AVR Bootloader FAQ

Written by Brad Schick <schickb@gmail.com> with contributions and editing by Cliff Lawson.
Last updated: May 19, 2009

Planning to create a custom bootloader? This document answers many of the questions you may have. Most of the concepts covered here are not discussed thoroughly in datasheets, and various tips for designing a robust bootloader are provided. While much of this information is available on avrfreaks.net, it is spread around in many different threads. The questions below start simple and get progressively more complex. To create a viable bootloader, you will probably want to understand at least the first 11 answers.

There is a lot to cover, so I will assume that you already know what a bootloader does, can develop in C, and are familiar with creating normal 8-bit AVR applications. This document is based on the avr-gcc tool chain using the avr-libc library and a typical makefile. If you are using different tools, the concepts in this document still apply but the code and makefile examples will need to be modified.

Also please note that the addresses shown in the examples are based on the AT90USB162, and **must** be adjusted to your particular AVR before reusing them.

Table of Contents

1. How do I get started?

Fundamentally, a bootloader is just a normal AVR application that is positioned in a special region of flash memory. In the simplest form, a bootloader is an AVR application with one extra linker flag added to its makefile:

```
LDFLAGS += -Wl,--section-start=.text=0x3000
```

That flag specifies the starting byte address of the bootloader. You can find this value in the datasheet for your AVR. Make sure you use the byte address and not the word address (most Atmel datasheets list word addresses for flash regions, but some list byte addresses). Rather than hard-coding the bootloader's starting address, I'd suggest adding a constant to your makefile.

```
BOOTSTART = 0x3000
LDFLAGS += -Wl,--section-start=.text=$(BOOTSTART)
```

2. What are the NRWW and RWW regions?

This is covered in the datasheets, but the names can be confusing. They refer to different regions within the MCU's flash memory, and indicate what happens to the CPU while that region is being erased or written. The bootloader always resides in the no-read-while-write section (NRWW). Very often the entire NRWW area is reserved for the bootloader, but it doesn't need to be. When the NRWW region of flash is being erased or written, the CPU is halted because "no-reading" is allowed. The application is usually stored in the read-while-write (RWW) section. When this region of flash is being erased or written, the CPU can continue running as long as the code being read is in the NRWW section (aka the bootloader). While the RWW is being reprogrammed any attempt to read from it will return 0xFF's. Before the RWW section can be read again after programming, it must be re-enabled (more on that in question #3).

What you really need to know are the effects of the RWW and NRWW regions:

- The bootloader can reprogram the application (RWW) region while the bootloader continues to run
- The bootloader can not easily update its own code
- Applications very rarely update the bootloader
- Applications can update themselves, but it is usually better to let the bootloader do it

3. How does the bootloader update the application?

Getting application firmware from some source into the bootloader is up to you. Common communication channels are UART and USB. The protocol used over that channel can be something you make up or a standard protocol like AVR109 or DFU. If you implement a standard protocol you can use an existing tool like AVR Studio with AVR109 or Atmel's Flip with DFU. But these standard protocols tend to be a bit bloated, and if you implement a simple custom protocol your bootloader will often be smaller and cleaner. But you'll also need to devise your own solution for sending data to the bootloader.

It is common to send one flash page at a time to a bootloader. The bootloader needs to write these pages to the application's flash region (RWW). Avr-libc provides a header file called <avr/boot.h> that has everything you need to do this. The functions are pretty well documented and an example

is provided. A few tips:

- Use the handy *_safe* macros so that you don't forget to wait until the MCU is ready
- Make sure to erase each page before you write it
- *boot_page_fill* takes a byte address but writes a word at a time. In a loop you must increment an address by 2 bytes
- After programming, remember to re-enable the RWW section with the *boot_rww_enable* macro. Do this before reading or running the application or the RWW section will read as 0xFF's

4. Can a bootloader update fuses?

This is not possible with AVRs. You need an external programmer to do this.

