HP-IB SnifTer

Martin Hepperle, January 2018

The SnifTer is a very simple device connected to the HP-IB bus. It can act in two modes: „HP-IB Sniffer“ or „HP-IB Terminal“. A push button toggles between these two modes.

As an passive device The HP-IB Snifter only listens to the traffic on the bus. It never talks to the bus. As such it cannot slow down the data flow by handshaking and its internal buffer may be overflowed by fast data communication on the bus. This could, for example, be the case when a HP-IB hard disk is on the bus. If you use a slow controller, like a HP-85 and address devices like voltmeters or printers no buffer overflow should occur.

Careful: if you want to modify this into an active device (i.e. one which actively uses the control and/or data lines) you must use at least resistors or line transceivers to control the load on the bus and to avoid shorts! The latter (transceivers like the 75160 and 75161 chips) is recommended. The version shown here is directly connected to the HP-IB bus and the pins of the AVR chip should never be programmed to be outputs!

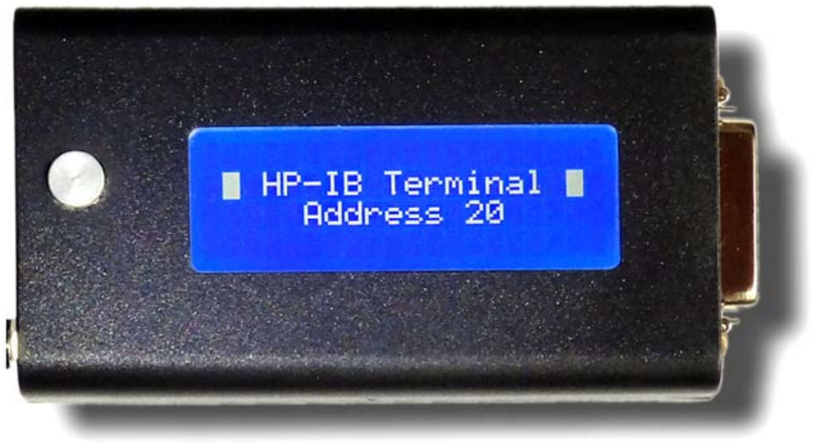
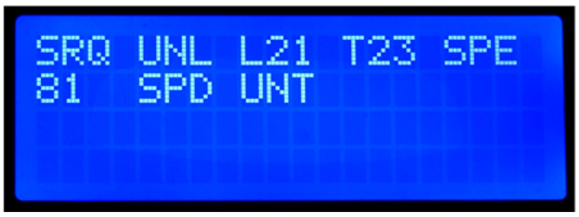


Figure 1: Startup screen of the Snifter booting up in terminal mode and showing its HP-IB address.

Sniffer Mode

When working with HP-IB devices it is sometimes useful to know what is going on on the HB-IB bus. Often it is not clear which commands a controller actually sends or what a response from a device really contains. As I did not have a sophisticated logic analyzer with HP-IB capability at this time, I created my own simple tool to listen to the data exchange on the HP-IB bus.

This mode provides a small display of HP-IB commands and data. Output is formatted in blocks of 3 characters per command resp. data byte. Commands are presented in form of 3 character mnemonics, data bytes are shown as decimal numbers.



```
SRQ UNL L21 T23 SPE
81  SPD UNT
```

Figure 2: The output on the LCD shows the SRQ event sent by a HP 3478A digital voltmeter. The serial poll result 81 has bits 0 („Single Trigger“), 4 („Front/Rear“ switch in „Front“ position) and 6 („External Trigger“) set.

`GTL`	Go To Local
`SDC`	Selected Device Clear
`PPC`	Parallel Poll Configure
`GET`	Group Execute Trigger
`TCT`	Take Control
`LLO`	Local Lockout
`DCL`	Device Clear
`PPU`	Parallel Poll Unconfigure
`SPE`	Serial Poll Enable

`SPD`	Serial Poll Disable
`UNT`	Untalk
`UNL`	Unlisten
`SRQ`	Service Request
`L`*`nn`*	Listen address *nn*
`T`*`nn`*	Talk address *nn*
`S`*`nn`*	Secondary Command *nn*
`nnn`	Data Byte *nnn*

Table 1: HP-IB Mnemonics shown in the display.

Terminal Mode

This mode can be used to output messages and data. For this purpose it supports a few escape and control sequences for cursor positioning and screen manipulation.
The terminal must be addressed as a listener. Its address is hardcoded in the source code (currently it is 20).

