Albrecht Weinert

AVR ATmega development report

A serial bootloader for ATmega based products – weAut_01, Arduino and akin

Rev. 1.9, Sept.19 2014

Prof. Dr.-Ing. Albrecht Weinert a-weinert.de
weinert – automation weinert-automation.de

Labor für Medien und verteilte Anwendungen (MEVA-Lab) meva-lab.de
Laboratory for Media and versatile Applications

Fachbereich Informatik der Hochschule Bochum
Computer Science department – Bochum University of Applied Sciences

A serial bootloader for ATmega based products – weAut_01, Arduino and akin

V01.01, 20.06.2013:	amendments by readers requests
V01.02, 08.09.2013:	Bootloader supports more targets; +small chapter on usage
V01.05, 05.11.2013:	major enhancements, title changed
V01.07, 30.11.2013:	minor corrections; some complements
V01.08, 12.03.2014:	some amendments due to operating experience
V01.09, 19.09.2014:	more target platforms

Version: V1.09

Last modified by A. Weinert at 31.10.2014

Note on numbering: There is one common numbering for figures, lists, tables etc.
Standard (Arabic) page numbering starts here – and not after the content table.
Most pdf readers won't handle the offset involved when directed to a page number.

Note on version control, SVN URL: https://ai2t.de/svn/albrecht/pub/AVRserBootl.odt

Note on publications: See also http://a-weinert.de/publication_en.html ,
http://blog.a-weinert.de/ and
http://blog.a-weinert.de/avrserbootl/

This document's URL: http://a-weinert.de/pub/AVRserBootl.pdf.
That might be newer if this is from elsewhere or on paper.

The bootloder's download URL
weinert-automation.de/files/openSource/opSour_weAutSys.zip

Table of content

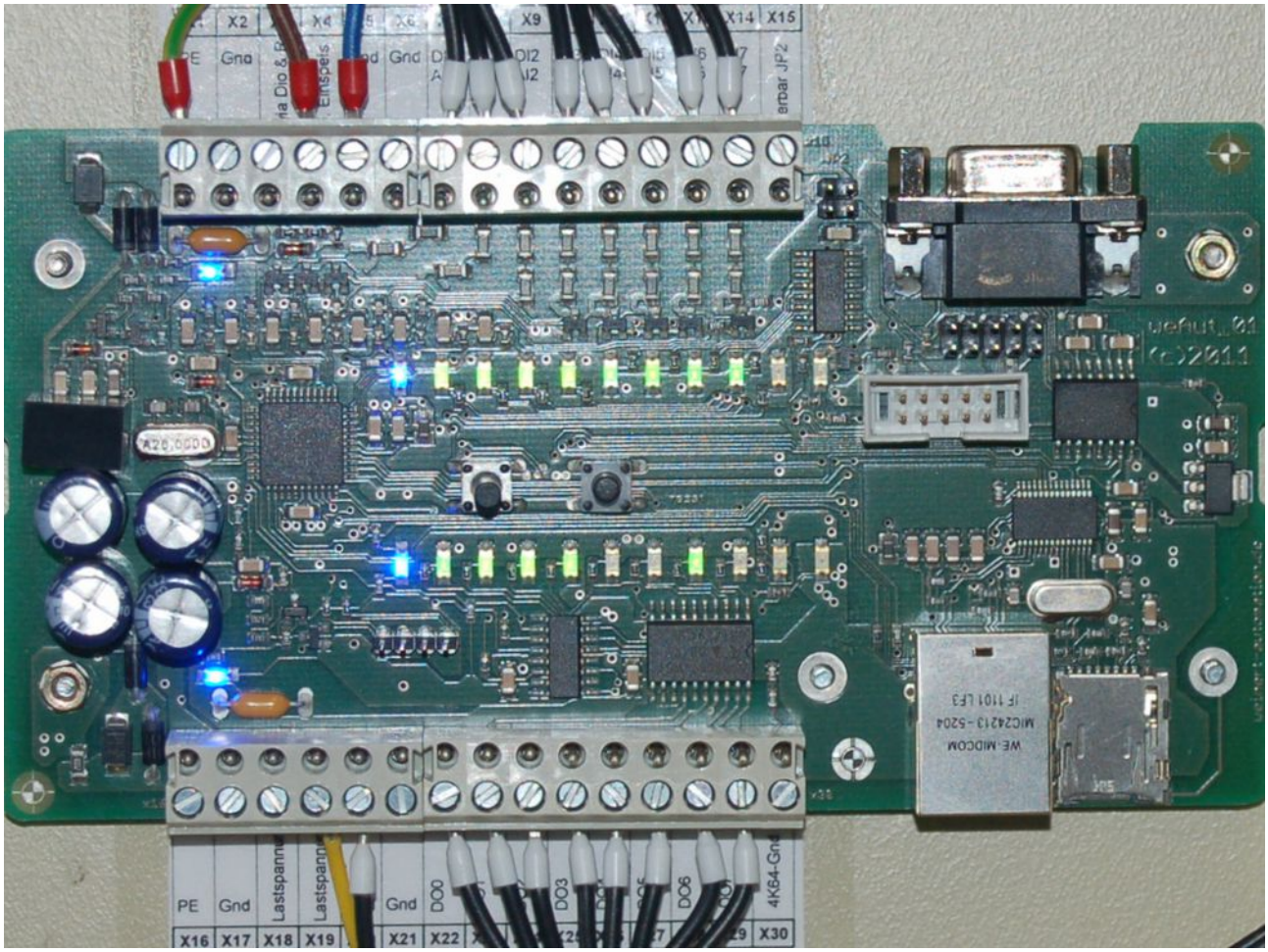

Figure 1: The automation module weAut_01 (weinert – automation 2012)

1. Motivation and Scope

This is about a serial bootloader for the 8bit AVR ATmega controllers – from small size devices with 16K flash and below and larger ones with up to 128 and 256K flash.

- The serial bootloader has proven operational experience on six + quite different ATmega based products.
- It enhances the usability and eases development work.
- This serial bootloader is available as open source.

1.1 Intended audience

This report is on the serial bootloader's

- development and usage (chapters 3 and 5) as well as
- the technical background, the protocol used and
- the implications by ATmega's computer architecture.

The latter two points are described in some elaborateness in chapters 2 and 4.

Hence this will be helpful for the experienced ATmega user just wishing to incorporate and use a (better) serial bootloader.

It might be even more beneficial for those developing own boot loaders software or otherwise interested on ATmega computer architecture and its implications to advanced program development in C.

Those who want to use Arduino boards just as cheap evaluation boards for ATmega based development, uncoupled from Arduino development tools and predefined (sketch) software, may find this especially useful.

1.2 Solution and targets

This serial bootloader was made for and tested on ATmega based modules as

- weAut_01 – automation module with industry standard process I/O,
- ArduinoMega2560 and ArduinoMegaADK – these are quite bare ATmega evaluation boards,
- ArduinoUno, ArduinoNano, eaysAVR – raw ATmega evaluation boards, too,

and other alike products.