5. What is BOOTLOADER_SECTION in <avr/boot.h> for?

Despite its name, the BOOTLOADER_SECTION macro is not useful for writing a bootloader. This macro just helps you re-position application functions into the NRWW flash section. The macro would probably have been better off named something like NRWW_SECTION.

The only use for the BOOTLOADER_SECTION macro is to add self programming functions to the application itself. The macro is needed because flash programming (the SPM opcode) can only be executed from the NRWW section. BOOT_LOADER_SECTION really just creates a special code section called ".bootloader" (another poor name choice), and then an additional linker flag positions that section somewhere in unused NRWW flash memory. This isn't a really common need.

Some people are tempted to use BOOTLOADER_SECTION to write an application and a bootloader as a single program, but this is not a good idea. A bootloader is best written as a stand-alone program that has no dependencies on the application. It will be more reliable and actually far easier to build your bootloader if it remains separate from the application.

6. How do I program the bootloader to the MCU?

Since the bootloader can't easily update itself and applications almost never modify the bootloader, you will need an external programmer to write the bootloader to the MCU. Example programmers include the STK500, AVRISP, AVR Dragon, JtagICE MKII, etc. The programming mode you'll use depends on your AVR. You don't need to do anything special to program a bootloader. If you have a working external programmer that can write the application and update fuses, it will work for the bootloader. The programer just reads the bootloader firmware file and writes it to the addresses specified within. The programmer doesn't notice or care that you happen to be writing to the NRWW flash region.

7. How do I program both the application and the bootloader?

Since you are hopefully building the application and bootloader separately, you will end up with two separate firmware hex files (for example, app.hex and boot.hex). The question then arises how to get them both onto the AVR. A few options:

Two step approach: Program your bootloader with an external programmer as described in question #6. You will probably also want to set fuses, including BOOTSZ and BOOTRST, at the same time. Then simply use the bootloader's normal communication mechanism to transfer and

program the application.

One step approach: Combine the app.hex and boot.hex files into a single file and use an external programmer to write the combined file to the MCU in one shot. You will probably also want to set fuses, including BOOTSZ and BOOTRST, at the same time.

I think the easiest way to combine hex files is with the *srec_cat* command-line tool. This tools is part of the *srecord* utilities. It comes with WinAVR and is very easy to install on a Unix like OS. This is the command you would use:

```
srec_cat app.hex -I boot.hex -I -o combined.hex -I
```

Cliff explains that you can also manually combined the app.hex and boot.hex files with a small edit: *“Every hex file has one last record in it which effectively says “the file ends here”. So after doing a [manual merge] one needs to edit the combined hex file and locate the termination record from the end of app.hex (now somewhere in the middle of the file) and remove it. The record has a type 01 – the type byte of an Intel hex record is the 4th hex byte so the record actually looks like “:00000001FF”. The whole line (the one in the middle, not the one at the end of the file) should be removed.”*

8. Can a bootloader use interrupts?

Yes, you just need to tell the CPU to jump to the interrupt vectors positioned within the bootloader rather than the vectors within the application. Somewhere early in the code for your bootloader (before using interrupts), add lines like this:

```
MCUCR = (1<<IVCE);
MCUCR = (1<<IVSEL);
```

Then when an interrupt occurs the program counter (PC) will be automatically set to the appropriate position within the bootloader's interrupt table. To work, the code sequence above must be compiled with optimizations enabled and a manual check should be made to ensure the resulting assembly writes occur within 4 machine cycles.

9. Should the bootloader or the application run first?

It is almost always better to set the BOOTRST fuse to run the bootloader after reset. When the bootloader runs after reset, you can easily reprogram the application even when the application is totally broken or absent.

Bootloaders work best as small, reliable, and infrequently changed programs. From the day you “ship” your device, you really never want to change the bootloader again. The application code, however, usually needs more freedom to change. Accordingly, the application might acquire some nasty bug causing it to fail. Or you may lose power while reprogramming a device and end up with a half written application. With a reliable bootloader running first, you can recover from these problems.

If the application is run on reset instead, you may be force to use an external programmer to recover from these problems. And if your device has no ISP header, you might have to throw it

away or start soldering.