Characters with codes < 32 are control codes and are generally replaced by a space character, except for:

7	bell	displays a bell symbol
8	backspace	move insertion point left and replace character by a space.
10	line feed	move insertion point down by one line. Scroll upwards if at bottom line.
12	form feed	clear display and home cursor
13	carriage return	move insertion point to first column on same line

Character codes above 127 will be shown in the proprietary character set of the LCD (depending on the character ROM, typically Japanese characters), no translation is performed.

On a HP-85 you can use the following BASIC code

```
PRINTER is 720,120
PRINT CHR$(27)& "J"
PRINT "Hello World"
```

to clear the screen and output the infamous greeting message. Note that the PRINT keyword appends space characters as it was originally designed for the internal printer.

To obtain fully controlled output you should use the SEND keyword like so:

```
SEND 7 ; MTA LISTEN 20 DATA 27,"J"
SEND 7 ; MTA LISTEN 20 DATA "Hello World"
```

Of course these two lines can be merged into a single line.

Implementation of HP and ANSI Escape Sequences

cursor up by one row (will stop at first row)	`HP: ESC A`	
	`VT: ESC [` *n* `A`	where: *n* = number of rows to move (1 if omitted)
cursor down by one row (will stop at last row)	`HP: ESC B`	
	`VT: ESC [` *n* `B`	where: *n* = number of rows to move (1 if omitted)
cursor right by one column (will stop at last column)	`HP: ESC C`	
	`VT: ESC [` *n* `C`	where: *n* = number of columns to move (1 if omitted)
cursor left by one column (will stop at first column)	`HP: ESC D`	
	`VT: ESC [` *n* `D`	where: *n* = number of columns to move (1 if omitted)
home	`HP: ESC H`	
clear screen and home cursor	`HP: ESC J`	
clear part of the screen	`VT: ESC [` *n* `J`	where: *n* = 0: from current (inclusive) to end of screen (default if omitted) *n* = 1: from start of screen to current (inclusive) column *n* = 2: from start of screen to end of screen (complete screen)
clear current line	`HP: ESC K`	
clear part of current line	`VT: ESC [` *n* `K`	where: *n* = 0: from current (inclusive) to last column (default if omitted) *n* = 1: from first to current (inclusive) column *n* = 2: from first to last column (complete line)
insert blank line at current line	`HP: ESC L`	
delete current line and move following lines up	`HP: ESC M`	
delete current character and move following characters left	`HP: ESC P`	
	`VT: ESC [` *n* `P`	where: *n* = number of characters to delete (1 if omitted)
insert mode on	`HP: ESC Q`	
insert mode off	`HP: ESC R`	

save current cursor position	`HP: ESC 7`
	`VT: ESC [ s`	

restore previously saved cursor position	`HP: ESC 8`
	`VT: ESC [ u`	

roll up	`HP: ESC S`	
roll down	`HP: ESC T`	
absolute cursor positioning	`HP: ESC &a` *col* `c` *row* `R`	

	`VT: ESC [` *row* `;` *col* `H`
	`VT: ESC [` *row* `;` *col* `f`
		where: *col* = 1 to n ASCII digits: column [0...19]
		row = 1 to n ASCII digits: row [0...3]

1.1. Theory of Operation

The firmware is triggered by interrupts linked to the DAV and the SRQ lines. Interrupt INT1 is set up to capture the falling edge of the DAV line, which indicates valid data. The falling edge of the SRQ signal triggers INT0 and creates an event when SRQ is asserted.

Inside the interrupt service routine the three 8-bit ports B, C and D are sampled and the eight data bits as well as the eight control line bits are packed into two bytes and stored in two circular FIFO buffers. Thus, each event takes two bytes in the buffers.

Outside of the interrupt routine, in the main loop, the current state of the push button is examined and if pressed the mode is toggled between „Sniffer" and „Terminal". Afterwards the code branches into the appropriate handler for each mode.

Inside these handlers the buffer is checked and if not empty the bytes found there are output. Thus the relatively slow output to the display does not interfere with incoming data. The buffer size is limited to about 600 events due to the 2KByte SRAM size of the Atmega168P chip. This is sufficient for slow data rates like standard instrument control commands or printer output.

