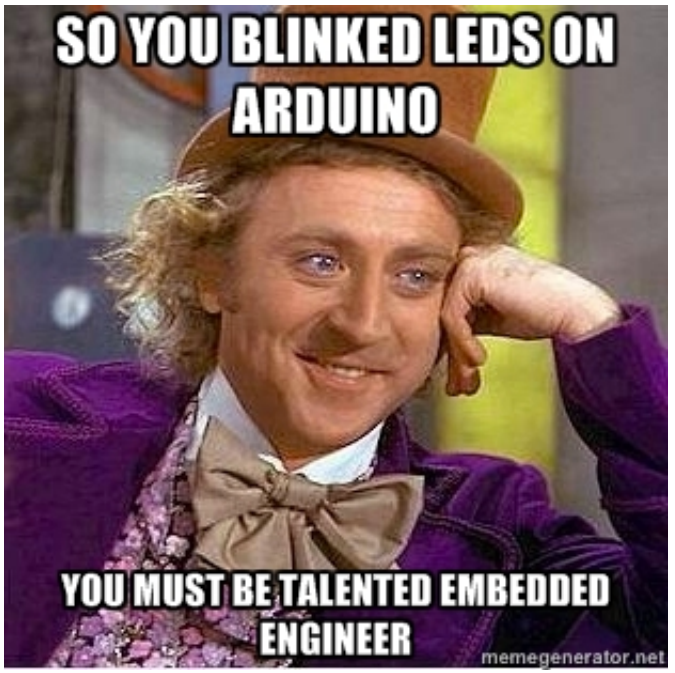
ESE350: Introduction to Embedded Systems

Lecture 4: Programming an embedded system from scratch



Rahul Mangharam
Electrical and Systems Engineering
Computer and Information Science
University of Pennsylvania

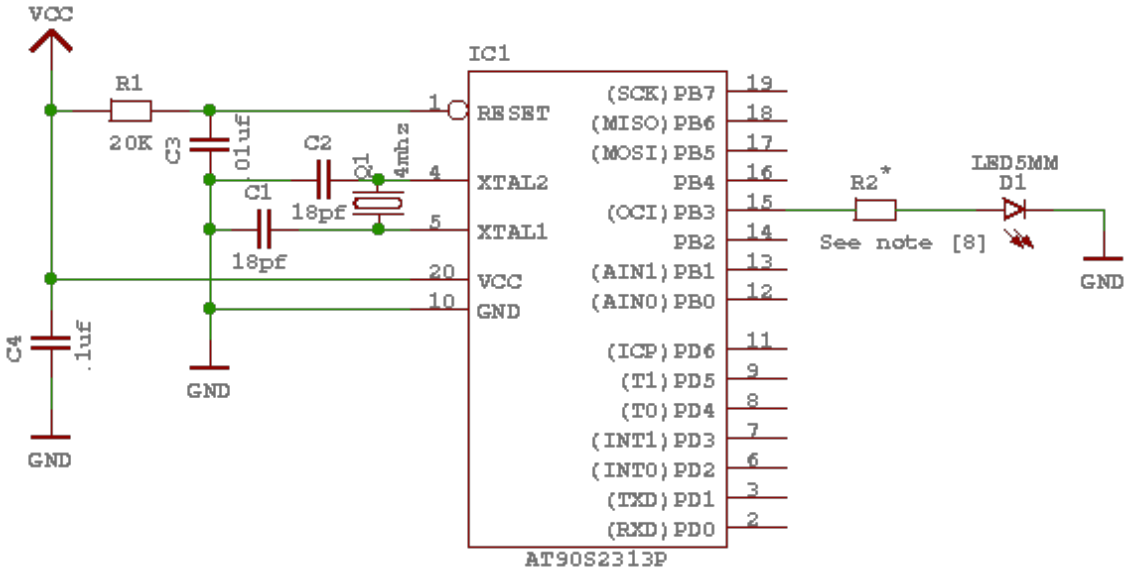
Many thanks to Jack Harkins and Eric Micallef
Guide based in part on http://www.atmel.com/webdoc/avrlibcreferencemanual/

Lecture outline

1. What happens when you compile your code?
2. How is an embedded system programmed for the first time?
3. Bootloaders for serial programming
4. STK500 Bootloader protocol to upload a new image
5. Walkthrough the `optiboot` bootloader code
6. Firmware over the air

1. What happens when you compile your code?

Let's start with a blink led demo example…



Schematic of circuit for demo project

So you built this circuit and wrote the code to blink the LED with a timer interrupt….

Let's start with a blink led example...

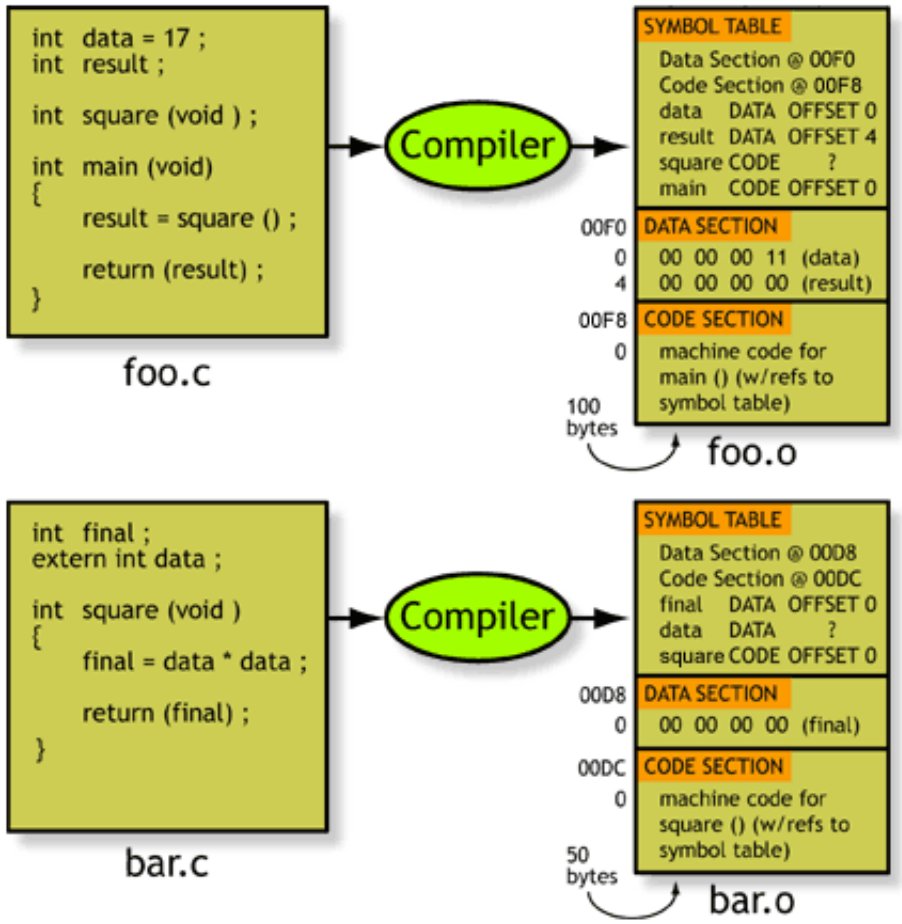
```
#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>

#include "iocompat.h"                   /* Note [1] */



enum { UP, DOWN };

ISR (TIMER1_OVF_vect)                   /* Note [2] */
{
    static uint16_t pwm;                /* Note [3] */
    static uint8_t direction;

    switch (direction)                  /* Note [4] */
    {
        case UP:
            if (++pwm == TIMER1_TOP)
                direction = DOWN;
            break;

        case DOWN:
            if (--pwm == 0)
                direction = UP;
            break;
    }

    OCR = pwm;                          /* Note [5] */
}
```

```
void
ioinit (void)                           /* Note [6] */
{
    /* Timer 1 is 10-bit PWM (8-bit PWM on some ATtinys).
*/
    TCCR1A = TIMER1_PWM_INIT;
    /* Start timer 1. */
    TCCR1B |= TIMER1_CLOCKSOURCE;

    /* Set PWM value to 0. */
    OCR = 0;

    /* Enable OC1 as output. */
    DDROC = _BV (OC1);

    /* Enable timer 1 overflow interrupt. */
    TIMSK = _BV (TOIE1);
    sei ();
}

int
main (void)
{

    ioinit ();

    /* loop forever, the interrupts are doing the rest */

    for (;;)                            /* Note [7] */
        sleep_mode();

    return (0);
}
```

Compiler

1. A software-development tool that translates high-level language programs into the machine-language instructions that a particular processor can understand and execute.

2. However, the object code that results is not yet ready to be run; at least a linker or link-step must follow



Compiling

```
$ avr-gcc -g -Os -mmcu=atmega8 -c demo.c
```

The compilation will create a **demo.o** file.

The -g is used to embed debug info. The debug info is useful for disassemblies and doesn't end up in the .hex files, so I usually specify it. Finally, the -c tells the compiler to compile and stop -- don't link.

…and Linking

Next we link it into a binary called demo.elf.

```
$ avr-gcc -g -mmcu=atmega8 -o demo.elf demo.o
```

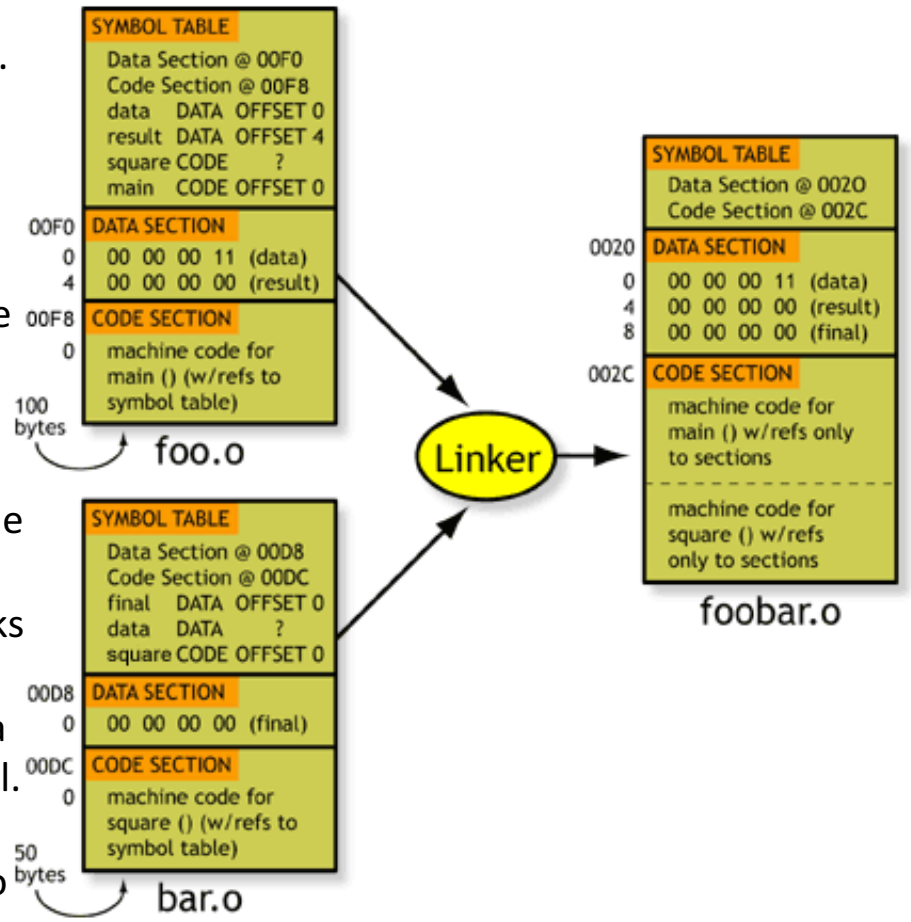
Linker

1. A software development tool that accepts one or more object files as input and outputs a relocatable program. The linker is thus run after all of the source files have been compiled and assembled into object files.

2. A simplified example showing how the linker resolves references across symbol tables and combines the code and data sections.

3. On the linker command line, you will specify a set of **object modules** and then a **list of libraries**, including the Standard C Library. The linker takes the set of object modules that you specify on the command line and links them together. Afterwards there will probably be a set of **"undefined references".** An undefined reference is a function call, with no defined function to match the call.

4. The linker will then go through the libraries, in order, to match the undefined references with function definitions that are found in the libraries



Examining the Object File

```
$ avr-objdump -h -S demo.elf > demo.lst
```

-S option disassembles the binary file and intersperses the source code in the output

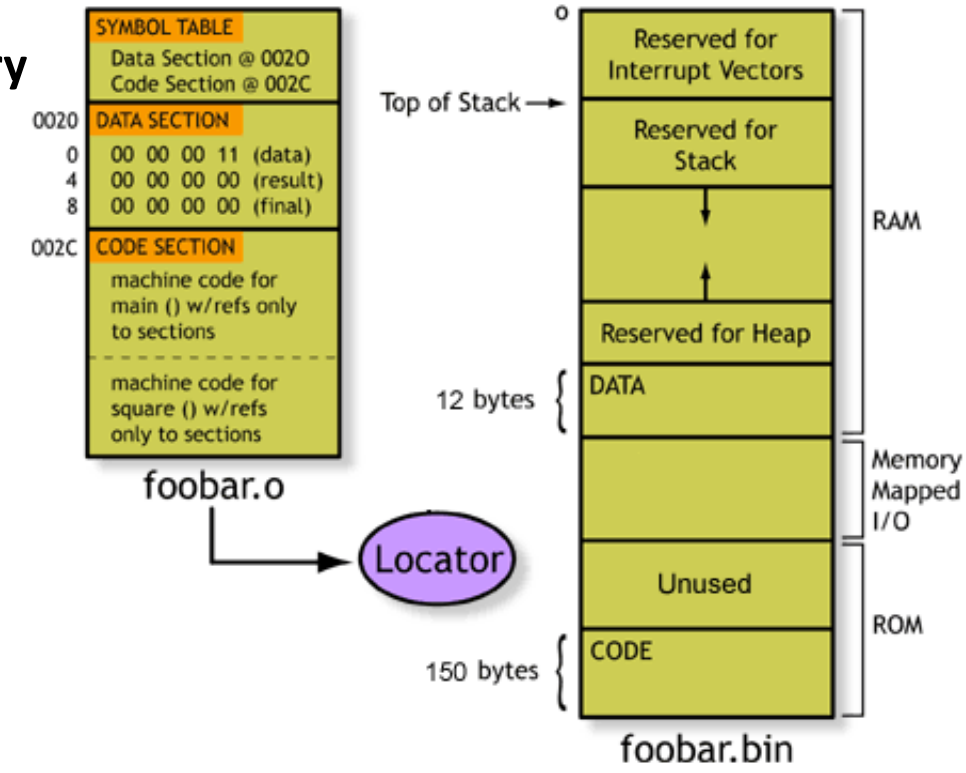
Here's the output as saved in the demo.lst file:

```
demo.elf:     file format elf32-avr
Sections: Idx Name Size VMA LMA File off Algn 0 .text 000000d8 00000000 00000000
00000074 2**1 CONTENTS, ALLOC, LOAD, READONLY, CODE 1 .bss 00000003 00800060
00800060 0000014c 2**0 ALLOC 2 .comment 0000002c 00000000 00000000 0000014c 2**0
CONTENTS, READONLY 3 .debug_aranges 00000068 00000000 00000000 00000178 2**3
CONTENTS, READONLY, DEBUGGING 4 .debug_info 00000339 00000000 00000000 000001e0
2**0 CONTENTS, READONLY, DEBUGGING 5 .debug_abbrev 00000102 00000000 00000000
00000519 2**0 CONTENTS, READONLY, DEBUGGING 6 .debug_line 00000215 00000000
00000000 0000061b 2**0 CONTENTS, READONLY, DEBUGGING 7 .debug_frame …..... .fina1
sections inserted by ld script. */ .section .fini0,"ax",@progbits cli d4: f8 94
cli 000000d6 <__stop_program>: __stop_program: rjmp __stop_program d6: ff cf
rjmp .-2 ; 0xd6 <__stop_program>
```

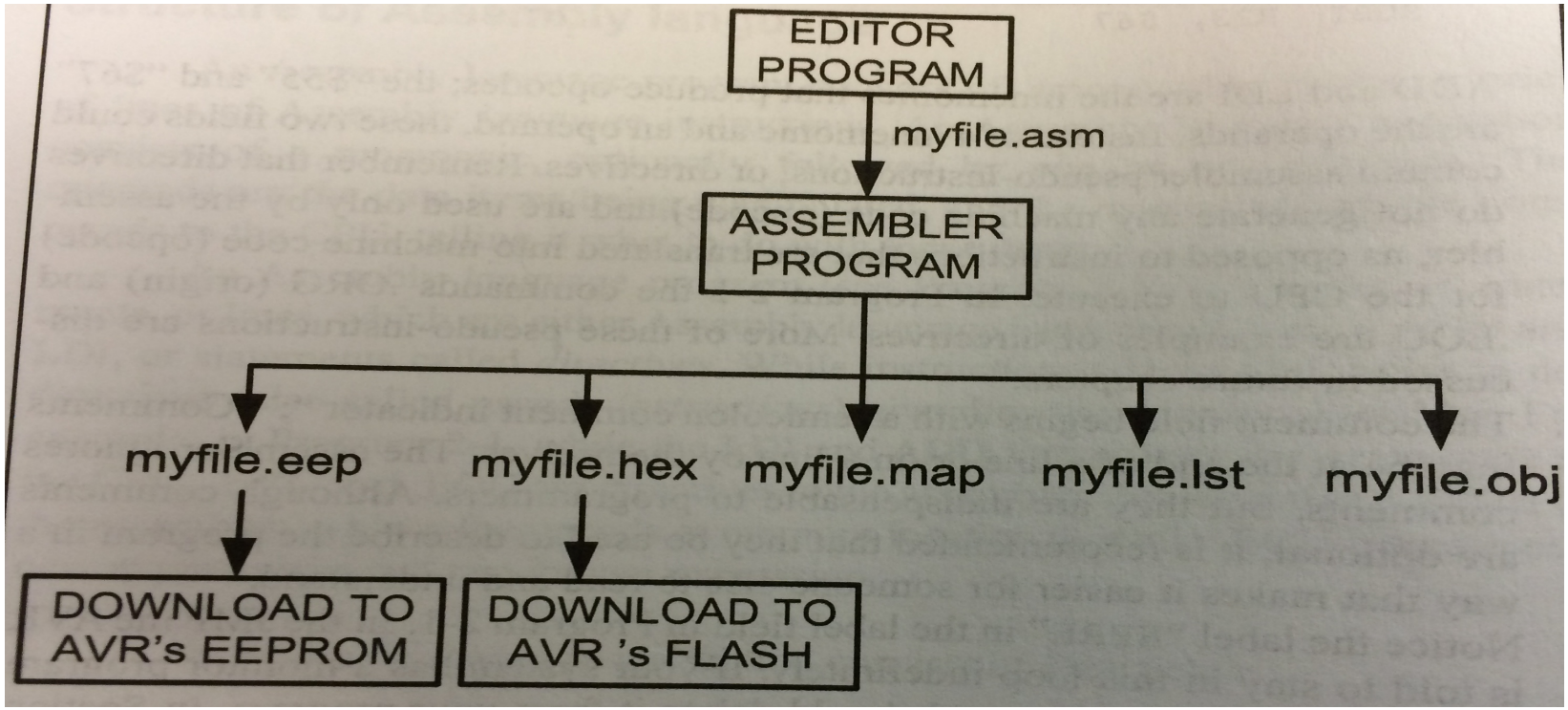
Locator (part of most linkers)

1. Maps relocatable program to **physical memory** specific to the particular target architecture

2. Maps .text .data .bss sections to their respective memory locations



From Atmel Studio to Executable Files



Linker Map files

A map file is useful for monitoring the sizes of your code and data. It also shows where modules are loaded and which modules were loaded from libraries.

