Katiana Bootloader for Arduino (version 2.1)

Overview

Katiana is a bootloader for Atmel AVR microcontrollers with built-in USB peripherals, which:

- enumerates on USB as a CDC Class device (aka virtual serial port),
- implements a subset of the AVR109 protocol, and
- is compatible with the Arduino IDE for uploading and running sketches.

Out of the box, it builds for the ATmega32U4 and fits into a 4KB bootloader section size. It should also be compatible with other Atmel AVR MCUs.

Why Another Bootloader?

Katiana is a step in the evolution of the *Caterina* bootloader written by Dean Camera and distributed with his LUFA software library. Caterina was modified by Arduino for use as the bootloader on some of their boards incorporating MCUs with built-in USB hardware (such as Leonardo and LilyPad USB). The original boot logic was rewritten by the Arduino folks, but the rest of Caterina was little changed.

Two main issues resulted in this author turning Caterina into Katiana:

- There was no provision for system clock prescaling other than 1:1.
- The bootloader does not enumerate on USB with a serial number.

Changes and Improvments

The Katiana bootloader is based on the Catiana loader distributed with Arduino releases as late as 1.8.4 (and maybe later). Here's a list of the more significant changes:

- Updated to build with LUFA version 170418.
- Now supports system clock prescaling (other than 1:1).
- 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.
- Sketches can get USB VID/PID from bootloader instead of boards.txt file.
- Optional address validation protects against uploads which would overwrite the bootloader.
    - Useful if BL1 lock bits are not set to protect boot sector.
- Communications between bootloader and sketch cannot be accidentally interfered with by the sketch.
- Various changes to reduce compiled flash image size.

Compatibility

This bootloader was specifically developed 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. It will *probably* also work with other ATmega MCUs with built-in USB peripherals and 128kB or less of flash memory.

Katiana responds to a subset of the Atmel AVR910 command protocol sufficient to support uploads from the Arduino IDE (through avrdude). Support for various subsets of AVR910 commands may be configured in the `AppConfig.h` file.

USB Interface Configuration

The bootloader configures the MCU's USB interface as follows:

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 flowchart below. This provides two 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 – the bootloader will remain running until a sketch has been uploaded.

Sketch Present?

When MCU flash has been erased and only the Katiana bootloader has been installed, memory location zero (the reset vector) will contain all ones (0xFFFF). When any sketch is loaded, it writes a word at location zero which will never be all ones. Examining the contents of the location in flash is therefore sufficient to determine if a sketch is present.

Power-on and Brown-out Resets

When these resets occur, and as long as there's a sketch loaded, the bootloader starts it without delay. If there's no sketch, the bootloader begins accepting AVR910 commands until a sketch has been loaded.

WDT Resets

This is one of the paths into the bootloader, and is the mechanism through which the Arduino IDE normally 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. The bootloader interferes with WDT reset events and this could make it more difficult for sketches to make use of WDT resets for their own purposes. Startup logic is designed to minimize the effect of this interference.

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 starting the sketch. The pre-defined location is SRAM is calle the *BootKey*, and when it contains 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 make a copy of the BootKey, before it is overwritten by the bootloader's own startup code.

External Resets

The first path into the bootloader is not available if the sketch disables the USB hardware (either accidentally or on purpose). A second path into the bootloader is provided to deal with this situation.

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 renders the CDC serial port non-functional. The user simply taps the reset button twice in less than 750ms to enter the bootloader. They then have a couple of seconds in which to begin an upload from the IDE.

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.

Sketch-Induced Resets

The bootloader startup logic presents yet one more possible way to get into the bootloader. The sketch itself can do the same thing that the core does when the serial port is touched at 1200 baud – activate the BootKey and then force a WDT reset. This all looks the same to Katiana and it will dutifully run the bootloader.

