chip45boot2 Studio 6 Notes

Hints for comiling/porting the chip45boot2 Studio 6 projects to other MCU targets.

INTRODUCTION

The chip45boot2 source code comes with two Atmel AVR Studio 6 sample project files. The picture on the right shows the content of the source code ZIP archive, which you should have received after purchasing the source code.

As the names imply, one Studio 6 project should be used for ATmega MCU targets, the other should be used for Xmega targets. They slightly differ in the number of source code files, since the Xmega targets are not supported by the regular boot.c/boot.h files of the AVR GNU toolchain. So we added local boot.c/boot.h files as well as two driver files sp_driver.c/sp_driver.h for handling the program memory access functions. You see the different project content, if you open the Studio 6 projects.
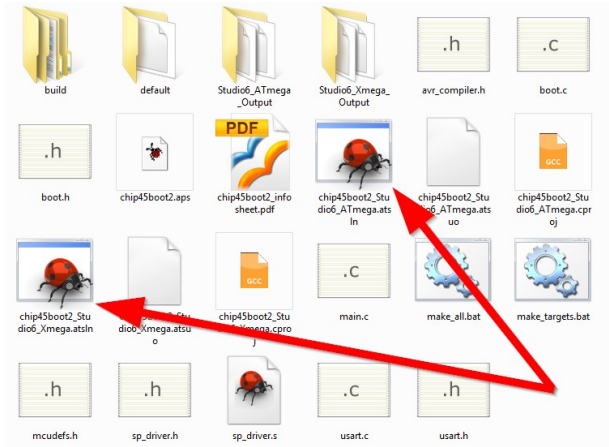
Make sure to always install the latest version of Studio 6, since we always use the latest version for compiling all targets!

Even though we use two batch files for automatically compiling the full set of targets and not a Studio 6 project for each target, these batch files use the AVR GNU toolchain from the latest Studio 6 installation. You don't have to use the batch files, of course, but you will later have to extract some information from them, so we mention them here in advance.

DOCUMENTATION

We assume, you have read and understood the chip45boot2 infosheet before proceeding to compile or port the bootloader for another target.

COMPILING OR PORTING? WHAT'S THE DIFFERENCE?

Compiling the bootloader for another target with Studio 6 means, that this particular target is already supported by the bootloader source code. That means, there exist target specific sections in make_targets.bat and mcudefs.h. In this case, the Studio 6 project settings have to be adjusted to the target and some information from make_targets.bat and mcudefs.h have to be entered into the project settings and you are done.

Porting the bootloader for a new target means, that the above mentioned is not the case. This means, that a new target specific section has to be added at least to mcudefs.h. Target specific informations have to be extracted from the MCU data sheet and maybe even from AVR GNU toolchain header files, like <avr/io.h>.

WHY IS THERE NO STUDIO 6 PROJECT FOR ALL SUPPORTED TARGETS AVAILABLE?

The more convenient a Studio 6 project is for you, who want's to edit code and compile for one fixed target configuration (MCU type, RS485 yes/no, USART0,1,2,...), the less it is for us. At the time we started this documentation, there exist 276 precompiled hexfiles for 47 different targets in the build directory. This would mean at least 47 Studio 6 project files and we would have to start each of them, adjust all parameters like RS485 yes/no, USART0,1,2,... before compiling one single target configuration – no way!

That's why we don't use Studio 6 for all targets, but use simple batch files like make_all.bat and make_targets.bat with a section for all targets and some loops for processing the different USARTs for a target and so on. If a new compiler version comes up or we do some modifications to the code, we test this for some particular targets and if it succeeds, we start the batch process, get some coffee and after a while we have 276 fresh hexfiles. ;-)

chip45boot2 Studio 6 Notes

COMPILING THE BOOTLOADER WITH STUDIO 6

If you open one of the Studio 6 project files, you should see a similar screen like the one to the right. One the right side you see the project explorer with the corresponding source files, dependencies, etc., on the left is both the editor window (assumed one source code file is opened for editing, if not, just double click on one in the explorer) as well as the settings screen for your project (we come to this in a minute).

For compiling the bootloader for the active configuration, you can select "Build Solutions" from the "Build" menu or just press "F7" (see picture below).