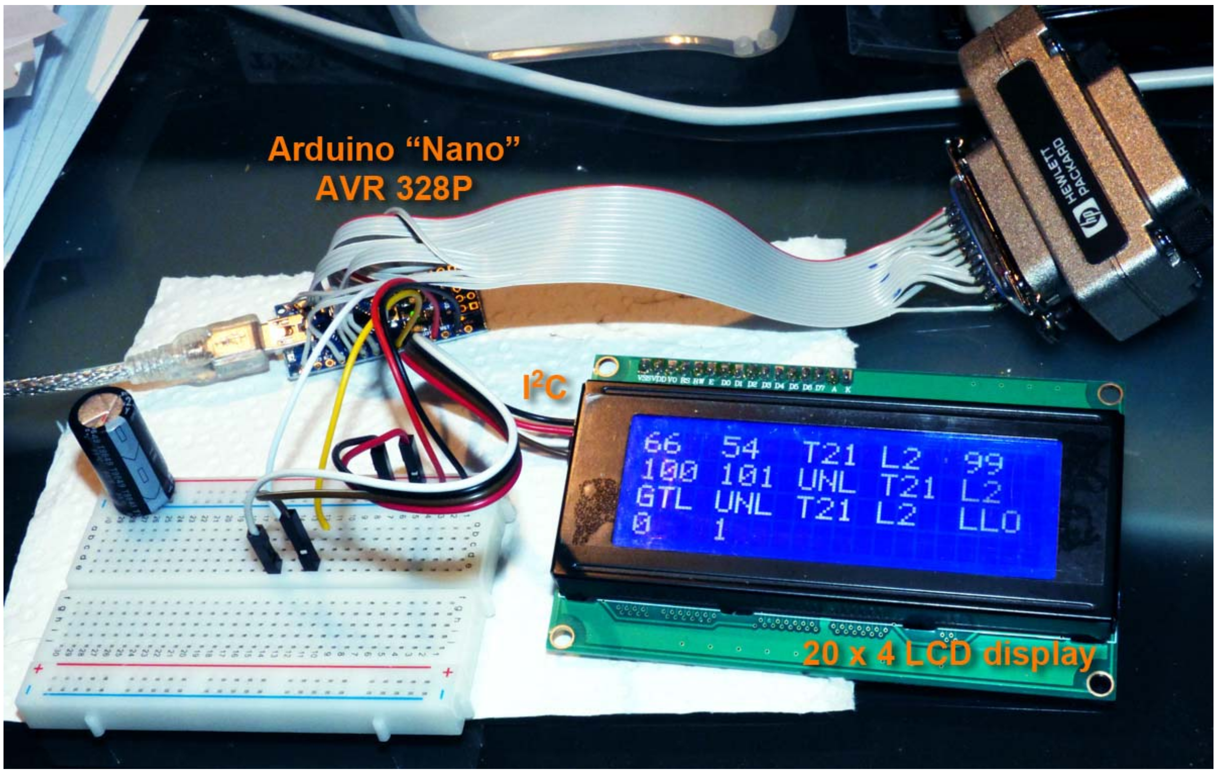
1.2. Theory of Manufacturing

The device is built around an Arduino Nano running at 16 MHz from a 5 V power supply. This chip has just enough lines to provide the desired functions, its SRAM memory is a bit on the smallish side, though. This affects the buffer size, which may overflow in certain applications. Besides the Arduino Nano I used a 4 lines × 20 characters LCD display module with a piggyback I^2C-to-parallel interface. This module also uses the 5 V operating voltage.
I soldered the ribbon cable from the HP-IB connector directly to the Arduino to avoid too many plugs. The drawback is, of course that you cannot easily recycle the Arduino, if that matters to you.
Another 4-wire ribbon cable connects the LCD module with the Arduinos SCL and SDA lines as well as 5 V and GND.
All components have been placed into a small plastic enclosure. For the display module a rectangulat cutout was milled into the top cover. The side of the bottom case received a similar cutout for the female HP-IB connector. The display itself was glued with double sided tape to the shell and a foam block (not shown in the picture) makes sure that it cannot drop into the case in case the tape would let go.

For mode toggling a simple push button was used. As no digital pins were available anymore I used the analog pin A6 for the button. The button was recycled from my old PaintJet operator panel and glued behind a hole in the upper case. When pushed, it connects the analog pin A6 to +5V. A proper „off" state is obtained by pulling A6 low by a 10 kΩ resistor[1]. This resistor was soldered directly between the A6 pad and GND on the ISP connector. A small aluminum disk was turned on my lathe and glued to the top of the button to obtain a pleasing look.
The current mode is remembered in EEPROM and reactivated when powered up again.