Arduino IDE Upload Process

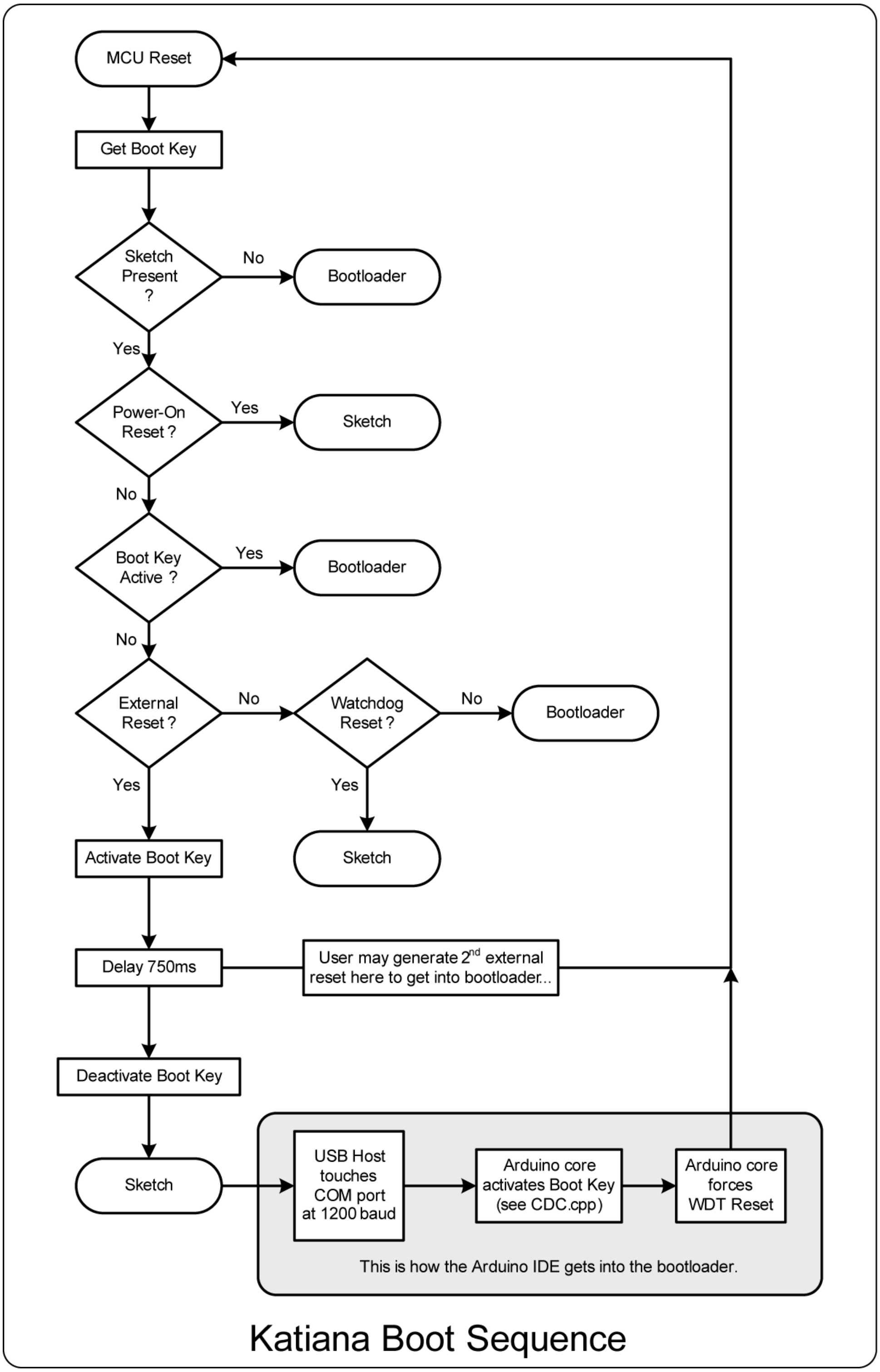
Many Arduino boards include a separate hardware component which implements the USB interface. On these boards it is possible for the host computer (e.g. Windows or Linux) to force a hardware reset on the Atmel MCU – and this provides an entry into the bootloader to upload a new sketch. These boards typically use the Optiboot bootloader.

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.