See also table 2 on page 7, The AVR µControllers (µC) used on these boards are this bootloader's programming target:

- ATmega2560
- ATmega1284P
- ATmega28P
- Atmega32L
- and more

See table 7 on page 12 for comparison. It will become obvious why the ATmega2560 and the platforms using it (like ArduinoMega e.g.) were the teasers.

This serial bootloader can easily be used with or adapted to any platform fulfilling two pre-conditions:

- an ATmega µC with bootloader support also known as
- self-programming capabilities

and

- a serial communication link connectable via COMx (Windows) respectively /dev/ttySx (Linux), be it via
    i. V.24 / RS232 SubD and e.g. Maxim's MAX202E as physical layer adapter and protection IC as in weAut_01 or
    ii. a USB to serial bridge implemented e.g. in an ATmega8U2 µC as with most newer Arduinos and many small ISP programmers

For solution ii – the USB to serial link – the corresponding USB driver will emulate a

COMx port on Windows respectively a

/dev/ttyUSBx (or similarly named) device on Linux.

Hint: Solution ii is also available in small cheap modules to give a true serial V.24 / SubD link to PCs and Laptops lacking it.

Hint2: Using solution ii with many USB2serial devices (µController) boards on Windows will populate the system with as many additional com-ports. Some tools (written in C/C++) have problems with two digit coms.
The workaround, both ugly and incomprehensible, is to prepend `\\.\` [sic!] to the comXY like in the example:

```
avrdude -p atmega328p -c avr109 -b 38400  -P \\.\com10  -v -t
```

1.3 Bootloader's Advantages

The main advantages of this serial bootloader are:

- No (more) need for extra programming hardware
- Using a standard communication link, often used anyway in normal operation
- Using a standard programming protocol
- Good integration / co-operation with system / application software
- Available without fee, open source
- Flexibility, covering a wide spectrum of ATmegas and target boards
- Arduino's “jail break” – Free Arduino!
- Utilising standard professional tool chains and libraries – like e.g. Protothreads

No programming hardware:

The bootloader is just an extra small program in the target µC residing in a protected high address range of the program memory.

Hint: In conformity with the prevalent bad habit we call the ATmega's program memory often “flash memory” or just “flash”. (That's the chip technology and not the role in the Harvard architecture.)

Standard communication link:

The serial link is provided for and often used in normal operation.
That is never the case for ISP or JTAG (programming) interfaces.

By utilising a “normal operation” link also for bootloading one needs no physical access to the board. An automation module, like weAut_01 e.g., is usually mounted in the controlled process's equipment cabinet or on an industrial robot's arm, where direct access may often be inconvenient or not desirable. With a (remote) serial link the switching from HMI communication to programming on a PC is as easy as clicking disconnect on the terminal program (HTerm e.g.) and starting the AVR programmer tool (like the well known AVRdude).

Standard programming (communication) protocol:

No new invention or documentation is needed here. The wide spread serial programmer protocol is (to a quite great extent) well documented as AVR109 in respective application notes [AVR109a, b]. And the protocol is well implemented in programming software – like the quasi standard AVRdude tool – on one side and in some programmers respectively bootloaders – like this one – on the other side.

Beware: But, alas, experience shows quite bad implementations – on both sides of the AVR109 protocol – can also be found.

Standard professional tool chain:

Thanks to using standard protocols and tools available this bootloader does not break the usage of the professional standard toolchain for embedded AVR / ATmega development. A professional toolchain might and will most often consist of

- Subversion,
- Eclipse,
- AVRgcc,
- tradition tools, like (GNU) make, grep, Gawk &c.,
- AVRdude as well as
- proven professional open source libraries like Adam Dunkels' Protothreads

The only change her by this bootloader is the simplified hardware handling and a slight change the programming scripts concerning AVRdude options.

Integration of bootloader software to the application program:

Motivated by its primary target – the automation module weAut_01 and its runtime weAutSys – this bootloader is designed to co-operate well with the system / application software. It is to be entered on every reset (by fuse setting) and from the application (by command, i.e. HMI / CLI). The bootloader will go (back) to the normal program's entry by a set of criteria and with little delay on a normal external reset or supply voltage restart.

The bootloader will always do the platform's basic initialisations – hence they can be omitted in the system / application software. And the bootloader functions and (flash final) variables are available ("linkable") to the normal software. That reduction in initalisation workload and more the provision of tested library functions (not to be repeated) can also reduce the net loss of flash space by introducing a bootloader – if perceptible anyhow.

Free Arduino!

This bootloader won't break the (standard) toolchains, professional libraries and workflows most professionals use in AVR / ATmega embedded development. That repeated, the bootloader obviously will open these tools to many Arduino boards. This is the "jail break" from "writing sketches" to "embedded programming". And it opens a wast range of quite cheap "naked" ATmega evaluation boards to professional development and teaching.

Flexibility, covering a wide spectrum of ATmegas:

The number of target boards and processors this bootloader was successfully used on is one or two handful. nevertheless, the ATmegas used covers the whole range of sizes and related architectural differences (or complications). Porting to further ATmegas with self-programming capability will hardly require any source code change.

Regarding the porting to other / new target boards (cf. table 2 on page 7) three aspects might require additions to the source code in form of an extra #elif-branch:

- using one or more LEDs to display bootloader state or liveliness
- using en extra (enter) switch or button to stay in the bootloader programme for a longer time
- doing target specific I/O and port initialisations in the bootloader

Summary

All advantages listed in this chapter were proven in practice.
An extra benefit was this serial bootloader's outperforming all (i.e. some half-dozen known / bought) ISP solutions in speed.

Remark: This is a bit astonishing, as a high speed synchronous link (ISP) is a technical base for much higher transmission speeds than is an asynchronous UART at 38400 8N1.

2. Targets

2.1 Modules

	weAut_01	ArduinoMega2560 (/ADK)
type	automation module	evaluation board
developer / manufacturer	weinert – automation Bochum Compudata Sprockhövel	open community multiple sources
µController	ATmega1284p	ATmega2560
memory	16K RAM / 128K Flash / 4K EEPROM	8K RAM / 256K Flash / 4K EE
CPU clock	20 MHz	16 MHz
supply 1	9..29V (load voltage) -0.5..+33V max.	6..10V (regulator), 5V (direct)
supply 2	9..30V (redundant sup.) -120..+33V	USB feed (5V)
protection	Supressor diodes, fuses	no supply protection
surveillance	Load voltage comparator, 4 LEDs	one green LED “on” (5V)
buffering	20 ms when both supplies drop out	none
process input digital (DI)	8 DI with settable thresholds and hyste-resis for 12V and 24V LV (Namur etc.)	no protected process I/O abundance of (µC/TTL) ports
analogue (AI)	those 8 DIs optionally usable as AI in three ranges up to 0..40V	max. 16 bare µC ports usable with ADC
protection	+-60V nominal; max. rating 260V~eff.	no I/O protection
status display	row of eight green LEDs near input clamps, freely usable (normally for DI)	none
process output digital (D=)	8 DO, High side switch: 0V or Load voltage (LV)	abundance of bare µC ports
protection	current, temperature, ESD	no I/O protection
status display	row of eight green LEDs near output clamps, state of DO driver input; two red LEDs: DO disable and DO fault	none
status LEDs	all LEDs listed above + one extra red	one yellow
keys, buttons	2: reset, Enter key (longer knob)	one button: reset
serial link	V.24; SubD 9; optional flow control	USB port with USB2serial chip
Ethernet	10M full duplex (no POI)	none
ext. memory	solid µSD slot; insert switch	none
USB	none	1 see above; ...ADK: +1 extra
programmer	ISP interface 10 pin, frame and slot	ISP 6 pin, mechanically open
free µC ports	officially none, 5 including two wire	about 78 free µC ports
see also	[We1, We2], fig. 1 on page 2	fig. 3 on page 8