The compiler output is shown in the "Output" windows on the bottom of the Studio 6.
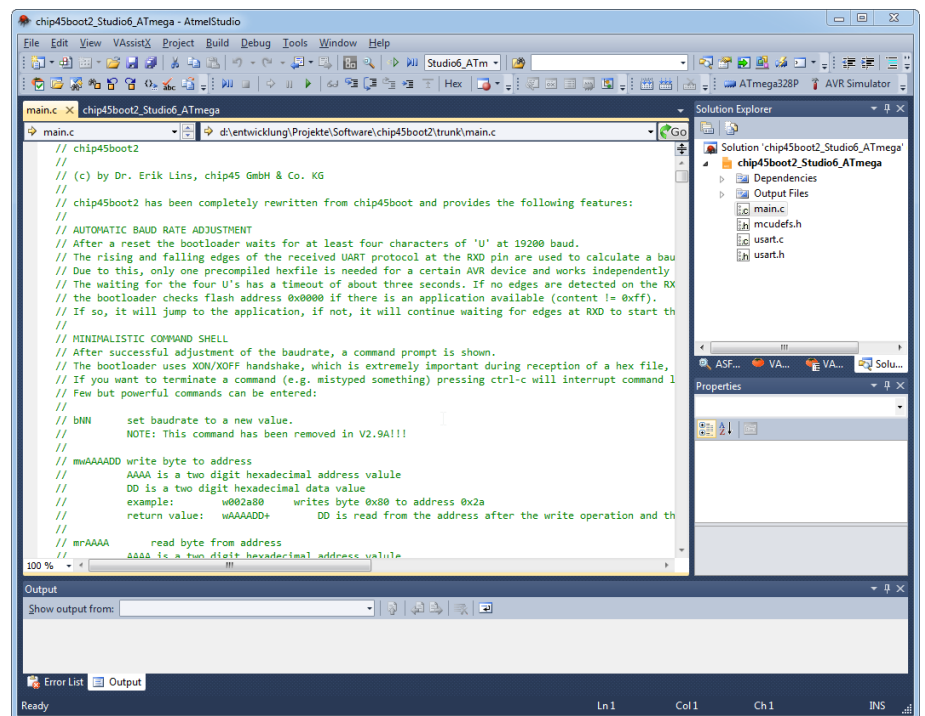
Finally you should see a similar content like shown in the picture below.
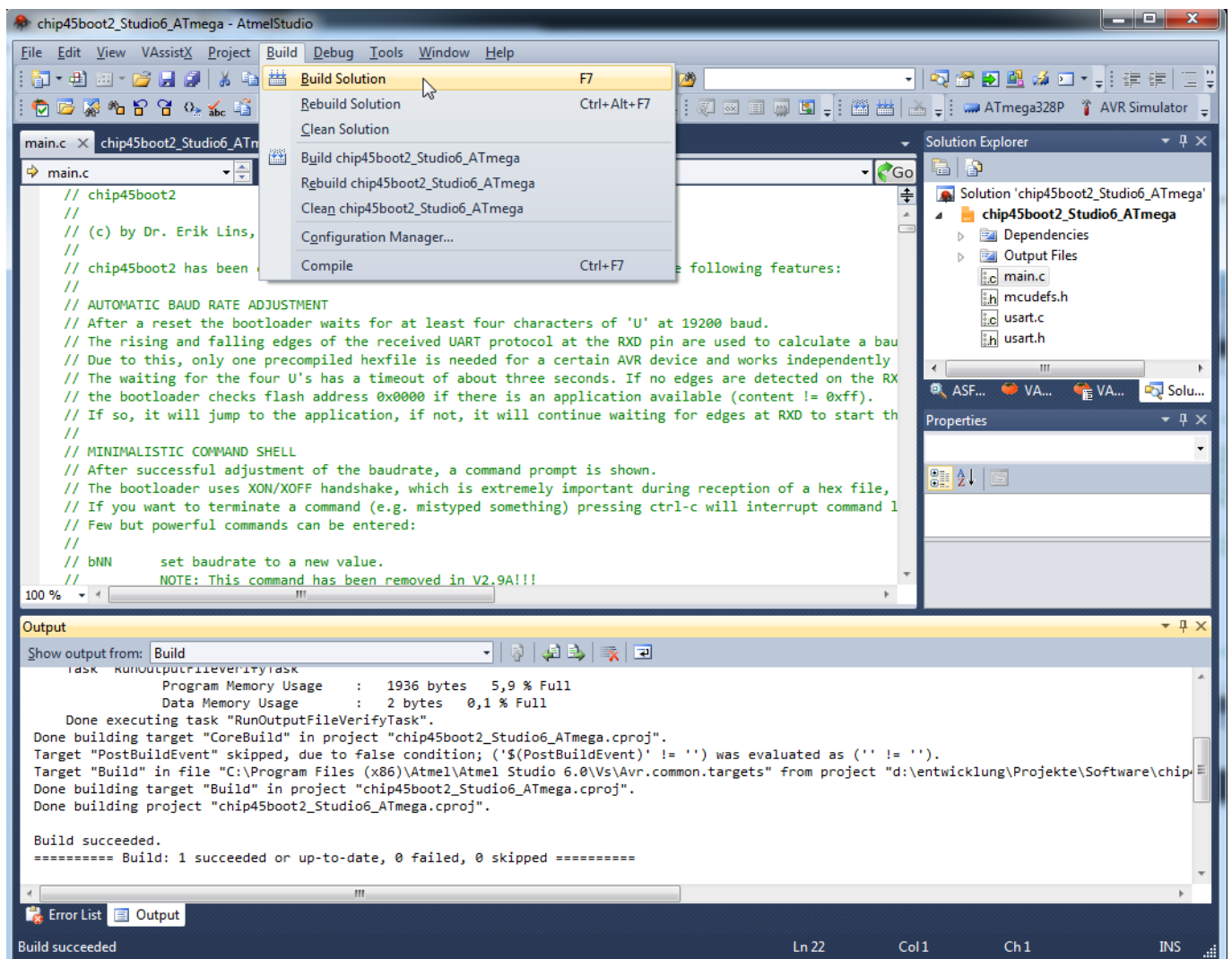
COMPILING FOR ANOTHER TARGET

The Studio 6 ATmega project is preset to generate a bootloader for the following configuration:

- ATmega328P device
- USART0
- no RS485 support

Let's assume you prefer another configuration, like this:

- ATmega644PA device
- USART1
- RS485 supported

No we apply the necessary modifications the the Studio 6 projects settings step by step.