__trampolines are where the linker puts function stubs

```
.rela.plt
 *(.rela.plt)

.text           0x0000000000000000       0xd8
 *(.vectors)
 *(.vectors)
 *(.progmem.gcc*)
                0x0000000000000000                . = ALIGN (0x2)
                0x0000000000000000                __trampolines_start = .
 *(.trampolines)
 .trampolines   0x0000000000000000        0x0 linker stubs
 *(.trampolines*)
                0x0000000000000000                __trampolines_end = .
 *(.progmem*)
                0x0000000000000000                . = ALIGN (0x2)
 *(.jumptables)
 *(.jumptables*)
 *(.lowtext)
 *(.lowtext*)
                0x0000000000000000                __ctors_start = .
```

Linker Map files

.text segment

.data segment

.bss segment

.eeprom segment

```
*(.fini2)
*(.fini2)
*(.fini1)
*(.fini1)
*(.fini0)
.fini0          0x00000000000000d4        0x4 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
*(.fini0)
                0x00000000000000d8                _etext = .

.data           0x0000000000800060        0x0 load address 0x00000000000000d8
                0x0000000000800060                PROVIDE (__data_start, .)
*(.data)
.data           0x0000000000800060        0x0 demo.o
.data           0x0000000000800060        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
.data           0x0000000000800060        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
.data           0x0000000000800060        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
*(.data*)
*(.rodata)
*(.rodata*)
*(.gnu.linkonce.d*)
                0x0000000000800060                . = ALIGN (0x2)
                0x0000000000800060                _edata = .
                0x0000000000800060                PROVIDE (__data_end, .)

.bss            0x0000000000800060        0x3
                0x0000000000800060                PROVIDE (__bss_start, .)
*(.bss)
.bss            0x0000000000800060        0x3 demo.o
.bss            0x0000000000800063        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
.bss            0x0000000000800063        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
.bss            0x0000000000800063        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-toolchain-de
*(.bss*)
*(COMMON)
                0x0000000000800063                PROVIDE (__bss_end, .)
                0x00000000000000d8                __data_load_start = LOADADDR (.data)
                0x00000000000000d8                __data_load_end = (__data_load_start + SIZEOF (.data))

.noinit         0x0000000000800063        0x0
                0x0000000000800063                PROVIDE (__noinit_start, .)
*(.noinit*)
                0x0000000000800063                PROVIDE (__noinit_end, .)
                0x0000000000800063                _end = .
                0x0000000000800063                PROVIDE (__heap_start, .)

.eeprom         0x0000000000810000        0x0
*(.eeprom*)
                0x0000000000810000                __eeprom_end = .
```

Linker Map files

```
.bss            0x0000000000800060        0x3
                0x0000000000800060                PROVIDE (__bss_start, .)
*(.bss)
.bss            0x0000000000800060        0x3 demo.o
.bss            0x0000000000800063        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-
.bss            0x0000000000800063        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-
.bss            0x0000000000800063        0x0 /data2/home/toolsbuild/jenkins-mcu-knuth/workspace/avr8-gnu-
*(.bss*)
*(COMMON)
                0x0000000000800063                PROVIDE (__bss_end, .)
                0x00000000000000d8                __data_load_start = LOADADDR (.data)
                0x00000000000000d8                __data_load_end = (__data_load_start + SIZEOF (.data))

.noinit         0x0000000000800063        0x0
                0x0000000000800063                PROVIDE (__noinit_start, .)
*(.noinit*)
                0x0000000000800063                PROVIDE (__noinit_end, .)
                0x0000000000800063                _end = .
                0x0000000000800063                PROVIDE (__heap_start, .)

.eeprom         0x0000000000810000        0x0
*(.eeprom*)
                0x0000000000810000                __eeprom_end = .
```

The **.text segment** (where program instructions are stored) starts at location 0x0.The last address in the .text segment is location 0xD8 (denoted by _etext), so the instructions use up 216 bytes of FLASH.

The **.data segment** (where initialized static variables are stored) starts at location 0x60, which is the first address after the register bank on an ATmega8 processor.
The next available addr in the .data segment is also loc 0x60, so the application has no initialized data.

The **.bss segment** (where uninitialized data is stored) starts at location 0x60.

The next available address in the .bss segment is at 0x63, so the app uses 3 bytes of uninitialized data.

Linker Map file

```
*(.fini2)
*(.fini2)
*(.fini1)
*(.fini1)
*(.fini0)
.fini0          0x00000000000000d4      0x4 /libgcc.a(_exit.o)
*(.fini0)
                0x00000000000000d8              _etext = .

.data           0x0000000000800060      0x0 load address 0x0000000d8
                0x0000000000800060              PROVIDE (__data_start, .)
*(.data)
.data           0x0000000000800060      0x0 demo.o
.data           0x0000000000800060      0x0 /exit.o
.data           0x0000000000800060      0x0 /libgcc.a(_exit.o)
.data           0x0000000000800060      0x0 /libgcc.a(_clear_bss.o)
*(.data*)
*(.rodata)
*(.rodata*)
*(.gnu.linkonce.d*)
                0x0000000000800060              . = ALIGN (0x2)
                0x0000000000800060              _edata = .
                0x0000000000800060              PROVIDE (__data_end, .)
```

```
.bss            0x0000000000800060      0x3
                0x0000000000800060              PROVIDE (__bss_start, .)
*(.bss)
.bss            0x0000000000800060      0x3 demo.o
.bss            0x0000000000800063      0x0 /exit.o
.bss            0x0000000000800063      0x0 /libgcc.a(_exit.o)
.bss            0x0000000000800063      0x0 /libgcc.a(_clear_bss.o)
*(.bss*)
*(COMMON)
                0x0000000000800063              PROVIDE (__bss_end, .)
                0x00000000000000d8              __data_load_start = LOADADDR
(.data)
                0x00000000000000d8              __data_load_end =
(__data_load_start + SIZEOF (.data))

.noinit         0x0000000000800063      0x0
                0x0000000000800063              PROVIDE (__noinit_start, .)
*(.noinit*)
                0x0000000000800063              PROVIDE (__noinit_end, .)
                0x0000000000800063              _end = .
                0x0000000000800063              PROVIDE (__heap_start, .)

.eeprom         0x0000000000810000      0x0
*(.eeprom*)
                0x0000000000810000              __eeprom_end = .
```

Generating Intel Hex format files

The ROM contents can be pulled from our project's binary and put into the file demo.hex using the following command:

```
$ avr-objcopy -j .text -j .data -O ihex demo.elf demo.hex
```

The resulting demo.hex file contains:

```
:1000000020E0A0E6B0E001C01D92A336B207E1F700
:100010001F920F920FB60F9211242F938F939F93DD
:1000200080916200882301F1813081F48091600029
:1000300090916100019790936100809360000009718
:1000400049F41092620080E090E004C0809160006A
:10005000909161009BBD8ABD9F918F912F910F90D0
:100060000FBE0F901F9018958091600090916100D5
:1000700001969093610080936000 8F3F23E0920788
:1000800049F781E0809362008FEF93E0E3CF83E84C
:100090008FBD8EB581608EBD1BBC1ABC82E087BB54
:1000A00084E089BF7894089583E88FBD8EB5816020
:1000B0008EBD1BBC1ABC82E087BB84E089BF7894EC
:1000C00085B7806885BF889585B78F7785BFF8CF5E
:0800D000F89400C0F894FFCF82
:00000001FF
```

What's in a HEX file?

- A HEX file consists of lines of program instructions, all of which start with a semicolon.
- Is created every time you compile a project.
- Is an ASCII text file that represents machine language and constant data.
- Tell the In-Circuit Serial Programmer where to place data into ROM and what data to place so our program can execute instructions correctly.
- Each line can be one of six different formats; we will only need to know three for this class.

Hex Record Format

Record Format

An Intel HEX file is composed of any number of HEX records. Each record is made up of five fields that are arranged in the following format:

```
:llaaaatt[dd...]cc
```

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits-which make up a byte-as described below:

- **:** is the colon that starts every Intel HEX record.
- ***ll*** is the record-length field that represents the number of data bytes (**dd**) in the record.
- ***aaaa*** is the address field that represents the starting address for subsequent data in the record.
- ***tt*** is the field that represents the HEX record type, which may be one of the following:
  00 - data record
  01 - end-of-file record
  02 - extended segment address record
  04 - extended linear address record
  05 - start linear address record (MDK-ARM only)
- ***dd*** is a data field that represents one byte of data. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the **ll** field.
- ***cc*** is the checksum field that represents the checksum of the record. The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two's complement.

We will talk about only the data record file as it is the most important. For an overview of the other record formats visit.

http://www.keil.com/support/docs/1584.htm

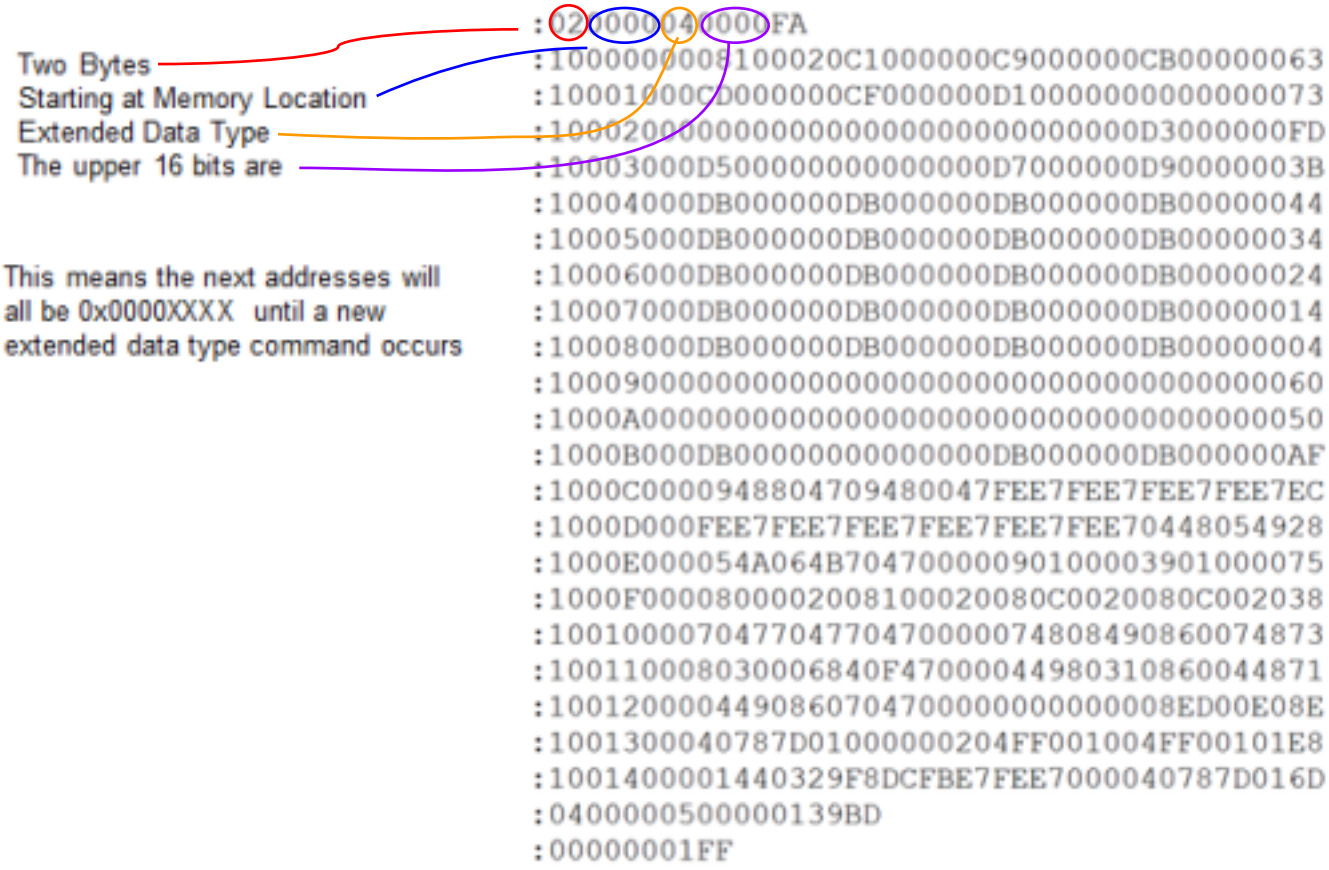
Compiled for starting address 0x0000

```
:020000040000FA
:1000000008100020C1000000C9000000CB00000063
:10001000CD000000CF000000D10000000000000073
:10002000000000000000000000000000D3000000FD
:1000300D5000000000000000D7000000D90000003B
:1000400DB000000DB000000DB000000DB00000044
:1000500DB000000DB000000DB000000DB00000034
:1000600DB000000DB000000DB000000DB00000024
:1000700DB000000DB000000DB000000DB00000014
:1000800DB000000DB000000DB000000DB00000004
:10009000000000000000000000000000000000060
:1000A000000000000000000000000000000000050
:1000B000DB0000000000000DB000000DB000000AF
:1000C000094880470948004 7FEE7FEE7FEE7FEE7EC
:1000D000FEE7FEE7FEE7FEE7FEE7FEE70448054928
:1000E000054A064B704700000901000039010000 75
:1000F000080000200810002008 0C0020080C002038
:10010000704770477047000007480849086007 4873
:10011000803000684 0F4700004498031086004 4871
:10012000449086070470000000000008ED00E08E
:1001300040787D01000000204FF001004FF00101E8
:10014000 1440329F8DCFBE7FEE7000040787D016D
:04000005000000139BD
:00000001FF
```

Compiled for starting address 0x4000

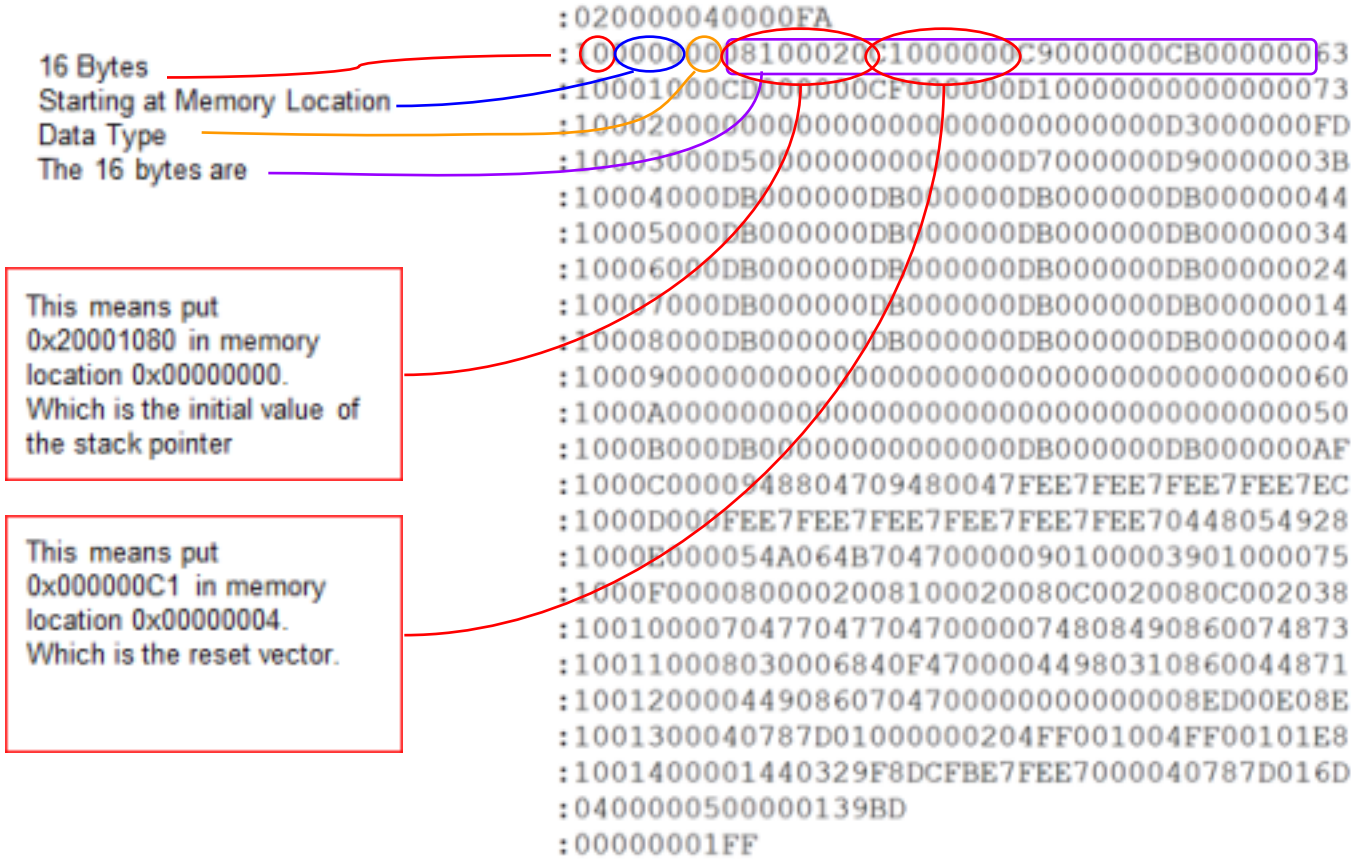
```
:020000040000FA
:1040000008100020C1400000C9400000CB40000063
:10401000CD400000CF400000D14000000000000073
:10402000000000000000000000000000D34000007D
:1040300D5400000000000000D7400000D94000003B
:1040400DB400000DB400000DB400000DB40000004
:1040500DB400000DB400000DB400000DB400000F4
:1040600DB400000DB400000DB400000DB400000E4
:1040700DB400000DB400000DB400000DB400000D4
:1040800DB400000DB400000DB400000DB400000C4
:10409000000000000000000000000000000000020
:1040A000000000000000000000000000000000010
:1040B000DB4000000000000DB400000DB400000AF
:1040C000094880470948004 7FEE7FEE7FEE7FEE7AC
:1040D000FEE7FEE7FEE7FEE7FEE7FEE704480549E8
:1040E000054A064B704700000941000039410000B5
:1040F000080000200810002008 0C0020080C0020F8
:10410000704770477047000007480849086007 4833
:1041100080300068 40F4700004498031086004 4831
:10412000449086070470000004000008ED00E00E
:1041300040787D01000000204FF001004FF00101A8
:10414000 1440329F8DCFBE7FEE7000040787D012D
:04000005000041397D
:00000001FF
```