Table 2: The bootloader's main target modules (ArduinoUno omitted here)

The motivation to start the bootloader's development was to enhance the weAut_01 automation board ([We1, We2], fig. 1 on page 2) and its runtime weAutSys ([We3]). Soon the ArduinoMega2560 respectively ArduinoMegaADK, fig. 3 was added as second target in the course of the work. A step from one target to two is the “original sin” opening the door for further flexibility to add further targets – so the porting to ArduinoUno, ArduinoNano, easyAVR and the like respectively to ATmega328P, Atmega32L etx. was comparatively easy.

In the end this bootloader is usable for all flash size – from small (<= 64K) to quite large (128 & 256K) – ATmegas with boot loader support respectively self programming capability.

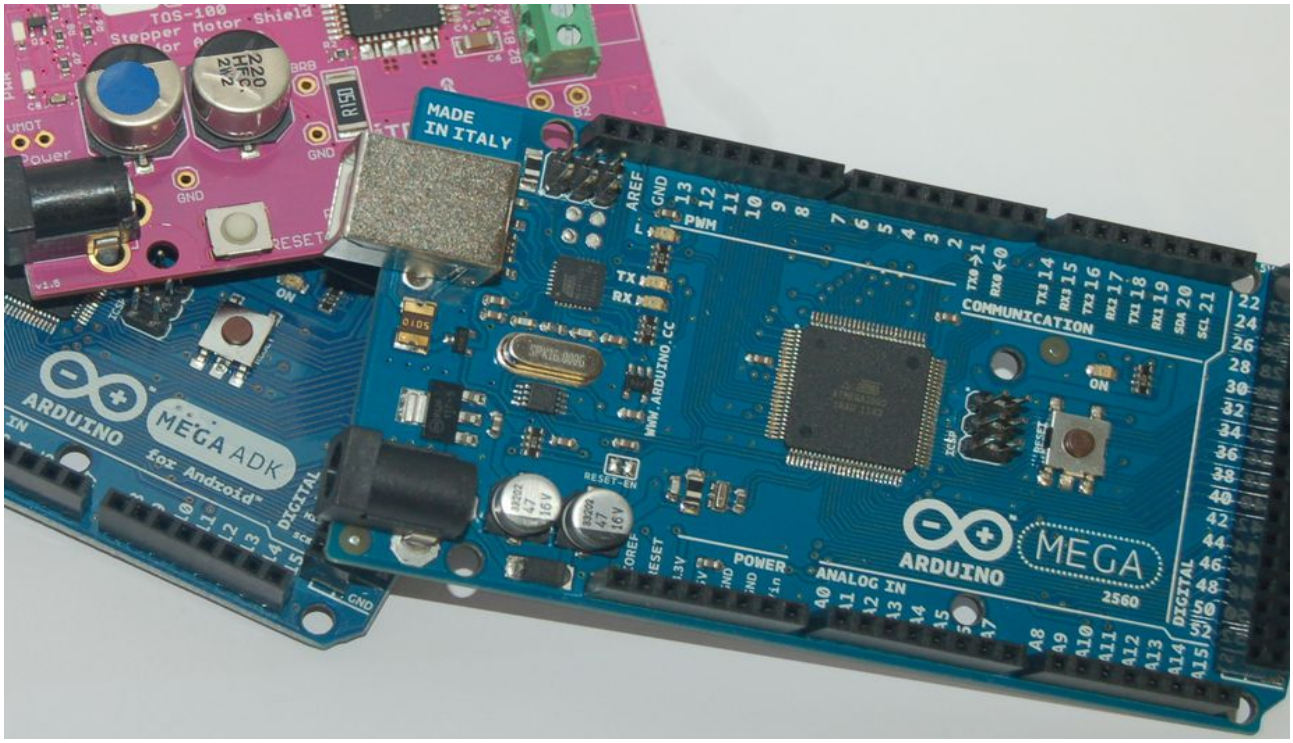

Figure 3: The ArduinoMega2560 (top), the ArduinoMegaADK (covered) and a “shield” (pink)

Table 1 on page 5 shows the bootloader's main targets in comparison. In a certain sense these both – an automation board with industry standard process I/O and a “naked” evaluation board – could not be more diverging. From the bootloader's point of view most differences shown in table 1 are of minor impact:

I **The enter key**:
The weAut_01 has a second button, named “Enter key”. The bootloader uses it as criterion to remain in bootloader or AVR109 protocol mode or at least use a quite long time-out before (re-) entering a normal system/application program. This extra “pressed button while reset” criterion is nice for the laboratory, and seems to have a long tradition.

But it can as well be omitted for

a) the Arduinos having no extra button nor a fixed port bit for boot enable

and for

b) the weAut_01 dwelling often physically inaccessible in the processes equipment cabinet.

II **The time-out light show**:

The weAut_01 has a row of 8 green LEDs parallel to the eight process DI/AI input clamps. These eight LEDs (controlled via SPI) are used by weAutSys to display the input states.

The bootloader uses them to display the countdown of a time-out. The maximum time-out of 72s starts with “filling” all 8 LEDs by dropping one light after another until all are on. This state is seen in the upper eight green LEDs on figure 1 on page 2. Then the inverse is done by dropping one “hole” after another until total darkness meaning timeout.

On the ArduinoMegas the bootloader just uses Port C (1 = LED on) for this handsome performance. The rationale is the double row connector at right side (that gives access to PortC) at the boards (and figure 3 's) right side is not used by most so called shields designed for the smaller Arduinos. Compare the red / pink board's size – that's a “shield” – with a blue ArduinoMega on figure 3, page 8.

But again, seeing the timeout run out, while the bootloader is eagerly waiting for programming software messages, is nice in the lab but useless for a remotely buried board and can as well be omitted – or should in case of the Arduinos.
The “lightshow” may be disabled (by C compiler macro setting) for the ArduinoMegas. This is, by the way, the default for ArduinoUno, as there is no usable 8bit DO port.

III **The USB / SubD interface difference**:

May be as a surprise, the difference of the both targets (cf. table 2 on page 7), most obvious when handling the programmer connection and software is totally transparent to the bootloader firmware, except for the “Caution 2” quality failure common to many Arduino boards – that is all Unos and Megas bought so far.