To be independent from an USB port a 5.5 mm barrel socket was added and connected to GND and the raw input voltage pin VIN on the Arduino.

[1] The analog inputs can also come in handy when you need more buttons than digital lines. With the help of a small resistor network you can translate each button into a different voltage level and read this with a single pin.



```
66  54  T21 L2  99
100 101 UNL T21 L2
GTL UNL T21 L2  LLO
0   1
```

Figure 3: Test assembly with the LCD module to show the HP-IB command mnemonics and data bytes. The output scrolls to that we see the last part of a conversation.

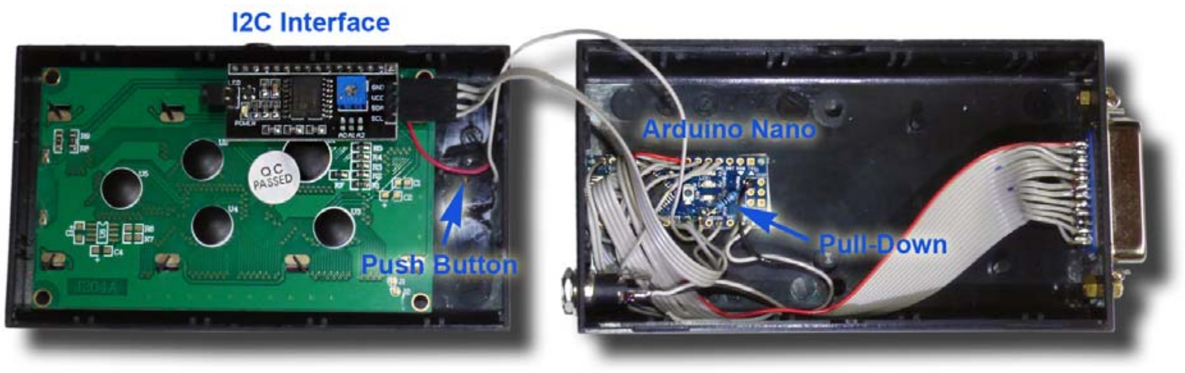


Figure 4: The final assembly of the device shows the few parts: one 4×20 LCD display, one Arduino Nano, one resistor, a push button, the HP-IB connector, a barrel plug and some ribbon cable. It can be powered either via the USB connector or by a 6 V power supply.

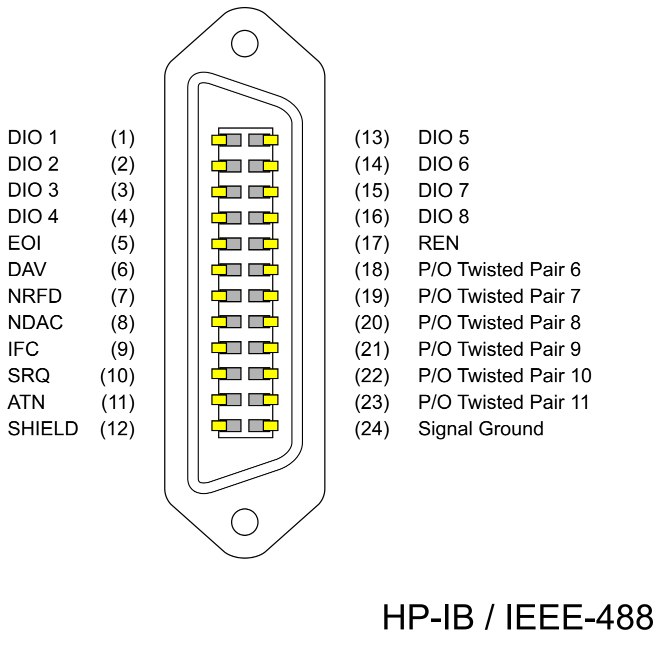


Figure 5: HP-IB Connector pin-out.

The HP-IB lines were connected according to the following table.