This is how Cliff puts it: *"I simply do not see the point of a bootloader without BOOTRST being used. My opinion is that a bootloader is a piece of code that you write once and is the ONLY piece of code in the entire system that must be proven faultless on day one (especially because it's difficult/impossible to update the bootloader in an AVR). You program it into the AVR once using ISP at which time you also program the fuses for clocks and so on. At the same time you also program the protection bits so that nothing can ever change the bootloader and you program BOOTRST. You now have an AVR that always starts into the immutable bootloader and it doesn't matter what happens in the app section as you have something safely stored in NRWW that can always dig you out of a hole."*

10. How does the bootloader know when to start the application?

There are many possibilities. If your device has a button or some other input mechanism, you can just signal your intent with a press. The bootloader would normally run the application immediately, and only remain resident itself when a button is depressed during reset. This is how the STK500 bootloader chip works.

Another common solution is for the bootloader to check an external communication channel for a particular symbol during startup. Again, the bootloader would normally run the application immediately, and only remain resident itself after checking an external communication channel and receiving a certain response or finding waiting data. The Atmel Butterfly is an example of this approach.

In our own case, neither of these solutions were ideal. Our device has no buttons and we wanted it to start the application very quickly under normal conditions. The only communication channel with the outside is USB, and unfortunately a USB device takes a while (in CPU terms) to be enumerated and begin communicating with the host.

So this is what we did: Assuming your AVR has EEPROM, you can store a byte value that indicates when the bootloader should continue running and when it should start the application. There are two parts to this approach. First somewhere early in your bootloader's code (like main) add lines like:

```
// Bootloader code
const uint8_t app_run = eeprom_read_byte(ADDR_APP_RUN);

if(app_run == APP_RUN)
{
    // In case the app is faulty, clear the eeprom byte so that
    // the BL will run next time. A properly running app should
    // set this back to APP_RUN.
    eeprom_write_byte(ADDR_APP_RUN, 0xFF);
    run_application();
}

// Only run the bootloader once, then go back to the app
// (comment out the next line during app development)
eeprom_write_byte(ADDR_APP_RUN, APP_RUN);
```

Then somewhere “late” in your application process, when you know fairly well that it is working properly, add this:

```
// Application code
eeprom_write_byte(ADDR_APP_RUN, APP_RUN);
```

This works as follows: If the application previously ran successfully the bootloader will read the APP_RUN byte and prepare to jump to the application. Just before the application is started, however, the APP_RUN byte is cleared. The application then re-writes the APP_RUN byte when it seems likely that it is functioning properly, and the sequence repeats. If the application fails before resetting the APP_RUN byte, however, the bootloader will stay resident upon the next reset. Unless you comment out that last line in the bootloader code above, the bootloader will only stay resident once and then retry the application on the following reset.

I'd also suggest providing some way to manually force the bootloader to continue running from within the application. You just need to clear the APP_RUN byte and prevent it from being rewritten later. We did this in our application with a USB command sent from a PC program.

One downside to this approach is that EEPROM on most Atmel controllers has a 100,000 erase/write endurance limit. Since this technique does two erase/write cycles for each startup, we get 50,000 restarts. For our application I figured an average of 10 restarts a day resulting in a lifespan of 13.7 years. This could easily be extended further with some EEPROM wear-spreading.

11. How does the bootloader start the application?

There is really only one choice, you must position the program counter (PC) to the start of the application (which is usually the reset vector). Assuming you have a normal avr-libc based application, the beginning of your application will be c-runtime code that initializes the stack and global variables and then continues on to your "main" function. You can position the program counter with an inline assembly jump or a function call to address 0. I think the jump is cleaner:

```
asm("jmp 0000");
```

But there is more to it than that. When you directly run the application like this, the MCU is not reset to a pristine state as it is after a reset. Register values and peripherals are not cleared and disabled. There are two common ways to handle this.