The CDC serial port exposed by the bootloader enumerates with a differnt USB product ID (aka PID) than does the serial port exposed by the sketch. The bootloader PID is set in **`Descriptors.c`** and might be `0x9207` for example. The sketch PID is set in the Arduino boards.txt file and might be `0x9208`. Because of this difference, 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`. This is part of the design of the upload process and it requires that PIDs be coordinated between the bootloader and the boards.txt file.

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

The bootloader can be configured to enumerate on USB with a custom hardware serial number. Without this feature (on the Windows OS 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. This doesn't happen with Arduino boards (such as the Uno) which report serial numbers. Having the bootloader report a serial number fixes this – the same board will always enumerate with the same COM port number.

For the to be truly useful, the sketch must also enumerate with a serial number for the same reason. Katiana makes the bootloader's serial number available to the sketch so that it may report a value without having to directly encode a value into every sketch. See the API section below for more on this.

There is an appendix at the end of this document which discusses the question of how to pick a serial number.

Enabling Custom USB S/Ns

To enable custom S/Ns, define the macro `CUSTOM_USB_SERIAL` in the **`AppConfig.h`** header file. The actual serial number the bootloader will report needs to be placed in the file **`UsbHdwrSerial.h`** prior to building the bootloader, and should look someithing like this example:

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

This file is not supplied with the Katiana package and must be created by the user. Technically, the serial number string could contain any Unicode characters in any language, but Katiana only supports ASCII characters and reports only an English language descriptor. See the appendix for more on restrictions imposed on USB serial numbers.

To keep the board on the same COM port, the serial number would need to be kept constant if the bootloader is ever re-flashed. It might be useful to put a physical label on each such board with the assigned serial number for this purpose.

Why a Separate File?

Some may wonder why the serial number is in a separate header file instead of being defined in the **`AppConfig.h`** file. Having the separate file allows for automated tools to automatically generate new serial numbers without needing to edit **`AppConfig.h`** every time.

Serial Number Validation

Some simple restrictions on serial numbers are applied during the build process. A check for valid ASCII charcters and loose limits on string length is about it.

As described in the appendix, it may make sense to place more tight restrictions on user-defined USB serial numbers. The restrictions suggested there may be imposed by defining the macro `STRICT_USB_SERIAL` in the **`AppConfig.h`** file. To allow unrestricted serial numbers, comment out the definition of `STRICT_USB_SERIAL` in **`AppConfig.h`**.

If strict validation is enabled, the S/N must be between 12 and 126 characters long and may only contain upper case hexadecimal characters (0-9, A-F). See the appendix for more about the requirement for unique numbers.

These restrictions are enforced during the build by a small program defined in **`ValidateUsbHdwrSerial.c`**. This program is compiled during the build using the native C compiler (`cc`). It can be easily modified to support a different set of validation rules.

Building Katiana

The distribution has been built and tested with Cygwin.

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. If custom USB serial numbers are enabled, create/edit the file **`UsbHdwrSerial.h`** with the desired USB hardware serial number. The script `NewUsbHdwrSerial.sh` may be used for this purpose if desired.
9. Edit **`Board/LEDs.h`** to match the LED assignments on your board (if any). If all LED options are disabled in **`AppConfig.h`** this step may be skipped.
10. Run `make`.
11. To generate HTML content in the `Documentation` folder, run `make doxygen`.
12. On Windows, edit `Katiana-Bootloader.inf` and change the USB VID/PID specified there to match the VID/PID combinations specified in **`Descriptors.c`** and in the `Boards.txt` file that will be used for building Arduino sketches. This step is *not* required if a pre-existing Arduino VID/PID combination is being used.

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

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.

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 with a freshly flashed bootloader, and they will be directed to the bootloader's COM port from the IDE.