Changing the MCU target

Select "Properties" from the "Project" Menu (it's the bottom entry and might also be labeled "chip45boot2_Studio6_ATmega Properties", depending on which file is selected in the Solution Explorer).

In the left part of the Studio 6 window, a new tab with project settings will show up. It has several vertical tabs (Build, Build Events, Toolchain, etc.) on it's left, where you please select the "Device" tab. A page with the currently selected ATmega328P will open, which shows some description on the target, the data sheet, supported tools and also a button "Change Device...". If you press this button a new window opens, where you can select a new target device.
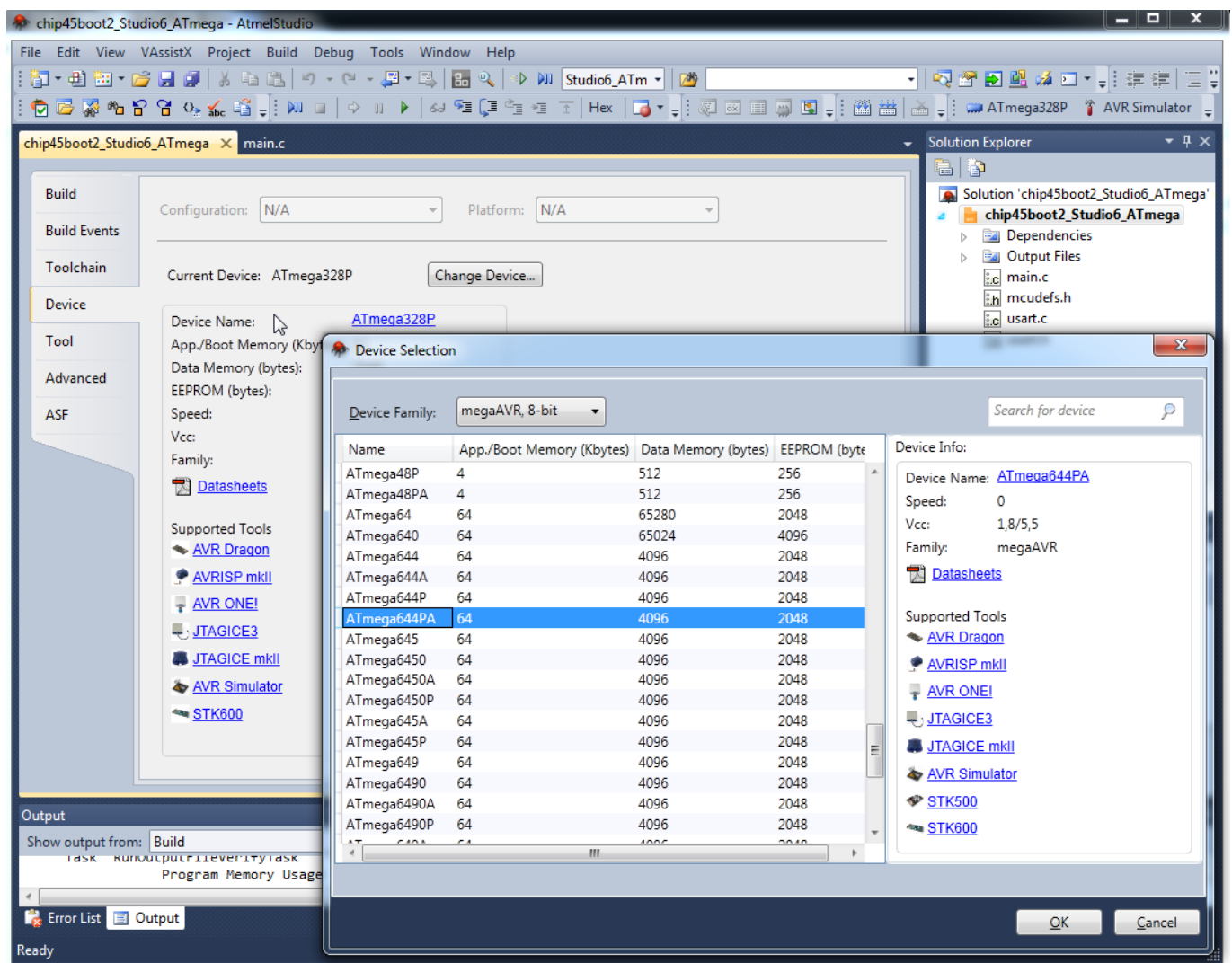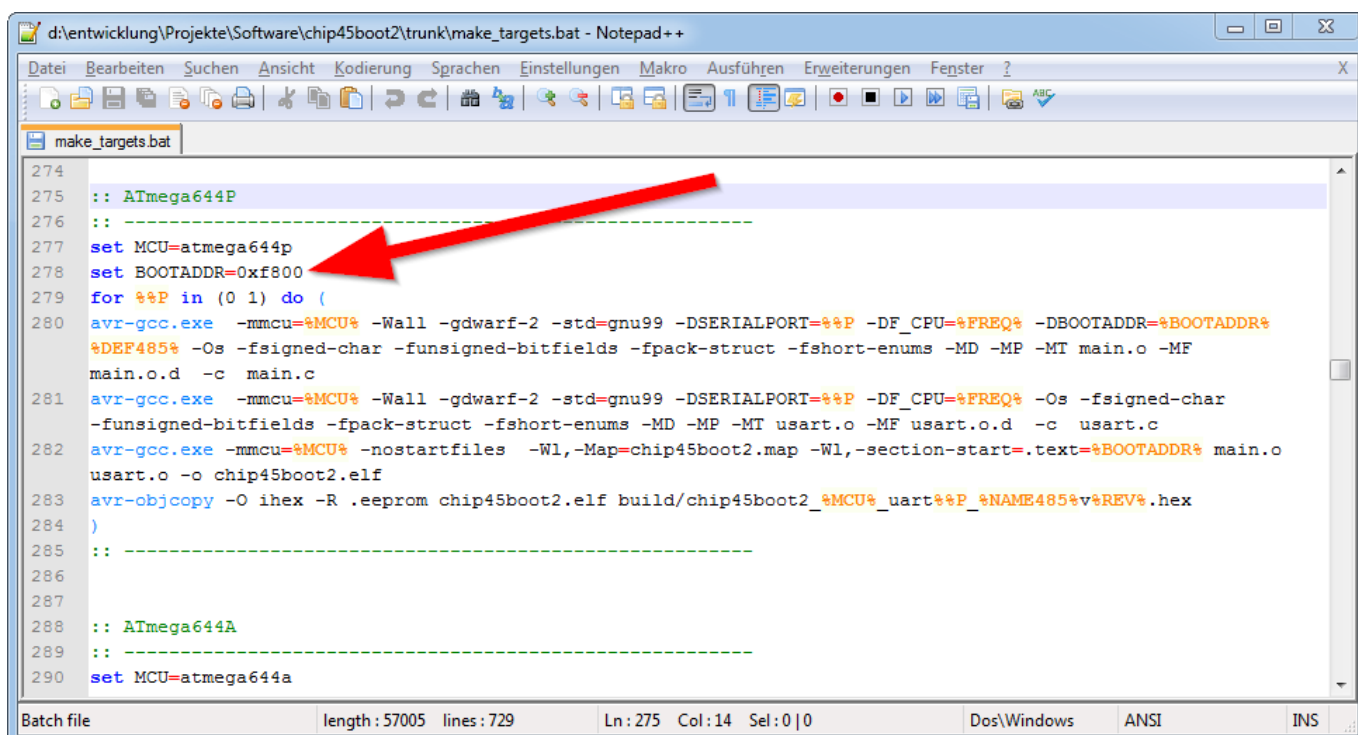