The first approach is to intentionally trigger a reset using the watchdog timer. This will reset the MCU to a pristine state and relaunch the bootloader (assuming you have the BOOTRST fuse set). If the condition that caused the bootloader to stay resident has gone away (i.e. you aren't pushing the button any longer), the bootloader should immediately run the application. It should start the application as soon as possible, before many of the MCU's resources are modified. Here is an example:

```
Disable_interrupt();
Wdt_change_16ms();
while(1);
```

Then somewhere near the beginning of the bootloader do the jump to address 0.

A second approach is for the bootloader to perform whatever cleanup is necessary and then jump directly to the application without a reset. The stock Atmel USB bootloader works this way. With a USB bootloader, it makes sense to shutdown USB and put interrupt calls back into application space before jumping:

```
Disable_interrupt();

// Shutdown USB cleanly
Usb_detach();
Usb_disable();
Stop_pll();

// Put interrupts back in app land
MCUCR = (1<<IVCE);
MCUCR = 0;

// Run the application
asm("jmp 0000");
```

The application should then explicitly configure the resources it uses. Personally I think it is good practice to write robust applications that fully initialize the resources they need anyway. Then the application doesn't depend on the bootloader's behavior and both can change independently.

12. How can the bootloader be sure the application is intact?

After reset, the bootloader will normally jump to the application and let it execute. But how can the bootloader be sure there is a valid non-corrupt application to start? What if a previous application update was interrupted when only half finished? There are many possible precautions you can take. Here are a few:

A) Whenever possible, it is a good idea to verify that an application update succeeded right after programming. This can be done by reading the application flash back out and comparing the result with the firmware file. If you don't want the bootloader to allow application downloads, you can send the entire firmware to the bootloader twice. Once to program it and a second time to compare it with what was just written. Of course, neither approach helps if there are deterministic errors in your communication channel or bootloader. Most standard programming protocols support a verification step. This approach does not detect random corruption.

B) Before the bootloader starts programming, have it erase the first application page. This will set the flash page to all 0xFF. Then program application pages in reverse order from the end towards the beginning. On startup, the bootloader can then check for a valid instruction (usually JMP or RJMP) at location 0 to determine if the entire application was programmed successfully. This approach likely requires a custom programming tool since most standard tools don't give you an option to program in reverse. This approach does not detect random corruption.

C) A variant of B is to program the entire application normally but reserve the last few bytes of application flash for a marker that holds some recognisable pattern like 0xBEEF. You can do this by modifying the firmware hex file with a tool like *srec_cat*. When the bootloader starts, it reads the marker bytes and if it finds the correct value it knows the complete application was written. This

will work with standard programming tools, assuming they program from the beginning of the firmware towards the end (which most probably do, but isn't guaranteed). This approach does not detect random corruption.

D) A much more thorough option is to use a checksum or CRC to validate the entire application each time the bootloader starts. The is similar to option C except rather than a predetermined marker, a computed CRC value is stored in the last few bytes of the application firmware. Again, you can do this with the *srec_cat* tool. Remember to account for unused space in the RWW section with padding or something. When the bootloader starts, it computes a CRC by reading the application and using the same CRC algorithm as *srec_cat*. This value is compared to the stored value and if they match the bootloader knows the application is valid. This will work with standard programming tools, and it does detect random corruption in the application. The downside is a longer delay before application startup.

E) If you are very concerned about corruption, you can also have the bootloader perform option D on itself with its own CRC value. Of course there isn't much a single bootloader can do if it detects corruption within itself beyond trying to signal a failure and shutting down. Time to get out the ISP programmer.

F) One nice side-effect of the EEPROM restart option discussed in question #10 is that it provides a measure of corruption detection automatically. If the application was half written or badly corrupted, it won't get a chance to set the APP_RUN byte and the bootloader will stay resident after the next reset. Instead of searching for a bad application, this approach just lets application failures signify the problem. In some ways this is even better than a CRC check because a valid CRC can be applied to an already corrupt firmware image long before programming. Of course the EEPROM approach is not perfect since the application might experience only minor corruption and fail after it writes APP_RUN.