With or without intermediate USB2serial conversion, in the end it is just UART0 (38400, 8N2 and no flow control) on all target ATmegas. Without 16/20MHz difference even the UART initialisation would not be distinguishable.

Hint / Caution: The ArduinoMegas USB2serial bridge uses the DTR on to off transition for a reset. Other Arduinos may use RTS instead or additionally. This can be useful, e.g. for remotely resetting / entering a bootloader. But of course, when used inadvertently, it can be a case of failures quite hard to uncover. If the user / system software has a HMI / CLI with an enter bootloader command, as has weAutSys (boot -load) this feature may be disabled by scratching a conducting path (as is done on figure 5 on page 10).

Hint / Caution 2: All known Arduinos won't use the same frequency generators respectively crystals for the main ATmega, the USB2serial bridge (ATmega8U2) – even if it's the same frequency (16MHz) and less than 30 mm distance. Every decent 16MHz (HC-20) crystals will deliver 15,997 .. 16,003 MHz. But alas – in some Arduinos you'll find so low grade SMD 16MHz-crystals giving rubbishy 15,90 MHz for the main µC. With one good crystal (for the USB µC) the deviation may become to much for the (20 mm spanning) serial link when using 38400,8N1.
Symptoms (caused by this unprofessional hardware design) are more or less random complains like e.g.:

avrdude: error: programmer did not respond to command: set addr

Remedies:
i) correct and or sync the µController clocks: that's by far the best solution as done in figures 5 and 6 on page 10. Compare figure 5 to figure 3 on page 8.
ii) try to get the USB2serial bridge to use 2 stop bits
iii) reduce to 19200 baud or even less
iv) calculate and use other UART prescaler settings by using the “bad” (measured “shitCristal”) frequency by something like Listing 4.

```
uint16_t uartDivide(uint32_t baudRate, uint8_t x2){
#if defined(anyArduino) && defined(shitCrystal)  // set shitCristal
  // to the measured wrong quartz frequency (e.g. 15941400 )
  return (uint16_t)( ((uint32_t)(shitCrystal))/ (baudRate * (x2 ? 8 : 16) ) );
#else
  return (uint16_t)( ((uint32_t)(F_CPU)) / (baudRate * (x2 ? 8 : 16) ) );
#endif
} // uartDivide(uint32_t, uint8_t)
```

Listing 4: Arduino correction for 15,94MHz at ATmega2560 and 16,000MHz at ATmega8U2

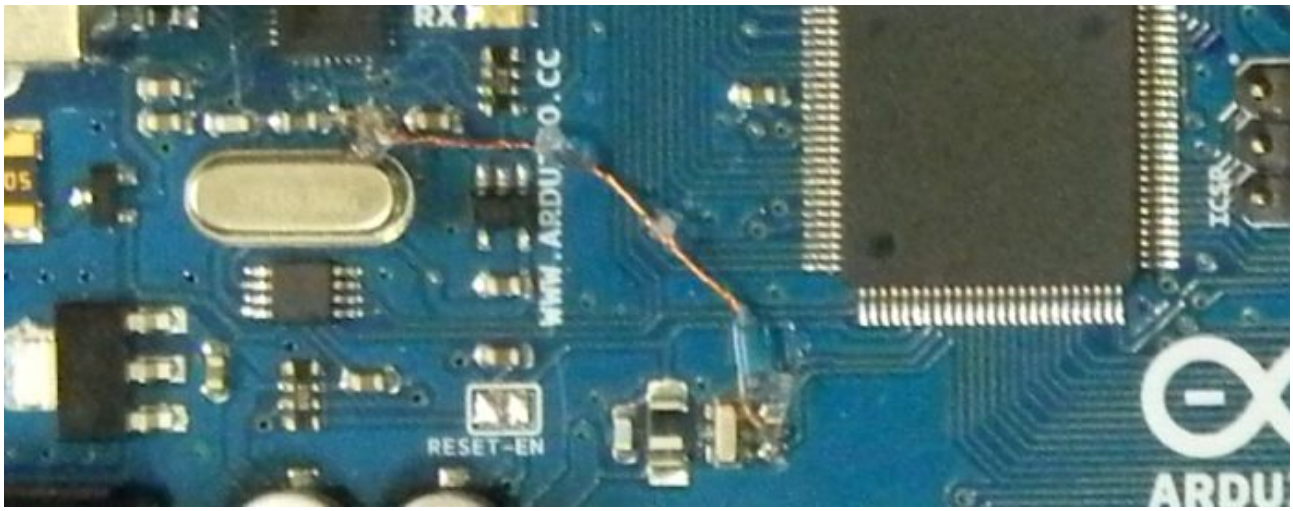

Figure 5: ArduinoMega2560: synchronising main µC's and USB2serial's (Atmega8U2) clocks

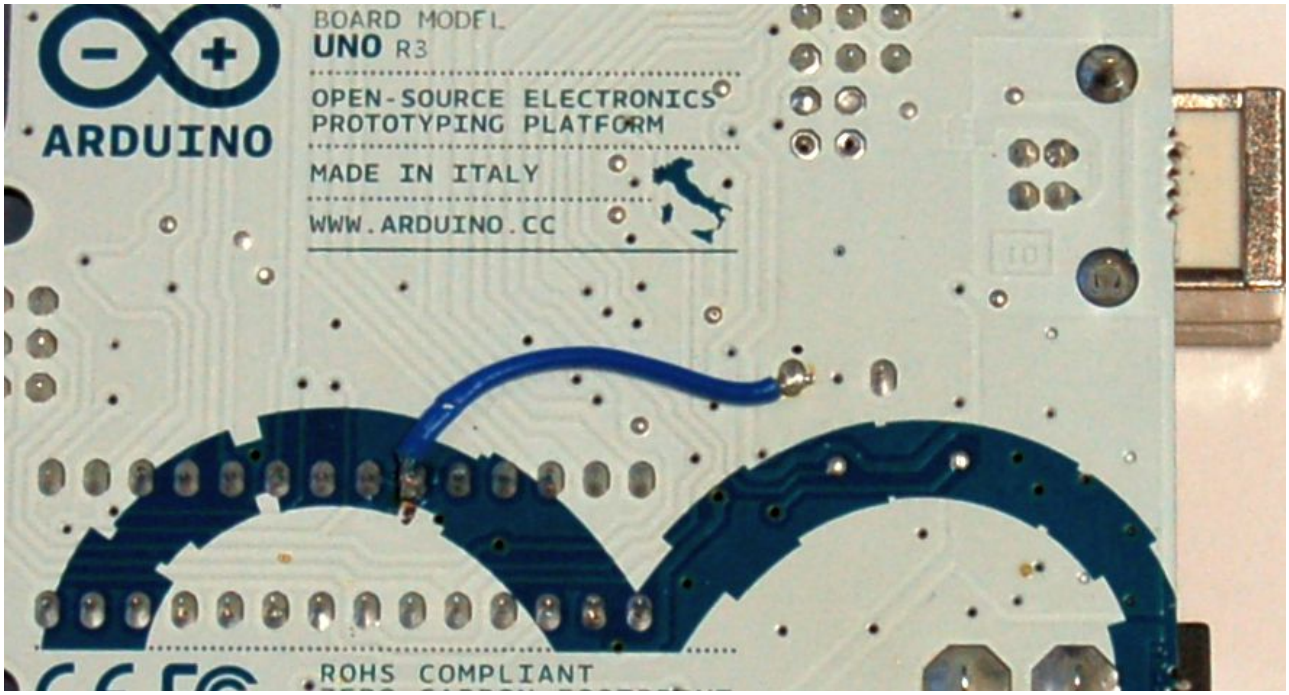

Figure 6: ArduinoUno (DIL): synchronising main µC's and USB2serial's (Atmega8U2) clocks