Please scroll to the ATmega644PA (or if you're bold, select your desired target here!), select it and close the window with "OK". Now the ATmega644PA is shown as the active target in the project properties window.

chip45boot2 Studio 6 Notes

Changing target specific memory settings

This is one of the most important issues when compiling a bootloader. Whilst a normal application resides at the beginning of the program memory at address 0x0000, the bootloader must be located in the boot block at the end of the program memory and the linker has to be told to locate the code at that address. And since the boot block can have different sizes, the start address can vary. On Xmega targets, the bootblock has a fixed 8kB size and is located above the regular program memory, but since program memory varies amongst device, the bootloader start address can vary also for Xmega targets.

The chip45boot2 is designed to fit into a 2kB boot block on ATmega devices. So the bootloader start address equals the flash size minus 2kB. For the 64kB ATmega644PA, this address is 65536-2048=63488 or as hex number 0xF800. Since the ATmega644PA is a supported target (or at least the mega644P and the mega644A are...), you find this information in the make_targets.bat batchfile. When you open this file with a text editor, search for ATmega644, you find three sections for ATmega644, ATmega644P and ATmega644A and one is shown in the following picture. The bootloader start address is marked red.

Now we need to put this address into the Studio 6 project settings.

Please open the Studio 6 project properties tab "Toolchain". On the right you will see a list of toolchain members, like compiler, linker, assembler, etc. Please open the entry "AVR/GNU Linker" → "Memory Settings". Now you see further memory settings to the right with the particular setting ".text=0x3C00" in the "FLASH segment" box. Look at the following picture to see a screenshot of this.

If you double click on that ".text=0x3C00" setting, a small dialog box comes up, where you can alter this setting.

Now we need to take an AVR speciality into account: The AVR architecture uses 16 bit opcodes and hence addresses program memory in words and not in bytes. According to this, also the Atmel datasheets use word addresses and we also need to tell the Studio 6 our desired bootloader start address as a word address, not a byte address. The calculation above as well as the setting in the batch file were byte address. So we need to devide those numbers by two to get a word address: 0xF800 / 2 = 0x7C00. You may now enter this number as the new value for the .text segment.

You can also cross-check this value in the ATmega644PA datasheet: The picture right shows a table for the possible boot block sizes and the corresponding bootloader start addresses. You find the 0x7C00 marked red.

So we can assume our calculation as correct and proceed.

164A/164PA/324A/324PA/644A/644PA/1284/1284P

25.8.16 ATmega644A/ATmega644PA Boot Loader Parameters

In Table 25-13 through Table 25-15, the parameters used in the description of the Self-Programming are given.