Nano	AVR	Ribbon	GP-IB	HP-IB	Mnemonics
D4–D7	PD4–7	1–4	1–4	DIO1–DIO4	**D**ata **I/O**
A0	PC0	9	5	EOI	**E**nd **O**r **I**dentify
D3	PD3 INT1	10	6	DAV	**DA**ta **V**alid
A3	PC3	11	7	NRFD	**N**ot **R**eady **F**or **D**ata
A2	PC2	12	8	NDAC	**N**ot **D**ata **AC**cepted
A1	PC1	13	9	IFC	**I**nter**F**ace **C**lear
D2	PD2 INT0	14	10	SRQ	**S**ervice **R**e**Q**uest
D12	PB4	15	11	ATN	**AT**tentio**N**
		n.c.	12	Shield	
D8–D11	PB0–3	5–8	13–16	DIO5–DIO8	**D**ata **I/O**
D13[2]	PB5	16	17	REN	**R**emote **EN**able
		n.c.	18–23	GND	
GND	GND	17	24	GND	Signal Ground

Table 2: Wiring scheme for HP-IB control and data lines and the ribbon cable.

[2] D13 is connected to the anode of a LED on all Arduino boards. The cathode of the LED is connected through a current limiting resistor to GND. While this is nice for the „Blinky“ example code, it draws current from the HP-IB bus when the REN line is high. Therefore I removed this LED (see Figure 6).