13. Can the bootloader use code built into the application?

Cliff says: *“No, no and 100% most definitely no.”*

I agree. Although it is technically possible, just say no. Your bootloader will be much more robust if it has zero dependencies on the application. The main purpose of a bootloader after-all is to erase and reprogram the application. You don't want to be arranging calls into the application around erasure of that same code.

I would not even store bootloader specific code that isn't part of the application in the RWW flash section. There is no full-proof way to protect it from accidental erasure and reprogramming. One full RWW erasure and your bootloader is potentially useless.

14. Can the application use code built into the bootloader?

Yes this is fairly easy, particularly if the code to be shared does not access global variables. Just don't try to achieve code sharing by building your bootloader and application as a single binary. The best approach is to build them separately and use function pointers to share code.

One very simple approach is to build the bootloader normally and then lookup the addresses of the functions you want to share in a map or disassembly file. You can then create hard-coded function pointers in the application using those addresses. This will work, but your two binaries will be tightly bound together. Every time the bootloader changes, those functions might move and you will need to update the addresses and rebuild your application(s). This is rather fragile, and also a

hassle while developing the bootloader. But if your bootloader is finished and you are sure that it will not change, this is a quick (and somewhat dirty) solution.

There is a better approach, however, that eliminates the problem of shared functions moving within the bootloader. If you think about it, the issue of functions changing locations is the same for interrupt handlers. How does the CPU know the addresses of your handlers? It doesn't. It only knows the address of a jump table that is added to your binary when linked. The CPU only requires this jump table (aka vector table) to be at a known location. You can use this technique to expose shared functions from a bootloader.

There are several ways to define a jump table. I think it is convenient to write a small assembly file:

```
.section .jumps,"ax",@progbits

// The gnu assembler will replace JMP with RJMP when possible
.global _jumptable
_jumptable:
    jmp shared_func1
    jmp shared_func2
    jmp shared_func3
```

Your jump table should use the names of the shared functions, and not their addresses. This is important because when you rebuild the bootloader you want the linker to automatically put the correct function offsets into the jump table. Then you just need to keep the jump table itself at a known location and the rest of the bootloader may change freely. To get the jump table into your bootloader, add the .S file to your makefile's ASRC line. Then add a linker flag to position it at a predetermined address that will not change. Often near the end of the bootloader:

```
JUMPSTART = 0x3FE0 # 32 bytes from the end of the AT90USB162 4kb boot section
LDFLAGS += -Wl,--section-start=.jumps=$(JUMPSTART)
```

You might need another flag to prevent the linker from throwing away your jump table since it will appear to be unused. This happens when you use the compiler flag *-ffunction-sections* along with the linker flags *--gc-sections* and *--relax*. If you are not sure, it doesn't hurt to add this anyway:

```
 LDFLAGS += -Wl,--undefined=_jumptable
```

The number of jumps that will fit in your table depends on both the amount of space you reserve for it and the size of the jumps. The size of the jumps depends on how far the jumps are. In bootloaders that are 8kb or less the jumps will be RJMP and only use 2 bytes each. The easiest way to know is to look at the bootloader's disassembly.

For another jump table technique that reuses the interrupt table at the beginning of a bootloader, see question #18. Also, some people suggest adding another layer of indirection to allow even the jump table to move around but I have not found this necessary.

After creating a jump table, the next step is to define function pointers that simplify calling the

shared functions through the jump table. You can do this with macros or inline functions. I prefer inline functions for the added type safety. To create the function pointers, open the disassembly for your bootlaoder and search for "_jumptable". Write down the byte address of each jump in the table. Once you have the addresses, create a header file something like this:

```
typedef void (*PF_VOID)(void);
typedef void (*PF_WHATEVER)(uint8_t);

static __inline__ void call_func1(void)
            { ((PF_VOID) (0x3FE0/2))(); }

static __inline__ void call_func2(void)
            { ((PF_VOID) (0x3FE2/2))(); }

static __inline__ void call_func3(uint8_t arg)
            { ((PF_WHATEVER) (0x3FE2/2))(arg); }
```

The hex numbers are the byte addresses you obtained from the disassembly file. They need to be divided by 2 to create word addresses since GCC does not do this automatically in this situation (this may be a GCC bug). Include this header in your application and call away...

```
call_func3(1);
```

15. Why can't global variables be accessed from shared functions?

Since you will have two totally separate binaries, each will have its own memory layout. The positions of global variables in the application are totally unrelated to the positions of globals in the bootloader.

