Features

- USB Protocol
  - Based on the USB DFU class
  - Autobaud (8/16 MHz crystal)
- In-System Programming
  - Read/Write Flash and EEPROM on-chip memories
  - Read Device ID
  - Full chip Erase
  - Start application command
- In-Application Programming
  - Software Entry-points for on-chip flash drivers

1. Description

The 8bits mega AVR with USB interface devices are factory configured with a USB bootloader located in the on-chip flash boot section of the controller.

This USB bootloader allows to perform In-System Programming from an USB host controller without removing the part from the system or without a pre-programmed application, and without any external programming interface.

This document describes the USB bootloader functionalities as well as the serial protocol to efficiently perform operations on the on chip Flash memories (Flash and EEPROM).
2. Bootloader Environment

The bootloader is located in the boot section of the on-chip Flash memory, it manages the USB communication protocol and performs read/write operations to the on-chip memories (Flash/EEPROM).

The USB bootloader is loaded in the “Bootloader Flash Section” of the on-chip Flash memory. The size of the bootloader flash section must be larger than the bootloader size. USB products are factory configured with the default on-chip USB bootloader and the required bootsection configuration.

Table 2-1. USB Bootloader Parameters

<table>
<thead>
<tr>
<th>Product</th>
<th>Flash Bootsection Size Configuration</th>
<th>VID / PID</th>
<th>Bootloader Start Address (word address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT90USB1287</td>
<td>4 KWord</td>
<td>0x03EB / 0x2FFB</td>
<td>0xf000</td>
</tr>
<tr>
<td>AT90USB1286</td>
<td></td>
<td>0x03EB / 0x2FF9</td>
<td>0x7800</td>
</tr>
<tr>
<td>AT90USB647</td>
<td>2 KWord</td>
<td>0x03EB / 0x2FFA</td>
<td>0x1800</td>
</tr>
<tr>
<td>AT90USB646</td>
<td></td>
<td>0x03EB / 0x2FF7</td>
<td>0x0800</td>
</tr>
<tr>
<td>AT90USB162</td>
<td></td>
<td>0x03EB / 0x2FF4</td>
<td>0x0800</td>
</tr>
<tr>
<td>ATmega32U4</td>
<td></td>
<td>0x03EB / 0x2FF3</td>
<td>0x0800</td>
</tr>
<tr>
<td>ATmega16U4</td>
<td></td>
<td>0x03EB / 0x2FF3</td>
<td>0x0800</td>
</tr>
</tbody>
</table>

Figure 2-1. Physical Environment

3. Bootloader Activation

As specified in the AT90USB datasheet, the bootloader can be activated by one of the following conditions:

- Regular application execution: A jump or call from the user application program. This may be initiated by a trigger such as a command received via USB, USART or SPI and decoded by the application.
• **Boot Reset Fuse** The Boot Reset Fuse (BOOTRST) can be programmed so that the Reset Vector points to the Boot Flash section start address after reset. Once the user code is loaded, a bootloader command (“start application”) can start executing the application code. Note that the fuses cannot be changed by the MCU itself. This means that once the Boot Reset Fuse is programmed, the Reset Vector will always point to the Bootloader Reset and the fuse can only be changed through the serial or parallel programming interface. The BOOTRST fuse is not active in the default factory configuration.

• **External Hardware conditions** The Hardware Boot Enable fuse (HWBE) can be programmed so that upon special hardware conditions under reset, the bootloader execution is forced after reset.

These different conditions are summarized in **Figure 3-1 on page 3**.

**Figure 3-1.** Boot Process

<table>
<thead>
<tr>
<th>Reset</th>
<th>Coarse Hardware conditions</th>
<th>BOOTRST = 0</th>
<th>PC = boot loader section</th>
<th>PC = 0000h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Software activation (jump)</td>
<td></td>
<td></td>
<td>Application Running</td>
</tr>
</tbody>
</table>

4. **Protocol**

4.1 **Device Firmware Upgrade Introduction**

Device Firmware Upgrade (DFU) is the mechanism implemented to perform device firmware modifications. Any USB device can exploit this capability by supporting the requirements specified in this document.

Because it is unpractical for a device to concurrently perform both DFU operations and its normal run-time activities, these normal activities must cease for the duration of the DFU operations. Doing so means that the device must change its operating mode; i.e., a printer is **not** a printer while it is undergoing a firmware upgrade; it is a PROM programmer. However, a
device that supports DFU is not capable of changing its mode of operation on its own. External (human or host operating system) intervention is required.

4.2 DFU Specific Requests

In addition to the USB standard requests, 7 DFU class-specific requests are used to accomplish the upgrade operations:

Table 4-1. DFU Class-specific Requests

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0001b</td>
<td>DFU_DETACH (0)</td>
<td>wTimeout</td>
<td>Interface (4)</td>
<td>Zero</td>
<td>none</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_DNLOAD (1)</td>
<td>wBlock</td>
<td>Interface (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010 0001b</td>
<td>DFU_UPLOAD (2)</td>
<td>wBlock</td>
<td>Interface (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010 0001b</td>
<td>DFU_GETSTATUS (3)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td></td>
<td>Status</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_CLRSTATUS (4)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>1010 0001b</td>
<td>DFU_GETSTATE (5)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td></td>
<td>State</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_ABORT (6)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td></td>
<td>none</td>
</tr>
</tbody>
</table>

4.3 DFU Descriptors Set

The device exports the DFU descriptor set, which contains:

- A DFU device descriptor
- A single configuration descriptor
- A single interface descriptor (including descriptors for alternate settings, if present)

4.3.1 DFU Device Descriptor

This descriptor is only present in the DFU mode descriptor set. The DFU class code is reported in the bDeviceClass field of this descriptor.

Table 4-2. DFU Mode Device Descriptor

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>12h</td>
<td>Size of this descriptor, in bytes</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>01h</td>
<td>DFU functional descriptor type</td>
</tr>
<tr>
<td>2</td>
<td>bcdUSB</td>
<td>2</td>
<td>0100h</td>
<td>USB specification release number in binary coded decimal</td>
</tr>
<tr>
<td>4</td>
<td>bDeviceClass</td>
<td>1</td>
<td>FEh</td>
<td>Application Specific Class Code</td>
</tr>
<tr>
<td>5</td>
<td>bDeviceSubClass</td>
<td>1</td>
<td>01h</td>
<td>Device Firmware Upgrade Code</td>
</tr>
<tr>
<td>6</td>
<td>bDeviceProtocol</td>
<td>1</td>
<td>00h</td>
<td>The device does not use a class specific protocol on this interface</td>
</tr>
<tr>
<td>7</td>
<td>bMaxPacketSize0</td>
<td>1</td>
<td>32</td>
<td>Maximum packet size for endpoint zero (limited to 32 due to Host side driver)</td>
</tr>
<tr>
<td>8</td>
<td>idVendor</td>
<td>2</td>
<td>03EBh</td>
<td>Vendor ID</td>
</tr>
<tr>
<td>10</td>
<td>idProduct</td>
<td>2</td>
<td>2FFBh</td>
<td>Product ID</td>
</tr>
<tr>
<td>12</td>
<td>bcdDevice</td>
<td>2</td>
<td>0x0000</td>
<td>Device release number in binary coded decimal</td>
</tr>
<tr>
<td>14</td>
<td>iManufacturer</td>
<td>1</td>
<td>0</td>
<td>Index of string descriptor</td>
</tr>
</tbody>
</table>
4.3.2 DFU Configuration Descriptor

This descriptor is identical to the standard configuration descriptor described in the USB DFU specification version 1.0, with the exception that the \( bNumInterfaces \) field must contain the value 01h.

4.3.2.1 DFU Interface Descriptor

This is the descriptor for the only interface available when operating in DFU mode. Therefore, the value of the \( bInterfaceNumber \) field is always zero.

Table 4-3. DFU Mode Interface Description

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>09h</td>
<td>Size of this descriptor, in bytes</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>04h</td>
<td>INTERFACE descriptor type</td>
</tr>
<tr>
<td>2</td>
<td>bInterfaceNumber</td>
<td>1</td>
<td>00h</td>
<td>Number of this interface</td>
</tr>
<tr>
<td>3</td>
<td>bAlternateSetting</td>
<td>1</td>
<td>00h</td>
<td>Alternate setting(^{(1)})</td>
</tr>
<tr>
<td>4</td>
<td>bNumEndpoints</td>
<td>1</td>
<td>00h</td>
<td>Only the control pipe is used</td>
</tr>
<tr>
<td>5</td>
<td>bInterfaceClass</td>
<td>1</td>
<td>FEh</td>
<td>Application Specific Class Code</td>
</tr>
<tr>
<td>6</td>
<td>bInterfaceSubClass</td>
<td>1</td>
<td>01h</td>
<td>Device Firmware Upgrade Code</td>
</tr>
<tr>
<td>7</td>
<td>bInterfaceProtocol</td>
<td>1</td>
<td>00h</td>
<td>The device does not use a class specific protocol on this interface</td>
</tr>
<tr>
<td>8</td>
<td>iInterface</td>
<td>1</td>
<td>00h</td>
<td>Index of the String descriptor for this interface</td>
</tr>
</tbody>
</table>

Note: 1. Alternate settings can be used by an application to access additional memory segments. In this case, it is suggested that each alternate setting employ a string descriptor to indicate the target memory segment; e.g., “EEPROM”. Details concerning other possible uses of alternate settings are beyond the scope of this document. However, their use is intentionally not restricted because the authors anticipate that implements will devise additional creative uses for alternate settings.

4.4 Commands Description

The protocol implemented in the AT90USB bootloader allows to:

- Initiate the communication
- Program the Flash or EEPROM Data
- Read the Flash or EEPROM Data
- Program Configuration Information
- Read Configuration and Manufacturer Information
- Erase the Flash
- Start the application

Overview of the protocol is detailed in “Appendix-A” on page 18.
4.5 Device Status

4.5.1 Get Status

The Host employs the DFU_GETSTATUS request to facilitate synchronization with the device. This status gives information on the execution of the previous request: in progress/OK/Fail/...

The device responds to the DFU_GETSTATUS request with a payload packet containing the following data:

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010 0001b</td>
<td>DFU_GETSTATUS (3)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>6</td>
<td>Status</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_CLRSTATUS (4)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>Zero</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 4-4. DFU_GETSTATUS Response

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bStatus</td>
<td>1</td>
<td>Number</td>
<td>An indication of the status resulting from the execution of the most recent request.</td>
</tr>
<tr>
<td>1</td>
<td>bwPollTimeOut</td>
<td>3</td>
<td>Number</td>
<td>Minimum time in milliseconds that the host should wait before sending a subsequent DFU_GETSTATUS. The purpose of this field is to allow the device to dynamically adjust the amount of time that the device expects the host to wait between the status phase of the next DFU_DNLOAD and the subsequent solicitation of the device's status via DFU_GETSTATUS.</td>
</tr>
<tr>
<td>4</td>
<td>bState</td>
<td>1</td>
<td>Number</td>
<td>An indication of the state that the device is going to enter immediately following transmission of this response.</td>
</tr>
<tr>
<td>5</td>
<td>iString</td>
<td>1</td>
<td>Index</td>
<td>Index of status description in string table.</td>
</tr>
</tbody>
</table>

Table 4-5. bStatus values

<table>
<thead>
<tr>
<th>Status</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>0x00</td>
<td>No error condition is present</td>
</tr>
<tr>
<td>errTARGET</td>
<td>0x01</td>
<td>File is not targeted for use by this device</td>
</tr>
<tr>
<td>errFILE</td>
<td>0x02</td>
<td>File is for this device but fails some vendor-specific verification test</td>
</tr>
<tr>
<td>errWRITE</td>
<td>0x03</td>
<td>Device id unable to write memory</td>
</tr>
<tr>
<td>errERASE</td>
<td>0x04</td>
<td>Memory erase function failed</td>
</tr>
<tr>
<td>errCHECK_ERASED</td>
<td>0x05</td>
<td>Memory erase check failed</td>
</tr>
<tr>
<td>errPROG</td>
<td>0x06</td>
<td>Program memory function failed</td>
</tr>
<tr>
<td>errVERIFY</td>
<td>0x07</td>
<td>Programmed memory failed verification</td>
</tr>
<tr>
<td>errADDRESS</td>
<td>0x08</td>
<td>Cannot program memory due to received address that is out of range</td>
</tr>
<tr>
<td>errNOTDONE</td>
<td>0x09</td>
<td>Received DFU_DNLOAD with wLength = 0, but device does not think it has all the data yet.</td>
</tr>
<tr>
<td>errFIRMWARE</td>
<td>0x0A</td>
<td>Device's firmware is corrupted. It cannot return to run-time operations</td>
</tr>
</tbody>
</table>
4.5.2 Clear Status

Each time the device detects and reports an error indication status to the host in response to a DFU_GETSTATUS request, it enters the dfuERROR state. After reporting any error status, the device can not leave the dfuERROR state, until it has received a DFU_CLRSTATUS request. Upon receipt of DFU_CLRSTATUS, the device sets status to OK and move to the dfuIDLE state. Once the device is in the dfuIDLE state it is then able to move to other states.

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0001b</td>
<td>DFU_CLRSTATUS (4)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>
4.5.3  Device State

The state reported is the current state of the device up to transmission of the response. The values specified in the \textit{bState} field are identical to those reported in DFU\_GETSTATUS.

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010 0001b</td>
<td>DFU_GETSTATE (5)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>1</td>
<td>State</td>
</tr>
</tbody>
</table>

4.5.4  DFU\_ABORT request

The DFU\_ABORT request forces the device to exit from any other state and return to the DFU\_IDLE state. The device sets the OK status on receipt of this request. For more information, see the corresponding state transition summary.

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010 0001b</td>
<td>DFU_ABORT (6)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>0</td>
<td>None</td>
</tr>
</tbody>
</table>

4.6  Programming the Flash or EEPROM Data

The firmware image is downloaded via control-write transfers initiated by the DFU\_DNLOAD class-specific request. The host sends between \texttt{bMaxPacketSize0} and \texttt{wTransferSize} bytes to the device in a control-write transfer. Following each downloaded block, the host solicits the device status with the DFU\_GETSTATUS request.

As described in the USB DFU Specification, "Firmware images for specific devices are, by definition, vendor specific. It is therefore required that target addresses, record sizes, and all other information relative to supporting an upgrade are encapsulated within the firmware image file. It is the responsibility of the device manufacturer and the firmware developer to ensure that their devices can process these encapsulated data. With the exception of the DFU file suffix, the content of the firmware image file is irrelevant to the host."

Firmware image:

- 32 bytes: Command
- X bytes: X is the number of byte (00h) added before the first significant byte of the firmware. The X number is calculated to align the beginning of the firmware with the flash page. X = start_address [32]. For example, if the start address is 00AFh (175d), X = 175 [32] = 15.
- The firmware
- The DFU Suffix on 16 Bytes.

\textbf{Table 4-7. DFU File Suffix}

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0</td>
<td>dwCRC</td>
<td>4</td>
<td>Number</td>
<td>The CRC of the entire file, excluding dwCRC</td>
</tr>
<tr>
<td>-4</td>
<td>bLength</td>
<td>1</td>
<td>16</td>
<td>The length of this DFU suffix including dwCRC</td>
</tr>
<tr>
<td>-5</td>
<td>ucDfuSignature</td>
<td>3</td>
<td>5 : 44h</td>
<td>The unique DFU signature field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6 : 46h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7 : 55h</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td>bcdDFU</td>
<td>2</td>
<td>BCD</td>
<td>DFU specification number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0100h</td>
<td></td>
</tr>
</tbody>
</table>
4.6.1 Request From Host

4.6.1.1 Write Command

The write command is 6 bytes long. In order to meet with the USB specification of the Control type transfers, the write command is completed with 26 (= 32 - 6) non-significant bytes. The total length of the command is then 32 bytes, which is the length of the Default Control Endpoint.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>idVendor</td>
<td>2</td>
<td>ID</td>
<td>The vendor ID associated with this file. Either FFFFh or must match device’s vendor ID</td>
</tr>
<tr>
<td>-12</td>
<td>idProduct</td>
<td>2</td>
<td>ID</td>
<td>The product ID associated with this file. Either FFFFh or must match the device’s product ID</td>
</tr>
<tr>
<td>-14</td>
<td>bcdDevice</td>
<td>2</td>
<td>BCD</td>
<td>The release number of the device associated with this file. Either FFFFh or a BCD firmware release or version number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0001b</td>
<td>DFU_DNLOAD (1)</td>
<td>wBlock</td>
<td>Interface (4)</td>
<td>Length</td>
<td>Firmware</td>
</tr>
</tbody>
</table>

4.6.1.2 Firmware

The firmware can now be downloaded to the device. In order to be in accordance with the Flash page size (128 bytes), X non-significant bytes are added before the first byte to program. The X number is calculated to align the beginning of the firmware with the Flash page. 

\[
X = \text{start address} \mod \text{Flash page size} = 128
\]

For example, if the start address is 00AFh (175d), 

\[
X = 175 \mod 128 = 47
\]

4.6.1.3 DFU Suffix

The DFU suffix of 16 bytes is added just after the last byte to program. This suffix is reserved for future use.
4.6.1.4 Answers from Bootloader

After each program request, the Host can request the device state and status by sending a DFU_GETSTATUS message.
If the device status indicates an error, the host must send a DFU_CLRSTATUS request to the device.

4.7 Reading the Flash or EEPROM Data

The flow described below allows the user to read data in the Flash memory or in the EEPROM data memory. A blank check command on the Flash memory is possible with this flow.

This operation is performed in 2 steps:

- DFU_DNLOAD request with the read command (6 bytes)
- DFU_UPLOAD request which correspond to the previous command.
4.7.1 First Request from Host
The Host sends a DFU Download request with a Display command in the data field.

```
|--------------------|---------|---------|---------|---------|---------|----------------------------------|
| Id_display_data    | 00h     | 01h     | 02h     | start_address | end_address | Display FLASH Data
| 03h                |         |         |         |         |         | Blank Check in FLASH            |
| 01h                |         |         |         | start_address | end_address | Display EEPROM Data             |
```

4.7.2 Second Request from Host
The Host sends a DFU Upload request.

4.7.3 Answers from the Device
The device sends to the Host the firmware from the specified start address to the specified end address.
4.7.4 Answers from the Device to a Blank Check Command

The Host controller sends a GET_STATUS request to the device. Once internal blank check has been completed, the device sends its status.

- If the device status is "OK":
  the device memory is then blank and the device waits for the next Host request.
- If the device status is "errCHECK_ERASED":
  the device memory is not blank. The device waits for an DFU_UPLOAD request to send the first address where the byte is not 0xFF.

4.8 Reading Configuration Information or Manufacturer Information

The flow described hereafter allows the user to read the configuration or manufacturer information.

4.8.1 Requests From Host

To start the programming operation, the Host sends DFU_DNLOAD request with the Read command in the data field (2 bytes).

```
SETUP
DFU_DNLOAD
OUT
Read_command (2 bytes)
IN
ZLP
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id_read_command</td>
<td>05h</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
<td>Read Bootloader Version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01h</td>
<td></td>
<td></td>
<td></td>
<td>Read Device boot ID1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>02h</td>
<td></td>
<td></td>
<td></td>
<td>Read Device boot ID2</td>
</tr>
<tr>
<td></td>
<td>01h</td>
<td>30h</td>
<td></td>
<td></td>
<td></td>
<td>Read Manufacturer Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31h</td>
<td></td>
<td></td>
<td></td>
<td>Read Family Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60h</td>
<td></td>
<td></td>
<td></td>
<td>Read Product Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>61h</td>
<td></td>
<td></td>
<td></td>
<td>Read Product Revision</td>
</tr>
</tbody>
</table>

4.8.2 Answers from Bootloader

The device has two possible answers to a DFU_GETSTATUS request:

- If the chip is protected from program access, an "err_VENDOR" status is returned to the Host.
- Otherwise, the device status is "OK". The Host can send a DFU_UPLOAD request to the device in order to get the value of the requested field.
4.9 Erasing the Flash

The flow described below allows the user to erase the Flash memory.

The Full Chip erase command erases the whole Flash.

4.9.1 Request from Host

To start the erasing operation, the Host sends a DFU_DNLOAD request with a Write Command in the data field (2 bytes).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id_write_command</td>
<td>00h</td>
<td>FFh</td>
<td></td>
<td></td>
<td></td>
<td>Full chip Erase (bits at FFh)</td>
</tr>
</tbody>
</table>

4.9.2 Answers from Bootloader

The device has two possible answers to a DFU_GETSTATUS request:

- If the chip is protected from program access, an “err_WRITE” status is returned to the Host.
- Otherwise, the device status is “OK”.

4.10 Starting the Application

The flow described below allows to start the application directly from the bootloader upon a specific command reception.

Two options are possible:

- Start the application with an internal hardware reset using watchdog.
  When the device receives this command the watchdog is enabled and the bootloader enters a waiting loop until the watchdog resets the device.

- Start the application without reset.
  A jump at the address 0000h is used to start the application without reset.

To start the application, the Host sends a DFU_DNLOAD request with the specified application start type in the data field (3 or 5 bytes).

This request is immediately followed by a second DFU_DNLOAD request with no data field to start the application with one of the 2 options.
Important note:
The bootloader performs a watchdog reset to generate the “hardware reset” that allows to execute the application section. After a watchdog reset occurs, the AVR watchdog is still running, thus the application should take care to disable watchdog at program start-up (otherwise the application that does not manage the hardware watchdog will run in an infinite reset loop).

4.11 Request From Host

![Diagram of command setup and upload]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id_write_command</td>
<td>04h</td>
<td>03h</td>
<td>00h</td>
<td></td>
<td></td>
<td>Hardware reset</td>
</tr>
<tr>
<td>04h</td>
<td></td>
<td>01h</td>
<td>address</td>
<td></td>
<td></td>
<td>LJMP address</td>
</tr>
</tbody>
</table>

4.12 Answer from Bootloader

No answer is returned by the device.

5. Security

When the USB bootloader connection is initiated, the bootloader automatically enters a read/write software security mode (independent of the product lock bits settings). This allows to protect the on-chip flash content from read/write access over the USB interface. Thus the only DFU command allowed after a USB bootloader connection is a “Full Chip Erase” command.

After this “Full Chip Erase” has been received and properly executed, all DFU commands are allowed, and thus the on-chip flash can be reprogrammed and verified.
6. Accessing the Low level Flash Drivers

The AT90USB USB bootloader is located in the boot section of the on-chip flash memory, meanwhile the bootloader section is the unique memory location allowed to execute on-chip flash memory write operations (SPM instruction is decoded only in this section).

Thus applications which require on-chip flash write access can perform calls to specific entry points located in the USB bootloader.

The USB bootloader provides several Application Programming Interfaces (API) that allows the application to access low level flash drivers located in the boot section. These APIs allow the following operations:

- Page Erase
- Page Write
- Load word in the temporary page buffer

Figure 6-1. USB bootloader API calls
The API are located at absolute addresses in the USB bootloader firmware and accept specific registers values as parameters. These parameters are compatible with a C compiler calling convention and thus can be called directly with function pointer declared as in the example below:

The full assembly code for the flash API drivers is given in "Appendix-B" on page 20.
7. Using the USB bootloader for In System Programming

Flip software is the PC side application used to communicate with the USB bootloader (Flip is available for free on the Atmel website).

For detailed instructions about using Flip and USB bootloader, please refer to AVR282: “USB Firmware Upgrade for AT90USB” (doc 7769).
8. Bootloader History

The following table shows the different bootloader revision and associated changes.

<table>
<thead>
<tr>
<th>Product</th>
<th>Bootloader Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT90USB1287</td>
<td>1.0.1</td>
<td>Initial Revision</td>
</tr>
<tr>
<td>AT90USB1286</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT90USB647</td>
<td>1.0.0</td>
<td>Initial Revision</td>
</tr>
<tr>
<td>AT90USB646</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT90USB162</td>
<td>1.0.1</td>
<td>Allow to use 16MHz cristal with 3.3V power supply and CKDIV8 fuse.</td>
</tr>
<tr>
<td>AT90USB82</td>
<td>1.0.5</td>
<td>Improved USB autobaud process</td>
</tr>
<tr>
<td>ATmega32U4</td>
<td>1.0.0</td>
<td>Initial Revision</td>
</tr>
<tr>
<td>ATmega16U4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9. Appendix-A

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id_prog_start</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init FLASH programming</td>
</tr>
<tr>
<td>01h</td>
<td></td>
<td>start_address</td>
<td>end_address</td>
<td></td>
<td>Init EEPROM programming</td>
<td></td>
</tr>
<tr>
<td>Id_display_data</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Display FLASH Data</td>
</tr>
<tr>
<td>03h</td>
<td></td>
<td>start_address</td>
<td>end_address</td>
<td></td>
<td>Blank Check in FLASH</td>
<td></td>
</tr>
<tr>
<td>01h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Display EEPROM Data</td>
</tr>
<tr>
<td>02h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Id_write_command</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
<td>FFh</td>
<td>Full chip Erase (bits at FFh)</td>
</tr>
<tr>
<td>04h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00h</td>
<td>Hardware reset</td>
</tr>
<tr>
<td>03h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01h</td>
<td>LJMP address</td>
</tr>
<tr>
<td>Id_read_command</td>
<td>00h</td>
<td></td>
<td></td>
<td></td>
<td>00h</td>
<td>Read Bootloader Version</td>
</tr>
<tr>
<td>05h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01h</td>
<td>Read Device boot ID1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>02h</td>
<td>Read Device boot ID2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01h</td>
<td>Read Manufacturer Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>03h</td>
<td>Read Family Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>02h</td>
<td>Read Product Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01h</td>
<td>Read Product Revision</td>
</tr>
<tr>
<td>Id_change_base</td>
<td>00h</td>
<td>03h</td>
<td>00h</td>
<td>01h</td>
<td>PP</td>
<td>Select “PP” 64kBytes flash page number</td>
</tr>
</tbody>
</table>
### Table 9-2. DFU Class-specific Requests

<table>
<thead>
<tr>
<th>bmRequestType</th>
<th>bRequest</th>
<th>wValue</th>
<th>wIndex</th>
<th>wLength</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0001b</td>
<td>DFU_DETACH (0)</td>
<td>wTimeout</td>
<td>Interface (4)</td>
<td>Zero</td>
<td>none</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_DNLOAD (1)</td>
<td>wBlock</td>
<td>Interface (4)</td>
<td>Length</td>
<td>Firmware</td>
</tr>
<tr>
<td>1010 0001b</td>
<td>DFU_UPLOAD (2)</td>
<td>wBlock</td>
<td>Interface (4)</td>
<td>Length</td>
<td>Firmware</td>
</tr>
<tr>
<td>1010 0001b</td>
<td>DFU_GETSTATUS (3)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>6</td>
<td>Status</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_CLRSTATUS (4)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>Zero</td>
<td>none</td>
</tr>
<tr>
<td>1010 0001b</td>
<td>DFU_GETSTATE (5)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>1</td>
<td>State</td>
</tr>
<tr>
<td>0010 0001b</td>
<td>DFU_ABORT (6)</td>
<td>Zero</td>
<td>Interface (4)</td>
<td>Zero</td>
<td>none</td>
</tr>
</tbody>
</table>
10. Appendix-B

```assembly
;**************************************************************************
**
; Copyright (c) Atmel.
**
; $Name: flash_boot_drv.s90,v $ $Revision: 1.7 $ $Id: flash_boot_drv.s90,v 1.7 2005/10/03 15:50:12 $
;**************************************************************************
** PURPOSE:
; This file contains the low level driver for the flash access
;**************************************************************************
**
FLASH_DRV(16)

;____ INCLUDES

#define ASM_INCLUDE
#include "config.h"

;**************************************************************************
**
; This is the absolute table entry points for low level flash drivers
; This table defines the entry points that can be called
; from the application section to perform on-chip flash operations:
;
; entry_flash_page_erase_and_write:
; R18:17:R16: The byte address of the page
;
; entry_flash_fill_temp_buffer:
; data16 : R16/R17: word to load in the temporary buffer.
; address: R18/R19: address of the word in the temp. buffer.
;
; entry_flash_prg_page:
; R18:17:R16: The byte address of the page
;
; entry_flash_page_erase:
; R18:17:R16: The byte address of the page
;
;**************************************************************************
**
ASEG FLASH_END-0x0001B
entry_flash_page_erase_and_write:
```
JMP flash_page_erase_and_write
entry_flash_read_sig:
  JMP flash_read_sig
entry_flash_read_fuse:
  JMP flash_read_fuse
entry_flash_fill_temp_buffer:
  JMP flash_fill_temp_buffer
entry_flash_prg_page:
  JMP flash_prg_page
entry_flash_page_erase:
  JMP flash_page_erase_public
entry_lock_wr_bits:
  JMP lock_wr_bits

RSEGBOOT

;****************************************************************************************
**
; NAME: flash_page_erase_and_write
;------------------------------------------------------------------------
--
; PARAMS: R18:17:R16: The byte address of the page
;------------------------------------------------------------------------
--
; PURPOSE: This function can be called for the user application
; This function performs an erase operation of the selected target page and
; the launch the prog sequence of the same target page.
; This function allows to save the 256 bytes software temporary buffer in
; the application section
;****************************************************************************************
**
flash_page_erase_and_write:
  PUSH R18
  RCALL flash_page_erase
  POP R18
  RCALL flash_prg_page
  RET

;****************************************************************************************
**
; NAME: flash_prg_page
;------------------------------------------------------------------------
--
; PARAMS: R18:17:R16: The byte address of the page
;------------------------------------------------------------------------
--
; PURPOSE: Launch the prog sequence of the target page
**flash_prg_page:**

```assembly
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
MOV R31, R17
MOV R30, R16 ;move adress to z pointer (R31=ZH R30=ZL)
OUT RAMPZ, R18
LDI R20, $05 ;(1<<PGWRT) + (1<<SPMEN))
OUT SPMCsr, R20; argument 2 decides function (r18)
SPM ;Store program memory
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
RCALL flash_rww_enable
RET
```

**flash_page_erase:**