Table 25-13. Boot Size Configuration[(1)]

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Application Flash Section	Boot Loader Flash Section	End Application Section	Boot Reset Address (Start Boot Loader Section)
1	1	512 words	4	0x0000 - 0x7DFF	0x7E00 - 0x7FFF	0x7DFF	0x7E00
1	0	1024 words	8	0x0000 - 0x7BFF	0x7C00 - 0x7FFF	0x7BFF	0x7C00
0	1	2048 words	16	0x0000 - 0x77FF	0x7800 - 0x7FFF	0x77FF	0x7800
0	0	4096 words	32	0x0000 - 0x6FFF	0x7000 - 0x7FFF	0x6FFF	0x7000

chip45boot2 Studio 6 Notes
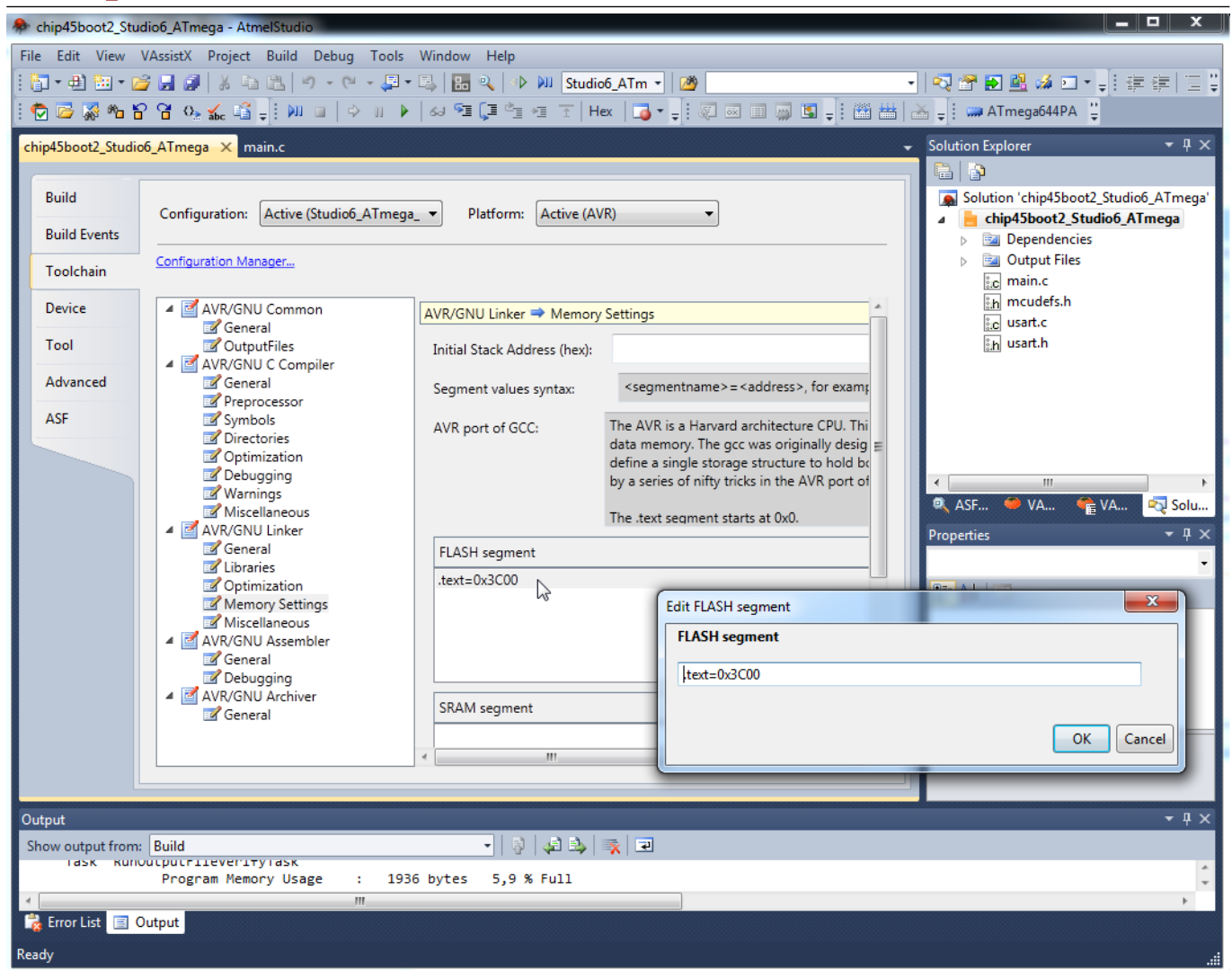
"Why the hell don't you use the correct word address in your batch file?"
The Studio 6 tries to hide the actual AVR-GCC compiler issues from us and make us feel convenient with using things, we might know from the data sheet, like word addresses. In fact the avr-gcc doesn't care about that Atmel-using-word-addresses stuff and since it is a general purpose compiler also available for other MCU architectures, it uses byte addresses as most other architectures do. Since we call the compiler directly from our batch files, we need to tell him the real (and correct...) byte addresses. Even the Studio 6 will call the avr-gcc later with our computed word address multiplied by two. You can check this in the Output window part later. To confuse more, the former AVR Studio 4 did use byte addresses. Thanks Atmel!

Changing the USART and/or RS485 support
The bootloader source code already has code included for all possible USARTs of a target as well as code for RS485 support. It's mostly conditionally compiled into the bootloader by #ifdef … #endif directives in the main code as well as in mcudefs.h, where all the USART specific stuff, like registers and IO pins are declared. To tell the compiler our wish for the USART and RS485 support, we need to set this as symbols in the project settings. In the "Toolchain" tab, we open "AVR/GNU C Compiler" → "Symbols" and see the predefined symbols on the right. See the following screenshot for details.