Hex Record Format



This means the next addresses will all be 0x0000XXXX until a new extended data type command occurs

Hex Record Format



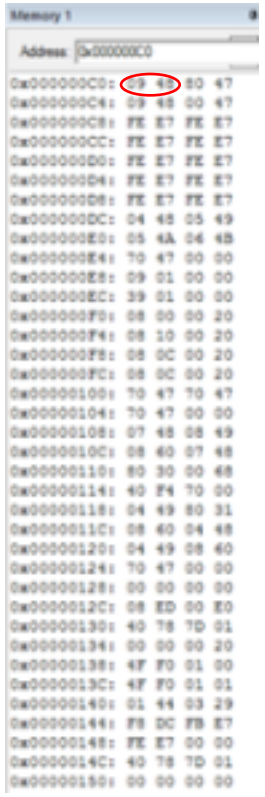
Hex Record Format

```
:020000040000FA
:10000000081000200C1000000C9000000CB00000063
:10001000CD000000CF000000D10000000000000073
:10002000000000000000000000000000D3000000FD
:10003000D500000000000000D7000000D90000003B
:10004000DB000000DB000000DB000000DB00000044
:10005000DB000000DB000000DB000000DB00000034
:10006000DB000000DB000000DB000000DB00000024
:10007000DB000000DB000000DB000000DB00000014
:10008000DB000000DB000000DB000000DB00000004
:10009000000000000000000000000000000000060
:1000A000000000000000000000000000000000050
:1000B000DB000000000000000DB000000DB000000AF
:1000C000094804709480047FEE7FEE7FEE7FEE7EC
:1000D000FEE7FEE7FEE7FEE7FEE70448054928
:1000E0000054A064B704700000901000039010000075
:1000F000080000200810002008C0020080C002038
:10010000704770477047000007480849086007487 3
:10011000803000684 0F4700004498031086004487 1
:1001200004490860704700000000000008ED00E08E
:10013000407870100000204FF001004FF00101E8
:10014000014403 29F8DCFBE7FEE7000040787D016D
:0400000500000139BD
:00000001FF
```

16 Bytes
Starting at Memory Location
Data Type
The 16 bytes are

First
instruction!

First instruction at
reset vector



Hex Record Format

```
:020000040000FA
:10000000081000020C1000000C9000000CB00000063
:10001000CD000000CF000000D10000000000000073
:10002000000000000000000000000000D3000000FD
:10003000D500000000000000D7000000D90000003B
:10004000DB000000DB000000DB000000DB00000044
:10005000DB000000DB000000DB000000DB00000034
:10006000DB000000DB000000DB000000DB00000024
:10007000DB000000DB000000DB000000DB00000014
:10008000DB000000DB000000DB000000DB00000004
:10009000000000000000000000000000000000060
:1000A000000000000000000000000000000000050
:1000B000DB00000000000000DB000000DB000000AF
:1000C000094880470948004 7FEE7FEE7FEE7FEE7EC
:1000D000FEE7FEE7FEE7FEE7FEE7FEE70448054928
:1000E000054A064B704700000901000039010000 75
:1000F000080000200810002008 0C0020080C002038
:10010000704770477047000007480849086007 4873
:10011000803000684 0F4700004498031086004 4871
:10012000044908607047000000000008ED00E08E
:1001300040787D01000000204FF001004FF00101E8
:10014000014403 29F8DCFBE7FEE7000040787D016D
:04000005000001 39BD
:00000001FF
```

4 Bytes
Starting at Memory Location
Data Type 05, Start of linear address
Data of linear start address

This is the entry location into the **main** routine, NOT the reset vector. Notice the endianess. The main routine starts at 0x00000139

Hex Record Format

```
:020000040000FA
:10000000081000020C1000000C9000000CB00000063
:10001000CD000000CF000000D10000000000000073
:10002000000000000000000000000000D3000000FD
:10003000D500000000000000D7000000D90000003B
:10004000DB000000DB000000DB000000DB00000044
:10005000DB000000DB000000DB000000DB00000034
:10006000DB000000DB000000DB000000DB00000024
:10007000DB000000DB000000DB000000DB00000014
:10008000DB000000DB000000DB000000DB00000004
:10009000000000000000000000000000000000060
:1000A000000000000000000000000000000000050
:1000B000DB00000000000000DB000000DB000000AF
:1000C000094880470948004 7FEE7FEE7FEE7FEE7EC
:1000D000FEE7FEE7FEE7FEE7FEE7FEE70448054928
:1000E000054A064B704700000901000039010000 75
:1000F000080000200810002008 0C0020080C002038
:10010000704770477047000007480849086007 4873
:10011000803000684 0F4700004498031086004 4871
:10012000044908607047000000000008ED00E08E
:1001300040787D01000000204FF001004FF00101E8
:1001400001440329F8DCFBE7FEE7000040787D016D
:0400000500000139BD
:00000001FF
```

0 data bytes
Address of 0x0000
(usually ignored)
End of file data type

This is the entry location into the **main** routine, NOT the reset vector. Notice the endianess. The main routine starts at 0x00000139

Data Record Example

Data Records

The Intel HEX file is made up of any number of data records that are terminated with a carriage return and a linefeed. Data records appear as follows:

```
:10246200464C5549442050524F46494C4500464C33
```

This record is decoded as follows:

```
:10246200464C5549442050524F46494C4500464C33
|||||||||||                               CC->Checksum
|||||||||DD->Data
|||||||TT->Record Type
|||AAAA->Address
|LL->Record Length
:->Colon
```

where:

- **10** is the number of data bytes in the record.
- **2462** is the address where the data are to be located in memory.
- **00** is the record type 00 (a data record).
- **464C...464C** is the data.
- **33** is the checksum of the record.

Data record example

Hex line:
:10457000**0F0A1F9**0020694**00F0B0F9**1CBC5DF81C

: = intel format:
10 = 16 bytes of data:
4570 = starting memory address
00 = data line

Data word aligned (**00F0A1F9**, 00206946, **00F0B0F9**, 1CBC5DF8)

Check sum = 1C

Writing to memory
0x4570 = 0x00F0A1F9
0x4574 = 0x00206946
0x4578 = 0x00F0B0F9
0x457B = 0x1CBC5DF8

Calculating the checksum:
(0x01 + NOT(0x10 + 0x45 + 0x70 + 0x00 + 0x00 + 0xF0 + 0xA1 + 0xF9 + 0x00 + 0x20 + 0x69 + 0x46 + 0x00 + 0xF0 + 0xB0 + 0xF9 + 0x1C + 0xBC + 0x5D + 0xF8))

2. How is an embedded system programmed for the first time?

Let’s look at Fuses and Lock Bits

Fuses

Everyone knows that AVR MCUs have, in general, three memory areas: FLASH, which is dedicated to program code, SRAM for run-time variables and EEPROM, which can be used by user code to store data that have to be preserved when MCU is turned off. Now, the lock/fuses form a fourth memory area available for programming. This holds a few bytes that contain those bits. For example, ATmega328P has four of them – one byte for lock bits and three bytes for fuses: low, high and extended.

The fuses determine how the chip will act, whether it has a bootloader, what speed and voltage it likes to run at, etc. Note that despite being called 'fuses' they are re-settable and don't have anything to do with protection from overpowering (like the fuses in a home).

The AVR microcontroller (ATmega16) consists of sixteen fuse bits - low fuse and high fuse bits.

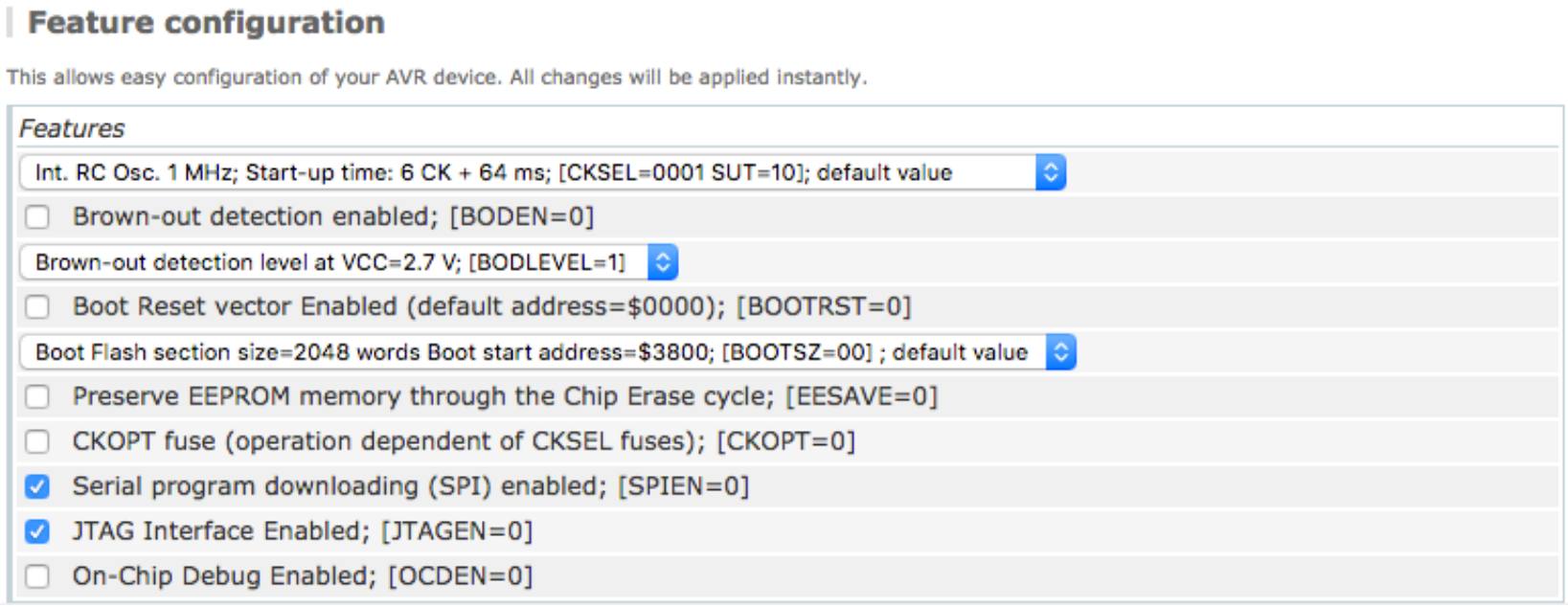
High Fuse bits:

Fuse Bit	OCDEN	JTAGEN	SPIEN	CKOPT	EESAVE	BOOTSZ1	BOOTSZ0	BOOTRST
Bit No.	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Default values	1	0	0	1	1	0	0	1

Low Fuse bits:

Fuse Bit	BODLEVEL	BODEN	SUT1	SUT0	CKSEL3	CKSEL2	CKSEL1	CKSEL0
Bit No.	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Default values	1	1	1	0	0	0	0	1

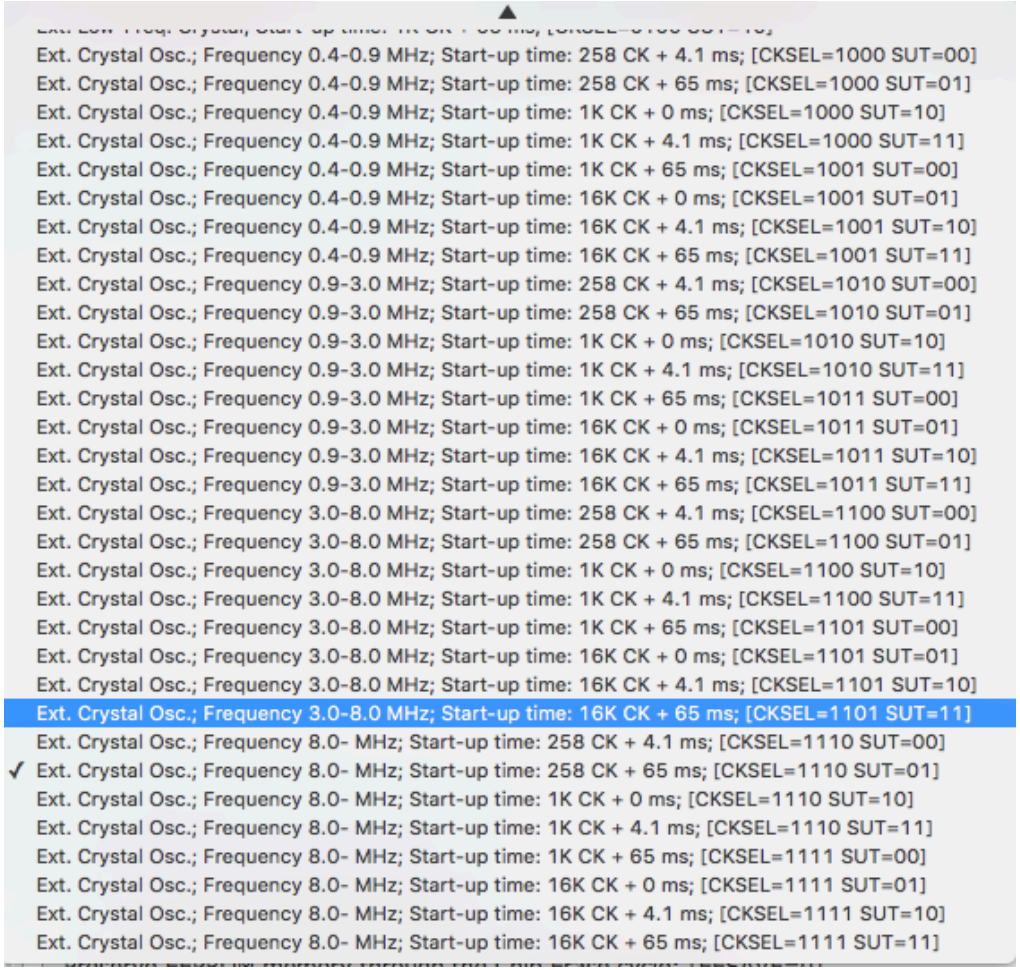
Fuses - configuration



Fuses - configuration

The AVR is set to an internal oscillator of 1MHz

You can change the crystal oscillator frequency to suit your platform by setting the CKSEL fuses



Fuses - configuration

Manual fuse bits configuration

This table allows reviewing and direct editing of the AVR fuse bits. All changes will be applied instantly.
Note: ☐ means unprogrammed (1); ☑ means programmed (0).

Bit	*Low*	*High*
7	☐ **BODLEVEL** Brown out detector trigger level	☐ **OCDEN** Enable OCD
6	☐ **BODEN** Brown out detector enable	☑ **JTAGEN** Enable JTAG
5	☐ **SUT1** Select start-up time	☑ **SPIEN** Enable Serial programming and Data Downloading
4	☑ **SUT0** Select start-up time	☐ **CKOPT** Oscillator Options
3	☑ **CKSEL3** Select Clock Source	☐ **EESAVE** EEPROM memory is preserved through chip erase
2	☑ **CKSEL2** Select Clock Source	☑ **BOOTSZ1** Select Boot Size
1	☑ **CKSEL1** Select Clock Source	☑ **BOOTSZ0** Select Boot Size
0	☐ **CKSEL0** Select Clock Source	☐ **BOOTRST** Select Reset Vector

Fuses - configuration

Current settings

These fields show the actual hexadecimal representation of the fuse settings from above. These are the values you have to program into your AVR device. Optionally, you may fill in the numerical values yourself to preset the configuration to these values. Changes in the value fields are applied instantly (taking away the focus)!

Low	*High*	*Action*	*AVRDUDE arguments*
0x E1	0x 99	Apply values Defaults Apply manual changes to the values on the left side, or load factory default values for the selected device.	`-U lfuse:w:0xe1:m -U hfuse:w:0x99:m` Select (try triple-click) and copy-and-paste this option string into your avrdude command line. You may specify multiple -U arguments within one call of avrdude.

Fuses - example

ATmega48 is shipped with internal 8MHz RC oscillator set as an active clock source and a clock divider CKDIV8 fuse set. This results in 1MHz system clock. It's usually too slow for me, and I always turn this fuse off to get full 8MHz.

Table 28-7. Fuse low byte.

Low fuse byte	Bit no.	Description	Default value
CKDIV8[4]	7	Divide clock by 8	0 (programmed)
CKOUT[3]	6	Clock output	1 (unprogrammed)
SUT1	5	Select start-up time	1 (unprogrammed)[1]
SUT0	4	Select start-up time	0 (programmed)[1]
CKSEL3	3	Select clock source	0 (programmed)[2]
CKSEL2	2	Select clock source	0 (programmed)[2]
CKSEL1	1	Select clock source	1 (unprogrammed)[2]
CKSEL0	0	Select clock source	0 (programmed)[2]

Note:
1. The default value of SUT1..0 results in maximum start-up time for the default clock source. See Table 9-9 on page 34 for details.
2. The default setting of CKSEL3..0 results in internal RC oscillator @ 8MHz. See Table 9-8 on page 34 for details.
3. The CKOUT fuse allows the system clock to be output on PORTB0. See "Clock output buffer" on page 35 for details.
4. See "System clock prescaler" on page 36 for details.