2.2 Controllers

The main targets use three different controllers – the ATmega328P, ATmega1284P and the ATmega2560. Table 7 on page 12 gives their properties, differences and AVR-gcc (include file) handling in a nutshell. The small ones, the ATmega328P and consorts, are not critical regarding the aspects discussed here.

The difference most impacting to this bootloader is the flash size (>=128kByte).

IV **The flash size difference**:

The difference of 256K to 128KByte flash size may seem less important as both sizes leave the 64K range addressable with one 16 bit or two 8 bit registers. For ATmegas the most prominent example is the Z-register or r30/r31 pair [AVR1].
But it's a bit more complicated.

Background:

The ATmegas are RISCs (reduced instruction set computers) in Harvard architecture. In contrast the intel80x86s e.g. – 8086 to 80586 (Pentium) etc. – are von Neumann CISCs.

Harvard means “architectually” separated data and program memory, here the ATmegas' RAM for data – mostly quite small – and the (always much larger) flash for program memory.

RISC for the ATmegas means (besides being unable to divide) almost all machine instructions being the same size – 16 bit and very few 32 bit – and same speed – mostly one processor clock period and a few up to four. As all machine instructions size is 16 bit or one word respectively integer multiples, instruction addresses always point to words. Hence a 128KByte program memory is to be seen as a 64K instruction word memory. That can be handled by 16 bit program counters and jump / call / return addresses.

Remark: It might be noted that there are (cross-) C compilers around with bugs on handling Harvard subtleties.

In just that sense the flash address in the AVR109 protocol (“A” command) is a word address. Insofar the ATmega1284p is harmless and the original sin breaking the 16 bit address range (and the “A” and “H” commands) is committed by the ATmega2560.

ATmega2560 programming trouble – a word of caution:

And right that is the point where some programmer software, (ISP) programmers and also bootloader software start trouble. Some just can't handle all words above 128Kbyte in one or the other way. This often goes undetected a long while by users – and, alas, the developers, too – as most programs are just much smaller.
“Hello world” won't get anyone into that troubles. Those products break down when the programs grow – or when it comes to burning a boot loader.

	ATmega2560	ATmega1284p	ATmega328P	ATmega32L
properties for C programming	avr/iom2560.h (+ avr/iomxx0_1.h)	avr/iom1284p.h	avr/iom328p.h	avr/iom32.h
#defines for memory sizes and organisation	SPM_PAGESIZE 256 RAMEND 0x21FF XRAMEND 0xFFFF E2END 0xFFF E2PAGESIZE 8 FLASHEND 0x3FFFF	SPM_PAGESIZE 256 RAMEND 0x40FF XRAMSIZE 0 XRAMEND RAMEND E2END 0xFFF E2PAGESIZE 8 FLASHEND 0x1FFFF	SPM_PAGESIZE 128 RAMEND 0x08FF XRAMSIZE 0 XRAMEND RAMEND E2END 0x3FF E2PAGESIZE 4 FLASHEND 0x07FFF	SPM_PAGESIZE 128 RAMEND 0x085F XRAMSIZE - XRAMEND RAMEND E2END 0x3FF E2PAGESIZE 4 FLASHEND 0x07FFF
#defines for AVR type	SIGNATURE_0 0x1E SIGNATURE_1 0x98 SIGNATURE_2 0x01	SIGNATURE_0 0x1E SIGNATURE_1 0x97 SIGNATURE_2 0x05	SIGNATURE_0 0x1E SIGNATURE_1 0x95 SIGNATURE_2 0x0F	SIGNATURE_0 0x1E SIGNATURE_1 0x95 SIGNATURE_2 0x02
recomm. fuses	L: FF H: D0 E: FD	L: F7 H: D0 E: FC	L: FF H: D0 E: FD	L: E0 H: D0
RAM size flash size boot flash part EEPROM	8K 256Kbyte max. 9192 bytes 4K	16K 128Kbyte max. 9192 bytes 4K	8K 32Kbyte max. 4096 bytes *) 4K	2K 32Kbyte max. 4096 bytes *) 1K
CPU clock	max. 16 MHz	max. 20 MHz	max. 20 MHz	max. 16 MHz
Pins	100	40	28 (DIL) / 32	40 (DIL) / 44
I/O ports	86	32	23	32
usable as AI	max. 16	max. 8	max. 6 / 8	max. 8
SPI	1	1	1	1
UART	4 (one optional as SPI)	2 (one optional as SPI)	1	1
data sheet	doc2549.pdf [AVR3]	doc8059.pdf [AVR2]	doc8161.pdf [AVR5]	doc2503.pdf [AVR8]
used in e.g.	ArduinoMega2560	weAut_01	ArduinoUno, Nano	easyAVR (standard V.7)

Table 7: The bootloaders target controllers (used / tested as of September 2014) *) Max. bootsize is with both bootsize fuses programmed (00).

mySmartUSB's bug and workaround:

One of the worst examples here is the otherwise quite good mySmartUSB. That's a nice and fast ISP programmer. At least some versions intermix all the flash addresses 0x3Exyz with 0x1Exyz (here given in byte units). That slump occurs both when burning and when checking a hexfile ([intel1]). With (hence) no error reported the flash is miss-programmed. This happens e.g. when controlled by AVRdude via the AVRISP protocol.

Question / problem, hence: How (the hell) bring in a bootloader
when no other programmer is around.

N.b.: Most bootloaders – including the one we introduce here – can't modify or replace itself. That's no problem, usually, as bootloaders (should) change by orders of magnitude less often than the application program.

Listings 8 and 9 show the makeup of a hexfile (probably for bootloaders) starting at address 0x1E000 respectively 0x3E000. As the mySmartUSBs in question interchange those addresses, one just has to interchange those addresses on first and the last but one line of those exemplary hex-files.

```
:020000021000EC
:10E00000.....................<many lines>.....
:040000031000E00009
:00000001FF
```

Listing 8: Frame of a 0x1Exyz hexfile (probably a ATmega1280 bootloader)

```
:020000023000CC
:10E00000.....................<many lines>.....
:040000033000E000E9
:00000001FF
```

Listing 9: Frame of a 0x3Exyz hexfile (probably a ATmega2560 bootloader)

Note: The last line giving the start address in intel-segmentation (unused here) may be left as is.