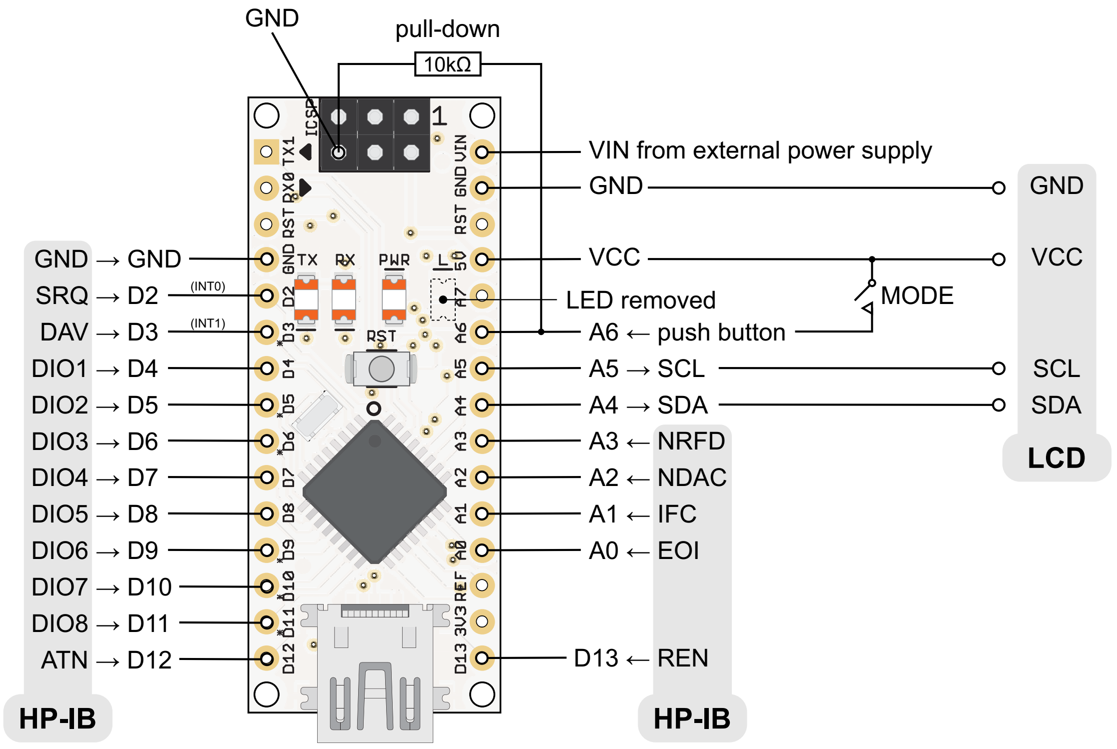


Figure 6: **Wiring scheme of the Arduino Nano.**

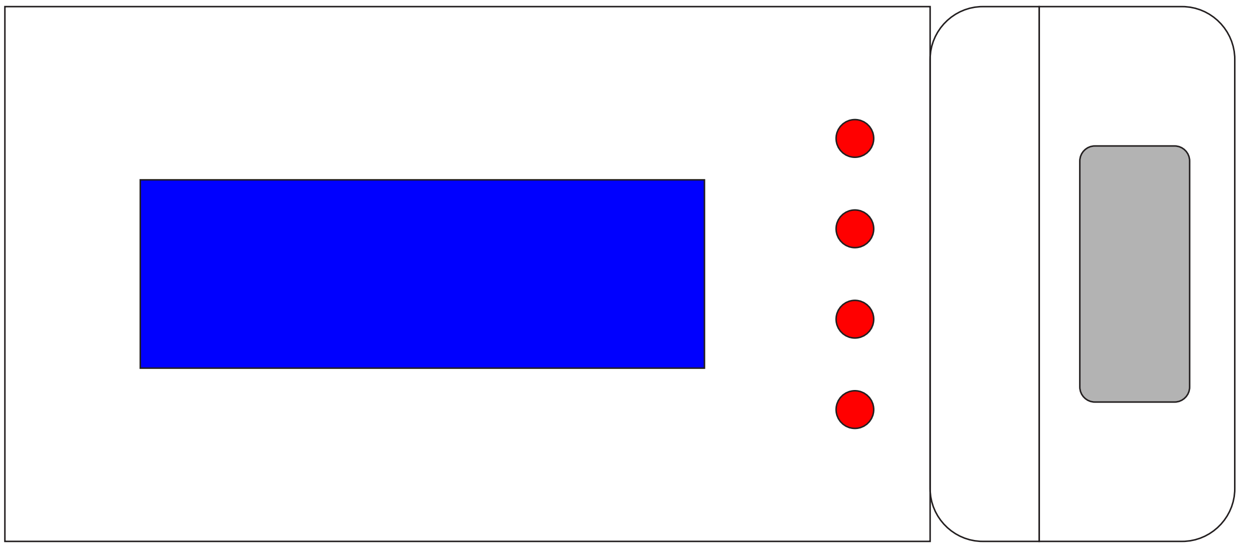


Figure 7: **Cutout plan for my case. The initial plan was to use multiple buttons, but in the end I used only a single button.**

The Arduino Code

The firmware is split into several modules. In principle these are a bit long and could be split into smaller chunks, but I left this exercise to the reader. All these files reside in the same directory, they are not packed into separate „libraries“. Also, I avoided C++ constructs to minimize the memory footprint.

HP_IB_SnifTer.ino	definitions, global variables, setup procedure and the main loop
HPIB.h	some common declarations
handleSniffer.cpp	code for the Sniffer mode
handleTerminal.cpp	code for the Terminal mode