Let's say a shared bootloader function does mistakenly read or write a global variable directly. When the bootloader was linked, the position of that global was essentially hard-coded into its executable code. When the bootloader itself is running, that position is correct and everything works. But when the application is running and calls that same function, the hard-coded position is wrong. It is wrong because when the application is running, its memory layout is present not the bootloader's layout. So you can not access globals directly from shared functions without potentially trashing data in the application.

Shared functions need to be told the address of non-local data at runtime. This is easier than it sounds; just have callers pass pointers to global data as an argument to the function. When the bootloader is running, it must pass the address of a global that was defined within the bootloader. And when the application is running, it must pass the address of a global defined within the application. If a number of variables are required, it is handy to group them together in a struct and pass a pointer to the struct. Something like:

```
// In a shared header file
typedef struct {
    uint8_t val1;
    uint16_t val2;
} globals_t;

// The shared function
void func4(globals_t *vars) {
    vars->val1 = 0;
    vars->val2 = 512;
}

// Globally defined in each binary
globals_t g_vars;

// Calling the shared function
call_func4(&g_vars);
```

If for some reason you can not pass a parameter to a shared function, see question #17 below.

16. Can the application use an IRS built into the bootloader?

This is actually less work than sharing a normal function because the default interrupt vectors already provide a jump to locate the ISR. Just code and build the bootloader's ISR normally (but again, without accessing globals directly). Then look at a map or disassembly file and find the interrupt table. Write down the address of the vector for the ISR(s) you want to share. You don't want the address of your handler (which the vector jumps to), you need the address of the vector itself. The address will be near the starting address of the bootloader.

Then add an ISR to your application that does nothing but jump to the bootloader's interrupt vector. For example:

```
// This must be declared "naked" because we want to let the
// bootloader function handle all of the register push/pops
// and do the RETI to end the handler.
void USB_GEN_vect(void) __attribute__((naked));

ISR(USB_GEN_vect)
{
    asm("jmp 0x302C");
}
```

This will not work without the "naked" attribute in the declaration. Without this, the application's ISR will push values onto the stack that will never get pop'ed and bury the return address. This would happen because you are not calling the bootloader's ISR, but jumping to it directly. The return (RETI) at the end of the bootloader's ISR will not return to the application's ISR but rather back to the code that was running when the ISR was invoked. Also note that with inline ASM like this, GCC automatically converts from a byte address to a word address so you don't need to divide by 2.

If you don't want to think about as many details, you can remove the entire declaration with "naked" and use a macro or inline function to *call* the shared handler through a function pointer. The function pointer would look just like those in question #14. But you will pay for this simplicity with a bunch of useless register pushes and pops entering and leaving the application's ISR.

17. How can a shared ISR access global data?

This is the same situation as question #15, but the solution is not as easy since you can't pass a pointer on the stack to an ISR. There are a number of possible solutions, but since this tutorial is already rather long I am not going to cover them in detail. Two options include using a GPIO register to hold a pointer to global data or reserving a section of SRAM at a known address to hold global data. If you are planning to do either, I suggest reading the following thread that discusses various options: http://tinyurl.com/q3fpud. For our own bootloader and applications, we used the reserved SRAM idea and it is working well.

18. Can I save space in a bootloader with few or no ISRs?

Yes, you can often trim 100+ bytes from the bootloader. This idea applies equally to normal applications, but the savings are often more significant for a small bootloader. Some AVRs have 40 or more interrupts, each taking 4 bytes in the interrupt vector table. Not only can you often slim that down, but you can also re-purpose unused vectors as shared function jumps (discussed in question #14).