```assembly
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
MOV R31, R17
MOV R30, R16 ;move adress to z pointer (R31=ZH R30=ZL)
OUT RAMPZ, R18
LDI R20, $03 ;(1<<PGERS) + (1<<SPMEN))
OUT SPMCsr, R20; argument 2 decides function (r18)
SPM ;Store program memory
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
RCALL flash_rww_enable CAUTION DO NOT ACTIVATE HERE or
;you will loose the entire page buffer content !!
RET
```

;***************************************************************
**
; NAME: flash_page_erase
;-------------------------------------------------------------
--
; PARAMS: R18:17:R16: The byte address of the page
;-------------------------------------------------------------
--
; PURPOSE: Launch the erase sequence of the target page
;-------------------------------------------------------------
--
; NOTE: This function does nt set the RWWSE bit after erase. Thus it does not
; erase the hardware temporary temp buffer.
; This function is for bootloader usage
;-------------------------------------------------------------
--
; REQUIREMENTS:
;***************************************************************
**

;*********************************************************************
;****************************************************************************************

**
; NAME: flash_page_erase_public
;-----------------------------------------------
--
; PARAMS: R18:17:R16: The byte address of the page
;-----------------------------------------------
--
; PURPOSE: Launch the erase sequence of the target page
;-----------------------------------------------
--
; NOTE: !!!!This function set the RWWSE bit after erase. Thus it
; erase the hardware temporary temp buffer after page erase
;****************************************************************************************

**

flash_page_erase_public:
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
MOV R31,R17
MOV R30,R16 ;move adress to z pointer (R31=ZH R30=ZL)
OUT RAMPZ, R18
LDI R20,$03 ;(1<<PGERS) + (1<<SPMEN))
OUTSPMCSR, R20; argument 2 decides function (r18)
SPM ;Store program memory
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
RCALL flash_rww_enable
RET

;****************************************************************************************

**
; NAME: flash_rww_enable
;-----------------------------------------------
--
; PARAMS: none
;-----------------------------------------------
--
; PURPOSE: Set RWSE bit. It allows to execute code in the application section
; after a flash prog (erase or write page)
;****************************************************************************************

**

flash_rww_enable:
RCALL WAIT_SPMEN ;Wait for SPMEN flag cleared
LDI R20,$11 ;(1<<WWSRE) + (1<<SPMEN))
OUT SPMCSR, R20; argument 2 decides function (r18)
SPM ;Store program memory
RJMP WAIT_SPMEN ;Wait for SPMEN flag cleared
; NAME: flash_read_sig
;--------------------------------------------------------------------------
--
; PARAMS:
; Return: R16: signature value
;--------------------------------------------------------------------------
--
; PURPOSE: Read hardware signature byte. The byte is selected through the addr
; passed as argument (See product data sheet)
;**************************************************************************
**
flash_read_sig:
RCALL   WAIT_SPMEN  ;Wait for SPMEN flag cleared
MOV     R31,R17
MOV     R30,R16     ;move address to Z pointer (R31=ZH R30=ZL)
OUT     RAMPZ, R18
LDI     R20,$21     ;(1<<SPMEN) | (1<<SIGRD))
OUT SPMCSR, R20; argument 2 decides function (r18)
LPM                ;Store program memory
MOV     R16, R0      ;Store return value (1byte->R16 register)
RJMP    WAIT_SPMEN  ;Wait for SPMEN flag cleared

; NAME: flash_read_fuse
;--------------------------------------------------------------------------
--
; Return: R16: fuse value
;--------------------------------------------------------------------------
--
; PURPOSE: Read fuse byte. The fuse byte is selected through the address passed
; as argument (See product datasheet for addr value)
;**************************************************************************
**
flash_read_fuse:
RCALL   WAIT_SPMEN  ;Wait for SPMEN flag cleared
MOV     R31,R17
MOV     R30,R16     ;move address to Z pointer (R31=ZH R30=ZL)
OUT     RAMPZ, R18
LDI     R20,$09     ;(1<<SPMEN) | (1<<BLBSET))
OUT SPMCSR, R20; argument 2 decides function (r18)
LPM                ;Store program memory
MOV     R16, R0      ;Store return value (1byte->R16 register)
RJMP    WAIT_SPMEN   ;Wait for SPMEN flag cleared

/*F************************************************************************
**
* NAME: flash_fill_temp_buffer
*------------------------------------------------------------------------
--
* PARAMS:
* data16 : R16/R17: word to load in the temporary buffer.
* address: R18/R19: address of the word.
* return: none
*------------------------------------------------------------------------
--
* PURPOSE:
* This function allows to load a word in the temporary flash buffer.
*------------------------------------------------------------------------
--
* EXAMPLE:
* fill_temp_buffer(data16, address);
*------------------------------------------------------------------------
--
* NOTE:
* the first paramater used the registers R16, R17
* The second parameter used the registers R18, R19
***************************************************************************/

flash_fill_temp_buffer:
    MOV     R31,R19     ;move adress to z pointer (R31=ZH R30=ZL)
    MOV     R30,R18
    MOV     R0,R17      ;move data16 to reg 0 and 1
    MOV     R1,R16
    LDI     R20,(1<<SPMEN)
    OUT SPMCSR, R20; r18 decides function
    SPM                ; Store program memory
    RJMP   WAIT_SPMEN   ; Wait for SPMEN flag cleared

;*F************************************************************************
**
; NAME: lock_wr_bits
;------------------------------------------------------------------------
--
; PARAMS:   R16: value to write
;------------------------------------------------------------------------
--
; PURPOSE:
;***************************************************************************/

lock_wr_bits:
RCALL WAIT_SPMEN ; Wait for SPMEN flag cleared
MOV R0,R16
LDI R18,((1<<BLBSET)|(1<<SPMEN))
OUT SPMCSR, R18 ; r18 decides function
SPM ; write lockbits
RJMP WAIT_SPMEN ; Wait for SPMEN flag cleared

;**************************************************************************
**
; NAME: wait_spmen
;--------------------------------------------------------------------------
--
; PARAMS: none
;--------------------------------------------------------------------------
--
; PURPOSE: Performs an active wait on SPME flag
;**************************************************************************
**
WAIT_SPMEN:
   MOV R0, R18
   INR18, SPMCSR ; get SPMCR into r18
   SBRC R18,SPMEN
   RJMP WAIT_SPMEN ; Wait for SPMEN flag cleared
   MOV R18, R0
   RET

END
11. Document Revision History

11.1 7618B 03/08
1. Removed references to DFU Functional Descriptor throughout the document.

11.2 7618C 07/08
1. Update for AT90USB162/82, AT90USB64x, ATmega32U4 and ATmega16U4.
2. Update bootloader revision history.