After the first upload, the sketch will appear on a different port. The IDE's port setting must be changed to this new port for subsequent uploads. If uploads to the new port are failing, try adding the custom board with build flags as shown below. When this fails, it is usually caused by the the core (`CDC.cpp`) using the wrong address for the BootKey.

```
<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.

Custom USB Descriptors

See the API section for more on how to get the sketch to use the bootloader's USB device descriptor.

If the custom USB serial number option is enabled in the bootloader it is possible for the sketch to read the bootloader's USB serial number and use it as well. To do this, copies of `USBCore.cpp` and **`USBAPI.h`** in the `Arduino-Core` folder of this distribution must be copied into the Arduino cores subdirectory. You may wish to make backup copies of the original files before replacing them. The core files provided in this distribution are valid for Arduino version 1.8.4 and may need to be modified to work with future Arduino versions.

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

This section is mostly for background. Arduino IDE users may skip over it to start with.

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.

Arduino

No additional changes should be required for uploads to work from the Arduino IDE. It is also possible to upload both flash and EEPROM data in the same operation with a modification to the IDE. A custom `platform.txt` file is required for this purpose. Place it in a subdirectory named `hardware/breadboard/avr` relative to the sketch's parent directory.

Make a copy of the `tools.avrdude` section in the standard `platform.txt` file and rename it – to `eepdude` for example. Then modify the arguments on the lines `tools.eepdude.upload.pattern` and `tools.eepdude.upload.pattern` to include this:

> `"-Ueeprom:w:{build.path}/{build.project_name}.eep:i"`

A custom board will also need to be defined in a `boards.txt` file which specifies the new `eepdude` programmer, like this:

> `[board name].upload.tool=eepdude`

The only issue with this setup is that uploading a sketch which does not define any EEPROM data may cause avrdude to report an error.

API for Sketches

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

The API also provides access to custom USB descriptors programmed into the bootloader. 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_USB_SERIAL        (uint8_t *)(KATIANA_TABLE_START + KATIANA_TABLE_SIZE - 6)

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

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

With the exception of `BOOTKEY`, these addresses 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 );
```

The Arduino core looks at the boot signature to determine where the BootKey is located.

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 Descriptor API

With a modified Arduino core file (`USBCore.cpp`) it is possible to automatically coordinate the sketch's USB VID, PID and serial number with the values programmed into the bootloader.

Note that as long as 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.

Arduino Core Modification

This is accomplished by defining global weak functions to obtain a custom device and serial number descriptors. A **"weak"** version of this function in the core returns a default result, indicating that there are no custom descriptors available. In this case, the core behavior is unchanged.

The sketch may override these functions, providing a copy of the descriptors from 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 is defined. This scheme does not require any special compile time flags.

Here's an example of what the modifications to USBCore.cpp might look like to implement this behavior. First, the weak functions that can be overridden in a sketch:

```
//
uint8_t USB_GetCustomSerialNumber(const uint8_t **buf);
uint8_t USB_GetCustomDescriptor(const uint8_t **buf);
//
uint8_t __attribute__ ((weak)) USB_GetCustomSerialNumber(const uint8_t **buffer)
{
    if (!buffer) return 0;
    *buffer = 0;
    return 0;
}
uint8_t __attribute__ ((weak)) USB_GetCustomDescriptor(const uint8_t **buffer)
{
    if (!buffer) return 0;
    *buffer = 0;
    return 0;
}
//
```

And the modified `SendDescriptor()` function:

```
//
bool SendDescriptor(USBSetup& setup)
{
    ...existing code omitted...
    //
    if (USB_DEVICE_DESCRIPTOR_TYPE == t)
    {
        uint8_t freeit = USB_GetCustomDescriptor(&desc_addr);
        if (desc_addr)
        {
            ret = USB_SendControl(0, desc_addr, *desc_addr);
            if (freeit) free((void *)desc_addr);
            return ret;
        }
        else
        {
            desc_addr = (const u8*)&USB_DeviceDescriptorIAD;
        }
    }
    //
    ...more code omitted...
        else if (setup.wValueL == ISERIAL)
        {
            uint8_t freeit = USB_GetCustomSerialNumber( &desc_addr );
            if (desc_addr)
            {
                ret = USB_SendControl(0, desc_addr, *desc_addr);
                if (freeit) free((void *)desc_addr);
                return ret;
            }
        #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...
//
```

Custom USB Device Descriptor

`KATIANA_USB_DESCRIPTOR` contains the address in bootloader flash of the bootloader's USB device descriptor. This is useful for determining the USB PID/VID used by the bootloader. Often, the sketch will want to use the same VID and alter the PID (increment by one for example).