Hint: It is strongly recommended to delete the modified .hex-file after burning it.
It's now as buggy as the buggy ISP programmer used so.

Using byte addresses for the flash memory – why and how:

Above was truly said for a “word organised” Harvard-RISC an “addressing just words” approach could and partly will be used. But that was not the whole truth.

Also in Harvard architectures it makes great sense to allow final variables, like constant strings, invariable arrays &c., to live in program memory. The counterpart in the 80x86 architecture would be final data put into a “readable code segment”. To simplify this

approach's handling with the ATmegas (i.e. to please the compilers) byte addressing for those variables is helpful to almost necessary – at least for reading those variables.

And, when at it already and for symmetry reasons, also using byte addresses for writing them makes sense too. They're read-only by definition, but there is self programming ([AVR4, AVR1]), that lacking we would not occupy ourself with bootloading here.

And so on. In the end the Atmel or AVR computer architects opted for flash byte addressing in almost all cases, hence bringing also the ATmega1280 out of the 16 bit paradise – and some more bootloader and programmer software around into trouble.

It might be noted here however:
This serial bootloader works quite well in that respect with with AVRdude (>=5.5, [tool2]) thanks to an extension to the AVR109 protocol (probably by Jörg Wunsch?). The protocols extensions beyond [AVR4] are not widely documented but well implemented in both AVRdude and this bootloader.

ATmega's accessing locations in "big" flash:

Due to opting consistently for flash byte addressing, starting with the Atmega1280, the (indirect) addressing via a (combined) 16 bit register isn't sufficient any more neither for reading final variables, for self programming nor for any program flow branches: jump, call, return

The "solution" for the 17 respectively 18 bits needed (or up to 22 bits in future) is to

i. bring the address bits 16 (up) to 23 in an I/O port [sic!] named
EIND for program addresses and
evaluated by the `EICALL` and `EIJUMP` ([AVR1]) instructions,

ii. doing the same with an I/O port named
RAMPZ for variable addresses
evaluated by the instructions `EPLM` and `SPM`. ,

iii. using 3 stack bytes for all call and return on all ATmegas with large enough flash.

Necessarily this is a burden to the Assembler programmers and code generator designers. Nevertheless the state of art in compiler technology can make this all transparent to higher level languages. Since decades for embedded controllers that language was and still is C.

2.3 Critique of AVR GCC C compiler

But, alas, even for smaller flash memories AVR-GCC ([tool3]) is
putting all burden to the human programmer:

This is a severe criticism on this C compiler as this failure breaks all expectations on a programming language just above the lowest assembler level. And it breaks the robustness (even) C has with respect to variable (and array) access.

In a high level language we use variables and arrays in this way:

```
uint8_t  x = y + 31;     //  expression with a variable assigned to a variable
uint16_t h = o[331];     //  array element to variable
uint32_t g = *(pb[i]);   //  dereferenced pointer from pointer array to variable
```

That's clearly readable and easy to express. (Well yes, in the third line the C language's abstrusities start to show through a bit). All choosing and handling of address registers, address arithmetic as well as the mechanics of memory access is the compiler's job.

That is why we use C with ATmegas.

With AVR-gcc it's quite easy to move a final (read-only) variable or array from RAM to program memory respectively flash by just setting an attribute called PROGMEM in the definition, done by the macro INFLASH in Listing 10. So read-only elements are most easily moved from RAM to flash or the other way round.

By changing just one spot, the whole GCC toolchain up to the linker knows this variables nature – final in code space – and where to put it.

But strangely (to put it mildly) by doing that a variable or an array access becomes totally different. When the designer, quite easily moves the array outPatt, for example, from RAM to flash one has to touch all (!) uses of that variable in all source files, as shows Listing 10.

```
// The declaration needed for making outPatt usable in the file in question.
// In C we wouldn't care where the array outPatt is defined in the end
extern uint8_t const outPatt[]; //!< pattern to be output in state [situation]

// standard definition (elsewhere; i.e. in any of project's many C-files)
uint8_t const outPatt[] = { 0x7D, 0x86, 0x6D, 0x43, 0x6D, 0xF6,};

// standard array usage respectively C syntax
uint8_t toBeOutput = outPatt[situation]; // depending on situation's value
we ..


// putting outPatt in flash memory by the PROGMEM attribute (in macro INFLASH)
// is dead easy. But this breaks outPatt's usage in all source files around
INFLASH( uint8_t const outPatt[] ) = { 0x7D, 0x86, 0x6D, 0x43, 0x6D, 0xF6,};

// spoiled usage when in flash cause the gcc compiler won't handle that:
uint8_t toBeOutput = pgm_read_byte(&(outPatt[situation])); // [sic!] and worse
```

Listing 10: An array moved from RAM to flash

The necessary manipulation from

```
uint8_t toBeOutput = outPatt[situation];
```

to

```
uint8_t toBeOutput = pgm_read_byte(&(outPatt[situation]));
```

shown in Listing 10 (and that is a simple case) breaks all C's expressiveness concerning variable access. Bringing in 'arrays of arrays' (best mixed dwelling in RAM and flash) or pointers … the thing gets totally out of hand, unreadable and even more error prone.

Additionally one has to choose **`pgm_read_byte`** **`pgm_read_word`** and so on according to the size respectively type of the data item to handle.

It's an outright misdetermination not to give this task to the avr-gcc C Compiler!

To make things even worse: The programmer has to make the right choice from a range of eight or more access macros according to the item's type and location. E.g. use

```
pgm_read_byte_far(((ADD_TYPE)&byteItem) | baseAddress)
```

instead of

```
pgm_read_byte(&byteItem)
```

when 16 bit byte addresses won't outreach. And then one has the extra burden to correct respectively complete the 24 bit address truncated before by avr-ggc's C pointer arithmetic (being RAM centric and 16 bit for all variables).

For that decision and correction the programmer has to know (beforehand) where the variable in question will dwell in flash – again clearly the AVG-gcc toolchain's area of work.

Hint on flash variable addresses:

At present flash variables (INFLASH() macro / PROGMEM attribute) are put at the beginning of the flash, bringing them very probably below 64 Kbyte in case of application variables. Bootloader's flash variables will, of course, get addresses 0x1pxyz respectively 0x3pxyz for an ATmega1284P or an ATmega2560; see the macro `FAR_ADD(var)` in the file `include/boot109.h`.

An “all is OK signal” (??) just for functions :

For functions and labels the linker repairs pointers by a so-called “trampoline” trick.

Obviously, handling variables in flash is a yieldingly source of troubles for any programmer and bootloader software as well as for the human programmer. And not putting final variables to flash makes neither sense nor is possible when RAM size gets narrow.

3. Bootloader usage

Burning the serial bootloader (once)

Of course, before being able utilize the serial bootloader as an embedded program or to use its procedures, functions and constants in the application it has to be put once in high flash memory. A non-existing serial bootloader can't burn itself nor can't this bootloader modify or replace itself.

