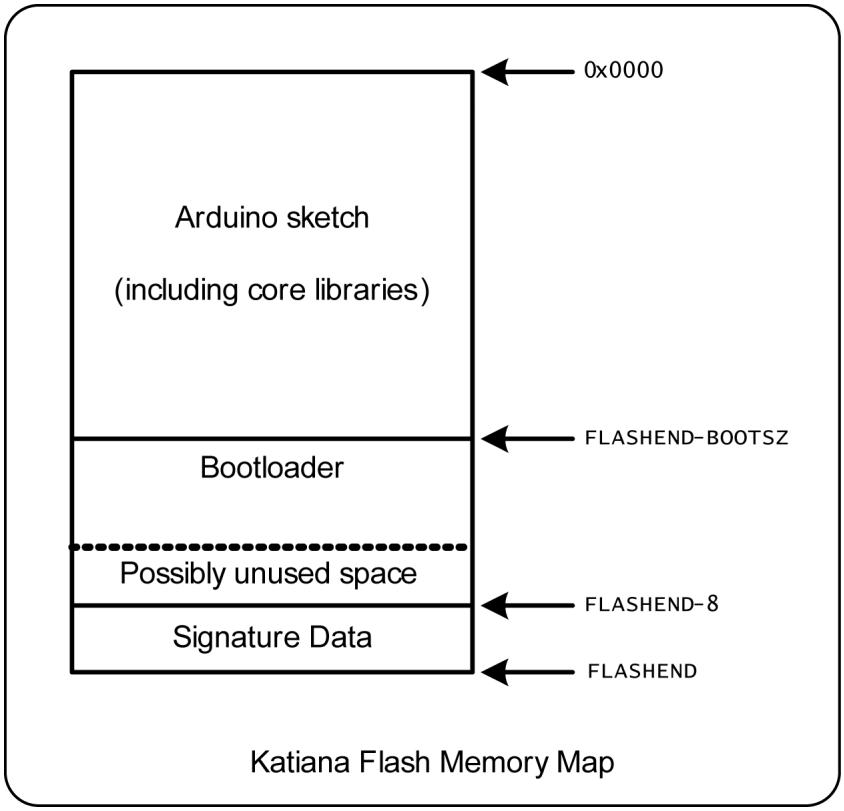
Device Memory Map

The boot sector size is determined by the `BOOTSZ` fuse bits. The makefile must be manually edited to indicate the following:

- Total amount of flash in kB
- Size of boot sector programmed in BOOTSZ fuses in kB.

If these are not accurately specified, the bootloader won't function.

The bootloader is always located at the start of the boot section. There may be some unused space between the end of bootloader and signature bytes, depending on boot section size and which bootloader options are enabled.



Known Issues:

The bootloader has only been tested on ATmega32U4 MCUs.

While this should work with other MCUs having flash sizes up to 128kB and internal USB hardware, only the ATmega32U4 MCU has been verified to work.

Project Options

Optional behavior and AVR910 protocol support can be enabled in the `Config/AppConfig.h` file. These are all just simple define's except as otherwise noted.

Define Name:	Location:	Description:
ENABLE_SECURITY_CHECKS	AppConfig.h	Guards against invalid flash and EEPROM addresses.
ENABLE_LED_SUPPORT	AppConfig.h	Enable da blinky lights.
LED_DATA_FLASHES	AppConfig.h	Flash LEDs on receipt and transmission of data.

LED_START_FLASHES	**AppConfig.h**	Number of flashes of the "L" LED at bootloader startup.
ENABLE_BLOCK_SUPPORT	**AppConfig.h**	Memory block read/write support. Must be enabled for Arduino uploads.
ENABLE_EEPROM_BYTE_SUPPORT	**AppConfig.h**	EEPROM memory byte read/write support.
ENABLE_FLASH_BYTE_SUPPORT	**AppConfig.h**	Flash memory byte read/write support.
ENABLE_LOCK_BYTE_WRITE_SUPPORT	**AppConfig.h**	Lock byte write support. Not Recommended.
CUSTOM_USB_SERIAL	**AppConfig.h**	Bootloader will enumerate with a hardware serial number on USB. Actual S/N is specified elsewhere.

It is highly recommended to enable the security check option. Without it, users can write to the bootloader flash section, or specify invalid addresses outside the valid range, resulting in writes to memory at unintended locations. While this should not happen with the Arduino IDE, if it does happen it will *brick* the bootloader and it will have to be re-programmed through the ISP port. Is it worth the risk?

Hardware Configuration

Some settings should also be checked in the makefile, and adjusted to match the hardware in which the bootloader will be installed.

Define Name:	Location:	Description:
MCU	makefile	Set this to your MCU (e.g. atmeta32u4).
F_USB, F_CPU	makefile	Set these to the input clock frequency (F_USB) and (optionally) prescaled system clock frequency (F_CPU) in Hertz. Note that F_USB must be either 8 or 16MHz or a compile error will be thrown.
FLASH_SIZE_KB	makefile	Set this to the size of flash memory in your MCU.
BOOT_SECTOR_SIZE_KB	makefile	Set this to the size of boot sector programmed inot the MCU's fuses.

Programming Conventions

Katiana is written in C. In general, the following conventions are used:

- Function names are Pascal case, like this: `PascalCase`.
- Variable names are Camel case, like this: `camelCase`.
- Macro names are all caps with underscores between words, like this: `MACRO_NAME`.
- In a few places, macros which can be used like a variable use Camel case.

Software License

The original source for this bootloader was BootloaderCDC.c licensed as follows by Dean Camera:

This heavily modified version of the original source is licensed by aweatherguy as follows:

> LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.