The modified Arduino core files contain a weak function for obtaining the sketch's device descriptor. Normally they do nothing, but can be overridden in the sketch to fetch the descriptor from bootloader flash. The file **`USB_CustomDescriptors.h`** in the `Arduino-Sketch` directory contains an example, reporduced here.

Both this example and the serial number retrieval below use this common function:

```
void __attribute__((noinline)) CacheDescriptor( uint8_t **sram, uint16_t progPtr )
{
    if (*sram) return;
    uint16_t prog = pgm_read_word(progPtr); // read pointer to descriptor from PROGMEM
    if (prog == 0x0000u || prog == 0xffffu) return; // not a valid pointer

    uint8_t cnt = pgm_read_byte( prog );     // 1st byte of descriptor is always length
    *sram = (uint8_t *)malloc(cnt);
    if (! *sram) return;
    uint8_t *dest = *sram;
    while (cnt--) *dest++ = pgm_read_byte( prog++ );
}
```

This shows how to obtain the bootloader's device descriptor. Notice that the Product ID is incremented by one so that the sketch reports a different value than the bootloader. Using this scheme, it is unfortunately still necessary to enter the correct VID/PID values in the `boards.txt` file used by the Arduino IDE.

```
uint8_t USB_GetCustomDescriptor(uint8_t **buf)
{
    if (!buf) return 0;
    *buf = 0;
    CacheDescriptor(buf, FLASHEND - 7);
    if (*buf)
    {
        // increment PID by one (or could toggle if desired).
        ((DeviceDescriptor *)buf)->idProduct += 1;
        return 1;   // non-zero indicates that caller must free memory when finished with it.
    }
    return 0;
}
```

Custom USB Serial Number

If the option is enabled, `KATIANA_USB_SERIAL` contains the address in bootloader flash of the USB serial number string descriptor. That can be accessed as shown below, overriding the core's weak function.

```
uint8_t USB_GetCustomSerialNumber(uint8_t **buf)
{
    if (!buf) return 0;
    *buf = 0;
    CacheDescriptor(buf, FLASHEND - 5);
    return (*buf) ? 1 : 0;   // non-zero indicates that caller must free memory when finished with
        it.
}
```

It returns the same serial number as the bootloader; this is okay since the sketch uses a different PID.

