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. Depending on options enabled, the build may exceed 4kB, but the options required for compatibility with the Arduino IDE will fit a 4kB boot sector.

When the bootloader is running, the board's RX and TX LEDs can optionally flash as data is being received from and sent to the CDC interface. The "L" LED can also be configured to flash a fixed number of times at bootloader startup.

Benefits

The Katiana bootloader offers some benefits compared 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.
- Bootloader no longer tries to execute sketch when none is present.

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. 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) to work. See the source code (`Katiana.c`) for detail on which AVR910

commands are supported.

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 flowchart below. Theis provides 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 – 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 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. 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*.

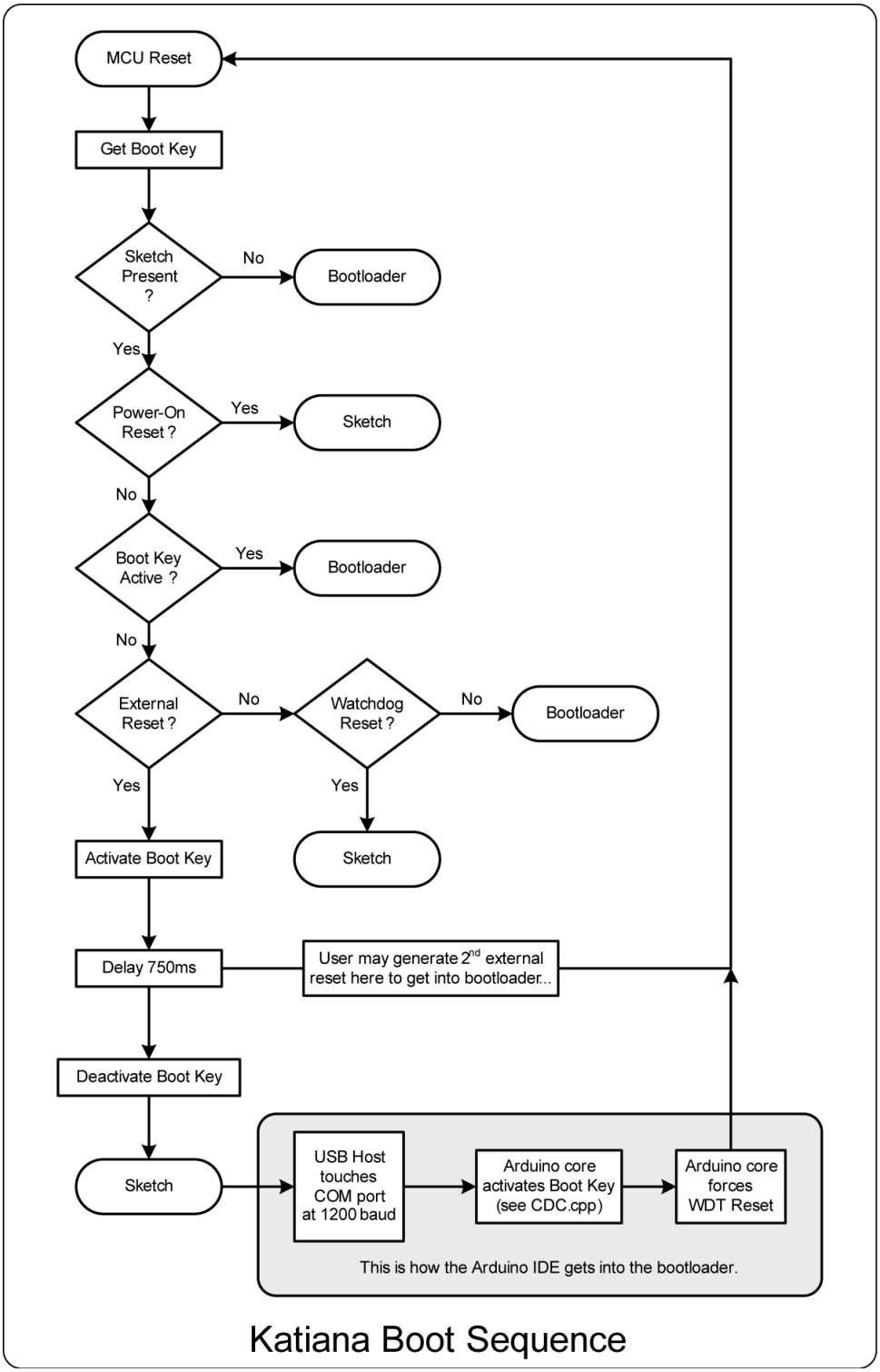
Beginning with IDE versions 1.6, the BootKey location is `(RAMEND-1)`. When a sketch is running this normally contains the return address for the **`main()`** function. Since **`main()`** never returns this is a safe place to pass a value to the bootloader. The bootloader has some code in its `.init0`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