Its default value is 0b01100010. CKDIV8 is seventh bit and we don't need to change any other bits, so only MSB needs to be changed. Hence the new value is 0b11100010=0xE2. The AVRdude command line is as follows:

```
avrdude – U lfuse:w:0xE2:m
```

Setting fuses with avrDUDE

If not drone right, you can brick you mcu

Reset Disable

This fuse turns the Reset pin into a normal pin instead of a special pin. If you turn this fuse on you cant program the chip using ISP anymore. I would suggest you never set this fuse unless you really mean to.

By default, chips that come from the factory have this turned off (that is, Reset is enabled)

avrdude -c usbtiny -p attiny2313 -U lfuse:w:<0xHH>:m
avrdude -c usbtiny -p attiny2313 -U hfuse:w:<0xHH>:m
avrdude -c usbtiny -p attiny2313 -U efuse:w:<0xHH>:m

Configuring a new microcontroller

A virgin microcontroller controller is configured at 1MHz internal RC oscillator with longest start-up and the JTAG pre-enabled. So the fuse bytes are as follows:

High Fuse bit = 1001 1001 = $99
Low Fuse bit = 1110 0001 = $E1

Important: Setting a fuse means writing a 0, not 1
fuse bit = 1 is UN-programmed (inactive);
fuse bit = 0 is programmed (active);

To set the microcontroller for high external frequency with longest start-up and JTAG disabled, the fuse settings are changed as following.

- CKSEL [3:0] = 1111
- SUT [1:0] = 11
- CKOPT = 0
- JTAGEN = 1

So,

High Fuse bit = 1100 1001 = $C9
Low Fuse bit = 1111 1111 = $FF

Standard Arduino Duemilanove settings. 16MHz slow power starting, Bootloader enabled with boot size 128 words (optiboot). Typical brown out voltage selected at 2.7V.

Low	*High*	*Extended*
0x FF	0x DE	0x 05

Lock bits

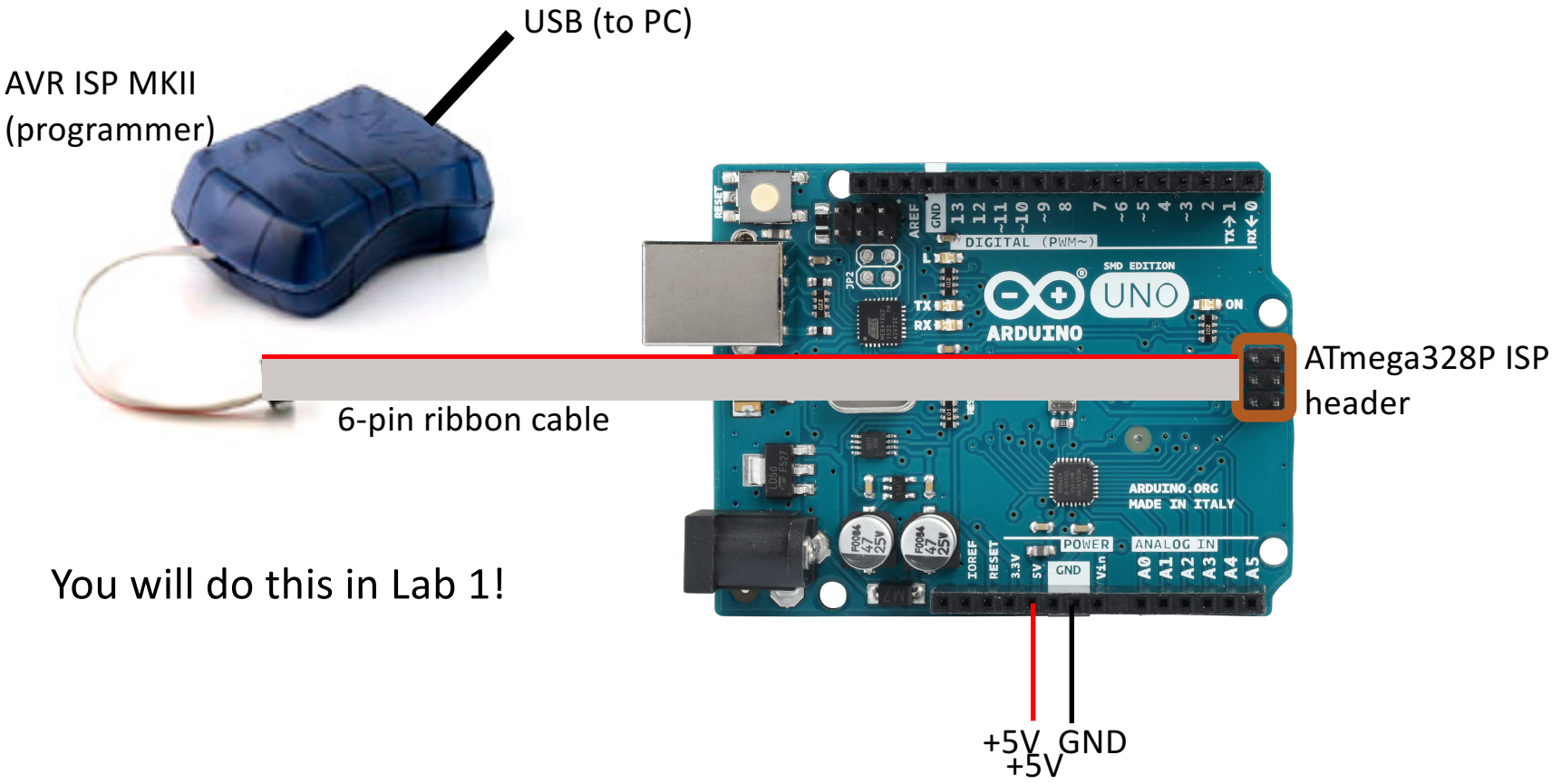
Bits	7	6	5	4	3	2	1	0
	-	-	BLB12	BLB11	BLB02	BLB01	LB2	LB1

- LB1 and LB2 bits are used to lock memory. Product developers always lock their microcontrollers to protect their intellectual property. So no one could read and duplicate.
- Other lock bit (BLB01, BLB02, BLB11 and BLB11) settings may be used to lock writing and reading to/from FLASH memory either from application area or bootloader section
- Usually you don’t change lock bits, but if you set any of those, lock bits are reset during chip erase. Just be careful ☺

Background

- Programming (flashing) a microcontroller may seem much like compiling a piece of code to run on a PC, but the two processes actually have substantial differences.
- When you compile code for a PC, you usually do so with the intent of running that program on the same PC that compiled it.
  - At least, on a PC with the same hardware and software architecture.
  - e.g. compile for Windows, run on Windows. Compile on Linux, run on Linux.
- Code for microcontrollers, however, must be compiled on a PC and then *uploaded* to the device later.
  - This is known as **cross-compiling**.
  - Key difference: the architectures (hardware or software) are not the same.

Flashing the MCU with an ICSP



You will do this in Lab 1!

Flashing the MCU with an ICSP

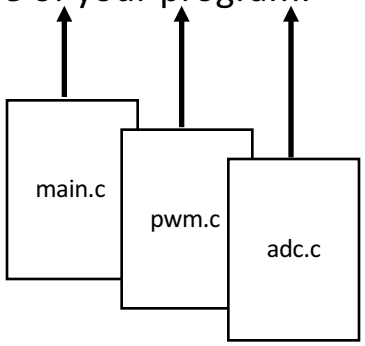
- We have to upload our compiled code to a MCU and need some way of placing the compiled code into some location in memory.
- One approach is to copy our compiled code directly to the device and have it run immediately from the device's execution start address.
  - This approach requires us to use a device known as a **programmer** to program the MCU. This programmer interfaces with a standard port on the MCU known as an **ISP header** (In-System Programmer) and sends program data from the PC to the MCU during programming.
  - This port is 6 pins wide and uses the SPI (Serial Peripheral Interface) protocol.

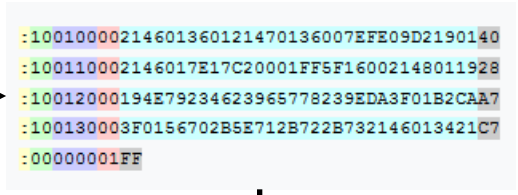
From Source Code to 328P

- There are several things that happen between when you click the "Compile" button in Arduino and when you actually get to run your code on the board.
    1. **Compilation/Linking**: the compiler takes in your Arduino source code and outputs a special file (known as a HEX file) that contains your program's binary data and where it will be placed in memory on the board.
    2. **Parsing the HEX file**: prior to programming the board, a program called avrdude on your PC parses the code into an address range it belongs to (known as a *page*) and the program instruction data itself.
    3. **Uploading the code**: the parsed HEX file data is then sent to the Arduino via a serial link, and the bootloader programs the board using this data.

From Source Code to 328P

Compiler compiles source code files into object files. Linker links these object files and produces a HEX file of your program.





avrdude (program on PC) parses the HEX file into AVR instruction opcodes and addresses (pages) where they will reside in program memory.

```
ADDRESS: 0x0000
LENGTH: 0xFF
INSNS: 0xDF, 0xA1, …
```

avrdude sends the parsed data over the serial link to the 328P. The bootloader on the 328P receives this information and programs the board.

1. Compilation/Linking	2. Parsing the HEX File	3. Uploading the Code

Arduino\hardware\arduino**boards.txt**

```
uno.name=Arduino Uno
uno.upload.protocol=arduino
uno.upload.maximum_size=32256   // maximum upload size. It's a 32K flash memory
device
uno.upload.speed=115200
uno.bootloader.low_fuses=0xff
uno.bootloader.high_fuses=0xde
uno.bootloader.extended_fuses=0x05
uno.bootloader.path=optiboot    // folder to use in the bootloaders repository
uno.bootloader.file=optiboot_atmega328.hex
uno.bootloader.unlock_bits=0x3F
uno.bootloader.lock_bits=0x0F
uno.build.mcu=atmega328p  // MCU type for this profile (Used with avrdude.conf file)
uno.build.f_cpu=16000000L
uno.build.core=arduino
```

avrDUDE

```
avrdude.exe -p m48 -c ftbb -P ft0 -e -U lfuse:w:0xE2:m -B 4800 -u
```

-p m48 – MCU type is ATmega48;

-c ftbb -P ft0 – my programming adapter is ftbb;

-e – erase chip before programming;

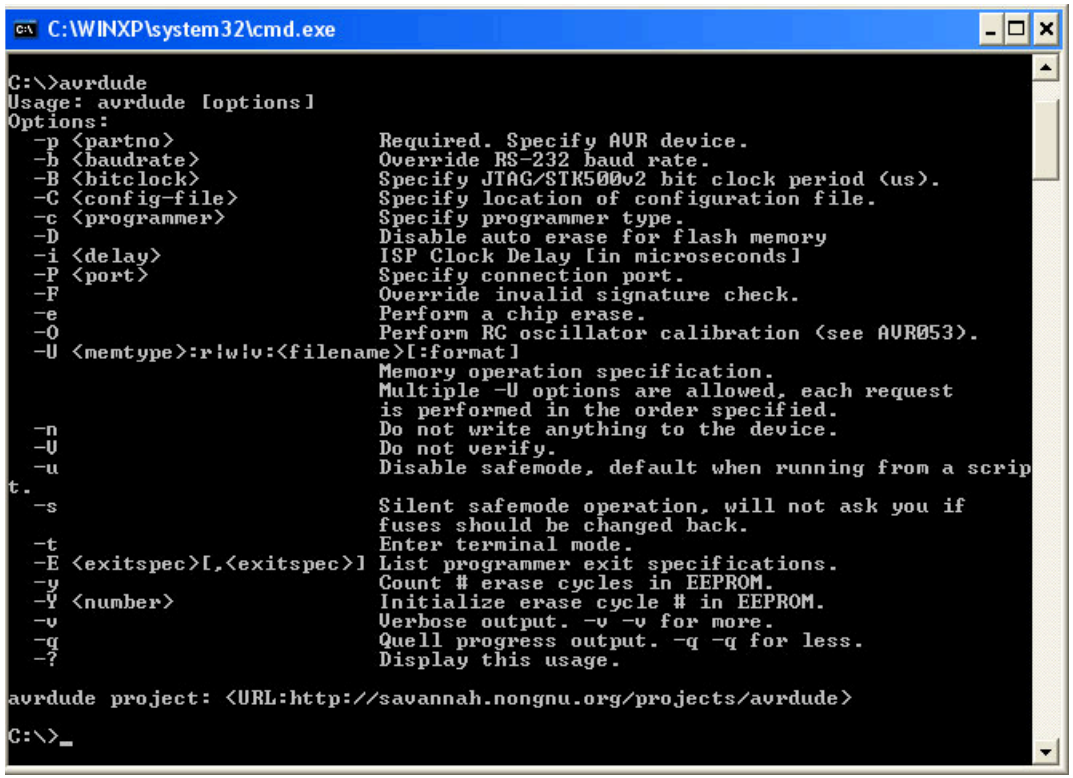
-U lfuse:w – write low fuse (lfuse) byte…

:0xE2:m – with 0xE2;

-B 4800 – program at lower speed;

-u – program anyway, ignore signature check.

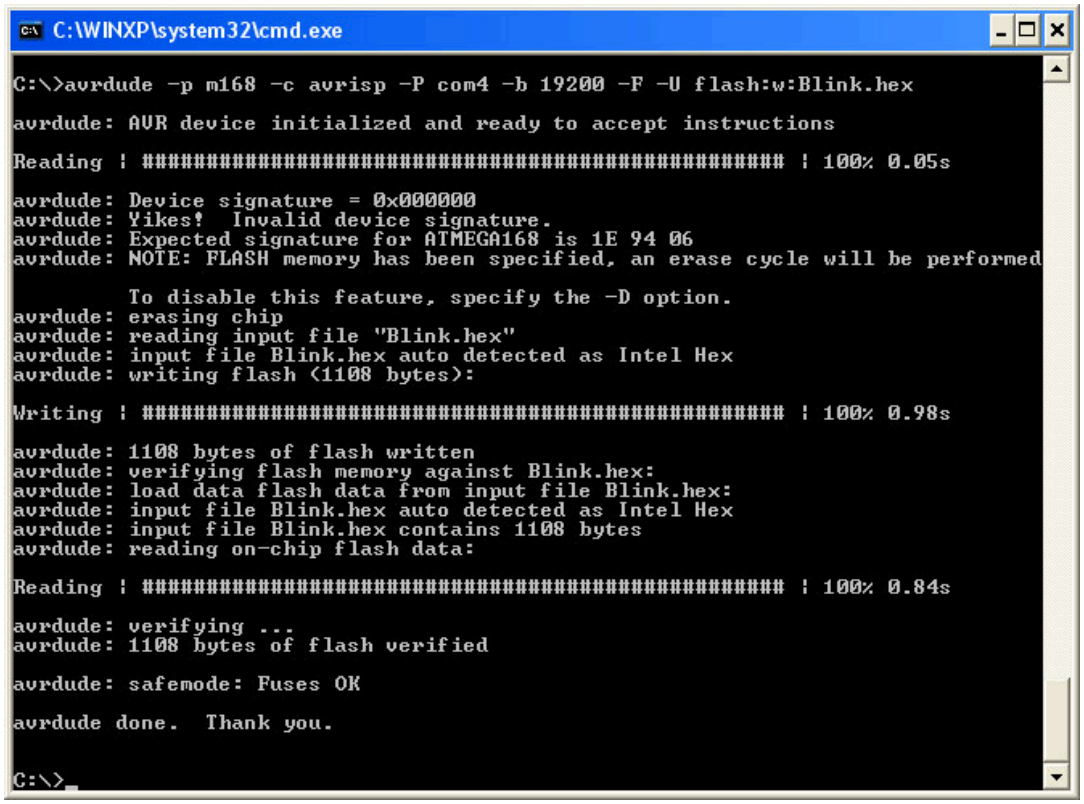
avrDUDE flashing a new image (1/2)



avrDUDE should go through the following steps:

1. Initializing the programmer
2. Initializing the AVR device and making sure its ready for instructions
3. Reading the device signature (**0x1e9406**) which confirms that the chip you specified in the command line (**avr328P**) is in fact the chip that the programmer is connected to
4. Erasing the chip
5. Reading the file and verifying its a valid file
6. Writing the flash
7. Verifying the flash

avrDUDE flashing a new image (2/2)



avrDUDE should go through the following steps:

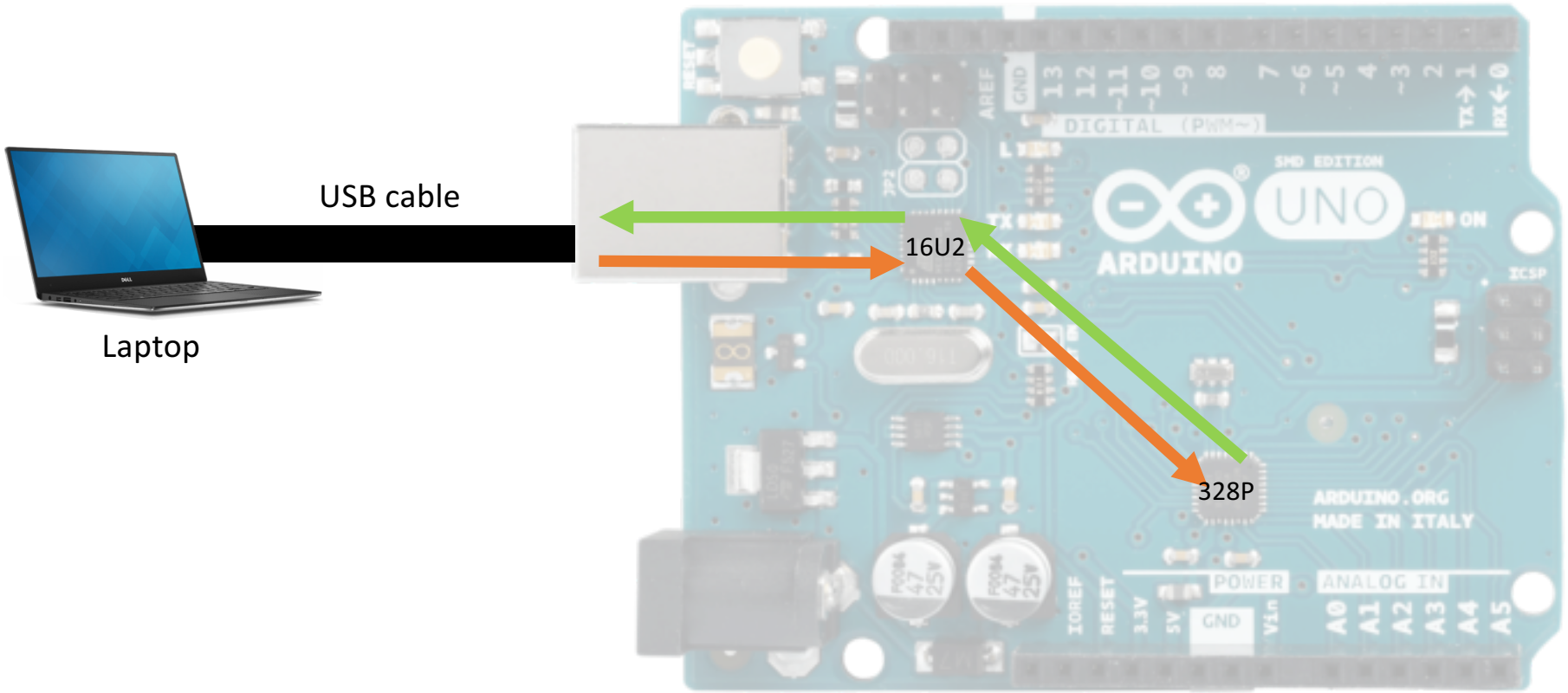
1. Initializing the programmer
2. Initializing the AVR device and making sure its ready for instructions
3. Reading the device signature (**0x1e9406**) which confirms that the chip you specified in the command line (**avr328P**) is in fact the chip that the programmer is connected to
4. Erasing the chip
5. Reading the file and verifying its a valid file
6. Writing the flash
7. Verifying the flash

3. Bootloaders for Serial Programming

- Using a programmer works, but it's not convenient since you still have to supply power and ground connections, either through the USB connector or via the power pins.
  - We can do better and combine the data and power links into one USB cable.