Therefore we need once or very seldom special (ISP) hardware programmers, like (e.g. / best) AVRisp mkII. To control them we can use the AVRdude programmer software by:

```
avrdude -p <target> -c avrisp2 -P usb -v -t
                                  -U flash:w:targetSerBootl.hex
```

Utilising the serial bootloader

As said this bootloader uses Atmels standard AVR109 protocol and can be used with any programming software knowing it. For the well known AVRdude here are some examples.

Use AVRdude interactively:

```
avrdude -p atmega1284 -c avr109 -b 38400  -P com1        -v -t
avrdude -p atmega2560 -c avr109 -b 38400  -P com9        -v -t
avrdude -p atmega328p -c avr109 -b 38400  -P \\.\com10  -v -t
```

For any other operation replace the -t by the respective AVRdude command option.
To save the flash's content, for example, use something like:

```
avrdude -p atmega1284 -c avr109 -b 38400 -P com1  -v -U flash:r:save.hex:i
```

Having the bootloader software talk to the programming software (AVRdude e.g.)

It is a sheer triviality – that may as well be forgotten in the stress of development work: To talk to the serial bootloader via AVR109 protocol the target µController has to execute the bootloader program. Normally, of course, it will execute the embedded application. AVRdude's endeavour to converse with the application serially, will – at best – end soon with more or less cryptic error reports.

According to the state machine described in the next chapter one may get into the bootloader software
a) by reset for a short time interval
or by
b) a (commanded) action of a suitable application program.

In a laboratory environment – with the target machine physically accessible – one will

- just release the rest button and the enter key for the direct or indirect (sript, make) AVRdude program start in the same second.

Some Arduinos with USB2serial converters use a signal change of DTR as reset. This “feature” would be dangerous for any a “real embedded application and better be scratched out. See “Hint / Caution” on page 9.

But anyway, the AVRdude software can't use this feature in the context of the AVR109 protocol. To use it precede AVRdude's start with an extra (Windows) command:

```
mode comXP dtr=on
avrdude -p atmegaXYZT -c avr109 -b 38400 -P comXP -V …....
```

4. Bootloader operation

4.1 Entering and leaving

This bootloader shall (by fuse setting) always be entered on any µController reset or re-start. It will then do the basic initialisations for the respective platform and then decide to either

- enter the application program or
- stay in / enter the AVR109 bootloader protocol mode waiting for a programming software's (AVRdude's) sign of life, e.g. an ESC character, via the serial link.

There are multiple criteria for that decision. The most important ones are:

- application flash is cleared ([00] == 0xFFFF):
  stay in the bootloader program (waiting for AVR109 communication)
- entered by call / jump from application, i.e. by (CLI) command:
  remain in bootloader (at least for a quite long time-out (> one minute))
- watchdog reset:
  re-enter system / application software immediately
  (It is considered to add “brown out” to the “do not enter bootlaoder” reset causes.)
- reset while "enter key” (on weAut_01) or other platform specific input criterion:
  remain in bootloader
  (As said, it is considered to completely remove that “traditional” option.)
- receive the exit bootloader ('E') command:
  enter application software after short time-out (and if flash not empty)
  (The timed out wait for next AVR109 command sequence or prelude allows for sequenced AVRdude commands, e.g., in a batch or make script.)
- time-out while waiting for an AVR109 command or sync byte (ESC)
  enter system / application software.

Entering the bootloader by reset is only feasible with physical access to the board, a remote reset control or a (remote) control of the supply. A remote reset control is seldom implemented. And the Arduino's DTR/RTS remote reset has to be considered as sinister in real applications. Switching off the supply (load voltage) might have undesirable consequences for process I/O or for other devices.

Hence it is desirable that all system / application program (put in by this bootloader) shall have a (CLI) command to enter the bootloader, as has weAutSys (boot -load). If such command is considered critical it might be kept secret from the inadvertent user or be further (password) protected.

4.2 Protecting the flash and EEPROM content

Besides above state machine's reluctance to stay in bootloader on doubtful reset conditions, the bootloader expects AVRdude's standard prelude command before allowing erase or write commands to be executed. For human on a terminal program (testing the bootloader directly) or a programmer software the consequence is to use the AVR109 protocol commands p, V, v and s before doing something serious.

This safety feature against unintentional bootloader operation has a down. Any deviating behaviour (like an improbable change of AVRdude's starting prelude) would stop the intended bootloader operation. That's nothing special nor negative – it's the usual safety availability trade-off.

4.3 Handling COM port driver bugs

On all systems tested yet opening a COMxy serial port test the transmit two spurious of about 40µs width about 580µs apart – or a multiple sequence of that pair. This inflicts all programs using serial interfaces on that workstation, be it AVRdude, vsst.exe, HTerm or what else. So this is a system respectively (FTDI) driver bug – and not a new one, as a little Web search revealed.

At not just very low baud rates the other station will interpret those spurious spikes as start bit and 0 to 6 data bits (usually one at 38400N). Then (as reports show) much goes wrong.

The protection feature just described would normally be triggered by these spurious spikes ending the whole thing before a chance to begin. Hence in this (“prelude checking”) state this serial bootloader will recognise and filter those buggy transmissions.

5. Bootloader integration

5.1 Initialisation and services

This bootloader's software has to be adapted to and tested with the respective platform and µController type. This has already been done for three quite different specimens.

As said, the bootloader shall be entered at every reset. And of course its first task will be all or a substantial part of the platform's initialisation. This is done in a way that this service will **not** have to be repeated by the system/application software – otherwise clearly the system/application software's first task.

This might save a few bytes and bring back some processor time spent for the bootloader.

5.2 Using bootloader functions and variables in the application

Omitting initialisation tasks already done by the bootloader is one integration step. But the real saving in flash space (and may be programming, documenting and testing effort) comes when using the bootloader's final variables, functions (and indirectly linked libraries) in the system application program.

There are some approaches around to this lucrative end – usually involving multiply indirect jump tables, miss-using unused (parts of) interrupt tables and the like. Not diminishing some of those approaches' ingenuity, they tend to get outright complicated, unreadable and error prone. They, too, violate the maxim

> “Let the (C) compiler and the linker do the work – … at least as far as possible”.

And having the bootloader's .h files and linker outcomes, it's possible here to follow this good principle. Trouble is the avr-gcc documentation and literature hardly giving any advice for re-using bootloader elements in applications. The most suggesting way is to link the bootloaders .elf file (containing all symbols and addresses) by:

```
avr-gcc .... -Wl,--just-symbols=boot109.elf ....
```

This, unfortunately, would run without complaints and will allow the usage of bootloader's variables and functions by the application so linked to it. . . . But do **not** use this!