The first entry in an interrupt table is the reset vector. Often bootloaders and applications need the reset vector because their executable code does not start right at the beginning of the firmware. Even if the reset vector is not currently required, it may be needed in the future after someone adds a new PROGMEM variable. So rather than totally removing the interrupt table from your bootloader, it is often better to just replace it with a smaller one. A minimal interrupt table can be written like this:

```
.section .blvects,"ax",@progbits

.global __vector_default
__vector_default:
    jmp  __init
```

This table only contains a reset vector that jumps to the c-runtime's __init function (this is what the default reset vector does). Assuming that JMP becomes a RJMP, you now have a 2 byte vector table. Add this .S file to your makefile's ASRC line.

The cleanest way to replace the default vector table is with a custom linker script. First figure out which linker script is being used for your AVR. In theory, the linker script should be named after an AVR's architecture. But I've found that to be untrue sometimes, so I located the linker script by renaming the directory that holds them (often C:\WinAVR\avr\lib\ldscripts on Windows and /usr/local/avr/lib/ldscripts on a Unix like OS) and then rebuilding. The linker will complain that it "cannot open linker script file ldscripts/avr3.x". That tells you the script file. Restore the original directory name and copy this file to your project directory. Then add the following line to the makefile's linker flags:

```
LDFLAGS += -T bootloader.x   # Or whatever you named the linker script
```

Now you need to modify the copied linker script to use your smaller interrupt table and dump the default. Open the linker script and search for "vectors". You should find existing text like:

```
  .text   :
  {
    *(.vectors)
    KEEP(*(.vectors))
```

Add a DISCARD line and change "vectors" to your section's name:

```
   /DISCARD/ : { *(.vectors); } /* Discard standard vectors */
  .text   :
  {
    *(.blvects)   /* Position and keep custom vectors */
    KEEP(*(.blvects))
```

If you have no ISRs, that is all you need to do. Another approach that I am not going to describe is to use the *-nostartfiles* and *-nodefaultlibs* linker flags with a custom initialization routine. This discards the default interrupt table along with the c-runtime startup code. For more details see these threads:

http://tinyurl.com/pkkwzt, http://tinyurl.com/qrsjjp, http://tinyurl.com/ptshkp

If your bootloader does use ISRs, you can still save space by replacing the full interrupt table with a smaller one. In our case, we only needed interrupt 11 (general USB on the AT90USB162) so we truncated the table and reused the slots before interrupt 11 as shared function jumps. When doing this, make sure you position the remaining interrupt vectors correctly. Each must be at the address listed for that interrupt in the datasheet (thus the 'nops' in the example below). By re-purposing a portion of the vector table, you eliminate the need for a separate jump table like the one shown in question #14. You just need the function pointers to call the re-purposed vectors. See the comments below for more details:

```
.section .bootvect,"ax",@progbits

; Custom vector table that eliminates wasted space after the last used
; vector (__vector_11, usb general). Also re-purpose the unused space
; between the reset vector and the usb vector for the jumps to shared
; code.
;
.global __vector_default
```

```
; There are 21 "word" spaces between __init and __vector_11. This fits
; 21 RJMPs or 10 JMPs. Since the bootloader is only "2K words" long,
; use RJMPs.
; - Don't change the order of these (unless it is before any devices
;   shipped)!
; - Add new entries by replacing nop's
; - Remove entries by replace them with nop's (without reordering)
__vector_default:
    rjmp  __init          ; 0x3000  !used interrupt!
    rjmp shared_func1     ; 0x3002
    rjmp shared_func2     ; 0x3004
    rjmp shared_func3     ; 0x3006
    rjmp shared_func4     ; 0x3008
    rjmp shared_func5     ; 0x300a
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    rjmp __vector_11      ; 0x302C  !used interrupt!
```

That's all for now. Feel free to send suggestions or additions. Thanks to all the freaks who have written helpful bootloader posts.