- The solution? Preload a *bootloader* onto the ATmega328 chip using the programmer, and then upload your code to the chip through a USB-to-serial connection.
  - In general terms, a **bootloader** is a program that runs on startup, configuring a computer or device before the main program or task begins.
  - When the device is reset, the Arduino bootloader checks the serial line for a new program to be uploaded. If one is present, the bootloader copies the program into flash memory and then runs whatever program is present there. (Note that a new program need not be present on each reset.)

Overview of Serial Communication
to program the ATmega328P via the ATmega16U2



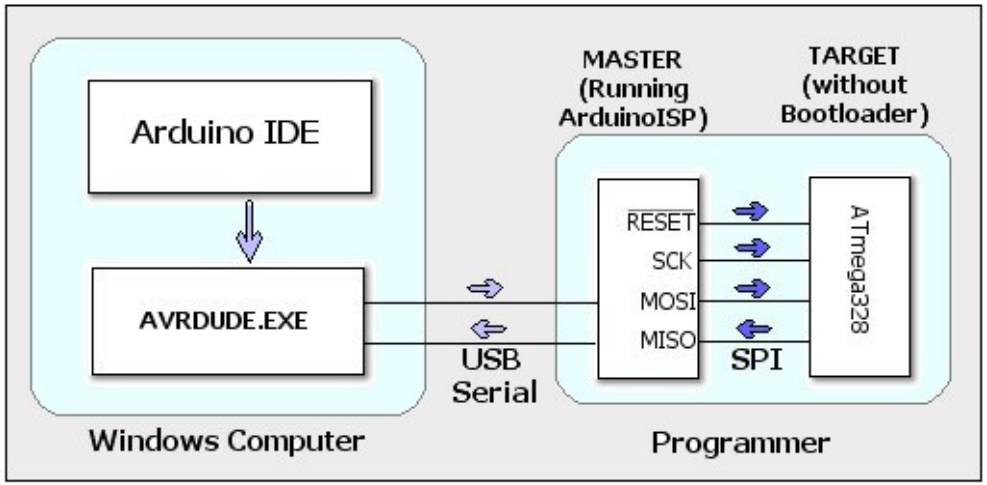
When I upload a new sketch to my AVR MCU using the Optiboot bootloader, what *really* happens?

What is sent to the MCU?

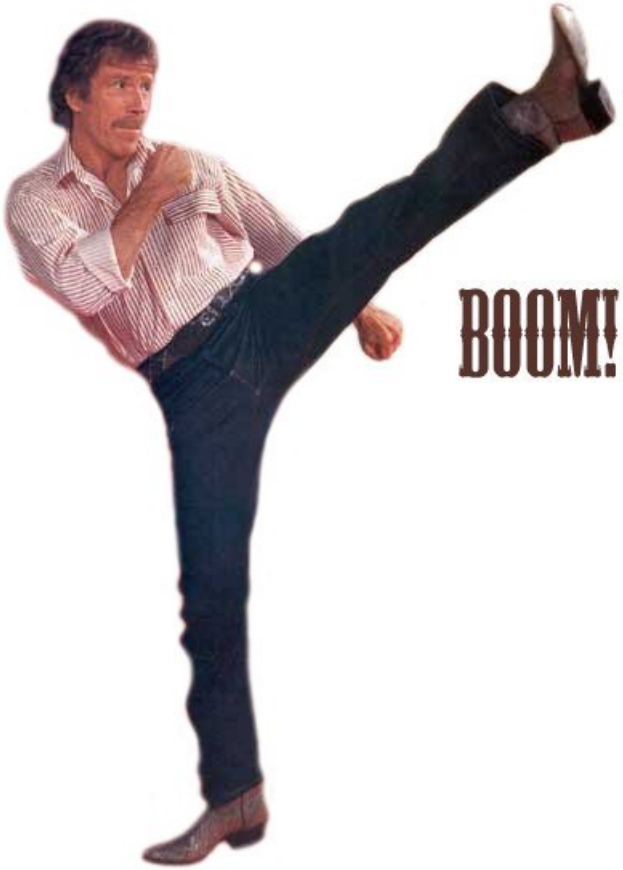
How does it respond?

What does "not in sync mean"?

What is "in sync" anyway?



What's a Bootloader?

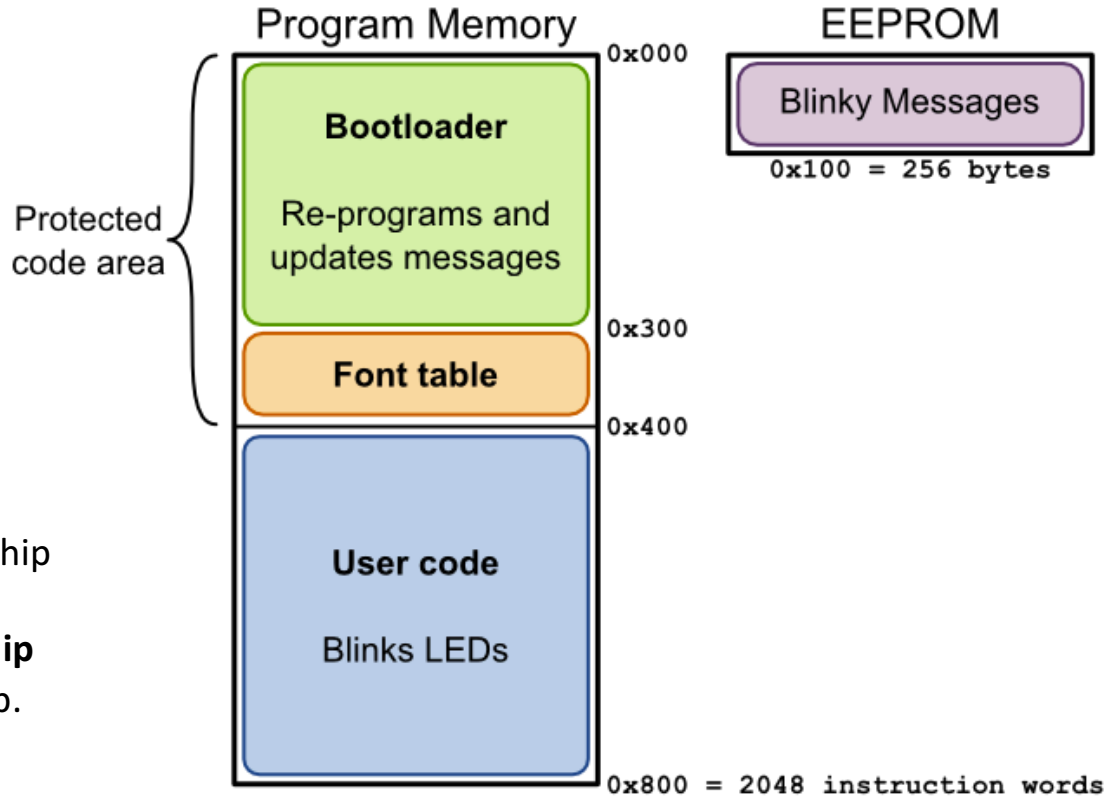


1. The bootloader is a little piece of code that allows you to program the flash memory of the Arduino's atmega328p via serial or USB instead of using an ICSP programmer.

2. After the avr compiler compiles everything down to an Intel hex file for downloading to the target board, it uses the avrdude command to program the Arduino via USB.

3. Most of the commands are from the stk500 module, except the signature reading is slightly different.

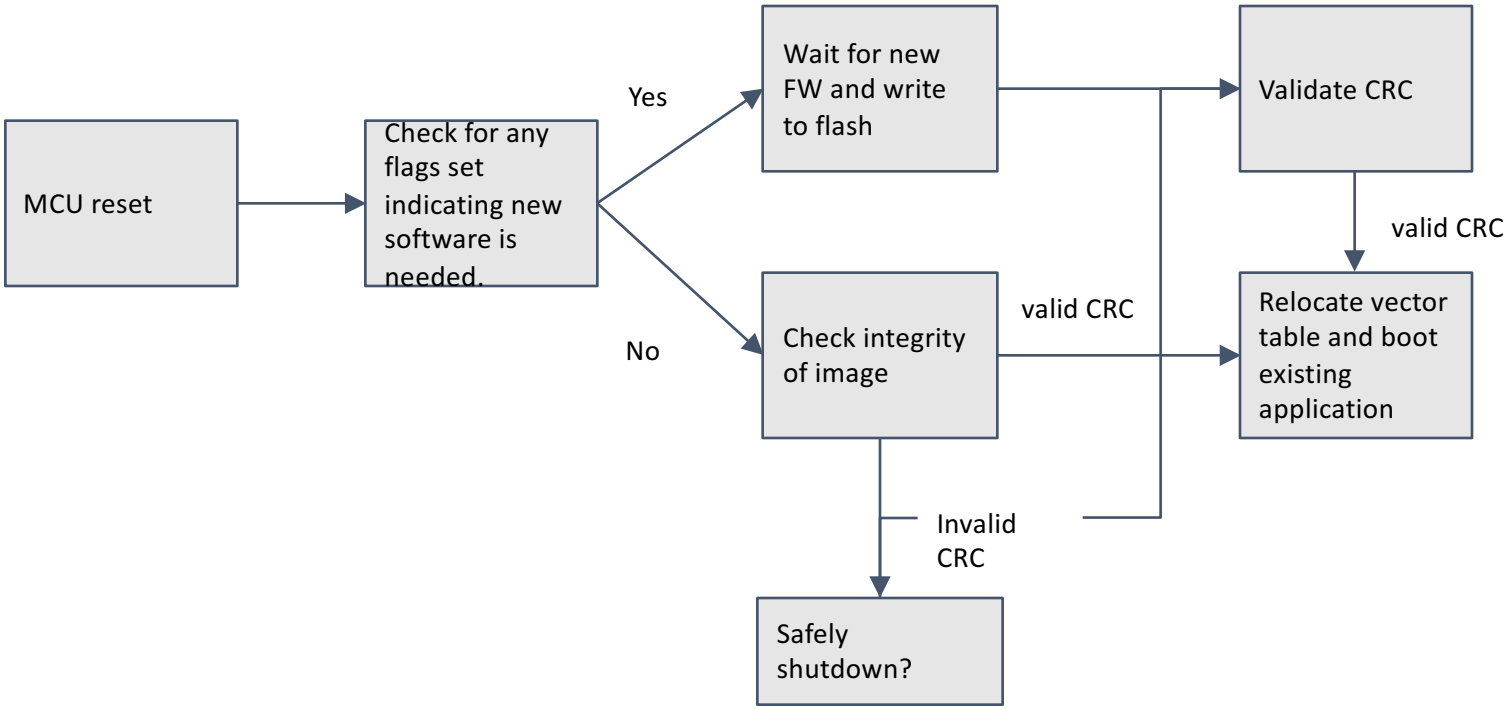
Bootloaders



Main tasks in the bootloading process:

1. Unlocking the bootloader section of the chip
2. Setting the the fuses on the chip
3. **Uploading the bootloader code to the chip**
4. Locking the bootloader section of the chip.

Bootloaders - Basic Flow



Bootloaders - Where to start?

The goals of a bootloader are to:

- Receive data from an external source.
- Place the data into specific memory regions.
- Validate CRC.
- Relocate vector table.
- Jump to user application.

4. Atmel STK500 protocol for avrDUDE

```
/* STK500 constants list, from AVRDUDE */
#define STK_OK              0x10
#define CRC_EOP             0x20  // 'SPACE
#define STK_GET_SYNC        0x30  // '0'
#define STK_GET_SIGN_ON     0x31  // '1'
#define STK_SET_PARAMETER   0x40  // '@'
#define STK_GET_PARAMETER   0x41  // 'A'
#define STK_SET_DEVICE      0x42  // 'B'
#define STK_SET_DEVICE_EXT  0x45  // 'E'
#define STK_ENTER_PROGMODE  0x50  // 'P'
#define STK_LEAVE_PROGMODE  0x51  // 'Q'
#define STK_CHIP_ERASE      0x52  // 'R'
#define STK_CHECK_AUTOINC   0x53  // 'S'
#define STK_LOAD_ADDRESS    0x55  // 'U'
#define STK_UNIVERSAL       0x56  // 'V'
#define STK_PROG_FLASH      0x60  // '`'
#define STK_PROG_DATA       0x61  // 'a'
#define STK_PROG_FUSE       0x62  // 'b'
#define STK_PROG_LOCK       0x63  // 'c'
#define STK_PROG_PAGE       0x64  // 'd'
#define STK_PROG_FUSE_EXT   0x65  // 'e'
#define STK_READ_FLASH      0x70  // 'p'
#define STK_READ_DATA       0x71  // 'q'
#define STK_READ_FUSE       0x72  // 'r'
#define STK_READ_LOCK       0x73  // 's'
#define STK_READ_PAGE       0x74  // 't'
#define STK_READ_SIGN       0x75  // 'u'
#define STK_READ_OSCCAL     0x76  // 'v'
#define STK_READ_FUSE_EXT   0x77  // 'w'
#define STK_READ_OSCCAL_EXT 0x78  // 'x'
```

Let's program the board
… and listen to the conversation on the serial bus

DTR and RTS are both toggled from High to low several times

Avrdude then sends the GET_SYNCH command, thows away whatever comes back and sends it again a couple of times

Avrdude/Arduino IDE:#30#20 STK_GET_SYNCH, Sync_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK
Avrdude/Arduino IDE: #30#20 STK_GET_SYNCH, Sync_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK
Avrdude/Arduino IDE: #30#20 STK_GET_SYNCH, Sync_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK

Get the version number of the bootloader (in this case it's 4.4) we could use this to branch to different variants of protocol.

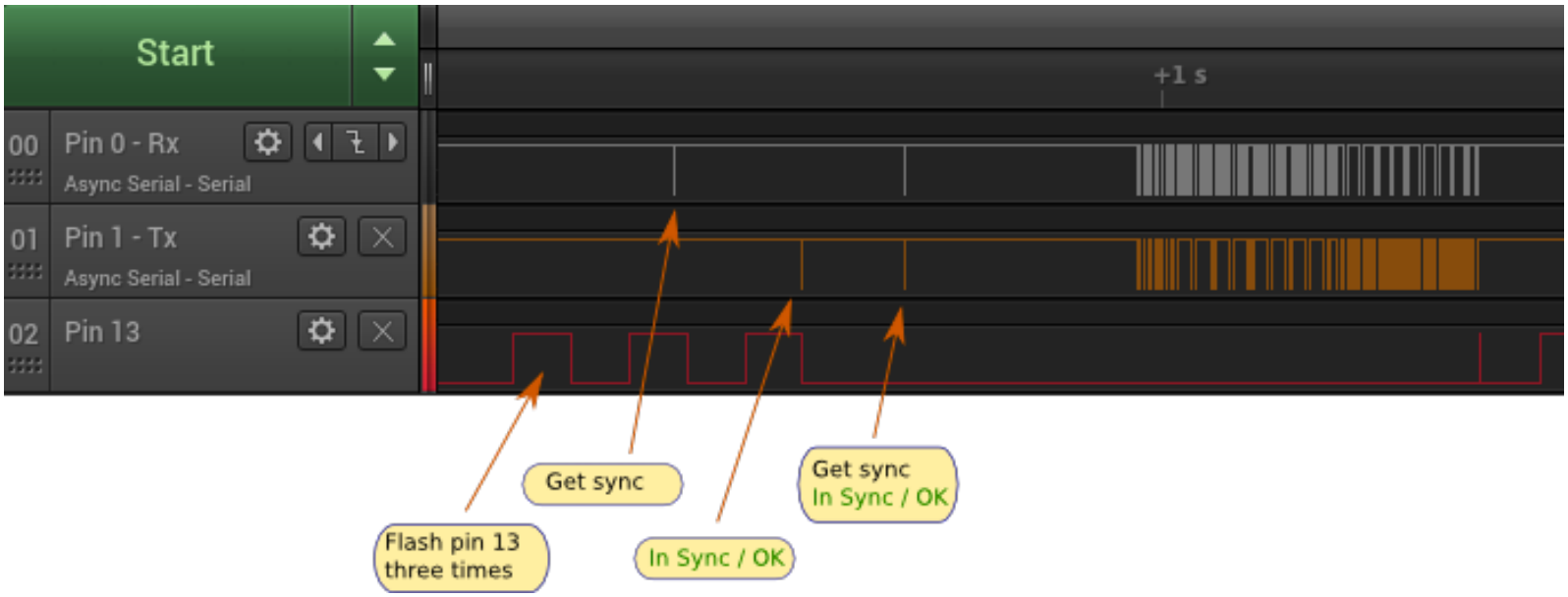
Avrdude/Arduino IDE: #41#81#20 STK_GET_PARAMETER, STK_SW_MAJOR, SYNC,CRC_EOP
Target/Arduino UNO: #14#04#10 STK_INSYNC, 0x04, STK_OK
Avrdude/Arduino IDE: #41#82#20 STK_GET_PARAMETER, STK_SW_MINOR, SYNC_CRC_EOP
Target/Arduino UNO: #14#04#10 STK_INSYNC, 0x04, STK_OK

Query: Get Sync? Reply: In Sync.
Query: Get parameter? (major version) Reply: version 4.
Query: Get parameter? (minor version) Reply: version 4.

