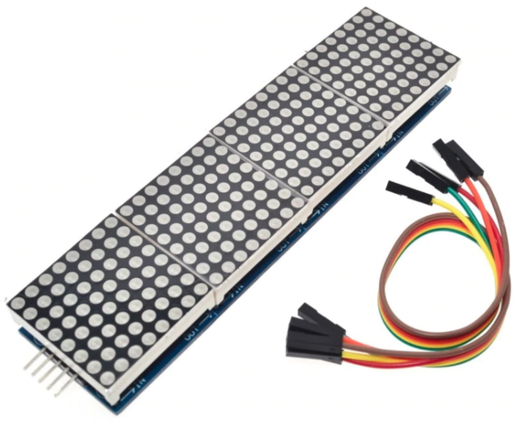
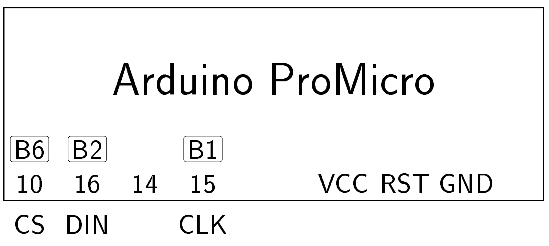
# TIME

**1. Program.** Display time from USB using MAX7219 module.





```
⟨ Header files 74 ⟩
⟨ Type definitions 47 ⟩
⟨ Global variables 8 ⟩
⟨ Functions 6 ⟩
⟨ Create ISR for connecting to USB host 24 ⟩
void main(void)
{
  ⟨ Connect to USB host (must be called first; sei is called here) 27 ⟩
  ⟨ Initialize display 2 ⟩
  uint8_t gotcha = 0;    /* used to protect ourselves from inadvertently introducing long-running
      code: time of execution of one loop iteration must not exceed interval between data arrivals */
  while (1) {
    ⟨ If there is a request on EP0, handle it 21 ⟩
    UENUM = EP2;
    if (UEINTX & 1 ≪ RXOUTI) {
      UEINTX &= ∼(1 ≪ RXOUTI);
      char time[9];
      int rx_counter = UEBCLX;
      while (rx_counter −−) time[7 − rx_counter] = UEDATX;
      UEINTX &= ∼(1 ≪ FIFOCON);
      time[8] = '\0';
      ⟨ Set brightness depending on time of day 20 ⟩
      if (gotcha) {
        display_write4(#0C, #01);    /* to be sure (it may be disabled in ⟨ Set brightness depending on
            time of day 20 ⟩, possibly via change-file) */
        strcpy(time, "99:99:99");    /* indicate failure */
        }
        time[5] = '\0';
        ⟨ Show time 3 ⟩
        if (gotcha) break;      /* freeze */
        gotcha = 1;      /* activate */
      }
      else gotcha = 0;      /* deactivate */
    }
  }
```

**2.** Initialization of all registers must be done, because they may contain garbage. First make sure that test mode is disabled, because it overrides all registers. Next, make sure that display is disabled, because there may be random lighting LEDs on it. Then set decode mode to properly clear all LEDs, and clear them. Finally, configure the rest registers and enable the display.

MAX7219 is a shift register. SPI here is used as a way to push bytes to MAX7219 (data + clock).

For simplicity (not to use timer), we use latch duration of 1us (min. is 50ns—$t_{CSW}$ in datasheet).

Note, that segments are connected as this: clock and latch are in parallel, DIN goes through each segment to DOUT and then to DIN of next segment in the chain.

⟨ Initialize display 2 ⟩ ≡

```
  PORTB |= 1 ≪ PB0;      /* on pro-micro led is inverted */
  DDRB |= 1 ≪ PB0;      /* disable SPI slave mode (SS port) */
  DDRB |= 1 ≪ PB1;      /* clock */
  DDRB |= 1 ≪ PB2;      /* data */
  DDRB |= 1 ≪ PB6;      /* latch */
  SPCR |= 1 ≪ MSTR | 1 ≪ SPR1 | 1 ≪ SPE;      /* SPR1 means 250 kHz FIXME: does native wire work
      without SPR1? does long wire work without SPR1? */
  display_write4(#0F, #00);
  display_write4(#0C, #00);
  display_write4(#09, #00);
  display_write4(#01, #00);
  display_write4(#02, #00);
  display_write4(#03, #00);
  display_write4(#04, #00);
  display_write4(#05, #00);
  display_write4(#06, #00);
  display_write4(#07, #00);
  display_write4(#08, #00);
  display_write4(#0A, #0F);
  display_write4(#0B, #07);
  display_write4(#0C, #01);
```

This code is used in section 1.

**3.** Buffer is used because in each device addresses are assigned to rows.

```
#define NUM_DEVICES 4
```

⟨ Show *time* 3 ⟩ ≡

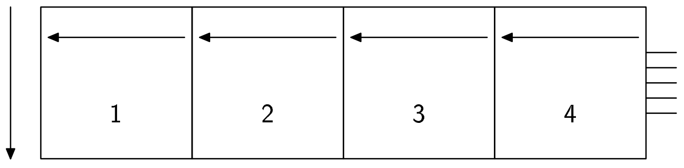
```
  uint8_t buffer[8][NUM_DEVICES * 8];
  ⟨ Fill buffer 4 ⟩
  ⟨ Display buffer 5 ⟩
```

This code is used in section 1.

**4.**
```
#define app(c)     /* append specified character to buffer */
        for (uint8_t i = 0; i < sizeof chr_##c/8; i++) buffer[row][col++] =
            pgm_read_byte(&chr_##c[row][i])
```

⟨ Fill buffer 4 ⟩ ≡
```
  for (uint8_t row = 0; row < 8; row++) {
    uint8_t col = 0;
    buffer[row][col++] = #00;
    for (char *c = time; *c ≠ '\0'; c++) {
      switch (*c) {
      case '0': app(0); break;
      case '1': app(1); break;
      case '2': app(2); break;
      case '3': app(3); break;
      case '4': app(4); break;
      case '5': app(5); break;
      case '6': app(6); break;
      case '7': app(7); break;
      case '8': app(8); break;
      case '9': app(9); break;
      case ':': app(colon);
      }
      buffer[row][col++] = #00;
    }
  }
```
This code is used in section 3.

**5.** Rows are output from right to left, from top to bottom. Left device is set first, right device is set last.



⟨ Display buffer 5 ⟩ ≡
```
  for (uint8_t row = 0; row < 8; row++) {
    uint8_t data;
    for (uint8_t n = 0; n < NUM_DEVICES; n++) {
      data = #00;
      for (uint8_t i = 0; i < 8; i++)
        if (buffer[row][n * 8 + i]) data |= 1 ≪ 7 − i;
      display_push(row + 1, data);
    }
    PORTB |= 1 ≪ PB6; _delay_us(1); PORTB &= ∼(1 ≪ PB6);     /* latch */
  }
```
This code is used in section 3.

**6.** ⟨ Functions 6 ⟩ ≡

```
void display_push(uint8_t address, uint8_t data)
{
  SPDR = address;
  while (~SPSR & 1 ≪ SPIF) ;
  SPDR = data;
  while (~SPSR & 1 ≪ SPIF) ;
}
```

See also section 7.

This code is used in section 1.

**7.** ⟨ Functions 6 ⟩ +≡

```
void display_write4(uint8_t address, uint8_t data)
{
  display_push(address, data);
  display_push(address, data);
  display_push(address, data);
  display_push(address, data);
  PORTB |= 1 ≪ PB6; _delay_us(1); PORTB &= ~(1 ≪ PB6);      /* latch */
}
```

**8.** ⟨ Global variables 8 ⟩ ≡

```
⟨ Character images 9 ⟩
```

See also sections 23, 33, 45, 48, 50, 68, 70, 71, and 72.

This code is used in section 1.

**9.** ⟨ Character images 9 ⟩ ≡

```
const uint8_t chr_0[8][5] PROGMEM = {
  {0, 1, 1, 1, 0},
  {1, 0, 0, 0, 1},
  {1, 0, 0, 0, 1},
  {1, 0, 0, 0, 1},
  {1, 0, 0, 0, 1},
  {1, 0, 0, 0, 1},
  {0, 1, 1, 1, 0},
  {0, 0, 0, 0, 0}
};
```

See also sections 10, 11, 12, 13, 14, 15, 16, 17, 18, and 19.

This code is used in section 8.