Without any warning this proceeding spells disaster, when symbol names crash. That can't be avoided as the .elf file contains also gcc generated symbols – for e.g. variable initialisation. The linker, thus offered alternatives, makes an uncontrollable (random) choice. That may lead to failures hardly traceable. We'll have to restrict us to user defined symbols mainly. Listing 11 shows one “howto” – that can / shall be buried in the makefile.

```
grep " [Tt] " boot109.sym | grep " _[_e][^m^u]" -v   \
                  | grep " main" -v     > boot109_ext.sym
Gawk '{print "--defsym " $3 "=0x" $1}' boot109_ext.sym    \
                                                  > boot109_ext.tab

avr-gcc   .....  -Wl,@boot109_ext.tab   ....
```

Listing 11: Filtering out the bootloader's user defined symbols and linking them elsewhere

The first filter in Listing 11 grabs the symbols from the T=“text” sections only. “text” here doesn't mean text but code [sic!]. The second filter removes all symbols starting with underline (_), that are those generated by the system or from secret libraries but not created by programmers with a certain sense of style. The second remove filter's exceptions are symbols starting with `__m` or `__u` which lets pass some library functions to implement basic arithmetic operators. If we have e.g. 32 bit divide in the bootloader the implementing functions must not be repeated in application flash. And finally the last filter simply removes the bootloader's start function main, which we never want to introduce to the application linker. Listing 12 shows a fraction of the (..._ext.tab) output.

```
--defsym setTheLed=0x0003e44e
--defsym uartDivide=0x0003e45a
--defsym initUART0=0x0003e498
--defsym sendSerByte=0x0003e51e
--defsym sendSerBytes_P=0x0003e52e
--defsym copyFirstSVNtokenP=0x0003e5a8
--defsym recvSerByte=0x0003e658
--defsym bootLoaderGreet=0x0003e67e
--defsym resetCauseText_P=0x0003e724
--defsym basicSystemInit=0x0003e772
--defsym toHMI8LEDchain=0x0003e7be
--defsym isFlashCleared=0x0003e7c2
--defsym appMain=0x0003e872
--defsym blockLoad=0x0003e8ba
--defsym __mulsi3=0x0003f2bc
--defsym __udivmodsi4=0x0003f2fa
--defsym __udivmodsi4_ep=0x0003f320
```

Listing 12: The bootloader's user defined symbols filtered and prepared (example, excerpt)

One has to use the full, actual and platform fitting symbol list, of course.

6. Resume

We provided a full serial featured bootloader implementing AVR109. That protocol is a standard by Atmel ([AVR4]) used in prevailing programming software, like the AVRdude tool ([tool2]).

This bootloader can read and “burn” both flash and EEPROM, read fuses, signatures and else. But (due to architectural restrictions) it can't write fuses or modify / replace itself. In those use cases we still need ISP (or JTAG) programmers.

Adapting this bootloader to a board like weAut_01 (by weinert – automation [We1], [We2]) or to Arduino boards makes special programming hardware dispensable.

Using a platform's standard communication link, remotely accessible anyway when utilised in the application, no direct or extra physical access is needed to “flash” a new program.

Additionally a standard protocol bootloader opens the Arduino boards to normal professional development tool chains and libraries, thus converting them from a “stage for sketches” to a cheap and versatile ATmega evaluation board for standard devopment.

The large and different flash sizes of the processors involved, see table 7 on page 12, indudes special problems to the pair of programming software and programmer or bootloader. The bootloader provided and reported on here solves these problems well.

Another design goal implemented was the bootloader's integration with application program. This concerns the transitions between bootloader and application program as well as usability of bootloader functions and final (flash) variables provided by this bootloader by the application software.

Please read also this bootloader's (generated) software documentation
in weAutSys' (online) documentation (→ modules → bootloader).

The bootloder's download URL is
weinert-automation.de/files/openSource/opSour_weAutSys.zip

A Abbreviations

38400 a standard UART baud rate

8N1 a standard UART setting, 8 bit, no parity, one stop bit

ADC analogue digital converter

AI analogue process input (from sensors)

API application programmer's interface

C C programming language

CISC complex instruction set computer

CLI command line interface

DI digital process input (from sensors)

DO digital process output (to actuators)

DRY “don't repeat yourself!” as approach

DTR data terminal ready – an extra UART (modem) signal

FTDI Name of a chip company specialised in USB2other solutions – synonymous for respective drivers, protocols and chips (like e.g. ATmega 16U2)

HMI Human machine interface

I/O Input / Output

ISP In system programming

JTAG Joint Test Action Group (or serial test access port)

LED light emitting diode

LV Load voltage (German Lastspannung), supply for process I/0 (24V in industrial automation systems , PLCs and heavy vehicles, 12V in facility / building equipment and small vehicles)

PLC programmable logic controller (small automation system)

POI power over Ethernet

RAM random access memory (readable and writeable)

RISC reduced instruction set computer

RTS request to send – an extra UART (modem) signal

SMD surface mounted device

SPI serial peripheral interface

sic! just so as written / cited! Believe it however incomprehensible.

USB	universal serial bus
USB2serial	replacing (legacy) V.24 / RS232 ports by an USB link
µC	µControllor, micro-controller
µSD	µSDcard, small (secure data) memory card

L References

[AVR1] Atmel, (doc0856.pdf)
8-bit AVR Instruction set

[AVR2] Atmel, (doc8059.pdf; preliminary)
8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash ATmega1284P

[AVR3] Atmel, (doc2549.pdf)
8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash
ATmega640/V ATmega1280/V Atmega1281/V ATmega2560/V Atmega2561/V

[AVR4] Atmel, (doc1644.pdf)
AVR109: Self-programming

[AVR5] Atmel, (doc2568.pdf)
AVR911: AVR Open Source Programmer

[AVR6] Atmel, (doc8171.pdf)
8-bit Atmel Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash
Atmega48PA Atmega88PA Atmega168PA ATmega328P

[AVR7] Atmel, (doc7799.pdf)
8-bit Atmel Microcontroller with with 8/16/32K Bytes of ISP Flash
ATmega8U2 Atmega16U2 ATmega32U2

[AVR8] Atmel, (doc2503pdf)
8-bit Atmel Microcontroller with with 32K Bytes of ISP Flash and USB
ATmega32 ATmega32L

[intel1] Intel, (HexFmt.pdf)
Hexadecimal Object File, Format Specification, Revision A, 1/6/88

[tool1] Richard M. Stallman, Roland McGrath, Paul D. Smith
GNU Make, A Program for Directing Recompilation, GNU make Version 3.82, July 2010

[tool2] Brian S. Dean, Jörg Wunsch
AVRDUDE, A program for download/uploading AVR microcontroller flash and eeprom
For AVRDUDE, Version 5.5, 29, October 2007

[tool3] GCC team
avr-libc 1.8.0 January 3 2012

[We1] Rolf Biesenbach, Albrecht Weinert
An economical approach for small sized automation tasks
April 2013, 9th International Symposium on Mechatronics and its Applications (ISMA13)

[We2] Albrecht Weinert
weAut_01 automation controller
user manual (German), Nov. 2011

[We3] Albrecht Weinert, weAutSys software documentation
generated by Doxygen, May 2013 or later, as .html and as .pdf