Here we meet again the bootloader start address as symbol. Also the bootloader needs to know about it's start address during runtime, to avoid for overwriting itself in case an incoming hexfile is larger than possible. BUT: At this point, the symbol must be given as byte address, since the bootloader thinks in bytes! So we take our previously calculated 0xF800 and put it here. No division by two necessary here! Please change the BOOTADDR to 0xF800.

The second symbol sets the desired USART for bootloader communication. In our example, we wanted to go for USART1, so and set the SERIALPORT to 1, which means USART1. On ATmega devices with more than two USARTs, you can go up to 3 here. On Xmega devices, you can go up to 7, but the meaning here is: 0=USARTC0, 1=USARTC1, 2=USARTD0, 3=USARTD1, etc. since the Xmega USARTs are named slightly different.

chip45boot2 Studio 6 Notes
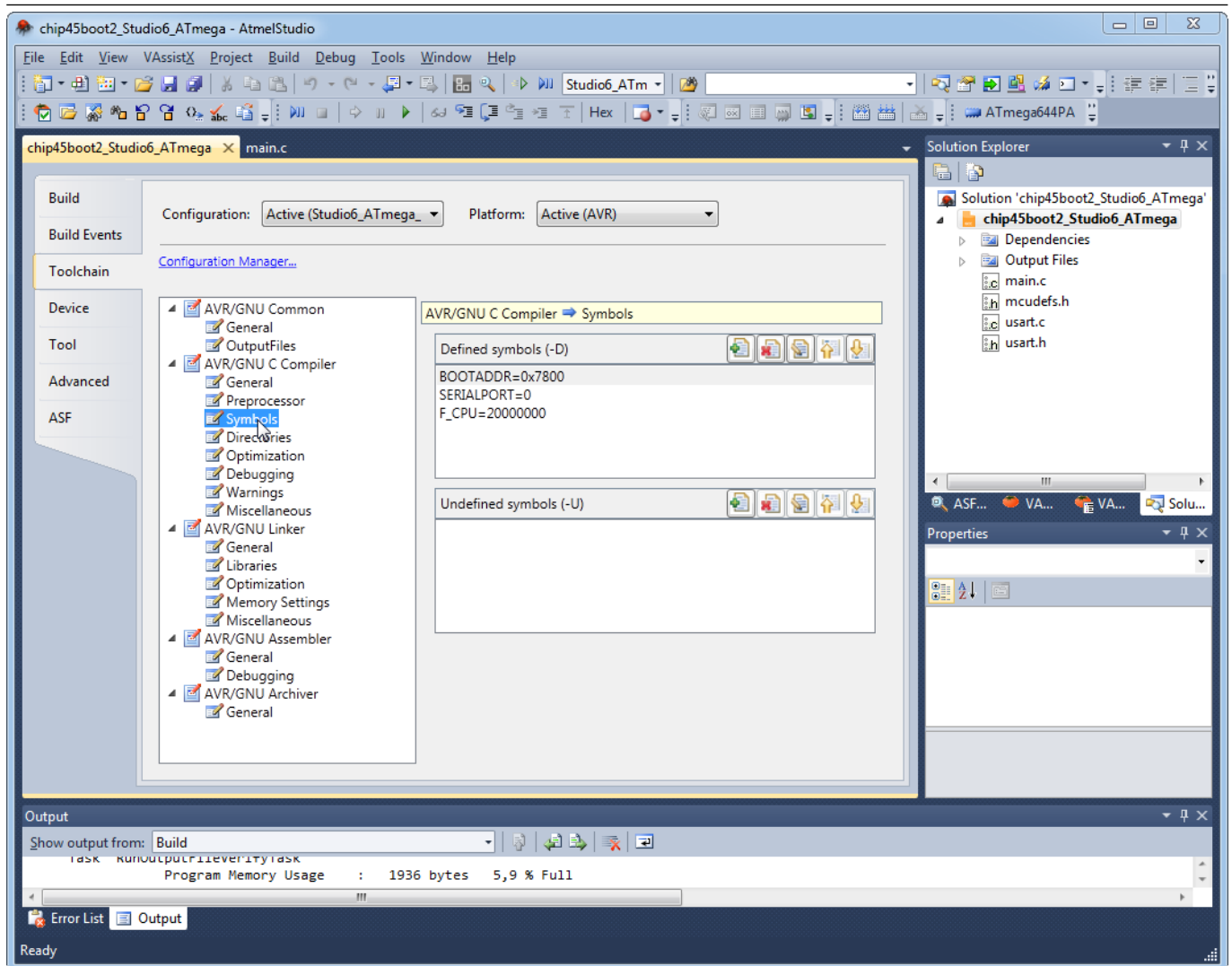
The next symbol F_CPU specifies the MCU clock frequency, but it is only used on very old ATmega targets, which cannot use the watchdog timer in interrupt mode. In those rare cases, a simple delay loop is used for timeout and the F_CPU setting is used to achieve an approximately 2 seconds timeout. On all other targets, this value can be ignored (except you add own code to the bootloader, which uses it...). So feel free to enter the correct frequency in Hz or ignore it.