Table 3: The firmware modules.

```
/*
  This file is part of the HP-IB Terminal / HP-IB Sniffer application.

  This application implements a passive device which monitors the HP-IB bus.
  It implements two modes: "Sniffer" and "Terminal". A pushbutton can be used
  to toggle between these modes.

  When the system detects a transition of the DAV to "true" (low) or a transition
  of the SRQ line to true (low)
  it captures the state of the data and control lines and stores them in a buffer.

  Later in its loop function it calls the handler function appropriate for ist
  current mode.

  Martin Hepperle, January 2018
*/


#include "HPIB.h"

#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <EEPROM.h>
#include <avr/eeprom.h>


extern void clearArea( int row_1, int col_1, int row_2, int col_2 );
extern void outputText (int r, int c, char * text );
extern void repaintLCDScreen();
extern void scrollUp();
extern void scrollDown();


#define DEVMODE_TERMINAL 0
#define DEVMODE_SNIFFER  1
int deviceMode = DEVMODE_TERMINAL;


// Port     76543210
// Port B 0b00111111    REN,  ATN,  DIO8, DIO7, DIO6, DIO5
// Port C 0b00001111    NRFD, NDAC, IFC,  EOI
// Port D 0b11111100    DIO4, DIO3, DIO2, DIO1, DAV,  SRQ, 0, 0

// connection of signal lines to Arduino
// Port D
#define SRQ_   2     // PD2 D2
#define DAV_   3     // PD3 D3
// Port B
#define ATN_   12    // PB4 D12
#define REN_   13    // PB5 D13
// Port D
#define EOI_   23    // PC0 A0
#define IFC_   24    // PC1 A1
#define NDAC_  25    // PC2 A2
```

```
#define NRFD_  26    // PC3 A3

// connection of data lines
// Port D
#define DIO1_  4     // PD4 D4
#define DIO2_  5     // PD5 D5
#define DIO3_  6     // PD6 D6
#define DIO4_  7     // PD7 D7
// Port B
#define DIO5_  8     // PB0 D8
#define DIO6_  9     // PB1 D9
#define DIO7_  10    // PB2 D10
#define DIO8_  11    // PB3 D11


// this interrupt is used to monitor the falling edge of SRQ
#define INT_SRQ INT0
// this interrupt is used to monitor the falling edge of DAV
#define INT_DAV INT1

extern void handleSniffer();
extern void handleTerminal();

/*
   Binäre Sketchgröße: 5.838 Bytes (von einem Maximum von 30.720 Bytes)

   After adding variables, you should check buffer size and adapt BUFLEN
   if necessary.
   Note that some (~256 bytes) free space is still needed for the stack in .data

   cd C:\Users\MARTIN~1\AppData\Local\Temp\build7944047484318042240.tmp\HP_IB.cpp.elf
   "D:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avr-size" -C --
mcu=atmega328p HP_IB_SniffTer.cpp.elf

    AVR Memory Usage
    ----------------
    Device: atmega328p

    Program:    8102 bytes (24.7% Full)
    (.text + .data + .bootloader)

    Data:       1676 bytes (81.8% Full)
    (.data + .bss + .noinit)
*/


// we use two bytes to store data and control lines for each event
byte bufControl[BUFLEN];
byte bufData[BUFLEN];
int readPointer  = 0;
int writePointer = 0;
int maxUsed = 0;

LiquidCrystal_I2C lcd(0x27,20,4);  // set the LCD address to 0x27 for 20 x 4 display
// we maintain our own screen buffer
char lcdBuffer[LCD_COLUMNS*LCD_ROWS+1]; // including '\0' byte as terminator
byte idx;

// we reduce SRAM memory footprint by putting some constant strings into FLASH RAM
static char PROGMEM startupTerminal[] = "\377 HP-IB Terminal \377";
static char PROGMEM startupSniffer[]  = "\377 HP-IB  Sniffer \377";
static byte PROGMEM bell[8]      = { 0x04, 0x0e, 0x0e, 0x0e, 0x1f, 0x00, 0x04 };
static byte PROGMEM backslash[8] = { 0x00, 0x10, 0x08, 0x04, 0x02, 0x01, 0x00 };
static byte PROGMEM tilde[8]     = { 0x00, 0x00, 0x09, 0x15, 0x12, 0x00, 0x00 };
static byte PROGMEM rubout[8]    = { 0x10, 0x10, 0x10, 0x1F, 0x10, 0x10, 0x10 };



// ==================================================================================
/*
  This interrupt service routine is called when SRQ is falling low.
  We do nothing here.
 */
ISR(INT0_vect)
{
  if ( deviceMode == DEVMODE_SNIFFER )
  {
    // special case of SRQ trigger
    bufControl[writePointer] = 0xFF;
    bufData[writePointer]    = 0xFF;

    // advance to next slot
    writePointer = (writePointer+1) % BUFLEN;

    checkBuffer();
  }
}

// ====================================================================================

/*
  This interrupt service routine is called when valid data is on the bus
  (DAV drops low).

  We save the state of all control and data lines for later processing
  in the main loop().
 */
ISR(INT1_vect)
{
  byte PB = PINB;