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. The compiler/liner arrange for functions in the `.init0` section to be automatically called prior to any other initialization code being executed. The first of these will have access to the un-sullied value of R2. Other than having the `.init0` section attribute, the function 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 ...
}
//
```

Here's a good question: what happens if multiple functions are declared in seciton `.init0`? Which one will run first? If the function to save `r2` is not first, there could be problems.

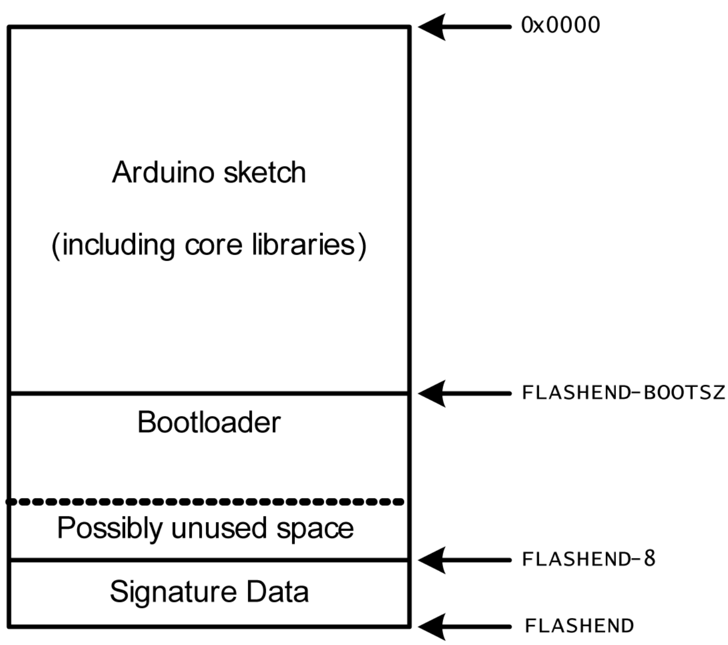
Flash Memory Map

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. If the bootloader is too large for the declared boot sector size, it will overlap the signature table and the linker will fail.



Katiana Flash Memory Map

Changes/Bug Fixes

Changes from the previous bootloader (Arduino version of Caterina) and fixes for bugs are listed here.

Atmel data sheets are wrong about flash page size.

More than one such document states that listed page sizes are words. This is *wrong*. For example, the true flash page size for the ATmega32U4 is 128 bytes – not 128 words as is claimed in the data sheet.

SPM Operations w/Interrupts Enabled

Caterina executes self-programming sequences with interrupts enabled. This is a bug and could result in failure of programming operations, and has been fixed.

Brown-Out Reset Action

The previous design would always enter the bootloader on a brown-out reset, but immediately jump to the sketch on a poer-on reset. This seemed inconsistent, and brown-out resets are now treated identically to power-on resets – the sketch is immediately started. To change this back to the way it was, see the function `SketchStartLogic()` in `Katiana.c`.

Timer 1

Timer 1 was previously run in count-up mode with overflow interrupts enabled. It is now run in CTC (clear on compare match) mode with compare match interrupts enabled. This makes it possible to achieve a relatively constant interrupt rate regardless of the MCU input clock frequency.

32-bit Variables Removed

Doing math with 32-bit variables is expensive. Because the AVR910 command set limits (word) addresses to 16 bits, Katiana does all of its address math with unsigned 16-bit variables. It takes a bit more work (dealing with RAMPZ) to do this, but the end result is a significant reduction in flash image size.

AVR910 Chip Erase Command

Support of this command is now optional.

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.

Only block read/write has been tested.

Options for flash and EEPROM byte support and lock byte writing have not been tested at all. They are a direct copy from the original Caterina bootloader.

Project Options

Optional behavior and subsets of AVR910 protocol support can be configured in the `Config/AppConfig.h` file. The contents of this file in the distribution have sufficient options enabled for Arduino IDE uploads to work. 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.
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_FLASH_ERASE_SUPPORT	AppConfig.h	Enables chip erase command.
ENABLE_LOCK_BYTE_WRITE_SUPPORT	AppConfig.h	Enables writing to the lock byte.
CUSTOM_USB_SERIAL	AppConfig.h	Bootloader will enumerate with a hardware serial number on USB.
STRICT_USB_SERIAL	AppConfig.h	Strict tests applied to USB serial number.
USB_VID	AppConfig.h	The Vendor ID to be enumerated.
USB_PID	AppConfig.h	The Product ID to be enumerated.

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	makefile	Input clock frequency on XTAL pin(s). Optional, default is F_CPU.
F_CPU	makefile	System clock frequency in Hz (after optional prescaling).
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.
- Indentation levels are four spaces and tab characters are not generally used.

Line Endings

The author edits many of the source files in VisualStudio and they may contain either LF or CR/LF line endings. It is recommended to use git's `autocrlf` option to manage this.

Appendix: USB VIDs and PIDs

If this bootloader is being installed on an existing Arduino board such as Leonardo or LilyPad USB, it might be okay to keep the VID and PID already assigned to the board. The board is still manufactured by the same vendor, and a serial number is simply being added.

If this is a custom board, the waters become a bit murky. First there's the legal issues, exemplified by this comment in Arduino's `Makefile` for the Caterina bootloder:

```
#
# USB vendor ID (VID)
# reuse of this VID by others is forbidden by USB-IF
# official Arduino LLC VID
# VID = 0x2341
#
```

Technically, using a VID you don't have rights to is illegal even it is only for personal use. If you want your own VID along with the rights to all 65,536 associated PIDs it will cost you at least (US)$5000. Yeah, that works for me. There are other alternatives available for both personal and commercial developers which may be found by performing an internet search on "usb vid pid for development work".

That said, some people may decide to reuse the VIDs/PIDs from an existing board. Legal issues aside, what other ramifications are there to doing this?

First of all, selecting a VID which is not currently assigned to anyone may not work. Some operating systems may not be happy if the board enumerates with a Vendor ID which has not been officially assigned and published by the USB-IF.

Valid USB Serial Numbers

The base USB standard does not specify limitations on the S/N length or what characters it may contain. It only requires a UTF-16 lil' endian Unicode string. For Katiana, the serial number must be 126 characters or less and only ASCII characters between 0x20 and 0x7E inclusive are allowed. The makefile will apply a test for these restrictions and the build fails if the S/N is not valid.

The standard for the USB Mass Storage Class / Bulk-Only Transport (Revision 1.0, Sept 31, 1999) is much more restrictive. The option `STRICT_USB_SERIAL` applies additional tests in keeping with that standard:

- The serial number must be at least 12 characters long.
- It must contain only upper case hexadecimal characters (0-9 and A-F).

This standard also requires that the 80-bit value comprised of VID (16-bits), PID (16-bits), and last 12 characters of S/N (effectively 48 bits) be unique. There is no way for the build to enforce this requirement.

The bootloader stores the serial number both in flash (as an ASCII string), and SRAM (as a Unicode string). Long serial numbers will consume a significant amount of both memories. The serial number location in flash is made available to sketches.

Unique USB Serial Numbers

To start with, if your are building commercial products for sale and have an officially assigned VID/PID combination, concerns raised below about unique serial numbers are moot.

Consider the requirement that the equivalent 80 bits of VID-PID-S/N need to be unique. Heck, since this *isn't* a mass storage product, the requirement may only be that the VID-PID-and entire serial number be unique. The worst case scenario is that lots of hobbyists would use the same VID/PID.

What are the odds that two such boards wind up with the exact same serial number? And what would happen if they did?

In theory, duplication of serial numbers is a bad thing that should never be allowed to happen and all folks should have their own VID/PID numbers to avoid this. In the real world, for hobbyists this isn't practical. If there do happen to be two different Arduino boards which have the exact same VID-PID-S/N this won't really be a problem as long as they are not plugged into the same USB host (i.e. PC). The bottom line is that as long as your personal use and distribution of boards is limited in scope, this is *not* necessarily a huge problem.

Pseudo-Random Serial Numbers

If you just start assigning numbers like "000000000001", "000000000002" and so on, chances are someone else has done this too. A very good way to mitigate the risks is to assign semi-random 48-bit (or longer) serial numbers to each unit. A shell script is included with the bootloader package which generates pseudo-random serial numbers (with lots of semi-meaningless overkill). It gets the serial number from a cryptographic hash of some semi-random text, including:

- 16 bytes of pseudo-random values from /dev/random
- Current date and time
- Long listing of the current directory with lines randomly shuffled
- Some extra "salt"

```
#!/bin/sh
salt='Salt of the Earth'  # change this as you please
hash=md5sum  # can change to sha256sum or other hash commands known to the shell
sum=`( od -N 16 -t x /dev/random ; date ; ls -l | shuf ; echo "$salt" ) 2>&1 | $hash`
serial=${sum:0:12}
serial=`echo $serial | tr [:lower:] [:upper:]`
echo \#pragma once >NewUsbHdwrSerial.h
echo \#define USB_HDWR_SERIAL \"$serial\" >> NewUsbHdwrSerial.h
#
```

The salt can be modified by each individual to further reduce odds of serial number duplication. If you use this or a similar technique, the odds of you having the same serial number as anyone else are extremely small.

The example above generates 12-character serial numbers and this should be more than adequate for most purposes. Longer serial numbers can be generated by taking a bigger subset of the hash (as many as 96 charactes with sha384), and this will further decrease the odds of ever having a duplicate serial number. That can done by changing the `${sum:0:12}` on the fourth line of the shell script above to `${sum:0:32}` to get a 32-character serial number for example.

The shell script above is provided with the Katiana distribution and is named `NewUsbHdwrSerial.sh`.

Appendix: 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: