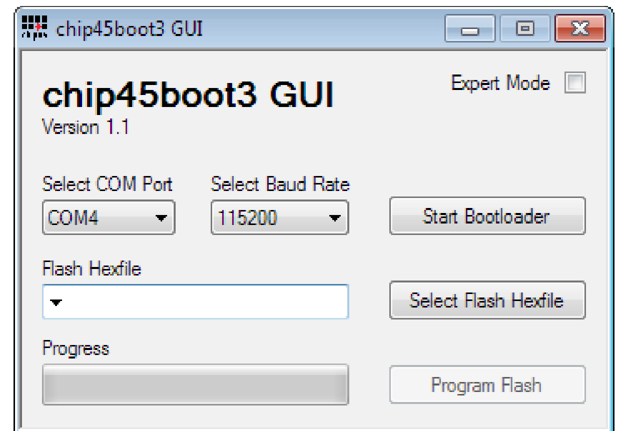
chip45boot3 Infosheet

chip45 Better Embedded.

Customizable Bootloader For ATmega and Xmega AVR Devices.

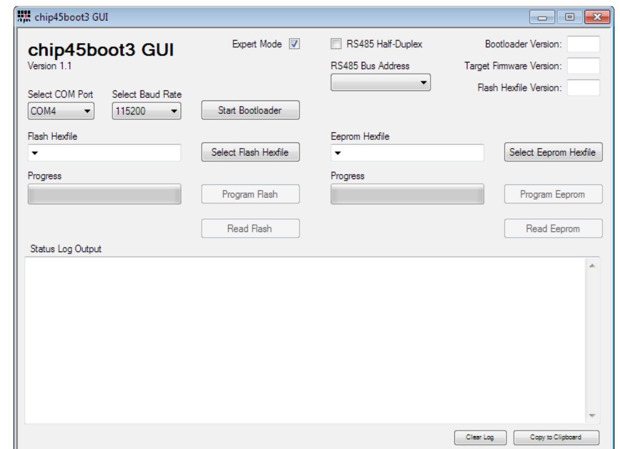
Overview



The chip45boot3 bootloader is a highly customizeable bootloader for all ATmega and Xmega devices with bootloader support (i.e. separate bootblock in flash and boot reset vector).

A fast binary protocol with strong 16 bit CRC checksum ensures a reliable communication with a host PC. Flash and EEPROM memory of the target can be written or read out and the written content is verified against the original data.

The bootloader can read out a firmware version from the exisiting flash memory and compare it against the version of the new firmware hexfile. A warning is shown, if an older version should be programmed.

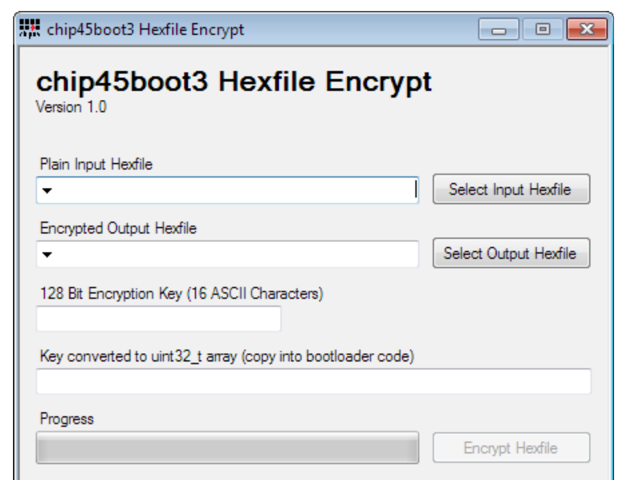


Communication is performed through any UART of the target device with a host PC (e.g. USB UART converter, RS232, RS485). Other microcontroller interfaces can be adopted relatively easy due to the very modular design of the bootloader.

Application firmware content can be XTAE encrypted optionally making it impossible for attackers to reverse engineer the firmware.

Memory footprint starts at below 2k for small ATmega devices with just flash programming support.

The bootloader is written in C with Atmel Studio 7 and the Windows PC GUI is written with Visual C# Express. Full source code is included of bootloader, PC GUI as well as an PC encryption tool for XTEA encryption of hex files.



Main Features

- Automatic baud rate detection
- Read and write of target flash memory
- Read and write of target EEPROM memory
- Read target firmware version
- RS485 half-duplex support
- optional XTAE encryption

Scope Of Delivery

If you order chip45boot3 in our online shop, you will receive the full source code as ZIP archive by email. Since the ZIP includes executables of the GUI and encrypt tool, it might happen, that your mail server reject the attachment. If we receive an error message from your mail server we will contact you and offer to download it from a temporary download link. If you don't receive an email with the source code following your order, please contact us and we will send a download link.

Version History

Xmega USB Controller Support

Up to rev118 the chip45boot3 provided support for the internal USB controller of certain Xmega devices. This USB support was based on the LUFA USB framework. Since this feature was requested only very rarely and often lead to compiler issues with latest Studio 7 versions, we decided to drop this on later releases.

If you are interested in Xmega USB controller support, please contact us by email and you can receive this version additionally.

chip45boot2 Compatibility

The older chip45boot2 is a simple bootloader and uses a plain ASCII protocoll of transmitting a native Intel hex file through a UART. This is NOT compatible with chip45boot3 and it is not possible to use the chip45boot3 GUI with chip45boot2 bootloader and vice versa.

Recent Versions

After version rev118 (see above) we did major reorganizations of the code. All USB related code and the corresponding conditional compilation macros have been removed and some minor bugs have been fixed. The example projects for ATmega and Xmega devices have been updated to the latest Atmel Studio 7 version.

ZIP Content Overview

When extracting the ZIP file you should see the following content:

Name	Änderungsdatum	Typ	Größe
chip45boot3_target_code	29.11.2017 12:16	Dateiordner	
chip45boot3_windows_encrypt_tool	29.11.2017 12:16	Dateiordner	
chip45boot3_windows_gui	29.11.2017 12:16	Dateiordner	
chip45boot3_Infosheet.pdf	29.11.2017 12:17	Foxit Reader PDF Document	1.015 KB

The source code is distributed over three main subfolders:

- **chip45boot3_target_code** with target bootloader source code and Studio 7 example projects
- **chip45boot3_windows_encrypt_tool** with the XTAE encryption tool for hex file encryption and key generation
- **chip45boot3_windows_gui** with the main PC bootloader communication tool

Additionally this Infosheet is included in the ZIP file.

Target Code

When looking into the bootloader target code folder you see the following files:

Name	Änderungsdatum	Typ	Größe
bootloader	29.11.2017 12:16	Dateiordner	
Debug	29.11.2017 12:16	Dateiordner	
mcu_headers	29.11.2017 12:16	Dateiordner	
ATmega_Example.componentinfo.xml	29.11.2017 12:11	XML-Dokument	4 KB
ATmega_Example.cproj	21.10.2017 20:20	ATMEL Studio 7.0 ...	12 KB
Xmega_Example.componentinfo.xml	14.11.2017 15:31	XML-Dokument	4 KB
Xmega_Example.cproj	21.10.2017 17:31	ATMEL Studio 7.0 ...	11 KB

The bootloader source code is split into the bootloader code in **bootloader** plus target specific C header files in **mcu_headers**. The folder **Debug** is generated during compilation of the bootloader and contains the final target bootloader files like .hex, .elf, .lss, .map etc.

Two Studio 7 example projects are included, one for ATmega devices and one for Xmega devices. They differ slightly since the Xmegas require some more c./h. files. The projects can be open with Studio 7 directly.

We will look into the projects in more detail below.

Windows GUI

The Windows GUI comes as Visual C# Express project. In the folder you find the chip45boot3_gui.sln Visual Studio solution file, which can be opened with Visual C# Express.

We will look into the project and give some customization hints later.

Encrypt Tool

Similar to the GUI the encrypt tool is also a Visual C# Express project and the above mentioned applies, too.

Target Bootloader Code

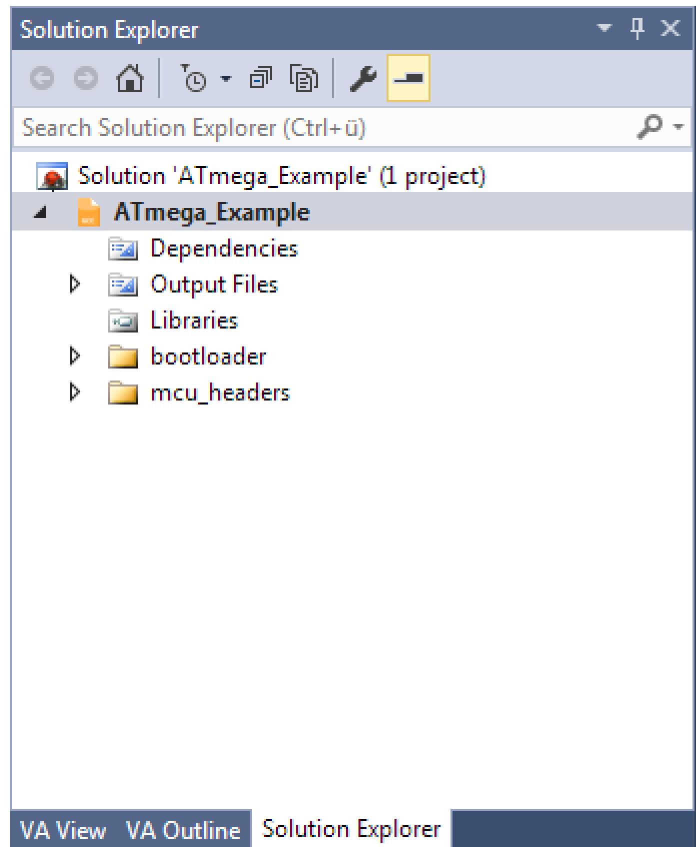
The bootloader comes with two separate target projects for ATmega and Xmega devices. At first we will open the projects, have a look into the project content and compile the examples.

Make sure you have the latest version of Atmel Studio 7 installed. You can download it here:

http://www.atmel.com/microsite/atmel-studio/

ATmega Studio 7 Project

Open the file **ATmega_Example.cproj** by double-clicking it or open it through Studio 7 File -> Open -> Project/Solution.



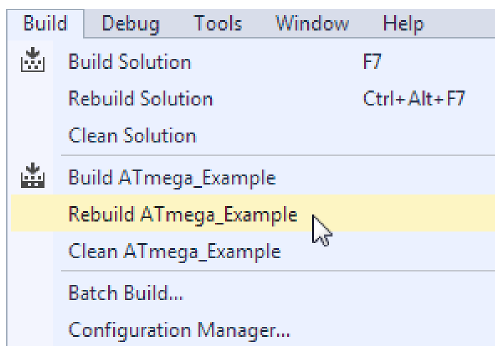
You should see the ATmega_Example project in the solution explorer on the right side of the Studio 7 window like shown on the right here.

Feel free to expand the **bootloader** folder to see the corresponding .c/.h source code files or **mcu_headers** for target specific header files.

We will look more into detail later when porting the bootloader to a particular target device.