**10.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_1[8][5] PROGMEM = {
  {0, 0, 1, 0, 0},
  {0, 1, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 1, 1, 1, 0},
  {0, 0, 0, 0, 0}
};
```

**11.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_2[8][5] PROGMEM = {
    {0, 1, 1, 1, 0},
    {1, 0, 0, 0, 1},
    {0, 0, 0, 0, 1},
    {0, 0, 0, 1, 0},
    {0, 0, 1, 0, 0},
    {0, 1, 0, 0, 0},
    {1, 1, 1, 1, 1},
    {0, 0, 0, 0, 0}
};
```

**12.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_3[8][5] PROGMEM = {
    {1, 1, 1, 1, 1},
    {0, 0, 0, 1, 0},
    {0, 0, 1, 0, 0},
    {0, 0, 0, 1, 0},
    {0, 0, 0, 0, 1},
    {1, 0, 0, 0, 1},
    {0, 1, 1, 1, 0},
    {0, 0, 0, 0, 0}
};
```

**13.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_4[8][5] PROGMEM = {
    {0, 0, 0, 1, 0},
    {0, 0, 1, 1, 0},
    {0, 1, 0, 1, 0},
    {1, 0, 0, 1, 0},
    {1, 1, 1, 1, 1},
    {0, 0, 0, 1, 0},
    {0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0}
};
```

**14.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_5[8][5] PROGMEM = {
    {1, 1, 1, 1, 1},
    {1, 0, 0, 0, 0},
    {1, 1, 1, 1, 0},
    {0, 0, 0, 0, 1},
    {0, 0, 0, 0, 1},
    {1, 0, 0, 0, 1},
    {0, 1, 1, 1, 0},
    {0, 0, 0, 0, 0}
};
```

**15.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_6[8][5] PROGMEM = {
   {0,0,1,1,0},
   {0,1,0,0,0},
   {1,0,0,0,0},
   {1,1,1,1,0},
   {1,0,0,0,1},
   {1,0,0,0,1},
   {0,1,1,1,0},
   {0,0,0,0,0}
};
```

**16.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_7[8][5] PROGMEM = {
   {1,1,1,1,1},
   {1,0,0,0,1},
   {0,0,0,1,0},
   {0,0,1,0,0},
   {0,1,0,0,0},
   {0,1,0,0,0},
   {0,1,0,0,0},
   {0,0,0,0,0}
};
```

**17.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_8[8][5] PROGMEM = {
   {0,1,1,1,0},
   {1,0,0,0,1},
   {1,0,0,0,1},
   {0,1,1,1,0},
   {1,0,0,0,1},
   {1,0,0,0,1},
   {0,1,1,1,0},
   {0,0,0,0,0}
};
```

**18.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_9[8][5] PROGMEM = {
   {0,1,1,1,0},
   {1,0,0,0,1},
   {1,0,0,0,1},
   {0,1,1,1,1},
   {0,0,0,0,1},
   {0,0,0,1,0},
   {0,1,1,0,0},
   {0,0,0,0,0}
};
```

**19.** ⟨ Character images 9 ⟩ +≡

```
const uint8_t chr_colon[8][6] $\boxed{\texttt{PROGMEM}}$ = {
  {0, 0, 0, 0, 0, 0},
  {0, 0, 1, 1, 0, 0},
  {0, 0, 1, 1, 0, 0},
  {0, 0, 0, 0, 0, 0},
  {0, 0, 1, 1, 0, 0},
  {0, 0, 1, 1, 0, 0},
  {0, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0}
};
```

**20.** ⟨ Set brightness depending on time of day 20 ⟩ ≡

```
if (strcmp(time, "21:00:00") ≥ 0 ∨ strcmp(time, "06:00:00") < 0) display_write4(#0A, #00);
if (strcmp(time, "06:00:00") ≥ 0 ∧ strcmp(time, "21:00:00") < 0) display_write4(#0A, #0F);
```

This code is cited in section 1.

This code is used in section 1.

**21.** No other requests except SET CONTROL LINE STATE come after connection is established. These are from *open* and implicit *close* in `time-write`. Just discard the data.

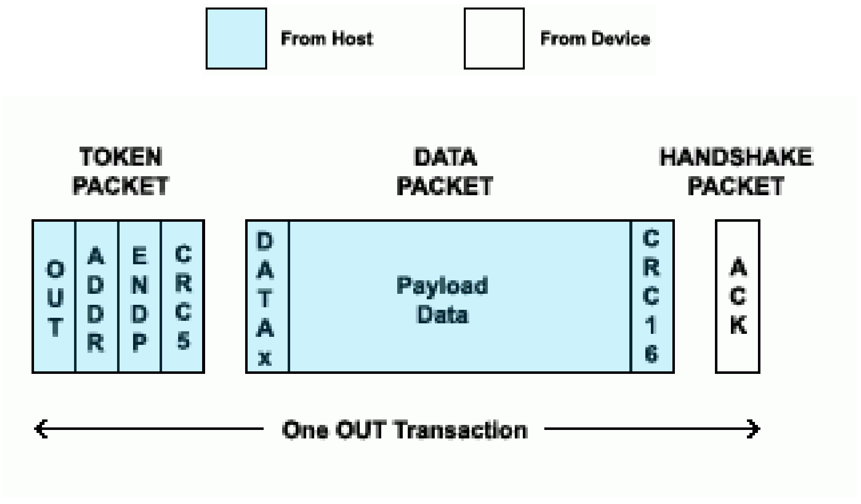
⟨ If there is a request on EP0, handle it 21 ⟩ ≡

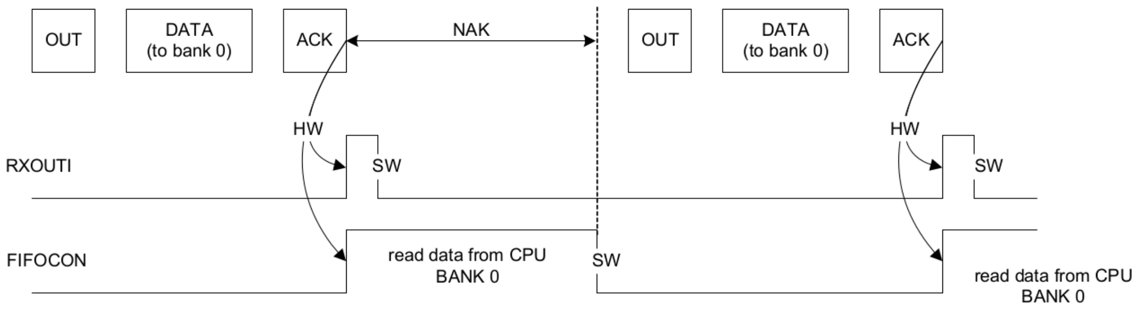
```
UENUM = EP0;
if (UEINTX & 1 ≪ RXSTPI) {
  UEINTX &= ∼(1 ≪ RXSTPI);
  UEINTX &= ∼(1 ≪ TXINI);      /* STATUS stage */
}
```

This code is used in section 1.

**22. OUT endpoint management.** (WARNING: these images are incomplete — they do not show possible handshake phases)

There is only one stage (data). It corresponds to the following transaction(s):





**23. Establishing USB connection.**

```
⟨ Global variables 8 ⟩ +≡
  volatile int connected = 0;
```

**24.** **#define** EP0 0 /* selected by default */
**#define** EP0_SIZE 32 /* 32 bytes† (max for atmega32u4) */

```
⟨ Create ISR for connecting to USB host 24 ⟩ ≡
  ISR(USB_GEN_vect)
  {
    UDINT &= ~(1 ≪ EORSTI);    /* for the interrupt handler to be called for next USB_RESET */
    if (¬connected) {
      UECONX |= 1 ≪ EPEN;
      UECFG1X = 1 ≪ EPSIZE1;    /* 32 bytes‡ */
      UECFG1X |= 1 ≪ ALLOC;
    }
    else {
      ⟨ Reset MCU 25 ⟩
    }
  }
```

This code is used in section 1.

**25.** Used in USB_RESET interrupt handler. Reset is used to go to beginning of connection loop (because we cannot use **goto** from within interrupt handler). Watchdog reset is used because in atmega32u4 there is no simpler way to reset MCU.