If it is "not in sync" that means that avrdude did not get the "in sync" response. Possible reasons:

- Wrong baud rate used
- Wrong serial port selected in IDE
- Wrong board type selected in IDE
- No bootloader installed
- Wrong bootloader installed
- Board not configured to use the bootloader (in the fuses)
- Some device plugged into pins D0 and D1 on the Arduino, interfering with serial commutations
- The USB interface chip (ATmega16U2) not working properly
- Wrong clock for the board
- Wrong fuse settings on the Atmega328P (eg. "divide clock by 8")
- Board/chip damaged
- Faulty USB cable (some USB cables provide power only, not for data, eg. cheap cables for USB fans)

Let’s program the board
… and listen to the conversation on the serial bus



Top line (gray) is sent **to** the Arduino, middle line (orange) is sent **from** the Arduino.

Let's program the board
… and listen to the conversation on the serial bus

Enter Programming mode:
Avrdude/Arduino IDE: #50#20 STK_ENTER_PROGMODE, SYNC_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK
Avrdude/Arduino IDE: #75#20 STK_READ_SIGN, SYNC_CRC_EOP (read device signature)
Target/Arduino UNO: #14#1E#95#0F#10 STK_INSYNC, (three byte signature), STK_OK

Here's where we actually get started. This specifies the address in Flash where the following PROGRAM_PAGE data goes.
Avrdude/Arduino IDE: #55#00#00#20 STK_LOAD_ADDRESS, 0x0000, SYNC_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK
And then the actual data. Note that the data is in the same order as the bytes in the Intel Hex file
Avrdude/Arduino IDE:
#64#00#80#46#0C#94#61#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C
#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94
#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E
#00#0C#94#7E#00#0C#94#9A#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00
#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C
#94#7E#00#00#00#00#00#24#00#27#00#2A#00#00#00#00#00#25#00#28#00#2B
#00#00#00#00#00#20
STK_PROGRAM_PAGE, 0x0080 (page size), 'F'(flash memory), data bytes…,SYNC_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK

Interpretation….

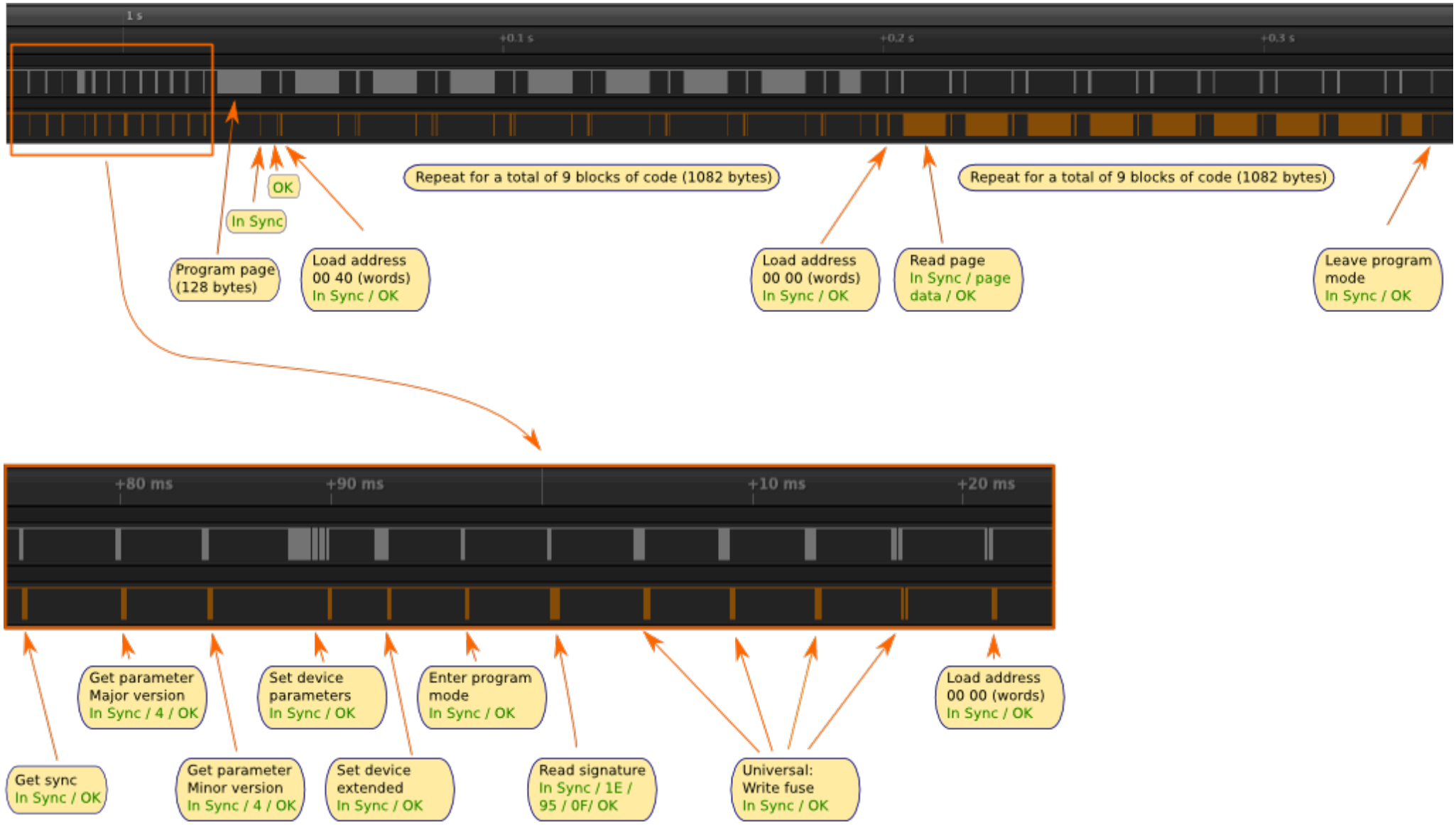
1. Enter program mode. Reply: In Sync/OK.

2. Read signature. Optiboot replies with 0x1E 0x95 0x0F **without actually reading the signature**.

3. Write fuses (four times). Optiboot **does not write the fuse** but just replies In Sync/OK.

4. Load address (initially 0x0000). The address is in words (ie. a word is two bytes). This sets the address for where the next page of data will be written.

5. Program page (up to 128 bytes are sent). Optiboot replies "In Sync" immediately. Then there is a pause of about 4 ms while it actually programs the page. Then it replies "OK".


1 s
+0.1 s
+0.2 s
+0.3 s
OK
In Sync
Program page
(128 bytes)
Load address
00 40 (words)
In Sync / OK
Repeat for a total of 9 blocks of code (1082 bytes)
Load address
00 00 (words)
In Sync / OK
Read page
In Sync / page
data / OK
Repeat for a total of 9 blocks of code (1082 bytes)
Leave program
mode
In Sync / OK
+80 ms
+90 ms
+10 ms
+20 ms
Get sync
In Sync / OK
Get parameter
Major version
In Sync / 4 / OK
Get parameter
Minor version
In Sync / 4 / OK
Set device
parameters
In Sync / OK
Set device
extended
In Sync / OK
Enter program
mode
In Sync / OK
Read signature
In Sync / 1E /
95 / 0F/ OK
Universal:
Write fuse
In Sync / OK
Load address
00 00 (words)
In Sync / OK

And then we repeat – set address, program page, etc.
Avrdude/Arduino IDE: #55#40#00#20 STK_LOAD_ADDRESS, 0x0040, SYNC_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK
A**vrdude/Arduino IDE:** STK_PROGRAM_PAGE, 0x0080 (page size), 'F'(flash memory), data bytes….,SYNC_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK
.
.
.
Then it reads several pages (probably to verify what it wrote)

Avrdude/Arduino IDE: #55#00#00#20 STK_LOAD_ADDRESS, 0x0000,SYNC_CRC_EOP
Target/Arduino UNO: #14#10 STK_INSYNC, STK_OK

Avrdude/Arduino IDE: #74#00#80#46#20 STK_READ_PAGE,0x0080 bytes, 'F' for flash, SYNC_CRC_EOP
Target/Arduino UNO #14 STK_INSYNC
Target/Arduino UNO #14#10 STK_INSYNC, STK_OK
#0C#94#61#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E
#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94
#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#9A#00#0C
#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00
#0C#94#7E#00#0C#94#7E#00#0C#94#7E#00#00#00#00#00#24#00#27#00#2A#00#00
#10 STK_OK

Interpretation….

1. Load address (now 0x0040). This is address 64 in decimal, ie. 128 bytes from the start of program memory.

2. Another page is written. This sequence continues until all the pages are written.

3. Load address (back to 0x0000). This is for verifying the write.

4. Read page (up to 128 bytes are read). This is for verifying. Note that even if the verify fails, the bad data is still written to the chip.

5. Leave programming mode.

Finishing up…

repeat several times at different addresses

.
.

And then leaves programming mode
Avrdude/Arduino IDE: #51#20 STK_LEAVE_PROGMODE, SYNC_CRC_EOP
Target/Arduino UNO #14#10 STK_INSYNC, STK_OK

Then we're done! Avrdude then toggles DTR/RTS to reset

5. Walkthrough the `optiboot` bootloader code

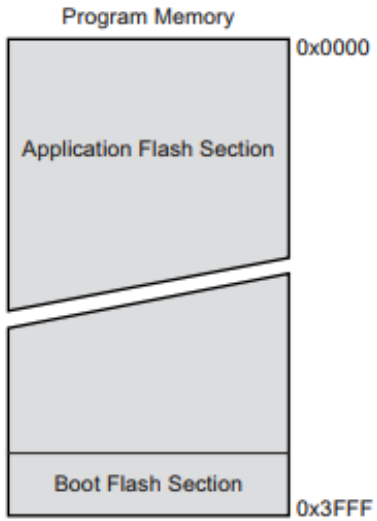
The Arduino Bootloader: Code Overview

- The Arduino bootloader can be broken down into two primary functions:
  - **Device Setup**: setting the stack pointer to the end of program memory (RAMEND), clearing the MCU status register (MCUSR), setting up the watchdog timer (more on this later).
  - **Command Loop**: waits for commands from the 16U2, such as getting the bootloader version (0xB1 and 0xB2), flashing a program to the program memory (STK_PROG_PAGE), and starting the user application (STK_ENTER_PROGMODE).

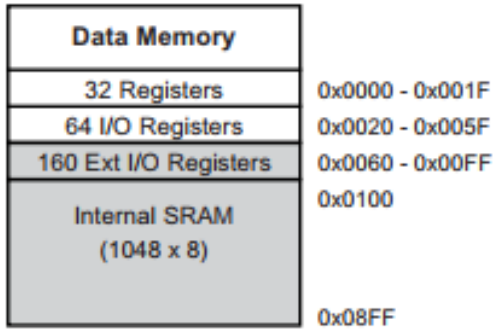
Refer to `optiboot_519.c` on Canvas for the rest of this section to make sense

Program and Data Memory

- The Arduino has two main types of memory:
  - **Flash (program) memory**: consists of *user application memory* (0x0000 – 0x3DFF) and *boot application memory (0x3DFF – 0x3E00)*
  - **Data memory**: consists of 32 *general-purpose registers* (0x0000 – 0x001F), 64 internal *I/O registers* (0x0020 – 0x005F), 96 *external I/O registers* (0x0060 – 0x00FF), and *internal SRAM* (stack and heap) (0x0100 – 0x08FF).



- There is a third type called **EEPROM** that can be used for persistent data storage (when the power to the MCU is turned off).
  - EEPROM stands for Electrically Erasable/Programmable Memory.



- Each of the three types has a separate address space.
  - This is called a **Harvard architecture**. Most MCUs have this type.
  - Distinct from a **Von Neumann architecture**, in which all memory types/segments share a common address space.

Setup: Setting the stack pointer

- Let's look at the code on the 328P.
- Most AVR chips that the bootloader supports (including the 328P) set the stack pointer to RAMEND (0x08FF) upon reset.
- Others (such as the ATmega8) don't; we must set it manually in software for these chips.

```
int main(void) {
  /* ... */
  #ifdef __AVR_ATmega8__
    SP=RAMEND;
  #endif
  /* ... */
```

Setup: Checking and clearing the status register

- The bootloader then saves and clears the status register (MCUSR).
  - We must save it first to detect whether we are starting the user application next.

```
uint8_t ch = MCUSR;
MCUSR = 0;
```

- If the watchdog timer reset the board, start the user application.
  - The timer runs out and resets the board after the 16U2 sends the STK_ENTER_PROGMODE command.
  - The EXTRF bit in MCUSR is set when an external reset (e.g. a push button reset) occurs, but not for a software-initiated (e.g. watchdog) reset.

```
if (!(ch & _BV(EXTRF))) appStart();
```

Setup: Setting up the UART

- The **UART** (Universal Asynchronous Receiver/Transmitter) is a device on the 328P that is used for communication with the 16U2.
  - We'll talk about UART in detail later in the course.

- The 16U2 sends bootloader commands over this channel, so we must configure it first.

```
UCSR0A = _BV(U2X0);                         // double speed mode
UCSR0B = _BV(RXEN0) | _BV(TXEN0);   // enable RX and TX
UCSR0C = _BV(UCSZ00) | _BV(UCSZ01); // frame and stop bits
UBRR0L = (uint8_t)( (F_CPU + BAUD_RATE * 4L) /
                    (BAUD_RATE * 8L) - 1 ); // set comm. rate
```

Setup: Setting up the watchdog timer

- The watchdog timer is used to reset the board and start the application.

- We start with a timeout of 1 second.
  - We will decrease this timeout value later to 16ms when we're ready to enter programming mode, but we set it now in case the 16U2 doesn't explicitly send the reset command, or the bootloader gets stuck.

```
watchdogConfig(WATCHDOG_1S);
```

Loop

- The bootloader enters a loop and obtains the next byte `ch` sent over the UART from the 16U2:

```
uint8_t ch = getch();
```

- Based on the byte sent, the 328P then performs some action, such as returning the firmware version, copying a program page into flash memory, or starting the user application.

```
uint8_t getch(void) {
    uint8_t ch;
    while(!(UCSR0A & _BV(RXC0)));
    if (!(UCSR0A & _BV(FE0))) {
        // No framing error; reset the watchdog.
        watchdogReset();
    }
    return ch;
}
```

This flag will be set when a framing error occurs, usually because of a bad baud rate.

Loop: Returning the hardware version

```
for (;;) {
  /* get character from UART */
  ch = getch();

  if(ch == STK_GET_PARAMETER) {
    unsigned char which = getch();
    verifySpace(); // check that the next char is a
                   // space; reset the board otherwise
    if (which == 0x82) {
      // Send optiboot version as "minor SW version"
      putch(OPTIBOOT_MINVER);
    } else if (which == 0x81) {
      putch(OPTIBOOT_MAJVER);
    } else {
      // GET PARAMETER returns a generic 0x03 reply
      // for other parameters to keep avrdude happy
      putch(0x03);
    }
    /* ... */
```

- This one's simple; it just returns the hardware version of optiboot (the bootloader name) over serial.
- The versions are defined as macros in the source code file (optiboot.c).

Loop: Flashing the firmware

- This one's more interesting. It consists of two parts: getting the address to write to and then writing the data page to that address.
- Let's start with getting the address:

```
if(ch == STK_LOAD_ADDRESS) {
  uint16_t newAddress;

  // Addresses are 16 bits wide, so we must get them in 1-byte
  // increments and then shift to form the whole address.
  newAddress = getch();
  newAddress = (newAddress & 0xff) | (getch() << 8);
  newAddress += newAddress; // Convert from 16-bit (word) address to byte address
  address = newAddress;     // note that address is a variable in main()
  verifySpace();
}
```

Loop: Flashing the firmware

- This one's more interesting. It consists of two parts: getting the address to write to and then writing the data page to that address.
- Let's start with getting the address:

```
if(ch == STK_LOAD_ADDRESS) {
  uint16_t newAddress;

  // Addresses are 16 bits wide, so we must get them in 1-byte
  // increments and then shift to form the whole address.
  newAddress = getch();
  newAddress = (newAddress & 0xff) | (getch() << 8);
  newAddress += newAddress; // Convert from 16-bit (word) address to byte address
  address = newAddress;     // note that address is a variable in main()
  verifySpace();
}
```

- **Question**: Why do we multiply `newAddress` by 2 here?