In case you want RS485 support in your bootloader, you need to add an additional symbol called "RS485" to the settings. Click on that green "+" icon and enter "RS485". The symbol list should now look like shown on the right.
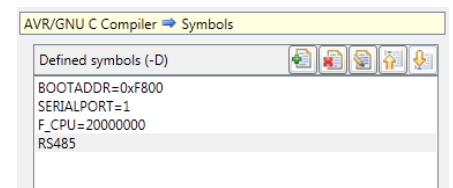
Compiling and testing
Press F7 to build the bootloader for the new target.
If everything is ok, you should see some scrolling-up of compiler messages in the Output Window, which should end with messages like this:
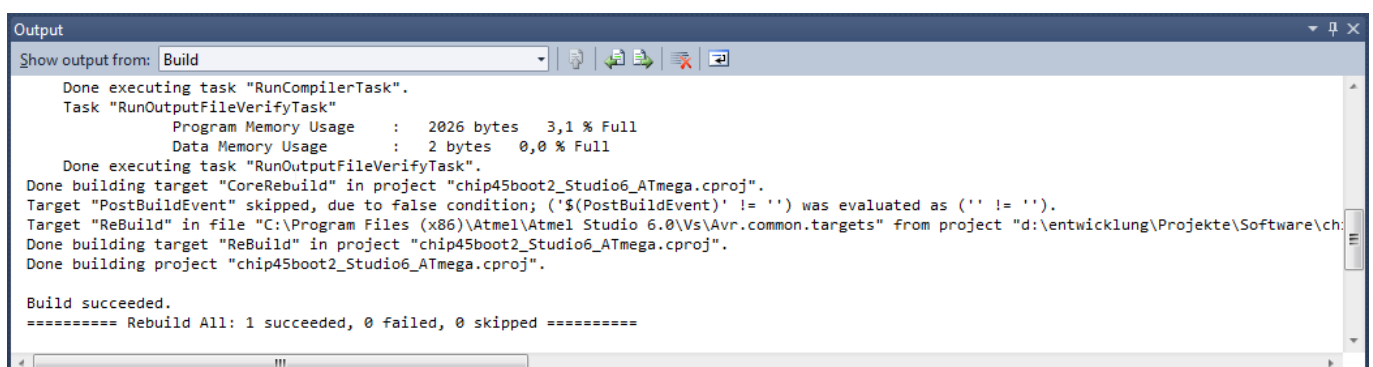
If you observe any errors or warnings, please double check all settings above and try again.

PORTING TO A NEW TARGET

We assume, that you have read and understood the above description on "Compiling for another target".

If so, porting the bootloader to a new target is not much different to the above, with one big exception:

There is no ready available section with definitions in mcudefs.h or make_targets.bat, where you can extract values like bootloader start address etc. from. But no problem, we will give some advice, how to fix that.

Add a new section to mcudefs.h

If you open mcudefs.h with Studio 6, you will see quite a lot of sections like this:

```
// set of definition for ATmega328P
#ifdef __AVR_ATmega328P__
            #define WATCHDOG
            #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 PIND
            #define myRXDPORT PORTD
            #define myRXD PIN0
            #define myTXD PIN1
            #define myDIRDDR DDRD
            #define myDIRPORT PORTD
            #define myDIR PIN4
#endif
```

Some section have subsections for different USARTs, but this is no big deal. So the first thing to do, is to find a section from a target, which is as similar as possible to our new target. Similar means, that it's a target with more or less equal "age" (since very old targets have this URSEL bit in it's USART register to take care of, or can't use the watchdog timer in interrupt mode), ideally the same quantity of USARTs (which is not so important, since you might just want a bootloader for one USART), the same amount of flash memory and a watchdog with the same interrupt capability. Example: If you want to make a bootloader for a modern ATmega1280 with 128k flash, watchdog interrupt, four USARTs, don't start with the section of the good old ATmega162 with 16k flash, which has that URSEL bit and only one USART and no watchdog timer interrupt.

Let's assume you want a bootloader for an ATmega169PA, which is currently not in the list of supported devices. It has 16k flash, one USART and no URSEL bit but it's watchdog timer doesn't provide an interrupt. A similar target would be an ATmega16 with also 16k flash, one USART without URSEL and no watchdog timer interrupt.

So we simply copy the ATmega168PA section to another location of the mcudefs.h file and first change the #ifdef line at the beginning with the correct MCU symbol definition. How do we know the correct name of that definition? A good guess would be to use __AVR_ATmega169PA__ instead of __AVR_ATmega16__. This is correct for many cases, but sometimes that symbol has a strange name, which also includes some chip revision numbers or so. This is more often the case for Xmega devices, than for ATmega.

The best (or most safe) way is, to look up that definition in the target-corresponding io.h header file in the AVR GNU toolchain directory. So we need to jump deeply into the Studio 6 installation directory, which on a standard Windows 7 installation is here:

c:\Program Files (x86)\Atmel\Atmel Studio 6.0\extensions\Atmel\AVRGCC\3.4.1.95\AVRToolchain\avr\include\avr\

The picture right shows the upper part of the content of that directory with the io.h marked red.