```
⟨ Reset MCU 25 ⟩ ≡
  WDTCSR |= 1 ≪ WDCE | 1 ≪ WDE;    /* allow to enable WDT */
  WDTCSR = 1 ≪ WDE;    /* enable WDT */
  while (1) ;
```

This code is used in section 24.

† Must correspond to UECFG1X of EP0.
‡ Must correspond to EP0_SIZE.

**26.** When reset is done via watchdog, WDRF (WatchDog Reset Flag) is set in MCUSR register. WDE (WatchDog system reset Enable) is always set in WDTCSR when WDRF is set. It is necessary to clear WDE to stop MCU from eternal resetting: on MCU start we always clear `WDRF` and WDE (nothing will change if they are not set). To avoid unintentional changes of WDE, a special write procedure must be followed to change the WDE bit. To clear WDE, WDRF must be cleared first.

Datasheet says that `WDE` is always set to one when `WDRF` is set to one, but it does not say if `WDE` is always set to zero when `WDRF` is not set (by default it is zero). So we must always clear `WDE` independent of `WDRF`.

This should be done right at the beginning of *main*, in order to be in time before WDT is triggered. We don't call *wdt_reset* because initialization code, that `avr-gcc` adds, has enough time to execute before watchdog timer (16ms in this program) expires:

```
eor r1, r1 (1 cycle)
out 0x3f, r1 (1 cycle)
ldi r28, 0xFF (1 cycle)
ldi r29, 0x0A (1 cycle)
out 0x3e, r29 (1 cycle)
out 0x3d, r28 (1 cycle)
call <main> (4 cycles)
```

At 16MHz each cycle is 62.5 nanoseconds, so it is 7 instructions, taking 10 cycles, multiplied by 62.5 is 625 nanoseconds.

What the above code does: zero r1 register, clear SREG, initialize program stack (to the stack processor writes addresses for returning from subroutines and interrupt handlers). To the stack pointer is written address of last cell of RAM.

Note, that ns is $10^{-9}$, us is $10^{-6}$ and ms is $10^{-3}$.

⟨ Disable WDT 26 ⟩ ≡

```
if (MCUSR & 1 ≪ WDRF)    /* takes 2 instructions if WDRF is set to one: in (1 cycle), sbrs (2 cycles),
        which is 62.5*3 = 187.5 nanoseconds more, but still within 16ms; and it takes 5 instructions if
        WDRF is not set: in (1 cycle), sbrs (2 cycles), rjmp (2 cycles), which is 62.5*5 = 312.5 ns more,
        but still within 16ms */
  MCUSR &= ~(1 ≪ WDRF);    /* takes 3 instructions: in (1 cycle), andi (1 cycle), out (1 cycle), which
        is 62.5*3 = 187.5 nanoseconds more, but still within 16ms */
if (WDTCSR & 1 ≪ WDE) {    /* takes 2 instructions: in (1 cycle), sbrs (2 cycles), which is 62.5*3 =
        187.5 nanoseconds more, but still within 16ms */
  WDTCSR |= 1 ≪ WDCE;    /* allow to disable WDT (lds (2 cycles), ori (1 cycle), sts (2 cycles)),
        which is 62.5*5 = 312.5 ns more, but still within 16ms) */
  WDTCSR = #00;
    /* disable WDT (sts (2 cycles), which is 62.5*2 = 125 ns more, but still within 16ms)* */
}
```

This code is used in section 27.

* '&=' must not be used here, because the following instructions will be used: `lds` (2 cycles), `andi` (1 cycle), `sts` (2 cycles), but according to datasheet §8.2 this must not exceed 4 cycles, whereas with '=' at most the following instructions are used: `ldi` (1 cycle) and `sts` (2 cycles), which is within 4 cycles.

**27.** ⟨ Connect to USB host (must be called first; *sei* is called here) 27 ⟩ ≡