External Resets

The 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.

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. 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.

To understand this process, one should know that 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.

USB VIDs and PIDs

If this bootloader is being installed on an existing Arduino board such as Leonardo or LilyPad USB, you should just keep the VID and PID already assigned to the board. End of story.

If this is some new thing, the waters become a bit murky. For development work, or if you're just making one-off hobby boards for your own use, reusing the VIDs/PIDs from Arduino boards is probably okay. 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".

Some operating systems may not be happy if the board enumerates with a VID/PID combination which has not been officially assigned and published by the USB folks.

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.

Valid USB Serial Numbers

The base USB standard does not specify limitations on the S/N length or what characters it may contain. The standard for the USB Mass Storage Class / Bulk-Only Transport (Revision 1.0, Sept 31, 1999) is much more restrictive. The recommendations adopted here comply 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).
- The 80-bit value comprised of VID (16-bits), PID (16-bits), and last 12 characters of S/N (effectively 48 bits) must be unique.

This driver enforces the minimum length and character value restrictions. Serial numbers longer than 126 characters (508 equivalent bits) are also prohibited.

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 entire serial number be unique (if longer than 12 characters). It is likely that lots of hobbyists would use this bootloader without modifying the VID/PID since they are valid numbers. This might even be legal since all they would be doing is adding a serial number to an official Arduino product. Technically however, it should be Arduino assigning the serial numbers to guarantee uniqueness, and not the end user.

What are the odds that two such boards wind up with the exact same serial number? And what would happen if they did? First, let's agree to skip over the question of legal issues.

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 80-bit (or longer) ID 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.

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.

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. For example a shell script is included with the bootloader package which generates pseudo-random serial numbers. It uses the MD5 hash of some 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
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`.

Enabling Custom USB S/Ns

The serial number needs to be placed in the file **`UsbHdwrSerial.h`** prior to building the bootloader, and should look like this:

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

The string must be between 12 and 126 characters long and may only contain upper case hexadecimal characters (0-9, A-F). Do not use the exact serial number shown above! See above for more on the requirement for unique numbers.

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. If custom USB serial numbers are enabled, create the file **`UsbHdwrSerial.h`** with the desired USB hardware serial number. The script `NewUsbHdwrSerial.sh` may be used for this purpose if desired.
9. Run `make`.
10. 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 must 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.

Custom USB Serial Numbers

If this option is enabled in the bootloader, 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.

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

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`

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) 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
//
```

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 );
```

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 serial number as an ASCII, null-treminated string.

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.

Two modifications are required to have the sketch enumerate with a serial number. Sample files with examples of these changes are included with the Katiana package.

Arduino Core Modification

With a modified Arduino core file (`USBCore.cpp`) it is possible to have the sketch report the same serial number 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.

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...
//
```

Sketch Modifications

The second modification is to add a function to the sketch which overrides the weak function that was added to the Arduino core above. Here is an example of that function:

```
//
uint8_t *usbSerial;
uint8_t usbSerialLength;
uint8_t usbSerialCached;
// ===> this overrides the weak function in USBCore.cpp <===
uint8_t USB_GetCustomSerialNumber(uint8_t **Buffer)
{
    if (! Buffer) return 0;
    *Buffer = usbSerial;
    if (usbSerialCached) return usbSerialLength;
    usbSerialCached = 1;
    // usbSerialLength is still zero from startup init here.
    uint16_t snptr = pgm_read_word(FLASHEND-7);         // get address of S/N
       string in flash
    if (snptr == 0x0000u || snptr == 0xffffu) return 0; // is there a valid address
       there?
    uint16_t p = snptr;
    uint8_t cnt = 0;
    while (cnt < 127)                                   // read string once to get
       the length
    {
        if (pgm_read_byte(p++) == 0) break;
        cnt++;
    }
    if ((cnt < 12) || (cnt > 126)) return 0;
    if ( ! (usbSerial = (uint8_t *)malloc(cnt + 1) ) )  // get memory for copy of
       S/N in SRAM
        return 0;
    *Buffer = usbSerial;
    usbSerialLength = cnt;
    uint8_t k;
    uint8_t *s = usbSerial;
    for (k = usbSerialLength; k; k--)                   // copy string from flash
       to SRAM
    {
     *s++ = pgm_read_byte(snptr++);
    }
    *s = 0;                                             // terminate the string
    return usbSerialLength;
}
//
```

A copy of this function is provided in the `Arduino-Sketch` subdirectory of the Katiana distribution.

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?

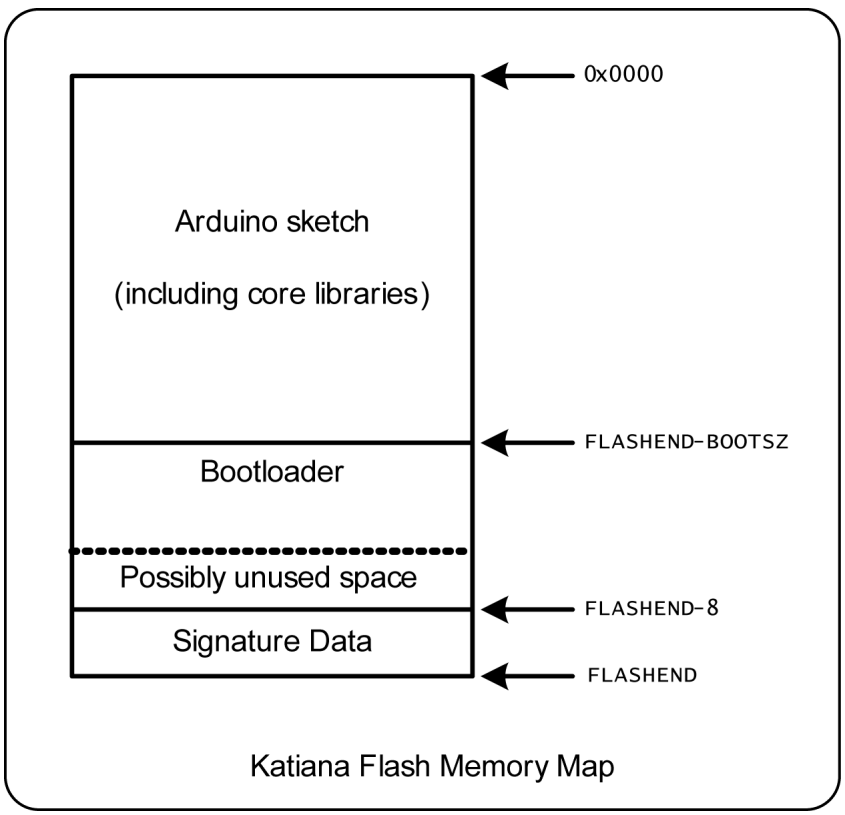
Flash Memory Map

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.



Changes and Bug Fixes

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

Executing Empty Flash

One low-level bug that is fixed in this design occured when the bootloader was first flashed into an erased

MCU. The original design would jump to location zero after eight seconds, and begin executing erased flash memory. Just what the MCU does with erased flash has been much debated but the discussion thread **here** seems to indicate that it either executes every such location, or every other such location, depending on the contents of register `r32`. Either way this was a minor (probably unnoticeable) bug and has been fixed in this bootloader.

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.

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:

> Copyright 2017 Dean Camera (dean [at] fourwalledcubicle [dot] com)
>
> Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
>
> The author disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

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

> Copyright (c) 2017, aweatherguy, all rights reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name "aweatherguy" nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL aweatherguy BE 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.