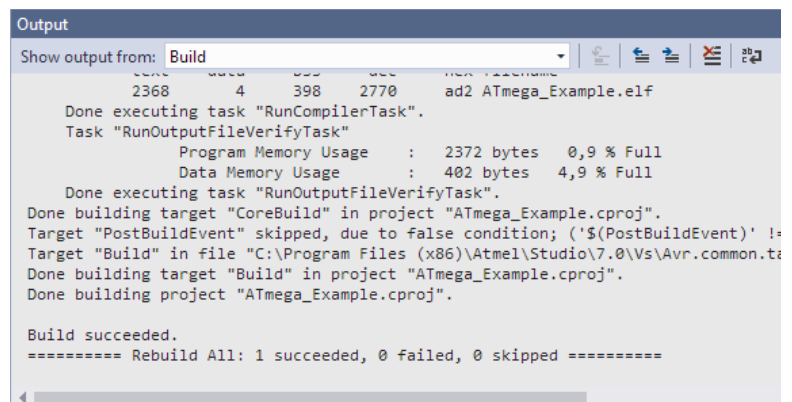
Compiling The Target Bootloader

Try to compile to bootloader for the current target device. Use **Rebuild ATmega_Example** from the Build menu:

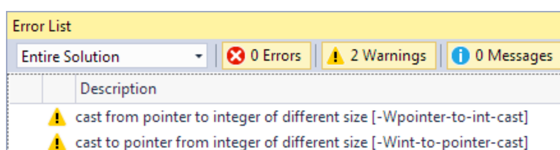


Rebuild instead of Build ensures, that the project is cleaned up prior to compilation, hence all source code files will be compiled and error checked freshly.

You should see the compilation progress in the Output window and should end up with a successful build like shown here:



Depending on your project compiler warning settings you might observe two warning messages, which can be ignored so far.



Xmega Studio 7 Project

The Xmega project is more or less similar to the ATmega project. Major differences are:

- Additional self-programming driver source files sp_driver.s/.h.
- Only xmega header files included in mcu_headers
- no BOOT_SECTION_START compiler symbol/macro in project settings, since for xmega devices this is already defined in regular AVR GNU toolchain header files

Common Source Code Files

We go through the source code files an an alphabetic order and give a very brief description of what happens in the particular files. All files are well documented with plenty of comments for the functions and most code lines.

Most files provide separate Init and deInit functions that are called on bootloader startup and on exit plus Do functions, which perform the relevant tasks.

ATmega/Xmega Files

`autobaud.c/.h`	The automatic baudrate detection is done here. USART pins are activated and timing of incoming autobaud characters are measured.
`clock.c/.h`	The Xmegas require some clock setting, whilst the ATmega clock setting is done through fuse bits.
`commands.c/.h`	All commands from the GUI are handled here. Most commands are related to a certain functionality compiled into the bootloader.
`crc.c/.h`	A CRC computation function is located here.
`flash.c/.h`	Provides functions to fill a flash page buffer with firmware data received from the GUI and to write that page into flash memory.
`host.c/.h`	Host communication functions. The whole host communication is performed by the two functions hostBufferReceive and hostBufferSend, which are using USART communication. If you want to port the bootloader to other interfaces like I2C or SPI, it should be necessary to port only these two functions.
`main.c/.h`	All other functions are called from here in an endless loop.
`startup.c/.h`	At a very early stage of the bootloader the function startupCheckStartCondition() is called. Since the bootloader always starts after reset, this function checks a condition if the bootloader should really be activated. Currently this is a high level on the USART RxD pin, which normally is the case when a remote USART is connected. This could be changed to e.g. reading another pin status with a jumper connected or could be reading an EEPROM cell, which were previously set by the application firmware.
`timeout.c/.h`	At certain positions a timeout is required, e.g. during startup when no (auto baud) characters are being received and the application should be started instead.
`usart.c/.h`	Basic USART communication functions used by host.c.
`xtae.c/.h`	The XTAE decryption of encrypted firmware date is done here.

`configure.h`	Some configuration issues are done here, like the bootloader version number. Since most other configuration is done through compiler symbols in the project settings, this file mostly takes care if all necessary symbols are actually set or not and throws a compiler error in case.

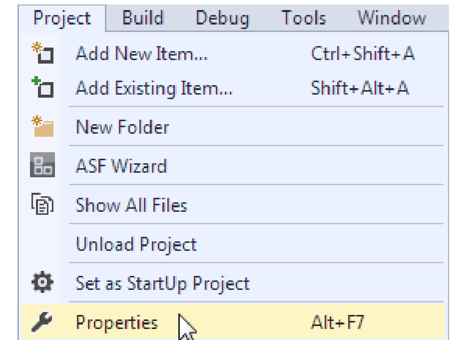
Additional Xmega-only Files

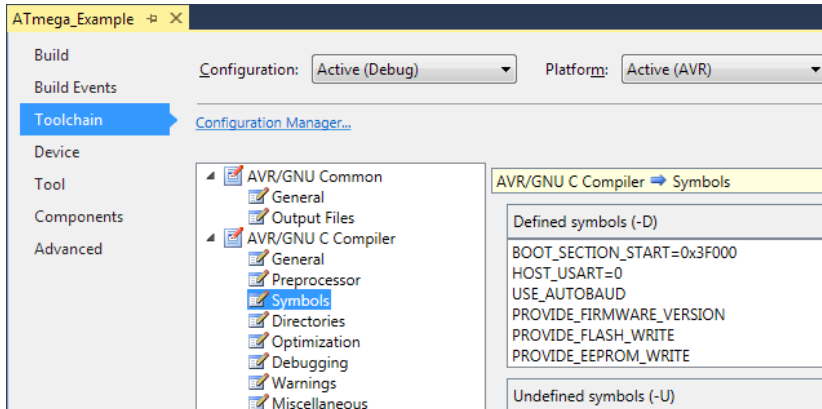
`sp_driver.s/.h`	This file provides some assembly files for Xmega self-programming functionality.

 © chip45 GmbH & Co. KG • 35396 Gießen • Germany • http://www.chip45.com

Customzing the Bootloader

The functionality of the bootloader is fully customizable through compiler macros without touching any line of code. Opening the project properties from the Project menu (or right click the project in the solution explorer) shows the project properties panel.





Select Toolchain -> AVR/GNU C Compiler -> Symbols to see the defined symbols for the current project. The symbol names are mostly self explanatory and we go through their effects now.

Functionality Symbols

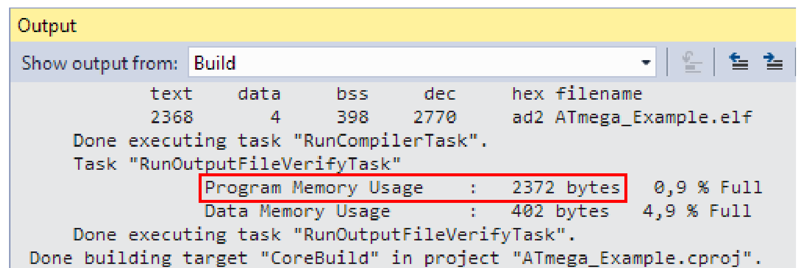
The following symbol names are known by the code and can be set/unset in the above project settings. Some are mandatory (e.g. the host usart port) and the compiler will throw an error during compilation if missing.

BOOT_SECTION_START=0xHHHH	For ATmega targets the size of the flash bootloader section can be selected by fuse bit settings, hence the bootloader start address is not fixed and has to be set through this symbol properly. See section Bootloader Start Address below for details. Xmega targets have a the boot block located above the application section with a fixed address and fixed size for each target device. The BOOT_SECTION_START symbol is predefined in the device io header file and must not be entered here.
HOST_USART=n	Select the target device USART port here. Valid values are numbers from 0 until the maximum USART number of the target. There is different meaning on ATmega and Xmega devices: ATmega: 0 : USART0 1 : USART1 … 3 : USART3 Xmega: 0 : USARTC0 1 : USARTC1 2 : USARTD0 3 : USARTD1 … 6 : USARTF0 7 : USARTF1 Some Xmegas are lacking certain USARTX1 ports, hence the corresponding numbers are missing/invalid. See the Xmega device header files in mcu_headers for examples. According to the selected USART port the USART pins are defined in the mcu header file.
USE_AUTOBAUD	Enables automatic baud rate detection of the bootloader. During connect sequence the GUI sends autobaud-characters and the bootloader measures the timing of the RX pin and adjusts its baud rate accordingly.
MANUAL_BAUDRATE=115200	If you don't want automatic baudrate detection, you can set the baudrate to a fixed value here. If using MANUAL_BAUDRATE you must not set USE_AUTOBAUD, since it has precedence over MANUAL_BAUDRATE.
F_CPU=14745600	In case the baudrate is specified manually it is necessary to set the device clock frequency here. If using automatic baudrate detection, you can omit this symbol.
USE_RS485	If defined the bootloader will assume a half-duplex RS485 connection and will drive an additional pin for the RS485 transceiver direction control. The pin is defined in the target mcu header file and can be adjusted to any pin you desire.

PROVIDE_FIRMWARE_VERSION	The bootloader can read out a firmware version from the applicatin flash content, provided it is defined during firmware compilation. See section Firmware Version below.
PROVIDE_FLASH_WRITE	If defined, the bootloader can program the target flash memory.
PROVIDE_FLASH_READ	If defined, the GUI can read out the target flash memory. This is not often used, since it is not wanted, that the firmware can be read out. Never use this together with encryption enabled, since this would allow to read out the firmware in a non-encrypted form!
PROVIDE_EEPROM_WRITE PROVIDE_EEPROM_READ	Similar to flash read/write this provides functionality to program or read EEPROM memory.
USE_ENCRYPTION	If defined, the bootloader will XTAE decrypt the firmware data sent from the GUI. If used, it is mandatory to also define the encryption key in the below form. You can copy&paste the proper definition from the PC hex file encrypt tool window.
ENCRYPTION_KEY={1936287828,1750365001,1635014757,2036681573} *Note: no spaces are allowed between the numbers and/or the curly brackets!*	

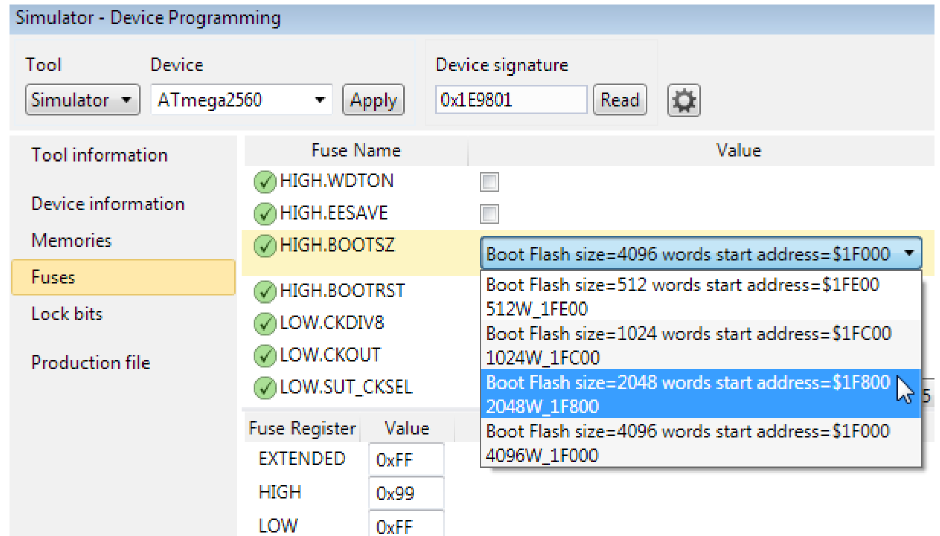
Bootloader Start Address

The size of the flash boot block is not fixed on ATmega devices and can be selected by fuse bits. It must be at least the size of the compiled bootloader. The get the actual size of the target bootloader you have to compile it. Make all necessary functionality symbol settings and Rebuild Projekt and you can see the size near the end of the Output window. The following screenshot shows the output with the bootloader size marked.



In this example the bootloader size is between 2kbytes and 4kbytes, hence a 4kbyte boot block has to be set by the fuse bits. The following screenshot shows the Device Programming window of Studio 7 with the selection of the boot block size. The screenshot shows the window for an ATmega2560 with a 4kbyte boot block.

Please note, that the drop down menu shows the boot block size in words instead of bytes! This is due to the AVR architecture using 16 bit machine opcodes. If you need a 4kbyte boot block as mentioned above, you have to select the 2048 words entry here.



For Xmega devices it is not necessary to do those boot block settings. Their boot block always starts above the application memory and has a fixed size for a target. Anyway it's good to have a look at the Program Memory Usage, since the code might be too large due to having too many options enabled.

Note: *Make sure that the BOOT_SECTION_START symbol must be set with the byte address!!! This is slightly confusing in this context, but is consistent for Xmega devices, which come with this symbol predefined in the io header files.*

Linker Address

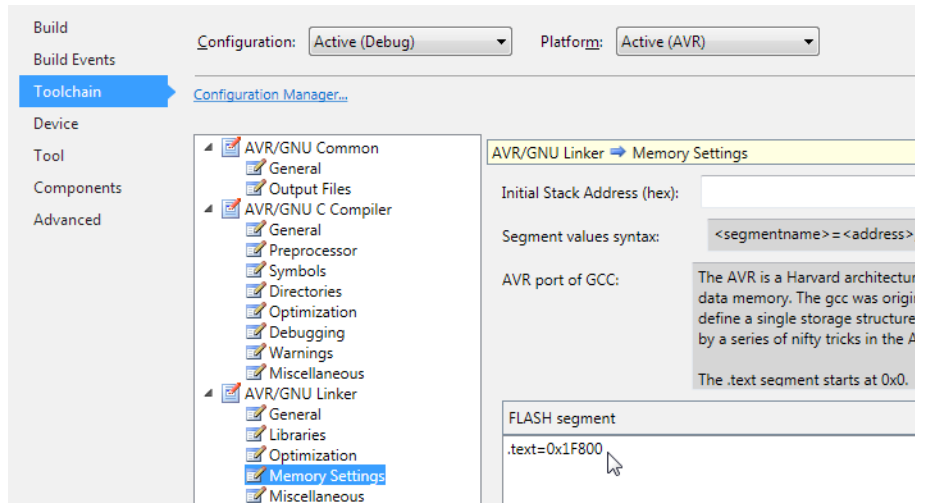
After compilation of all source files the GNU GCC linker composes the final program and locates its start address usually at address zero (0x0000), because this is the start address for application code on AVR

devices. Since a bootloader is located at the top end of the flash memory, we must let the linker know about this and tell him the start address, where the code should start.

Open the project properties as previously mentioned above and go to Toolchain -> AVR/GNU Linker -> Memory Settings. Under FLASH Segment you have to enter a new item and insert the bootloader start address in the following form:

```
.text=0x1F800
```

.text is the code section of the GCC memory layout and 0x1F800 is the bootloader start address as we set in the previous section.



Please note again, that there is a slight inconsistency between byte and word addresses used at certain positions:

- BOOT_SECTION_START in the compiler symbols has to be specified as byte address (only for ATmegas)
- .text in the linker settings has to be specified as word address

Example Calculation

If you use an ATmega2560, the flash size is 256kbytes, i.e. 262144 bytes (256*1024). If you want to use a 4kybyte (4096 byte) bootblock, the bootloader start address is (256-4)*1024 = 258048 = 0x3F000. This is the byte address and as word address it is 0x3F000 / 2 = 0x1F8000. So we set:

- `BOOT_SECTION_START=0x3F000`
- `.text=0x1F800`

Values From the Data Sheet

Alternatively you can also check the target device datasheet and check the table for the Boot Loader Parameters as shown below. Note that the addresses in AVR data sheets are given as word address, not byte address.

ATmega2560/2561 Boot Loader Parameters

In Table 29-13 through Table 29-15, the parameters used in the description of the Self-Programming are given.

Table 29-13. Boot Size Configuration, ATmega2560/2561[(1)]

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Appli-cation Flash Section	Boot Loader Flash Section	End Application Section	Boot Reset Address (Start Boot Loader Section)
1	1	512 words	4	0x00000 - 0x1FDFF	0x1FE00 - 0x1FFFF	0x1FDFF	0x1FE00
1	0	1024 words	8	0x00000 - 0x1FBFF	0x1FC00 - 0x1FFFF	0x1FBFF	0x1FC00
0	1	2048 words	16	0x00000 - 0x1F7FF	0x1F800 - 0x1FFFF	0x1F7FF	0x1F800
0	0	4096 words	32	0x00000 - 0x1EFFF	0x1F000 - 0x1FFFF	0x1EFFF	0x1F000

Porting to a New Target

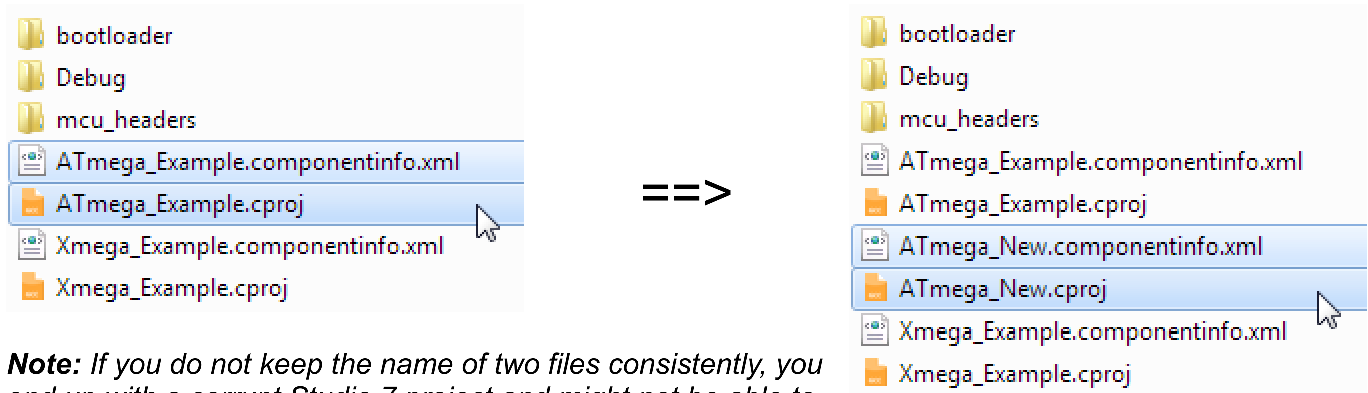
Porting the bootloader to a new target is quite simple and the necessary steps are decribed in the following.

We recommend to keep a copy the original example projects for reference without modifications.

Copy Studio 7 Project

Most simple way is to copy the whole target code folder to a new location and use/modify the example projects.

But if you want to use the bootloader on several targets and have them use the same code base, you probably want to copy on of the example projects into the same location. In that case is important to copy BOTH the .cproj file as well as the .componentinfo.xml file and give them the same name. See below for an example. You can then open the project with Studio 7 as usual.



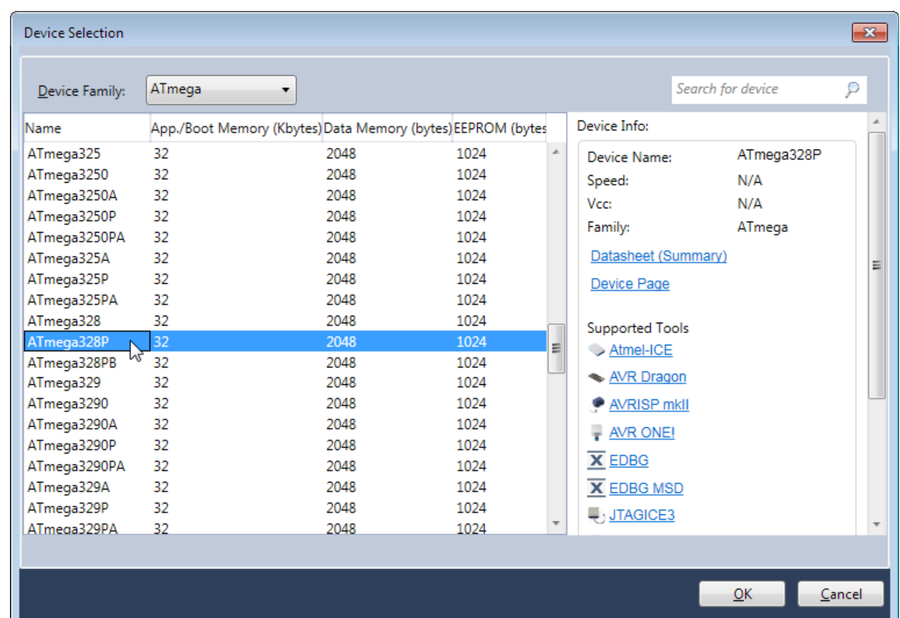
Note: *If you do not keep the name of two files consistently, you end up with a corrupt Studio 7 project and might not be able to change the device to a new one.*

Change the Target Device

Open the project properties and go to Device -> Change Device button. The Device Selection will open and you can select a new device from the list. In the example we switch from ATmega2560 to ATmega328P.

Note: *Make sure to not switch to an Xmega device in the ATmega project and vice versa.*

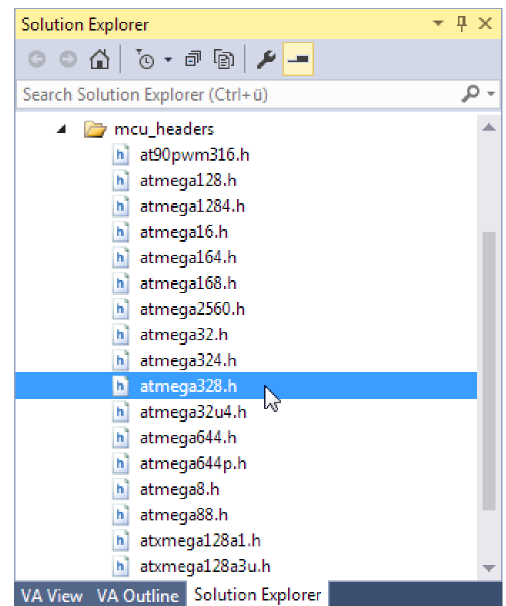
Studio 7 will then switch to the new device.



 © chip45 GmbH & Co. KG • 35396 Gießen • Germany • http://www.chip45.com

Check for Proper Header File

For each target device we need a proper header file in mcu_headers, which defines all the relevant things for that device.



Expand the folder mcu_headers in the Solution Explorer and check for the device. Luckily we find an atmega328.h. Since the ATmega328P and the ATmega328 do not differ noteworthy, we may use the atmega328.h also for the ATmega328P.

Open the header file and have a look into it...

`#define WATCHDOG_INT`

The bootloader normally uses the watchdog timer as timeout timer, since the watchdog timer runs at always the same clock rate independent from the actual crystal clock setting, hence timeout durations remain stable. On older ATmega devices the watchdog timer is not available as timer but just as reset generator. In that case a normal timer will be used for timeout, but the duration might have to be adjusted in the code.

If the device offers the watchdog timer as timer, you may define this symbol.

The next section describes the available USARTs of the device, see screenshot right.

```
// definitions for the serial port
#if (HOST_USART == 0)

    #define myUDR    UDR0
    #define myUBRRH  UBRR0H
    #define myUBRRL  UBRR0L
    #define myUCSRA  UCSR0A
    #define myUCSRB  UCSR0B
    #define myUCSRC  UCSR0C
    #define myUDRE   UDRE0
    #define myRXC    RXC0
    #define myTXC    TXC0
    #define myRXEN   RXEN0
    #define myTXEN   TXEN0
    #define myUCSZ0  UCSZ00
    #define myUCSZ1  UCSZ01
    #define myURSEL 0            // MCU type does not require URSEL bit to be set
    #define myRXDPIN PINE
    #define myRXDPORT PORTE
    #define myRXD PIN0
    #define myTXD PIN1

    #ifdef USE_RS485
        #define myRS485DDR DDRE
        #define myRS485PORT PORTE
        #define myRS485_DIRPIN PIN2
    #endif

#elif (HOST_USART == 1)
```

A set of USART register names with preceeding "my" will be defined for each USART and the bootloader will access the USART only through these "my" symbols. Make sure to also set the RXD pin definition properly, since this is used for automatic baud rate detection. An additional pin is defined as RS485 half-duplex direction pin.

In case no proper header file is available for your target, then choose one of a target which is as equal as possible to your target in terms of USARTs and watchdog. Copy it, rename it properly and edit its content and adjust the HOST_USART settings.

Finally the new header file has to be added to mcu.h in the bootloader folder. Open mcu.h and have a closer look into it. It is more or less a list of target devices and corresponding header files. See screenshot right.

```
#elif defined (__AVR_ATmega168__)
                                        #include "atmega168.h"
#elif defined (__AVR_ATmega168A__)
                                        #include "atmega168.h"
#elif defined (__AVR_ATmega168P__)
                                        #include "atmega168.h"


#elif defined (__AVR_ATmega328__)
                                        #include "atmega328.h"
#elif defined (__AVR_ATmega328P__)
                                        #include "atmega328.h"
```

Add a new block at a useful location and enter the proper __AVR_ATmega symbol plus the #include of the new header file you created.

As you can see, it is okay to use the same header file for for target devices, provided the set of HOST_USARTs is identical among them. E.g. you can use the same file for ATmega164, 324, 644 and 1284 since all have the same USART configuration.

If you are unsure about the correct __AVR_ATmega symbol, you can check the iomxxx.h header file and look up the symbol there.

Set Bootloader Start Address

As described above (Customizing The Bootloader -> Bootloader Start Address) we need to set the correct bootloader start address compiler symbol **`BOOT_SECTION_START`** and the linker symbol **`.text`**.

Compile The Bootloader

Finally do a Build or Rebuild project... If everything was set correct, it should lead to a successful compilation as described above (Target Bootloader Code -> Compiling The Target Bootloader).

Using a Version Number in the Target Firmware

The bootloader can read out the firmware version of the programmed firmware and can show it in the GUI, which will warn if an older firmware version should be programmed over a newer one. When receiving the read firmware version command, the bootloader will search the whole flash application memory for the string "FW_VERSION:" and - when found - read the major and minor version number from the following bytes. This can be done independently for application flash memory and eeprom memory.

Application Firmware Version

To set the application firmware version, you have to compile a version string into the application firmware. This can be done by the following code line in your application C file:

```
const unsigned char firmware_version[] PROGMEM = { "FW_VERSION:01.03" };
```

TBD: Check if it's necessary to uncheck "garbage collect unused areas" in the linker settings!

EEPROM Version

Similarly a version number can be set for the EEPROM memory generated by the compiler:

```
const unsigned char eeprom_version[] EEMEM = { "EE_VERSION:01.00" };"
```

Communication with the Bootloader

Connection Sequence

After power up or reset of the target device, the bootloader is started and checks the startup condition in startup.c. With active automatic baudrate detection, this startup condition is simply a high level on the RXD pin, which usually indicates a valid UART on the opposite side. Without autobaud the bootloader starts always. You are free to implement other startup conditions in startup.c startupCheckStartCondition(...), like sampling an IO pin for a set jumper or the content of an EEPROM cell or other things usuful for your application.

If the startup condition is met, the next step depends on active automatic baudrate calibration.

- With active autobaud, the bootloader assumes, that the opposite side (PC GUI) will send several 'U'-characters. The bootloader measures the length of the low-bits of at least five 'U'-character and calculates the baudrate. After calculation the bootloader tries to read three 'U'-characters to ensure the baudrate was properly calculated.
- With fixed baudrate, the bootloader sets this baudrate and just tries to read the three 'U'-characters.

After successful reception of three 'U'-characters, the bootloader sends a packet with "c45b3" as payload to establish the connection with the PC GUI. See following chapter for packet structure.

Communication Protocol

The chip45boot3 bootloader uses a bidirectional packet based protocol for communication with the PC GUI application.

Packet Structure

Each packet starts with an STX character and ends with an ETX character, which makes it very simple to be parsed within the bootloader host.c functions. The STX and ETX characters are unique for start and end of frame. If an STX or ETX character occurs as content of the payload, the character is being escaped by a prepended ESC character and the most significant bit of STX/ETX is set. The same applies for an ESC character being part of the payload. The following bytes are used for the three special characters:

- `STX: 0x02`
- `ETX: 0x03`
- `ESC: 0x1B`

To illustrate the escaping of the special characters, we use the following example payload:

```
AF-FE-02-DE-AD-1B-C0-FF-EE
```

The next line shows the resulting packet including STX, ETX, etc.:

```
02-AF-FE-1B-82-DE-AD-1B-9B-C0-FF-EE-03
02/03:     regular ETX and STX characters
1B:        ESC character for following byte
82:        0x02 with most significant bit set
9B:        0x1B with most significant bit set
```

This packet structure with unique characters, which cannot be part of the payload, makes it very simple to synchronize to the beginning of a frame and to recognize the end of frame with very few lines of code in the bootloader.

Payload Structure

The packet payload is composed of

- one command byte
- a varying amount of data bytes (maximum 64 data bytes!)
- two bytes of checksum (CRC-16 CCITT)

Communication Sequence

The communication is always initiated by a packet from the PC GUI to the bootloader. The bootloader processes the packet and the payload and responds with an own packet.

The answer from the bootloader starts with either

- 0xE0 indicating a CRC error
- the same command byte received by the PC GUI, indicating a command error
- the command byte or'ed with 0x80, indicating successful command processing

List of Command Bytes

The actually available command bytes may vary depending on the configuration of your particular bootloader. For example, if you do not set PROVIDE_FIRMWARE_VERSION in your project settings, the corresponding command will not be compiled into your bootloader, hence the command is not available and will not be answered by the bootloader.

The following table shows the list of existing commands, their description as well as sample packets from PC GUI and bootloader answer packets.

Command Name	Value	Description

`CMD_READ_VERSION_BOOTLOADER`	`0x11`	You can set a particular bootloader version number in configure.h, which will be requested by the PC GUI and displayed. It can be used to distinguish between certain bootloader versions, which may provide different features. The bootloader answeres with two major and minor version bytes.
		
		Example PC GUI Packet:
		`11-CRC-CRC`
		
		Example bootloader packet for version V1.02:
		`91-01-02-CRC-CRC`

`CMD_READ_VERSION_FIRMWARE`	`0x12`	The bootloader can read out a firmware version from the application flash memory, if this is compiled into the firmware.
		
		Example PC GUI Packet:
		`12-CRC-CRC`
		
		Example bootloader packet for version V2.03:
		`92-02-03-CRC-CRC`

`CMD_READ_VERSION_EEPROM`	`0x13`	Similar to the firmware version, the bootloader can read out a version from the eeprom memory.
		
		Example PC GUI Packet:
		`13-CRC-CRC`
		
		Example bootloader packet for version V3.04:
		`93-03-04-CRC-CRC`

`CMD_START_APPLICATION`	`0x18`	When receiving this command, the bootloader will quit and start the application firmware. Example PC GUI Packet: `18-CRC-CRC`

 © chip45 GmbH & Co. KG • 35396 Gießen • Germany • http://www.chip45.com

		Example bootloader packet: `98-CRC-CRC`

CMD_SET_ADDRESS	0x21	This command sets the start address for all subsequent flash or eeprom memory read or write commands.
		
		Example PC GUI Packet for address 0x12345678
		`21-12-34-56-78-CRC-CRC`
		
		Example bootloader packet:
		`A1-CRC-CRC`

CMD_FLASH_WRITE_DATA	0x22	The PC GUI can send up to 64 bytes of application flash memory data. The data is stored in an internal buffer until a flash page is full and is programmed to flash automatically. If the last flash page is not full, but should be written anyway, a `CMD_FLASH_WRITE_DATA` packet with zero data bytes has to be written to force programming of the last flash page. The PC GUI has to take care of the maximum 64 data bytes in the payload. If the buffer in the bootloader overruns, an command error will be returned.
		
		Example PC GUI Packet with 64 data bytes:
		`22-01-02-03-...-3E-3F-40-CRC-CRC`
		
		Example PC GUI Packet with zero data bytes to force page write:
		`22-CRC-CRC`
		
		Example bootloader packet:
		`92-CRC-CRC`

CMD_FLASH_READ_DATA	0x23	The PC GUI can read from flash memory with this command. After the command byte, the number of bytes to be read is set in the payload. Reading starts at the flash memory address set by `CMD_SET_ADDRESS` before.
		
		Example PC GUI Packet with 8 data bytes to read:
		`23-08-CRC-CRC`
		
		Example bootloader packet with 8 data bytes from flash:
		`93-01-02-03-04-05-06-07-08-CRC-CRC`

CMD_EEPROM_WRITE_DATA	0x24	Same as `CMD_FLASH_WRITE_DATA` but data is written to eeprom memory.
CMD_EEPROM_READ_DATA	0x25	Same as `CMD_FLASH_READ_DATA` but data is read from eeprom memory.