If you open that file with a text editor, you will see really many #if defined … #elif defined … directives like this:

```
#if defined (__AVR_AT94K__)
#  include <avr/ioat94k.h>
#elif defined (__AVR_AT43USB320__)
#  include <avr/io43u32x.h>
#elif defined (__AVR_AT43USB355__)
#  include <avr/io43u35x.h>
#elif defined (__AVR_AT76C711__)
#  include <avr/io76c711.h>
#elif defined (__AVR_AT86RF401__)
#  include <avr/io86r401.h>
#elif defined (__AVR_AT90PWM1__)
#  include <avr/io90pwm1.h>
#elif defined (__AVR_AT90PWM2__)
…
```

boot.h	19.11.2012 19:06	C Header File	26 KB
builtins.h	19.11.2012 19:06	C Header File	4 KB
common.h	19.11.2012 19:06	C Header File	8 KB
cpufunc.h	19.11.2012 19:06	C Header File	4 KB
crc16.h	19.11.2012 19:06	C Header File	2 KB
delay.h	19.11.2012 19:06	C Header File	2 KB
eeprom.h	19.11.2012 19:06	C Header File	25 KB
fuse.h	19.11.2012 19:06	C Header File	9 KB
interrupt.h	19.11.2012 19:06	C Header File	12 KB
io.h	19.11.2012 19:06	C Header File	20 KB
io43u32x.h	19.11.2012 19:06	C Header File	12 KB
io43u35x.h	19.11.2012 19:06	C Header File	11 KB
io76c711.h	19.11.2012 19:06	C Header File	11 KB
io86r401.h	19.11.2012 19:06	C Header File	8 KB
io90pwm1.h	19.11.2012 19:06	C Header File	35 KB
io90pwm2b.h	19.11.2012 19:06	C Header File	30 KB
io90pwm3b.h	19.11.2012 19:06	C Header File	30 KB
io90pwm81.h	19.11.2012 19:06	C Header File	22 KB
io90pwm161.h	19.11.2012 19:06	C Header File	19 KB
io90pwm216.h	19.11.2012 19:06	C Header File	40 KB
io90pwm316.h	19.11.2012 19:06	C Header File	40 KB
io90pwmx.h	19.11.2012 19:06	C Header File	45 KB
io90scr100.h	19.11.2012 19:06	C Header File	34 KB
io1200.h	19.11.2012 19:06	C Header File	6 KB
io2313.h	19.11.2012 19:06	C Header File	10 KB

Typ: C Header File
Größe: 19,1 KB
Änderungsdatum: 19.11.2012 19:06

Here you need to find an entry, which mates your desired target. Search for "mega169" and let's see, what we find:

```
#elif defined (__AVR_ATmega169__)
#  include <avr/iom169.h>
#elif (defined __AVR_ATmega169A__)
#include <avr/iom169a.h>
#elif defined (__AVR_ATmega169P__)
#  include <avr/iom169p.h>
#elif defined (__AVR_ATmega169PA__)
#  include <avr/iom169pa.h>
…
```

Bingo! The lowest entry is the correct one for our ATmega169PA and the one, we have to use for our new section in mcudefs.h:

```
// set of definition for ATmega169PA
#ifdef __AVR_ATmega169PA__
          #define myMCUSR MCUSR
          #define myWDCE  WDCE
          #define myUDR   UDR0
…
```

The next step would be to go through the section and check all the register names and bit names. As you can see above, the myMCUSR definition has to be adjusted from MCUCSR to MCUSR. Also very important are the USART RXD and TXD pins:

```
          #define myRXDPIN PIND
          #define myRXDPORT PORTD
          #define myRXD PIN0
          #define myTXD PIN1
```
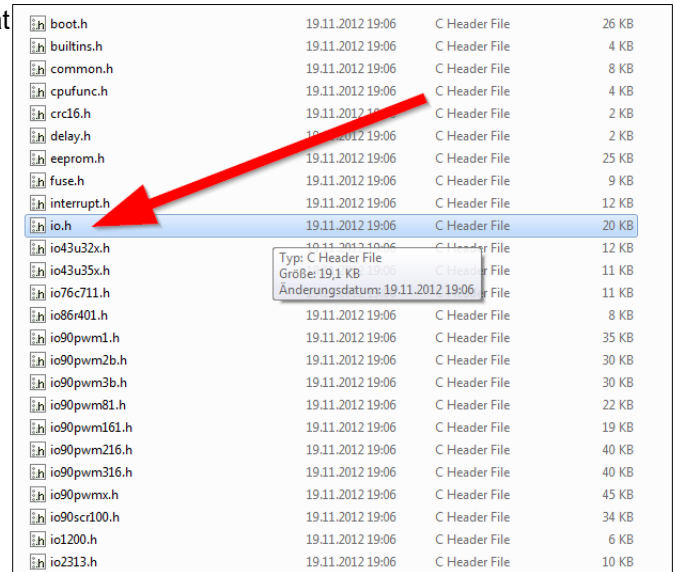
During automatic baud rate adjustment, the pins are not used through the USART peripheral, but are used as normal IO pins, so these settings are important to check.

The last three definitions with myDIR... are used for RS485 only, so you might ignore them, if RS485 is not desired.

Change the new target in the Studio 6 project settings
This has already been described in the first section for compiling the bootloader for another target and should be looked up there. The major steps are:

- get the bootloader start address for a 2kB bootblock from the datasheet (or from a similar target in make_targets.bat) and put it into the "Toolchain" → "Linker" → "Memory Settings". If extracted from make_targets.bat, don't forget to devide by two!
- adjust the symbols in "Toolchain" → "Compiler" → "Symbols" to your needs. Set the BOOTADDR (datasheet value * 2), chose SERIALPORT, probably correct F_CPU and if desired, enable RS485 support.

Build the new bootloader
If you chose a target, which is supported by the toolchain and did not make mistakes in mcudefs.h or the settings above, you should be able to compile the new target by pressing F7.

MIT License

Copyright (c) 2023 Erik Lins

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.