Loop: Flashing the firmware

- Now we'll actually flash the firmware by copying data:

```
if(ch == STK_PROG_PAGE) {
      uint8_t *bufPtr; uint16_t addrPtr;

      getch(); length = getch(); getch();  // get length of page

      // If we are in RWW section, immediately start page erase
      if (address < NRWWSTART) __boot_page_erase_short((uint16_t)(void*)address);

      // While that is going on, read in page contents
      bufPtr = buff; do { *bufPtr++ = getch(); } while (--length);

      // If we are in NRWW section, page erase has to be delayed until now.
      // Todo: Take RAMPZ into account
      if (address >= NRWWSTART) __boot_page_erase_short((uint16_t)(void*)address);

      verifySpace(); // Read command terminator, start reply

      /* continued on next slide */
```

Loop: Flashing the firmware

```
  // If only a partial page is to be programmed, the erase might not be complete.
  boot_spm_busy_wait();

  // Copy buffer into programming buffer
  bufPtr = buff; addrPtr = (uint16_t)(void*)address; ch = SPM_PAGESIZE / 2;
  do {
    uint16_t a;   a = *bufPtr++; a |= (*bufPtr++) << 8;
    __boot_page_fill_short((uint16_t)(void*)addrPtr,a);
    addrPtr += 2;
  } while (--ch);

  // Write from programming buffer
  __boot_page_write_short((uint16_t)(void*)address);
  boot_spm_busy_wait();
}
```

Loop: Resetting the board

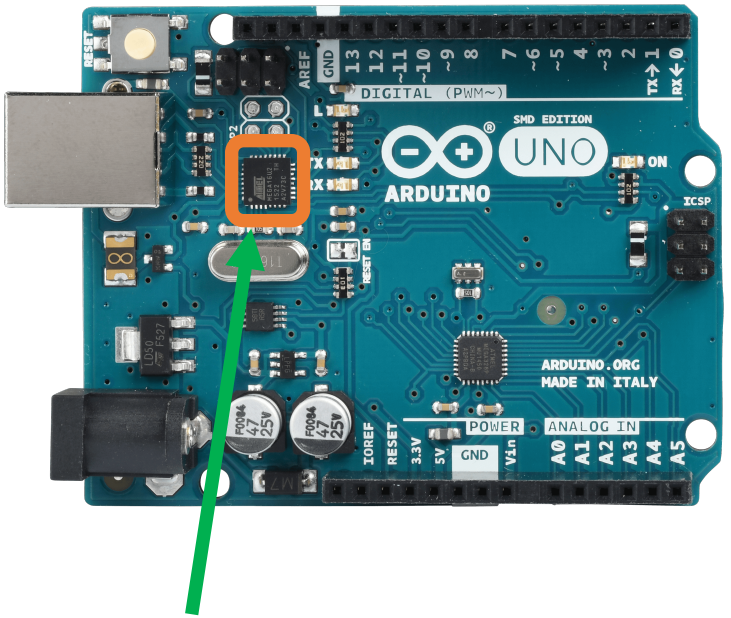
- avrdude will send the “Q” character when it’s done flashing and is ready to reset the board.
- The reset is handled by the verifySpace() function.
  - Before the loop begins, the MCU will check MCUSR (the status register) to see if a watchdog timeout occurred and will reset the board if one did occur.

```
} else if (ch == 'Q') {
  // Adaboot no-wait mod
  watchdogConfig(WATCHDOG_16MS);
  verifySpace();
```

```
void verifySpace() {
  // no more characters; shorten WD timeout
  if (getch() != CRC_EOP) {
    // watchdog will reset the board
    // wait until it does
    watchdogConfig(WATCHDOG_16MS); while (1);
  }
  putch(STK_INSYNC);
}
```

Programming the bootloader for the ATmega16U2

- Up to this point, we've focused on the ATmega328P (the chip that runs user programs).
  - We need a way to transmit data from the PC to the 328P, however.
- The Arduino also has another chip—the 16U2—that is used to handle serial communication between a PC and the 328P.
- This 16U2 chip acts as a bridge between the USB protocol (used by a PC) and UART protocol (used by the 328P).



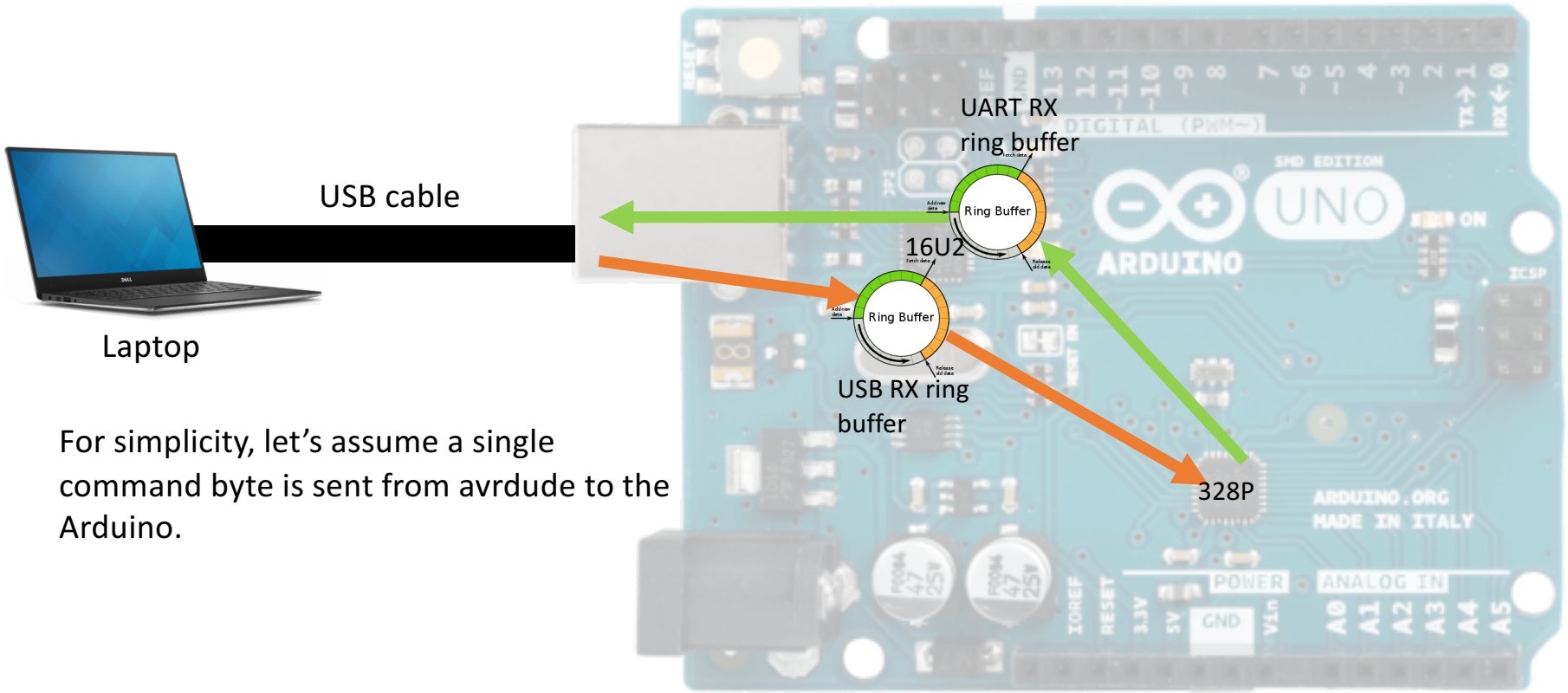
The 16U2

16U2: How does it work?

- The 16U2 is pre-loaded with firmware that stores incoming data from the PC and 328P before sending it to the receiving device.
  - The data is stored in two ring buffers: one for the USB-to-16U2 link, and the other for the 16U2-to-328P link.
  - Once the 16U2 is ready to send data (to either the 328P or the PC), it sends all of the data in the respective buffer at once, clearing it in the process.

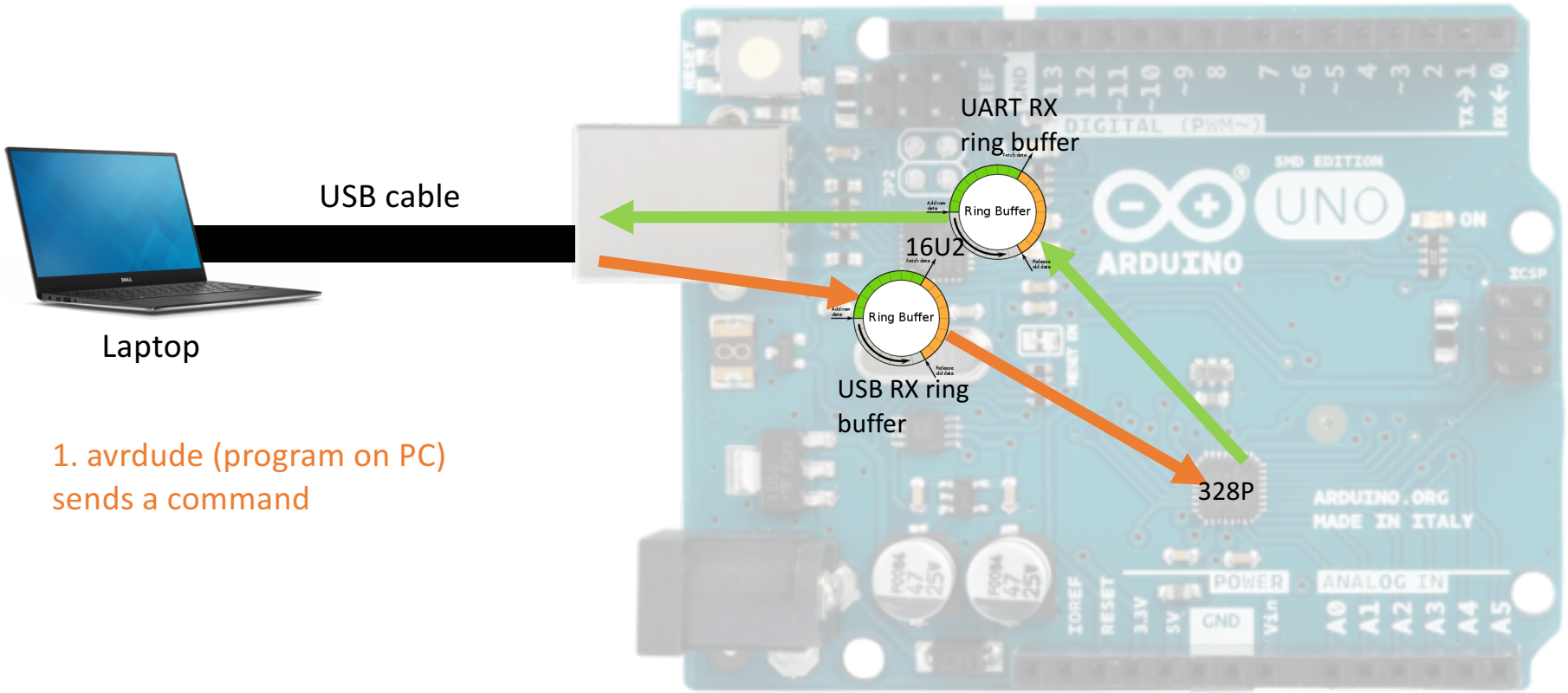
```
/** Circular buffer to hold data from the host before it is sent to the device via the serial port. */
RingBuff_t USBtoUSART_Buffer;

/** Circular buffer to hold data from the serial port before it is sent to the host. */
RingBuff_t USARTtoUSB_Buffer;
```

Overview of Serial Communication



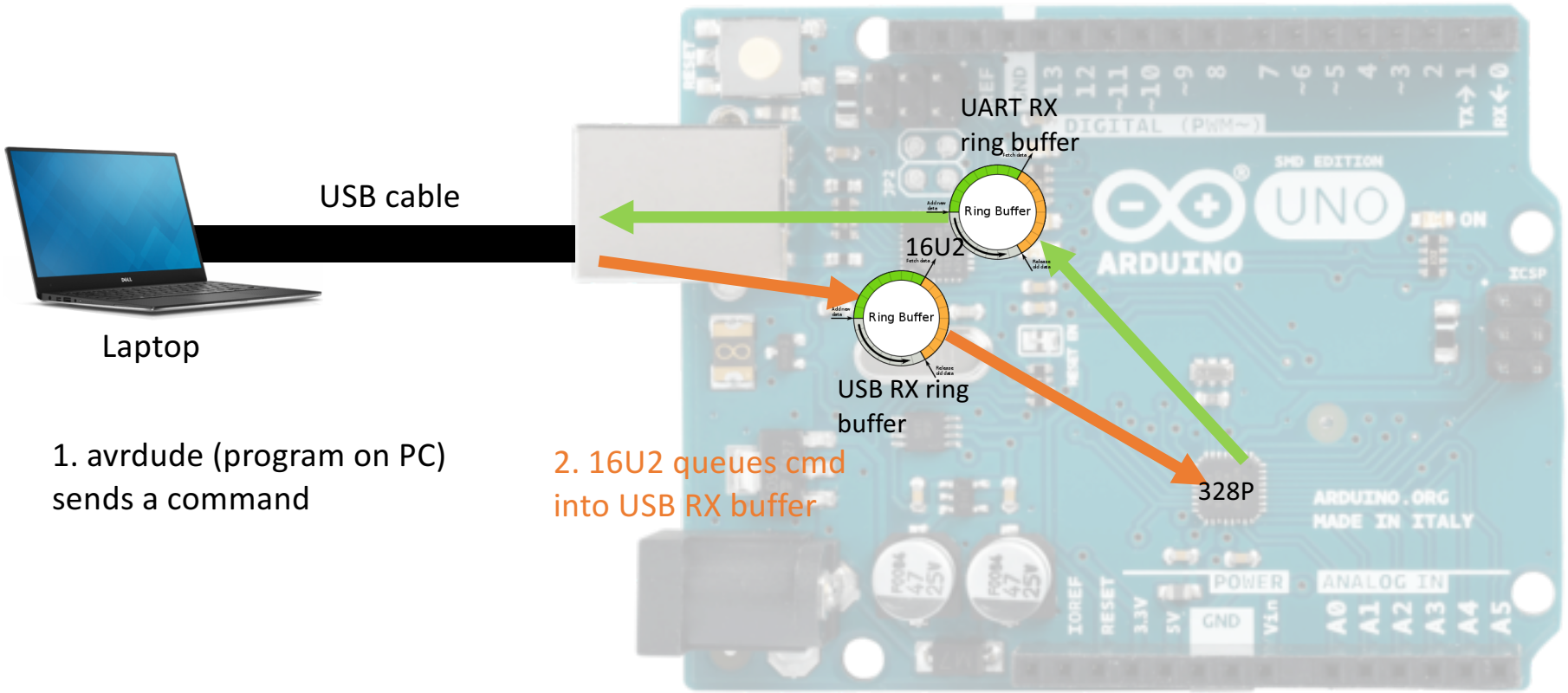
For simplicity, let's assume a single command byte is sent from avrdude to the Arduino.

Overview of Serial Communication

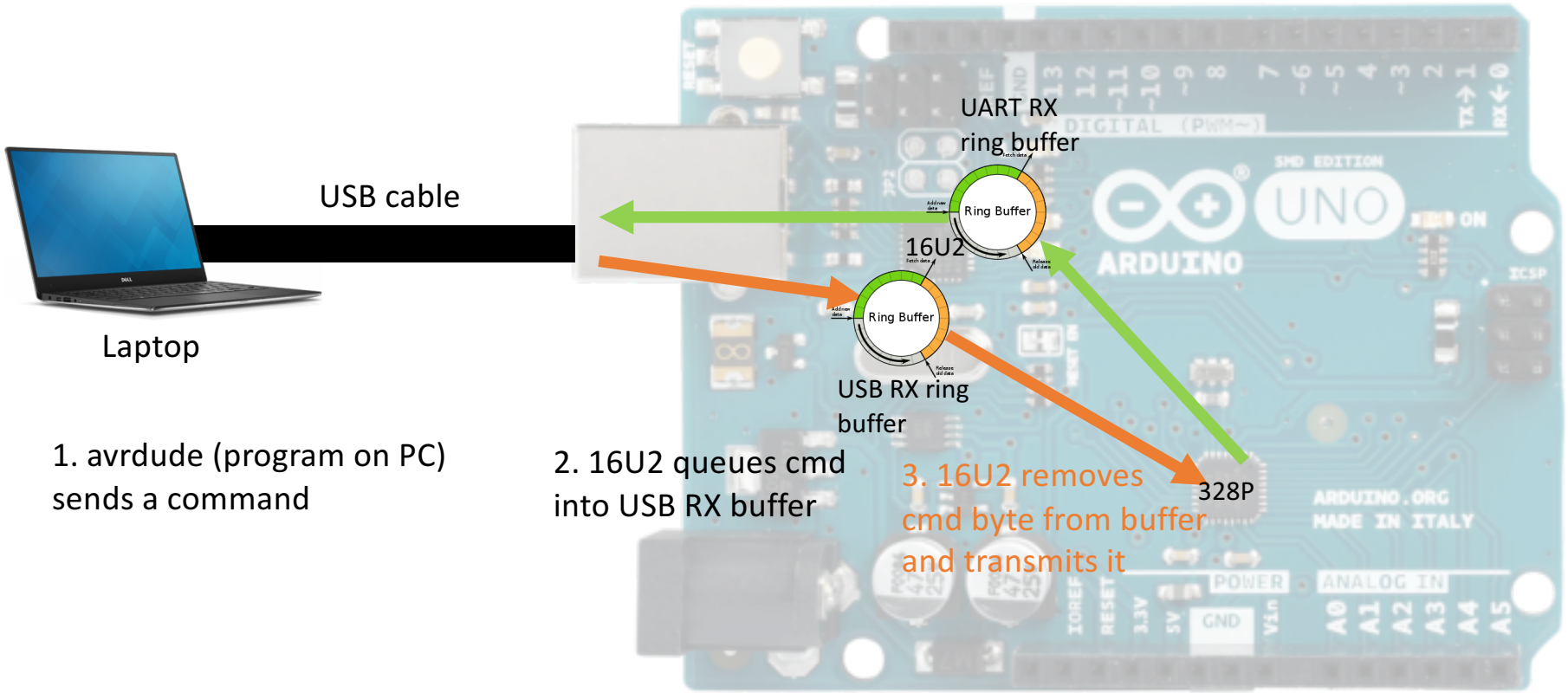


1. avrdude (program on PC) sends a command

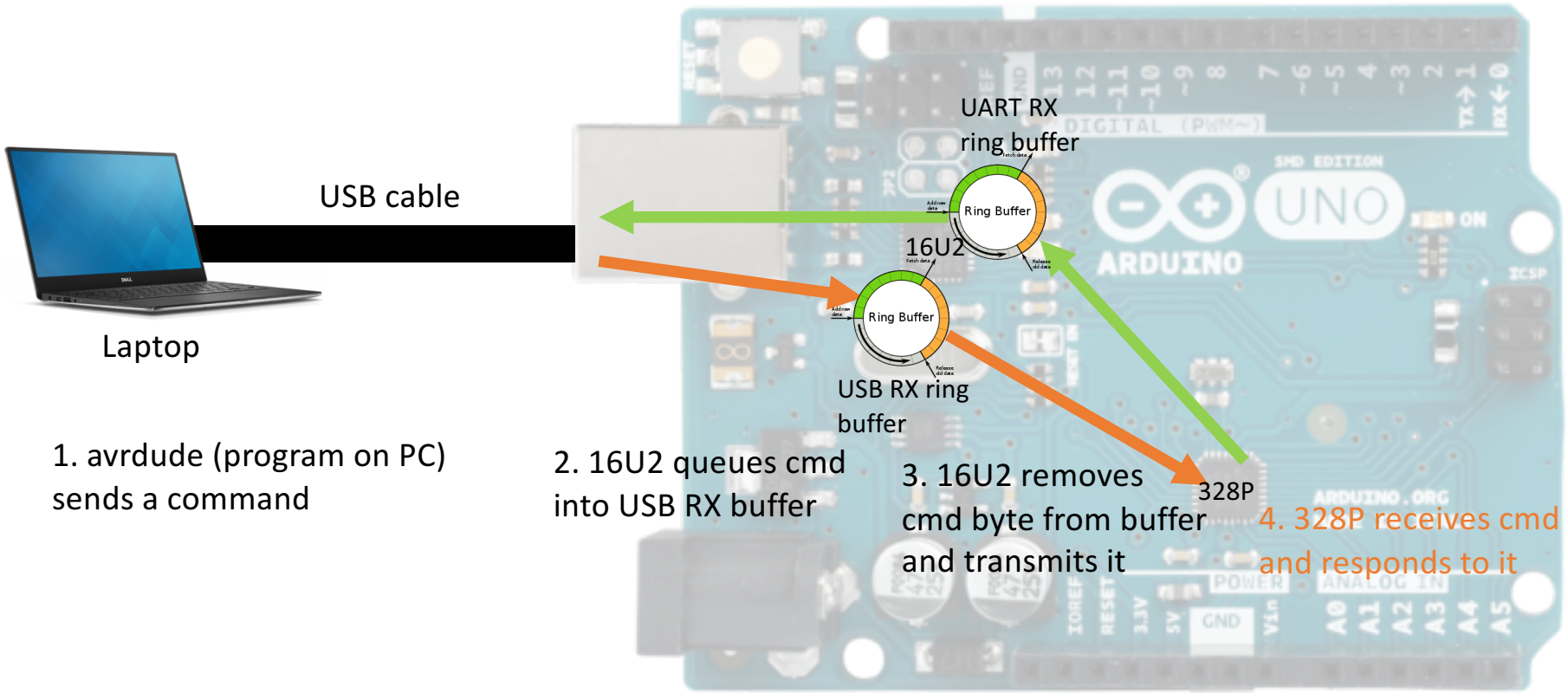
Overview of Serial Communication



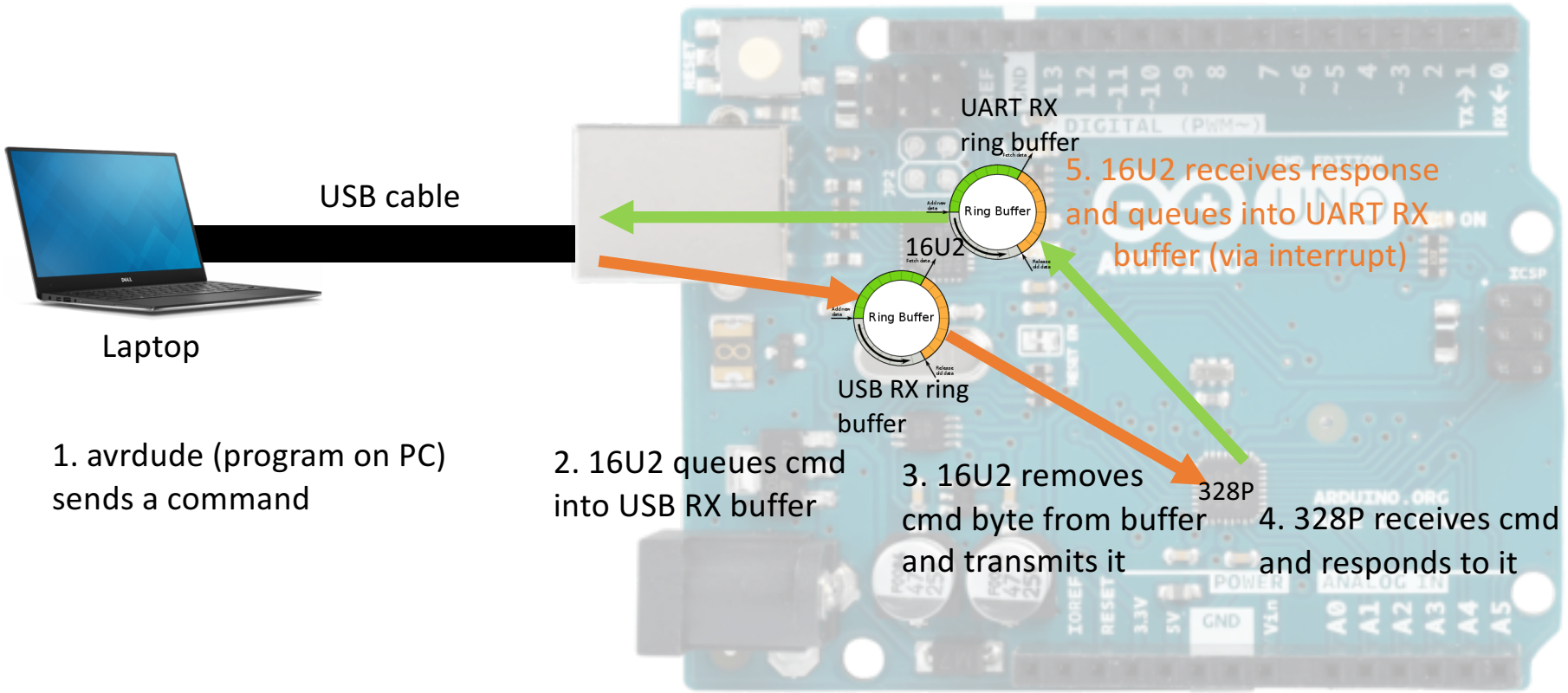
Overview of Serial Communication



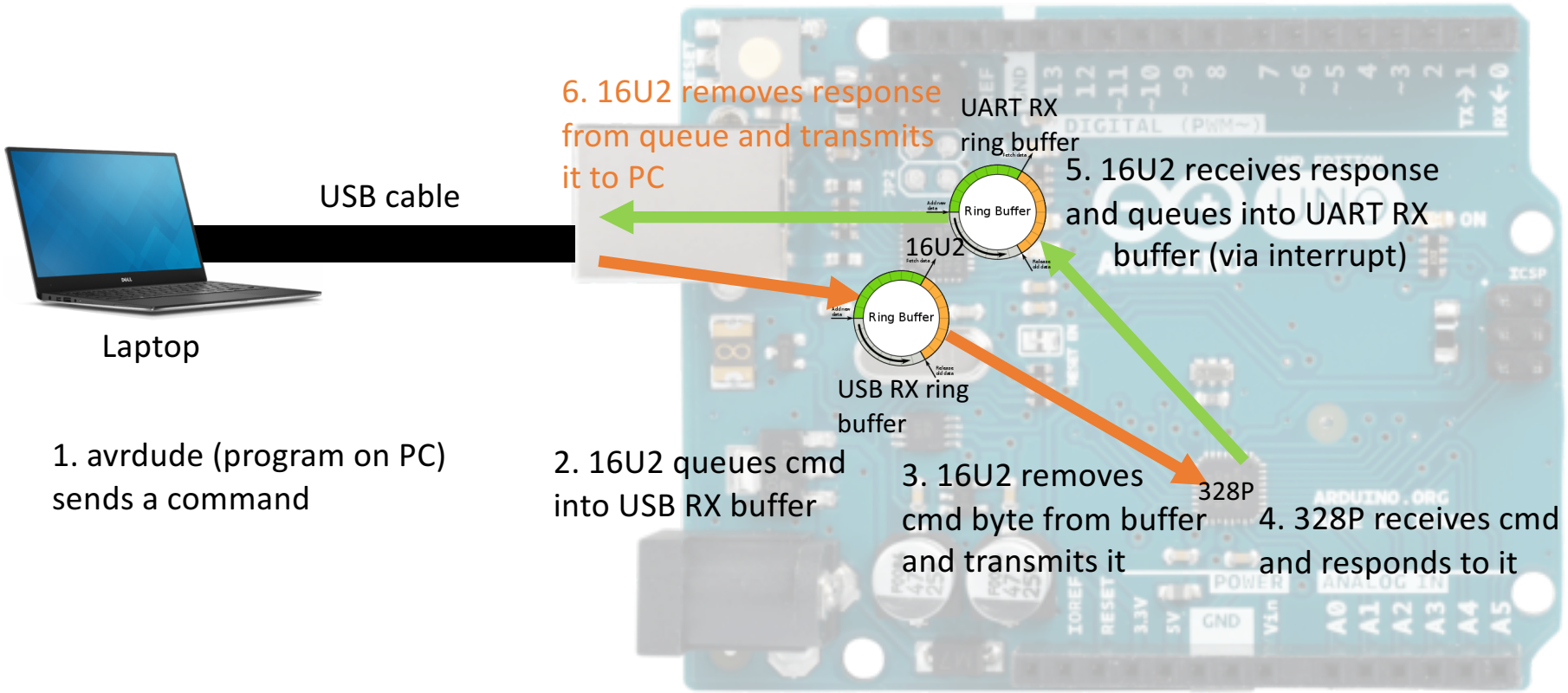
Overview of Serial Communication



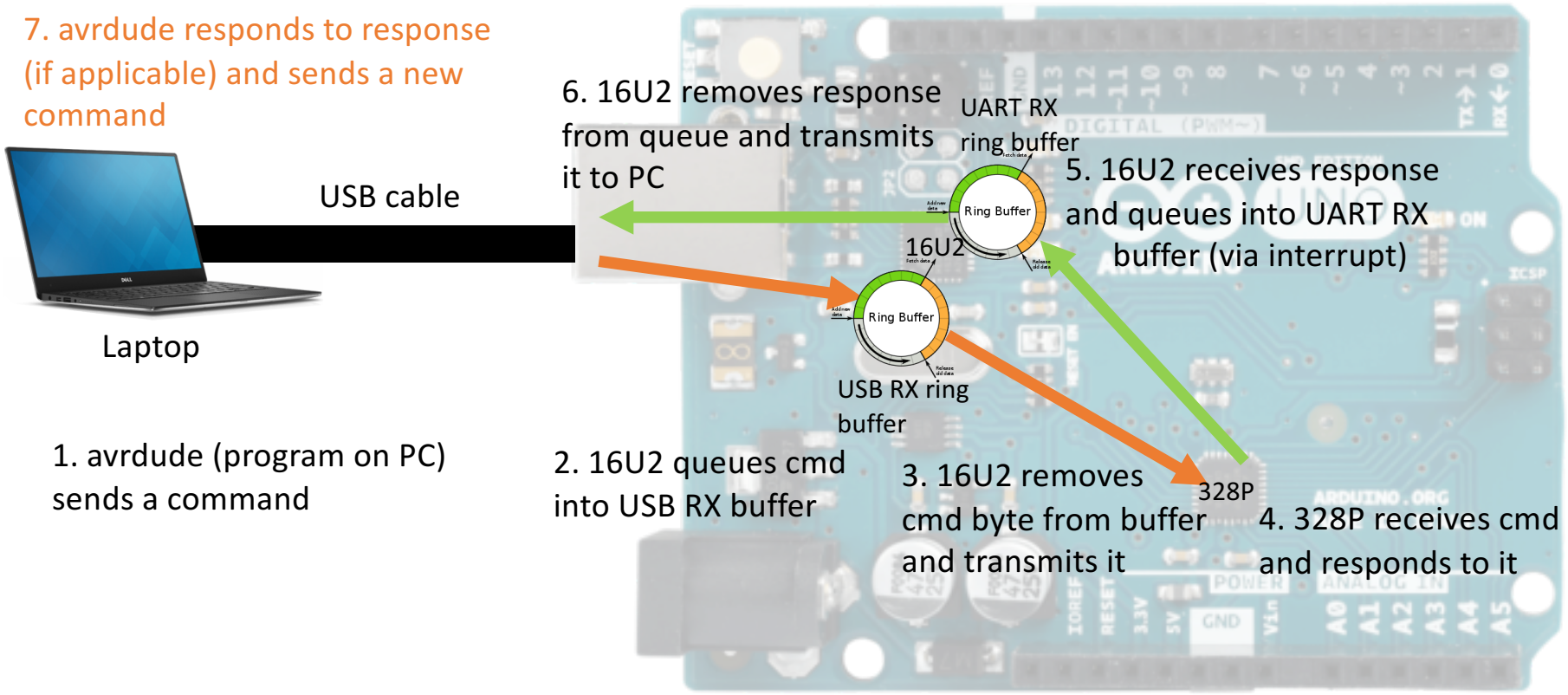
Overview of Serial Communication



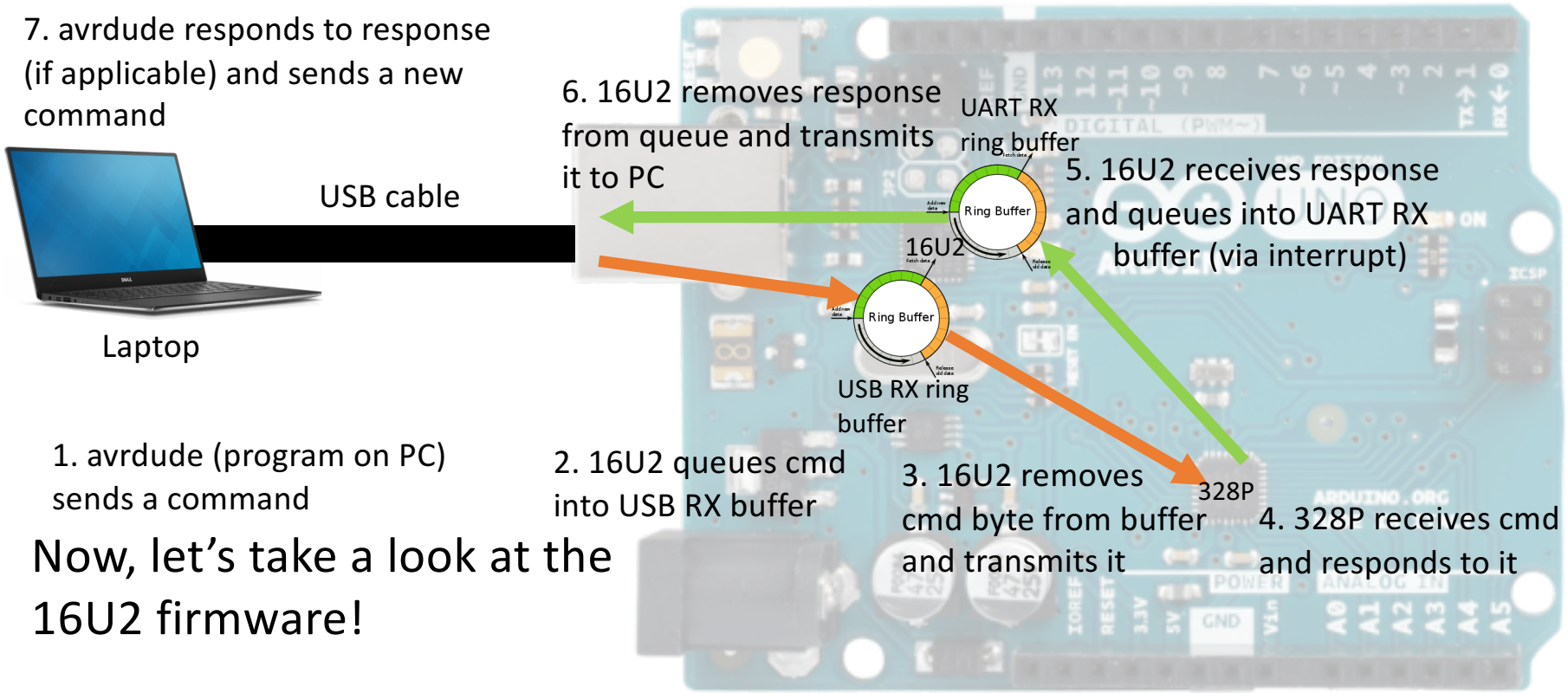
Overview of Serial Communication



Overview of Serial Communication



Overview of Serial Communication



Now, let's take a look at the 16U2 firmware!

16U2: Receiving data from the 328P

- The 16U2 firmware uses *interrupts* to receive data sent from the 328P that will later be passed on to the PC.
  - An **interrupt** is a software routine that runs whenever a specified hardware event, such as a data packet arrival or timer overflow, occurs.
  - You'll get a lot of exposure to interrupts in the next few labs after Lab 1.
- In this case, the interrupt retrieves the byte sent from the 16U2 and places it into the UART-to-USB ring buffer.

```
/** ISR to manage the reception of data from the serial port, placing received bytes into a circular buffer
 *  for later transmission to the host.
 */
ISR(USART1_RX_vect, ISR_BLOCK)
{
    uint8_t ReceivedByte = UDR1;

    if (USB_DeviceState == DEVICE_STATE_Configured)
      RingBuffer_Insert(&USARTtoUSB_Buffer, ReceivedByte);
}
```

16U2: Receiving data from the PC

- Unlike for UART, the 16U2 doesn't have an interrupt defined for receiving from a USB source.
- We must instead **poll** the device in the main loop to see if the PC sent the 16U2 data.

```
for (;;)
{
    /* Only try to read in bytes from the CDC interface if the transmit buffer is not full */
    if (!(RingBuffer_IsFull(&USBtoUSART_Buffer)))
    {
        int16_t ReceivedByte = CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);

        /* Read bytes from the USB OUT endpoint into the USART transmit buffer */
        if (!(ReceivedByte < 0))
          RingBuffer_Insert(&USBtoUSART_Buffer, ReceivedByte);
    }
```

16U2: Sending the buffered data

- The 16U2 sends data buffered from the 328P over USB if the UART-to-USB buffer is almost full or if the buffer flush timer (TCCR0B) has expired. This is done in the main loop.
  - Don’t worry about not knowing what TCCR0B is; it is a timer built into the chip, and you’ll find out soon in the next lab anyway ☺

```
/* Check if the UART receive buffer flush timer has expired or the buffer is nearly full */
RingBuff_Count_t BufferCount = RingBuffer_GetCount(&USARTtoUSB_Buffer);
if ((TIFR0 & (1 << TOV0)) || (BufferCount > BUFFER_NEARLY_FULL))
{
    TIFR0 |= (1 << TOV0);

    if (USARTtoUSB_Buffer.Count) {
        LEDs_TurnOnLEDs(LEDMASK_TX);
        PulseMSRemaining.TxLEDPulse = TX_RX_LED_PULSE_MS;
    }

    /* Read bytes from the USART receive buffer into the USB IN endpoint */
    while (BufferCount--)
      CDC_Device_SendByte(&VirtualSerial_CDC_Interface, RingBuffer_Remove(&USARTtoUSB_Buffer));
```

16U2: Sending the buffered data

- There's also another block in the main loop for sending data from the USB-to-UART buffer over UART:

```
/* Load the next byte from the USART transmit buffer into the USART */
if (!(RingBuffer_IsEmpty(&USBtoUSART_Buffer))) {
  Serial_TxByte(RingBuffer_Remove(&USBtoUSART_Buffer));

    LEDs_TurnOnLEDs(LEDMASK_RX);
    PulseMSRemaining.RxLEDPulse = TX_RX_LED_PULSE_MS;
}
```