```
⟨ Disable WDT 26 ⟩
UHWCON |= 1 ≪ UVREGE;
USBCON |= 1 ≪ USBE;
PLLCSR = 1 ≪ PINDIV;
PLLCSR |= 1 ≪ PLLE;
while (¬(PLLCSR & 1 ≪ PLOCK)) ;
USBCON &= ∼(1 ≪ FRZCLK);
USBCON |= 1 ≪ OTGPADE;
UDIEN |= 1 ≪ EORSTE;
sei();
UDCON &= ∼(1 ≪ DETACH);      /* attach after we prepared interrupts, because USB_RESET will arrive
      only after attach, and before it arrives, all interrupts must be already set up; also, there is no need to
      detect when VBUS becomes high — USB_RESET can arrive only after VBUS is operational anyway,
      and USB_RESET is detected via interrupt */
while (¬connected)
  if (UEINTX & 1 ≪ RXSTPI) ⟨ Process SETUP request 34 ⟩
```

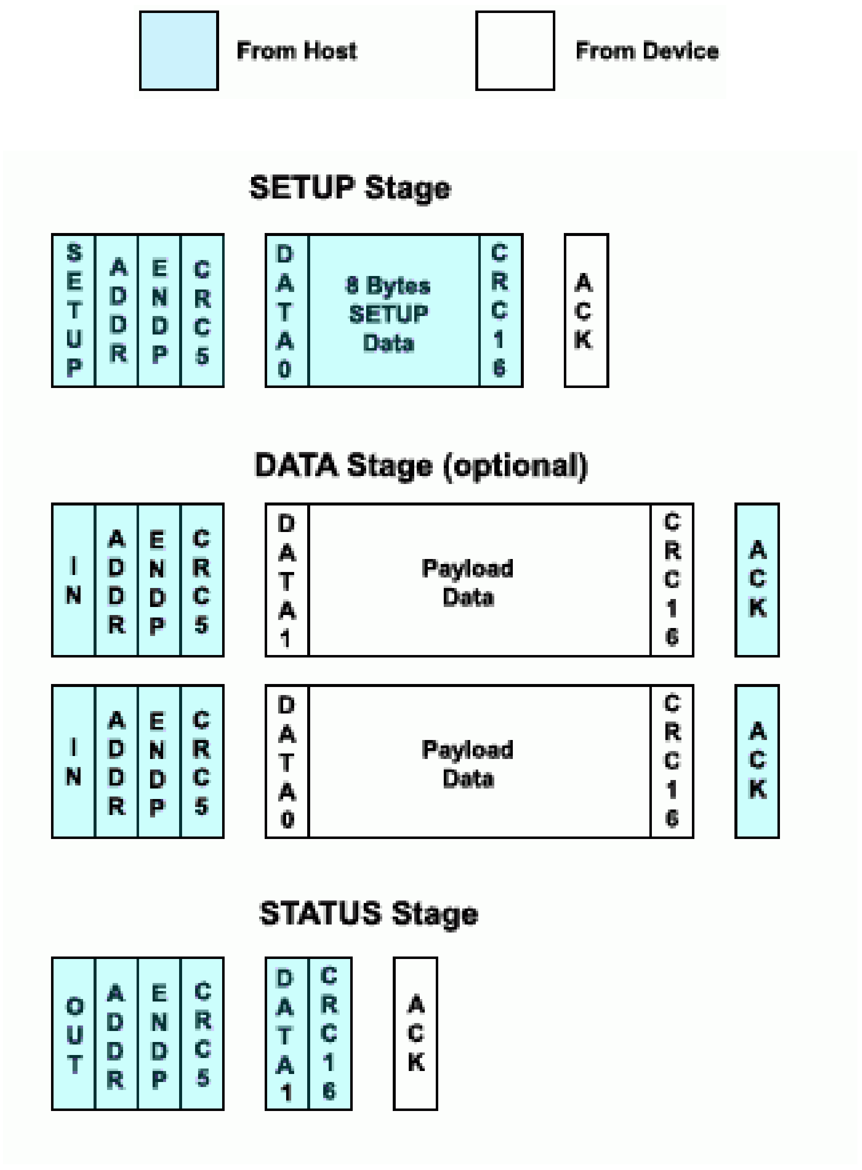
This code is used in section 1.

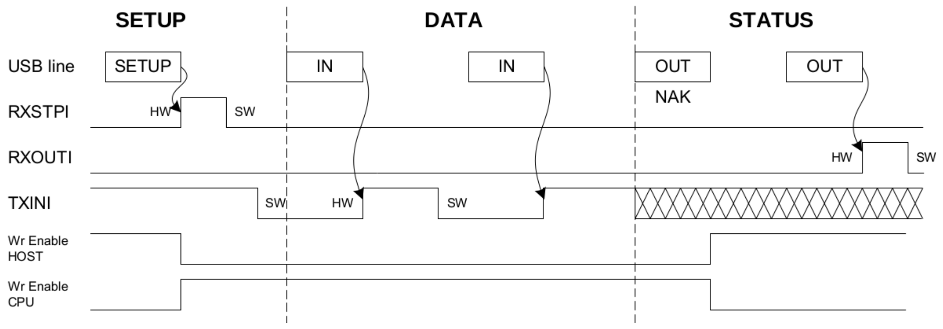
**28. Control endpoint management.** (WARNING: these images are incomplete — they do not show possible handshake phases)

Device driver sends a packet to device's EP0. As the data is flowing out from the host, it will end up in the EP0 buffer. Firmware will then at its leisure read this data. If it wants to return data, the device cannot simply write to the bus as the bus is controlled by the host. Therefore it writes data to EP0 which sits in the buffer until such time when the host sends a IN packet requesting the data.*

* This is where the prase "USB controller has to manage simultaneous write requests from firmware and host" from §22.12.2 of datasheet becomes clear. (Remember, we use one and the same endpoint to read *and* write control data.)

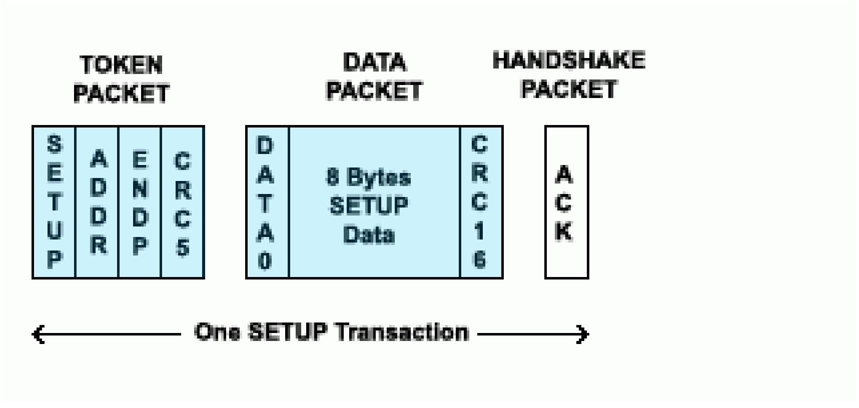
**29. Control read (by host).** There are the folowing stages*:

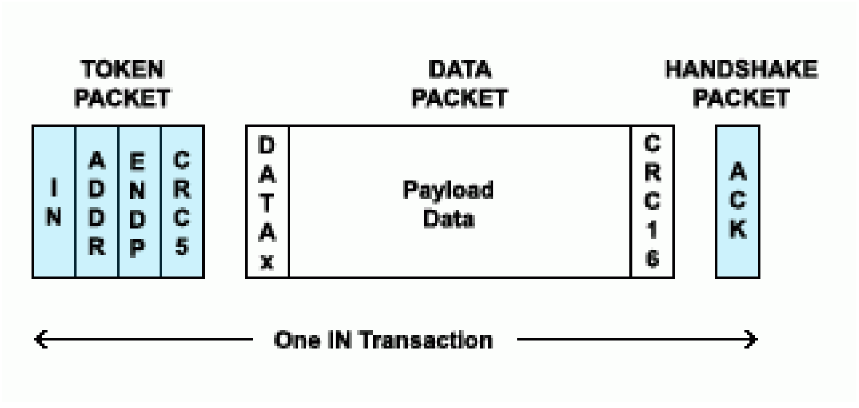


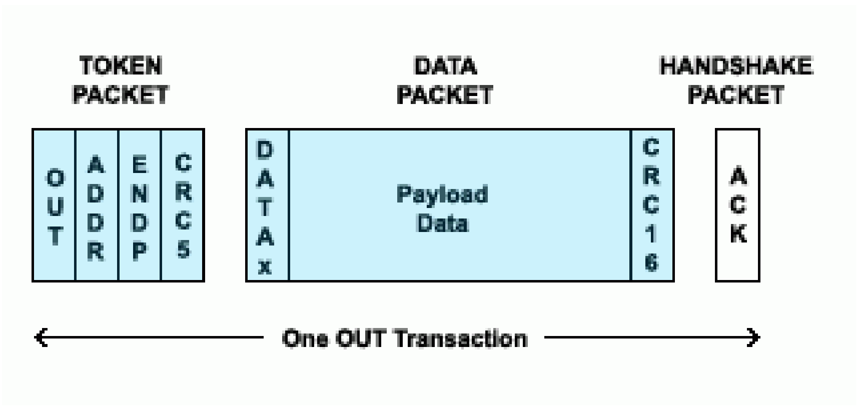


* Setup transaction ≡ Setup stage

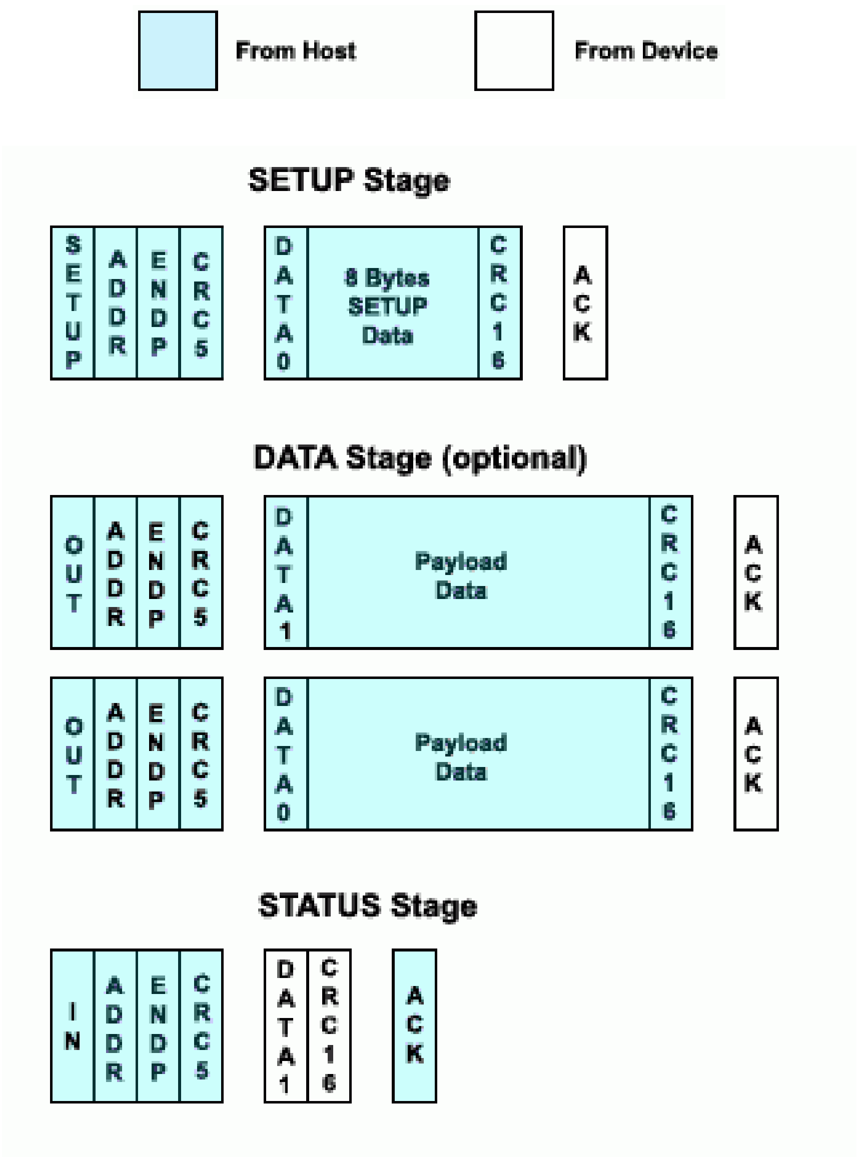
**30.** This corresponds to the following transactions:

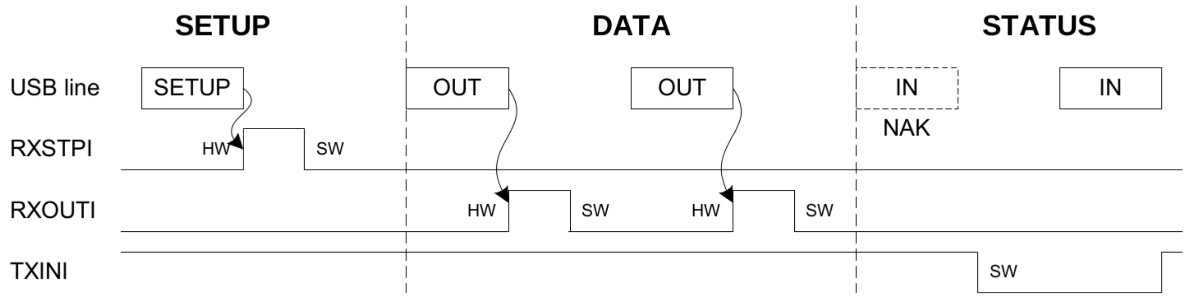






**31. Control write (by host).** There are the following stages*:

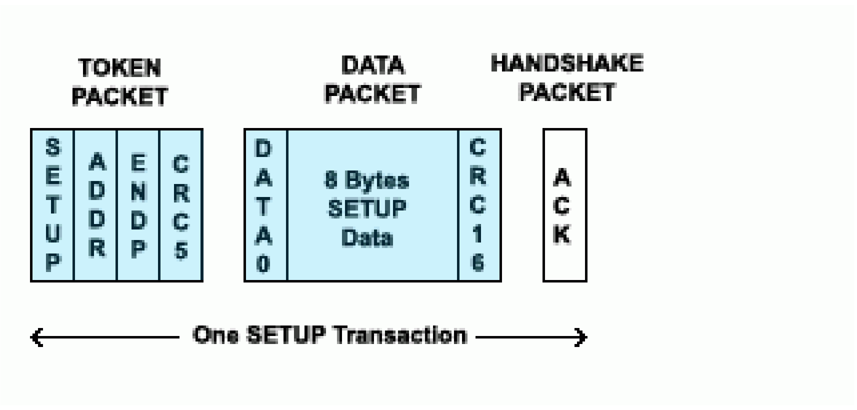


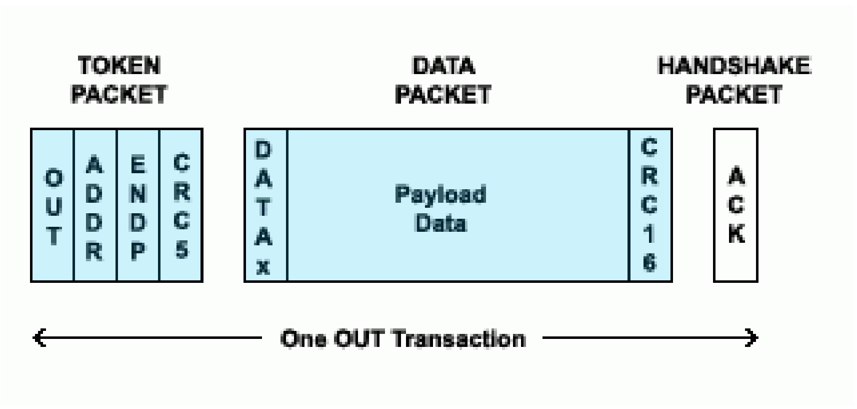


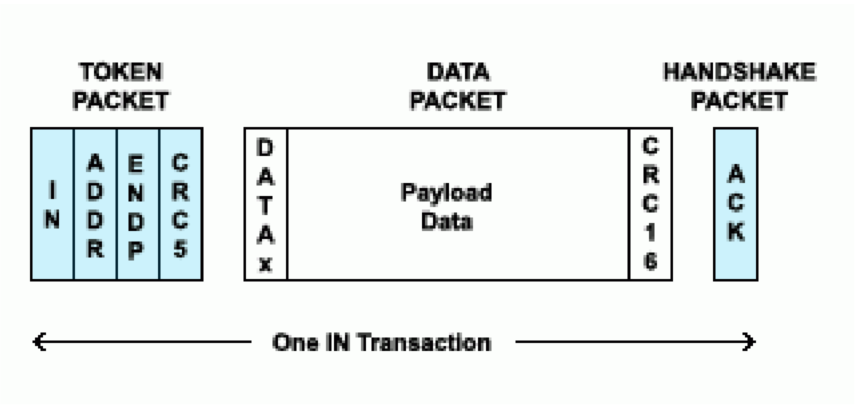
* Setup transaction ≡ Setup stage

Commentary to the drawing why “controller will not necessarily send a NAK at the first IN token” (see §22.12.1 in datasheet): If TXINI is already cleared when IN packet arrives, NAKINI is not set. This corresponds to case 1. If TXINI is not yet cleared when IN packet arrives, NAKINI is set. This corresponds to case 2.

**32.** This corresponds to the following transactions:







**33. Connection protocol.**

```
⟨ Global variables 8 ⟩ +≡
  U16 wValue;
  U16 wIndex;
  U16 wLength;
```

**34.** The following big switch just dispatches SETUP request.

```
⟨ Process SETUP request 34 ⟩ ≡
  switch (UEDATX | UEDATX ≪ 8) {
  case #0500:
    ⟨ Handle SET ADDRESS 35 ⟩
    break;
  case #0680:
    switch (UEDATX | UEDATX ≪ 8) {
    case #0100:
      ⟨ Handle GET DESCRIPTOR DEVICE 36 ⟩
      break;
    case #0200:
      ⟨ Handle GET DESCRIPTOR CONFIGURATION 38 ⟩
      break;
    case #0300:
      ⟨ Handle GET DESCRIPTOR STRING (language) 39 ⟩
      break;
    case #03 ≪ 8 | MANUFACTURER:
      ⟨ Handle GET DESCRIPTOR STRING (manufacturer) 40 ⟩
      break;
    case #03 ≪ 8 | PRODUCT:
      ⟨ Handle GET DESCRIPTOR STRING (product) 41 ⟩
      break;
    case #03 ≪ 8 | SERIAL_NUMBER:
      ⟨ Handle GET DESCRIPTOR STRING (serial) 42 ⟩
      break;
    case #0600:
      ⟨ Handle GET DESCRIPTOR DEVICE QUALIFIER 37 ⟩
      break;
    }
    break;
  case #0900:
    ⟨ Handle SET CONFIGURATION 43 ⟩
    break;
  case #2021:
    ⟨ Handle SET LINE CODING 44 ⟩
    break;
  }
```

This code is used in section 27.

**35.** No OUT packet arrives after SETUP packet, because there is no DATA stage in this request. IN packet arrives after SETUP packet, and we get ready to send a ZLP in advance.

```
⟨ Handle SET ADDRESS 35 ⟩ ≡
  wValue = UEDATX | UEDATX ≪ 8;
  UDADDR = wValue & #7F;
  UEINTX &= ∼(1 ≪ RXSTPI);
  UEINTX &= ∼(1 ≪ TXINI);      /* STATUS stage */
  while (¬(UEINTX & 1 ≪ TXINI)) ;
      /* wait until ZLP, prepared by previous command, is sent to host♯ */
  UDADDR |= 1 ≪ ADDEN;
```

This code is used in section 34.

**36.** When host is booting, BIOS asks 8 bytes in first request of device descriptor (8 bytes is sufficient for first request of device descriptor). If host is operational, *wLength* is 64 bytes in first request of device descriptor. It is OK if we transfer less than the requested amount. But if we try to transfer more, host does not send OUT packet to initiate STATUS stage.

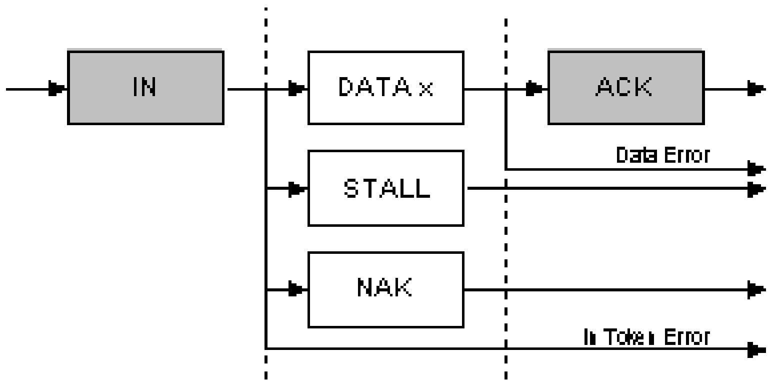
```
⟨ Handle GET DESCRIPTOR DEVICE 36 ⟩ ≡
  (void) UEDATX; (void) UEDATX;
  wLength = UEDATX | UEDATX ≪ 8;
  UEINTX &= ∼(1 ≪ RXSTPI);
  size = sizeof dev_desc;
  buf = &dev_desc;
  ⟨ Send descriptor 46 ⟩
```

This code is used in section 34.

♯ According to §22.7 of the datasheet, firmware must send ZLP in the STATUS stage before enabling the new address. The reason is that the request started by using zero address, and all the stages of the request must use the same address. Otherwise STATUS stage will not complete, and thus set address request will not succeed. We can determine when ZLP is sent by receiving the ACK, which sets TXINI to 1. See "Control write (by host)" in table of contents for the picture (note that DATA stage is absent).

**37.** A high-speed capable device that has different device information for full-speed and high-speed must have a Device Qualifier Descriptor. For example, if the device is currently operating at full-speed, the Device Qualifier returns information about how it would operate at high-speed and vice-versa. So as this device is full-speed, it tells the host not to request device information for high-speed by using "protocol stall" (such stall does not indicate an error with the device — it serves as a means of extending USB requests).

The host sends an IN token to the control pipe to initiate the DATA stage.



Note, that next token comes after RXSTPI is cleared, so we set STALLRQ before clearing RXSTPI, to make sure that STALLRQ is already set when next token arrives.

This STALL condition is automatically cleared on the receipt of the next SETUP token.

USB§8.5.3.4, datasheet§22.11.

```
⟨ Handle GET DESCRIPTOR DEVICE QUALIFIER 37 ⟩ ≡
  UECONX |= 1 ≪ STALLRQ;
     /* prepare to send STALL handshake in response to IN token of the DATA stage */
  UEINTX &= ∼(1 ≪ RXSTPI);
```
This code is used in section 34.

**38.** First request is 9 bytes, second is according to length given in response to first request.

```
⟨ Handle GET DESCRIPTOR CONFIGURATION 38 ⟩ ≡
  (void) UEDATX; (void) UEDATX;
  wLength = UEDATX | UEDATX ≪ 8;
  UEINTX &= ∼(1 ≪ RXSTPI);
  size = sizeof conf_desc;
  buf = &conf_desc;
  ⟨ Send descriptor 46 ⟩
```
This code is used in section 34.

**39.**
```
⟨ Handle GET DESCRIPTOR STRING (language) 39 ⟩ ≡
  (void) UEDATX; (void) UEDATX;
  wLength = UEDATX | UEDATX ≪ 8;
  UEINTX &= ∼(1 ≪ RXSTPI);
  size = sizeof lang_desc;
  buf = lang_desc;
  ⟨ Send descriptor 46 ⟩
```
This code is used in section 34.

**40.** ⟨ Handle GET DESCRIPTOR STRING (manufacturer) 40 ⟩ ≡

```
(void) UEDATX; (void) UEDATX;
wLength = UEDATX | UEDATX ≪ 8;
UEINTX &= ∼(1 ≪ RXSTPI);
size = pgm_read_byte(&mfr_desc.bLength);
buf = &mfr_desc;
⟨ Send descriptor 46 ⟩
```

This code is cited in section 69.

This code is used in section 34.

**41.** ⟨ Handle GET DESCRIPTOR STRING (product) 41 ⟩ ≡

```
(void) UEDATX; (void) UEDATX;
wLength = UEDATX | UEDATX ≪ 8;
UEINTX &= ∼(1 ≪ RXSTPI);
size = pgm_read_byte(&prod_desc.bLength);
buf = &prod_desc;
⟨ Send descriptor 46 ⟩
```

This code is cited in section 69.

This code is used in section 34.

**42.** Here we handle one case when data (serial number) needs to be transmitted from memory, not from program.

⟨ Handle GET DESCRIPTOR STRING (serial) 42 ⟩ ≡

```
(void) UEDATX; (void) UEDATX;
wLength = UEDATX | UEDATX ≪ 8;
UEINTX &= ∼(1 ≪ RXSTPI);
size = 1 + 1 + SN_LENGTH * 2;     /* multiply because Unicode */
⟨ Fill in sn_desc with serial number 73 ⟩
buf = &sn_desc;
from_program = 0;
⟨ Send descriptor 46 ⟩
```

This code is used in section 34.

**43.** Interrupt IN endpoint is not used, but it must be present (for more info see "Communication Class notification endpoint notice" in index).

```
#define EP1 1
#define EP2 2
#define EP3 3
#define EP1_SIZE 32    /* 32 bytes† */
#define EP2_SIZE 32    /* 32 bytes† */
#define EP3_SIZE 32    /* 32 bytes† */
```

⟨ Handle SET CONFIGURATION 43 ⟩ ≡

```
UEINTX &= ∼(1 ≪ RXSTPI);
UENUM = EP3;
UECONX |= 1 ≪ EPEN;
UECFG0X = 1 ≪ EPTYPE1 | 1 ≪ EPTYPE0 | 1 ≪ EPDIR;     /* interrupt†, IN */
UECFG1X = 1 ≪ EPSIZE1;     /* 32 bytes‡ */
```

† Must correspond to UECFG1X of EP1.

† Must correspond to UECFG1X of EP2.

† Must correspond to UECFG1X of EP3.

† Must correspond to ⟨ Initialize element 6 in configuration descriptor 58 ⟩.

‡ Must correspond to EP3_SIZE.

```
  UECFG1X |= 1 ≪ ALLOC;
  UENUM = EP1;
  UECONX |= 1 ≪ EPEN;
  UECFG0X = 1 ≪ EPTYPE1 | 1 ≪ EPDIR;    /* bulk†, IN */
  UECFG1X = 1 ≪ EPSIZE1;    /* 32 bytes‡ */
  UECFG1X |= 1 ≪ ALLOC;
  UENUM = EP2;
  UECONX |= 1 ≪ EPEN;
  UECFG0X = 1 ≪ EPTYPE1;    /* bulk†, OUT */
  UECFG1X = 1 ≪ EPSIZE1;    /* 32 bytes‡ */
  UECFG1X |= 1 ≪ ALLOC;
  UENUM = EP0;    /* restore for further setup requests */
  UEINTX &= ~(1 ≪ TXINI);    /* STATUS stage */
```

This code is used in section 34.

**44.** Just discard the data. This is the last request after attachment to host.

```
⟨ Handle SET LINE CODING 44 ⟩ ≡
  UEINTX &= ~(1 ≪ RXSTPI);
  while (¬(UEINTX & 1 ≪ RXOUTI)) ;    /* wait for DATA stage */
  UEINTX &= ~(1 ≪ RXOUTI);
  UEINTX &= ~(1 ≪ TXINI);    /* STATUS stage */
  connected = 1;
```

This code is used in section 34.

**45.** ⟨ Global variables 8 ⟩ +≡

```
  U16 size;
  const void *buf;
  U8 from_program = 1;    /* serial number is transmitted last, so this can be set only once */
  U8 empty_packet;
```

† Must correspond to ⟨ Initialize element 8 in configuration descriptor 59 ⟩.
‡ Must correspond to EP1_SIZE.
† Must correspond to ⟨ Initialize element 9 in configuration descriptor 60 ⟩.
‡ Must correspond to EP2_SIZE.

**46.** Transmit data and empty packet (if necessary) and wait for STATUS stage.

On control endpoint by clearing TXINI (in addition to making it possible to know when bank will be free again) we say that when next IN token arrives, data must be sent and endpoint bank cleared. When data was sent, TXINI becomes ‘1’. After TXINI becomes ‘1’, new data may be written to UEDATX.*

```
⟨ Send descriptor 46 ⟩ ≡
  empty_packet = 0;
  if (size < wLength ∧ size % EP0_SIZE ≡ 0) empty_packet = 1;
      /* indicate to the host that no more data will follow (USB§5.5.3) */
  if (size > wLength) size = wLength;      /* never send more than requested */
  while (size ≠ 0) {
    while (¬(UEINTX & 1 ≪ TXINI)) ;
    U8 nb_byte = 0;
    while (size ≠ 0) {
      if (nb_byte++ ≡ EP0_SIZE) break;
      UEDATX = from_program ? pgm_read_byte(buf++) : *(U8 *) buf++;
      size--;
    }
    UEINTX &= ~(1 ≪ TXINI);
  }
  if (empty_packet) {
    while (¬(UEINTX & 1 ≪ TXINI)) ;
    UEINTX &= ~(1 ≪ TXINI);
  }
  while (¬(UEINTX & 1 ≪ RXOUTI)) ;      /* wait for STATUS stage */
  UEINTX &= ~(1 ≪ RXOUTI);
```

This code is used in sections 36, 38, 39, 40, 41, and 42.

* The difference of clearing TXINI for control and non-control endpoint is that on control endpoint clearing TXINI also sends the packet and clears the endpoint bank. On non-control endpoints there is a possibility to have double bank, so another mechanism is used.

**47. USB stack.**

```
⟨ Type definitions 47 ⟩ ≡
  typedef unsigned char U8;
  typedef unsigned short U16;
```

See also sections 49 and 69.

This code is used in section 1.

**48. Device descriptor.**

Placeholder prefixes such as ‘b’, ‘bcd’, and ‘w’ are used to denote placeholder type:

| | |
|---|---|
| *b* | bits or bytes; dependent on context |
| *bcd* | binary-coded decimal |
| *bm* | bitmap |
| *d* | descriptor |
| *i* | index |
| *w* | word |

```
#define MANUFACTURER 1
#define PRODUCT 2
#define SERIAL_NUMBER 3

⟨ Global variables 8 ⟩ +≡
  struct {
    U8 bLength;
    U8 bDescriptorType;
    U16 bcdUSB;    /* version */
    U8 bDeviceClass;    /* class code assigned by the USB */
    U8 bDeviceSubClass;    /* sub-class code assigned by the USB */
    U8 bDeviceProtocol;    /* protocol code assigned by the USB */
    U8 bMaxPacketSize0;    /* max packet size for EP0 */
    U16 idVendor;
    U16 idProduct;
    U16 bcdDevice;    /* device release number */
    U8 iManufacturer;    /* index of manu. string descriptor */
    U8 iProduct;    /* index of prod. string descriptor */
    U8 iSerialNumber;    /* index of S.N. string descriptor */
    U8 bNumConfigurations;
  } const dev_desc PROGMEM = {
    18,    /* size of this structure */
    #01,    /* device */
    #0200,    /* USB 2.0 */
    #02,    /* CDC (§4.1 in CDC spec) */
    0,    /* no subclass */
    0,
    EP0_SIZE,
    #03EB,    /* VID (Atmel) */
    #2018,    /* PID (CDC ACM) */
    #1000,    /* device revision */
    MANUFACTURER,    /* (Mfr in kern.log) */
    PRODUCT,    /* (Product in kern.log) */
    SERIAL_NUMBER,    /* (SerialNumber in kern.log) */
    1    /* one configuration for this device */
  };
```

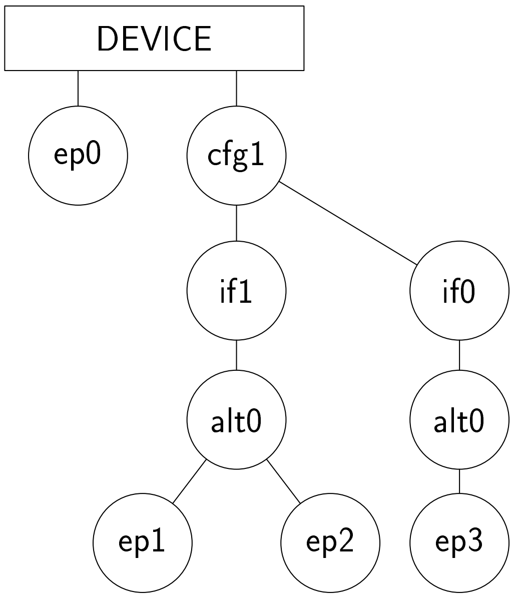
**49. Configuration descriptor.**

Abstract Control Model consists of two interfaces: Data Class interface and Communication Class interface.

The Communication Class interface uses two endpoints*, one to implement a notification element and the other to implement a management element. The management element uses the default endpoint for all standard and Communication Class-specific requests.

Theh Data Class interface consists of two endpoints to implement channels over which to carry data.

§3.4 in CDC spec.



```
⟨ Type definitions 47 ⟩ +≡
  ⟨ Type definitions used in configuration descriptor 54 ⟩
  typedef struct {
    ⟨ Configuration header descriptor 52 ⟩ el1;
    S_interface_descriptor el2;
    ⟨ Class-specific interface descriptor 1 62 ⟩ el3;
    ⟨ Class-specific interface descriptor 2 64 ⟩ el5;
    ⟨ Class-specific interface descriptor 3 66 ⟩ el6;
    S_endpoint_descriptor el7;
    S_interface_descriptor el8;
    S_endpoint_descriptor el9;
    S_endpoint_descriptor el10;
  } S_configuration_descriptor;
```

* Although CDC spec says that notification endpoint is optional, in Linux host driver refuses to work without it. Besides, notifocation endpoint (EP3) can be used for DSR signal.

**50.** ⟨ Global variables 8 ⟩ +≡
```
const S_configuration_descriptor conf_desc PROGMEM = {
  ⟨ Initialize element 1 in configuration descriptor 53 ⟩,
  ⟨ Initialize element 2 in configuration descriptor 55 ⟩,
  ⟨ Initialize element 3 in configuration descriptor 63 ⟩,
  ⟨ Initialize element 4 in configuration descriptor 65 ⟩,
  ⟨ Initialize element 5 in configuration descriptor 67 ⟩,
  ⟨ Initialize element 6 in configuration descriptor 58 ⟩,
  ⟨ Initialize element 7 in configuration descriptor 56 ⟩,
  ⟨ Initialize element 8 in configuration descriptor 59 ⟩,
  ⟨ Initialize element 9 in configuration descriptor 60 ⟩
};
```

**51. Configuration header descriptor.**

**52.** ⟨ Configuration header descriptor 52 ⟩ ≡
```
struct {
  U8 bLength;
  U8 bDescriptorType;
  U16 wTotalLength;
  U8 bNumInterfaces;
  U8 bConfigurationValue;
    /* number between 1 and bNumConfigurations, for each configuration† */
  U8 iConfiguration;    /* index of string descriptor */
  U8 bmAttibutes;
  U8 MaxPower;
}
```
This code is used in section 49.

**53.** ⟨ Initialize element 1 in configuration descriptor 53 ⟩ ≡
```
{
  9,    /* size of this structure */
  #02,    /* configuration descriptor */
  sizeof (S_configuration_descriptor),
  2,    /* two interfaces in this configuration */
  1,    /* this corresponds to '1' in 'cfg1' on picture */
  0,    /* no string descriptor */
  #80,    /* device is powered from bus */
  #32    /* device uses 100mA */
}
```
This code is used in section 50.

† For some reason configurations start numbering with '1', and interfaces and altsettings with '0'.

**54. Interface descriptor.**

```
⟨ Type definitions used in configuration descriptor 54 ⟩ ≡
  typedef struct {
    U8 bLength;
    U8 bDescriptorType;
    U8 bInterfaceNumber;   /* number between 0 and bNumInterfaces − 1, for each interface */
    U8 bAlternativeSetting;   /* number starting from 0, for each interface */
    U8 bNumEndpoints;   /* number of EP except EP 0 */
    U8 bInterfaceClass;   /* class code assigned by the USB */
    U8 bInterfaceSubClass;   /* sub-class code assigned by the USB */
    U8 bInterfaceProtocol;   /* protocol code assigned by the USB */
    U8 iInterface;   /* index of string descriptor */
  } S_interface_descriptor;
```

See also section 57.

This code is used in section 49.

**55.** ⟨ Initialize element 2 in configuration descriptor 55 ⟩ ≡

```
  {
    9,   /* size of this structure */
    #04,   /* interface descriptor */
    0,   /* this corresponds to '0' in 'if0' on picture */
    0,   /* this corresponds to '0' in 'alt0' on picture */
    1,   /* one endpoint is used */
    #02,   /* CDC (§4.2 in CDC spec) */
    #02,   /* ACM (§4.3 in CDC spec) */
    #01,   /* AT command (§4.4 in CDC spec) */
    0   /* not used */
  }
```

This code is used in section 50.

**56.** ⟨ Initialize element 7 in configuration descriptor 56 ⟩ ≡

```
  {
    9,   /* size of this structure */
    #04,   /* interface descriptor */
    1,   /* this corresponds to '1' in 'if1' on picture */
    0,   /* this corresponds to '0' in 'alt0' on picture */
    2,   /* two endpoints are used */
    #0A,   /* CDC data (§4.5 in CDC spec) */
    #00,   /* unused */
    #00,   /* no protocol */
    0   /* not used */
  }
```

This code is used in section 50.

**57. Endpoint descriptor.**

```
⟨ Type definitions used in configuration descriptor 54 ⟩ +≡
  typedef struct {
    U8 bLength;
    U8 bDescriptorType;
    U8 bEndpointAddress;
    U8 bmAttributes;
    U16 wMaxPacketSize;
    U8 bInterval;    /* interval for polling EP by host to determine if data is available (ms-1) */
  } S_endpoint_descriptor;
```

**58.** Interrupt IN endpoint serves when device needs to interrupt host. Host sends IN tokens to device at a rate specified here (this endpoint is not used, so rate is maximum possible).

```
#define IN (1 ≪ 7)
⟨ Initialize element 6 in configuration descriptor 58 ⟩ ≡
  {
    7,    /* size of this structure */
    #05,    /* endpoint */
    IN | 3,    /* this corresponds to '3' in 'ep3' on picture */
    #03,    /* transfers via interrupts† */
    EP3_SIZE,
    #FF    /* 256 (FIXME: is it 'ms'?) */
  }
```

This code is cited in section 43.

This code is used in section 50.

**59.**

```
⟨ Initialize element 8 in configuration descriptor 59 ⟩ ≡
  {
    7,    /* size of this structure */
    #05,    /* endpoint */
    IN | 1,    /* this corresponds to '1' in 'ep1' on picture */
    #02,    /* bulk transfers† */
    EP1_SIZE,
    #00    /* not applicable */
  }
```

This code is cited in section 43.

This code is used in section 50.

† Must correspond to UECFG0X of EP3.

† Must correspond to UECFG0X of EP1.

**60.** **#define** `OUT` $(0 \ll 7)$

⟨ Initialize element 9 in configuration descriptor 60 ⟩ ≡

```
{
  7,     /* size of this structure */
  #05,     /* endpoint */
  OUT | 2,     /* this corresponds to '2' in 'ep2' on picture */
  #02,     /* bulk transfers† */
  EP2_SIZE,
  #00     /* not applicable */
}
```

This code is cited in section 43.

This code is used in section 50.

**61. Functional descriptors.**

These descriptors describe the content of the class-specific information within an Interface descriptor. They all start with a common header descriptor, which allows host software to easily parse the contents of class-specific descriptors. Although the Communication Class currently defines class specific interface descriptor information, the Data Class does not.

§5.2.3 in CDC spec.

**62. Header functional descriptor.**

The class-specific descriptor shall start with a header. It identifies the release of the USB Class Definitions for Communication Devices Specification with which this interface and its descriptors comply.

§5.2.3.1 in CDC spec.

⟨ Class-specific interface descriptor 1 62 ⟩ ≡

```
struct {
  U8 bFunctionLength;
  U8 bDescriptorType;
  U8 bDescriptorSubtype;
  U16 bcdCDC;
}
```

This code is used in section 49.

**63.** ⟨ Initialize element 3 in configuration descriptor 63 ⟩ ≡

```
{
  5,     /* size of this structure */
  #24,     /* interface */
  #00,     /* header */
  #0110     /* CDC 1.1 */
}
```

This code is used in section 50.

† Must correspond to `UECFG0X` of `EP2`.

**64. Abstract control management functional descriptor.**
The Abstract Control Management functional descriptor describes the commands supported by the Communication Class interface, as defined in §3.6.2 in CDC spec, with the SubClass code of Abstract Control Model.
§5.2.3.3 in CDC spec.

```
⟨ Class-specific interface descriptor 2 64 ⟩ ≡
  struct {
    U8 bFunctionLength;
    U8 bDescriptorType;
    U8 bDescriptorSubtype;
    U8 bmCapabilities;
  }
```
This code is used in section 49.

**65.** *bmCapabilities*: Only first four bits are used. If first bit is set, then this indicates the device supports the request combination of `Set_Comm_Feature`, `Clear_Comm_Feature`, and `Get_Comm_Feature`. If second bit is set, then the device supports the request combination of `Set_Line_Coding`, `Set_Control_Line_State`, `Get_Line_Coding`, and the notification `Serial_State`. If the third bit is set, then the device supports the request `Send_Break`. If fourth bit is set, then the device supports the notification `Network_Connection`. A bit value of zero means that the request is not supported.

```
⟨ Initialize element 4 in configuration descriptor 65 ⟩ ≡
  {
    4,     /* size of this structure */
    #24,     /* interface */
    #02,     /* ACM */
    1 ≪ 2 | 1 ≪ 1
  }
```
This code is used in section 50.

**66. Union functional descriptor.**
The Union functional descriptor describes the relationship between a group of interfaces that can be considered to form a functional unit. One of the interfaces in the group is designated as a master or controlling interface for the group, and certain class-specific messages can be sent to this interface to act upon the group as a whole. Similarly, notifications for the entire group can be sent from this interface but apply to the entire group of interfaces.
§5.2.3.8 in CDC spec.

```
⟨ Class-specific interface descriptor 3 66 ⟩ ≡
  struct {
    U8 bFunctionLength;
    U8 bDescriptorType;
    U8 bDescriptorSubtype;
    U8 bMasterInterface;
    U8 bSlaveInterface[SLAVE_INTERFACE_NUM];
  }
```
This code is used in section 49.

**67.** **#define** SLAVE_INTERFACE_NUM 1

⟨ Initialize element 5 in configuration descriptor 67 ⟩ ≡

```
  {
    4 + SLAVE_INTERFACE_NUM,      /* size of this structure */
    #24,      /* interface */
    #06,      /* union */
    0,      /* number of CDC control interface */
    {
      1      /* number of CDC data interface */
    }
  }
```

This code is used in section 50.

**68. Language descriptor.**
This is necessary to transmit manufacturer, product and serial number.

⟨ Global variables 8 ⟩ +≡

```
  const U8 lang_desc[] PROGMEM = {
    #04,      /* size of this structure */
    #03,      /* type (string) */
    #09, #04      /* id (English) */
  };
```

**69. String descriptors.**
The trick here is that when defining a variable of type **S_string_descriptor**, the string content follows the first two elements in program memory. The C standard says that a flexible array member in a struct does not increase the size of the struct (aside from possibly adding some padding at the end) but gcc lets you initialize it anyway. **sizeof** on the variable counts only first two elements. So, we read the size of the variable at execution time in ⟨ Handle GET DESCRIPTOR STRING (manufacturer) 40 ⟩ and ⟨ Handle GET DESCRIPTOR STRING (product) 41 ⟩ by using *pgm_read_byte*.

TODO: put here explanation from `https://stackoverflow.com/questions/51470592/`

Is USB each character is 2 bytes. Wide-character string can be used here, because on GCC for atmega32u4 wide character is 2 bytes. Note, that for wide-character string I use type '**int**', not '**wchar_t**', because by '**wchar_t**' I always mean 4 bytes (to avoid using '**wint_t**').

⟨ Type definitions 47 ⟩ +≡

```
  typedef struct {
    U8 bLength;
    U8 bDescriptorType;
    int wString[];
  } S_string_descriptor;
#define STR_DESC(str) { 1 + 1 + sizeof str − 2, #03, str }
```

**70. Manufacturer descriptor.**

⟨ Global variables 8 ⟩ +≡

```
  const S_string_descriptor mfr_desc PROGMEM = STR_DESC(L"ATMEL");
```

**71. Product descriptor.**

⟨ Global variables 8 ⟩ +≡

```
  const S_string_descriptor prod_desc PROGMEM = STR_DESC(L"TEL");
```

**72. Serial number descriptor.**
This one is different in that its content cannot be prepared in compile time, only in execution time. So, it cannot be stored in program memory.

```
#define SN_LENGTH 20   /* length of device signature, multiplied by two (because each byte in hex) */
⟨ Global variables 8 ⟩ +≡
  struct {
    U8 bLength;
    U8 bDescriptorType;
    int wString[SN_LENGTH];
  } sn_desc;
```

**73.**

```
#define SN_START_ADDRESS #0E
#define hex(c) c < 10 ? c + '0' : c − 10 + 'A'
⟨ Fill in sn_desc with serial number 73 ⟩ ≡
  sn_desc.bLength = 1 + 1 + SN_LENGTH * 2;   /* multiply because Unicode */
  sn_desc.bDescriptorType = #03;
  U8 addr = SN_START_ADDRESS;
  for (U8 i = 0; i < SN_LENGTH; i++) {
    U8 c = boot_signature_byte_get(addr);
    if (i & 1) {
        /* we divide each byte of signature into halves, each of which is represented by a hex number */
      c >>= 4;
      addr++;
    }
    else c &= #0F;
    sn_desc.wString[i] = hex(c);
  }
```

This code is used in section 42.

**74. Headers.**

```
⟨ Header files 74 ⟩ ≡
#include <avr/boot.h>    /* boot_signature_byte_get */
#include <avr/interrupt.h>    /* ISR, USB_GEN_vect, sei */
#include <avr/io.h>    /* ADDEN, ALLOC, DDRB, DETACH, EORSTE, EORSTI, EPDIR, EPEN, EPSIZE1,
        EPTYPE0, EPTYPE1, FIFOCON, FRZCLK, MCUSR, MSTR, OTGPADE, PB0, PB1, PB2, PB6, PINDIV, PLLCSR,
        PLLE, PLOCK, PORTB, RXOUTI, RXSTPI, SPCR, SPDR, SPE, SPIF, SPR1, SPSR, STALLRQ, TXINI, UDADDR,
        UDCON, UDIEN, UDINT, UEBCLX, UECFG0X, UECFG1X, UECONX, UEDATX, UEINTX, UENUM, UHWCON, USBCON,
        USBE, UVREGE, WDCE, WDE, WDRF, WDTCSR */
#include <avr/pgmspace.h>    /* pgm_read_byte */
#include <string.h>    /* strcmp, strcpy */
#include <util/delay.h>    /* _delay_us */
```

This code is used in section 1.

**75. Index.**