  byte PC = PINC;
  byte PD = PIND;

  // Ports and data bits
  // (read labels from top to bottom)
  //
  //     PB         PC  3210   PD  32
  //  |  ..RA3210   ....NNIE   7654DS..
  //  |    ET           RDFO       AR
  //  V    NN           FACI       VQ
  //                    DC

  // bits in bufControl
  // 0:EOI
  // 1:IFC
  // 2:NDAC
  // 3:NRFD
  // 4:ATN
  // 5:REN
  // 6:SRQ
  // 7:DAV
  bufControl[writePointer] =
    ((PD & 0x0C)<<4) |  // 0x04+0x08       6:SRQ, 7:DAV
     (PB & 0x18)     |  // 0x10+0x20       4:ATN, 5:REN
     (PC & 0x0F);       // 0x01+0x02+0x04  0:EOI, 1:IFC, 2:NDAC, 3:NRFD

  // bufData: DIO (4 high bits PB: 00001111  4 low bits PD: 11110000)
  bufData[writePointer] = ( (PB << 4) | (PD >> 4) ) ^ 0xFF;

  // advance to next slot
  writePointer = (writePointer+1) % BUFLEN;

  // monitor buffer usage
  checkBuffer();
}

// ====================================================================================
/*
 Test buffer usage and check for buffer overrun.
 */
void checkBuffer()
{
  // monitor buffer usage by watermark
  int used = writePointer - readPointer;
  if ( used < 0 )        used += BUFLEN;
  if ( used > maxUsed )  maxUsed = used;

  if ( writePointer == readPointer )
  {
    // column, row
    lcd.setCursor(0,0);
    lcd.print ( "ERROR: BUFFER OVERRUN");
  }
}

// ==================================================================================

/*
 Define a user-defined character from a pattern stored in PROGMEM.
 */
void defineChar ( int n, prog_uint8_t * p )
{
  byte buffer[8];
  // copy from PROGMEM to stack in SRAM
  memcpy_P(buffer, p, 8);
  lcd.createChar ( n, buffer );
}

// ==================================================================================

/*
 Sets the device mnode and shows a corresponding message to the user.
 */
void selectMode ( byte newMode )
{
  char buffer[32];

  // set device mode
  if ( (deviceMode != DEVMODE_SNIFFER) &&
       (deviceMode != DEVMODE_TERMINAL) )
  {
    // out of range, use default
    deviceMode = DEVMODE_TERMINAL;
  }
  else
  {
    deviceMode = newMode;
  }

  if ( deviceMode != eeprom_read_byte(0) )
  {
    // if changed: save for next startup
    eeprom_write_byte ( 0, deviceMode );
  }


  // clear and home cursor
  clearArea ( 0,0, LCD_ROWS-1,LCD_COLUMNS-1);
  memset ( buffer, 0, sizeof(buffer) );
  if ( deviceMode == DEVMODE_TERMINAL )
    memcpy_P(buffer, startupTerminal, strlen_P(startupTerminal));
  else
    memcpy_P(buffer, startupSniffer, strlen_P(startupSniffer));
  outputText ( 3, 1, buffer );
  repaintLCDScreen();
  idx = 0;
}

// ==================================================================================

/*
  Where it all begins...
 */
void setup()
{
  char buffer[32];
  // Serial.begin(115200);

  // need init twice for reliable initialization?
  lcd.init();
  delay(500);
  lcd.init();
  lcd.backlight();

  selectMode ( eeprom_read_byte(0) );

  for ( int i=0 ; i<LCD_ROWS-1 ; i++ )
  {
    delay(150);
    scrollUp();
  }
  delay(150);
  scrollDown();
  delay(150);
  outputText ( 2, 5, "Address 20");  /// should match MY_ADDRESS
  repaintLCDScreen();

  // load special characters
  defineChar ( 0, bell );
  defineChar ( 1, backslash );
  defineChar ( 2, tilde );
  defineChar ( 3, rubout );

  delay(2000);
  clearArea ( 0,0, LCD_ROWS-1,LCD_COLUMNS-1 );

  // ------------------------------
  // All ports are inputs by default after reset.
  // Nevertheless we want to be sure...
  // You should NEVER set any port connected directly to the HP-IB bus to OUTPUT.
  // for OUTPUT you should use at least a resistor network or even better
  // proper line drivers!

  //                          7     6     5     4     3     2     1     0
  // Port B 0b11000000 0xC0               REN,  ATN,  DIO8, DIO7, DIO6, DIO5
  // Port C 0b11110000 0xF0   0,    RSET, SCL,  SDA,  NRFD, NDAC, IFC,  EOI
  // Port D 0b00000011 0x03   DIO4, DIO3, DIO2, DIO1, DAV,  SRQ,  TXD,  RXD

  // PB 0-5 == input
  DDRB &= 0xC0;
  // PC 0-3 == input
  DDRC &= 0xF0;
  // PD 2-7 == input
  DDRD &= 0x03;

  // diable interrupts first to avoid interrupt on change
  EIMSK = 0;
  // setup interrupts 0 and 1
  EICRA = (1<<ISC11) | (0<<ISC10) | (FALLING<<ISC01) | (FALLING<<ISC00) ;
  // finally enable the interrupts 1 and 0
  EIMSK |= (1 << INT_DAV) | (1 << INT_SRQ);

  // ADC is used for key check
```

```
#define ADC_FREE ((1<<ADEN) | (1<<ADSC) | (1<<ADATE) | (1<<ADPS2) | (1<<ADPS1) |
(1<<ADPS0))

    // disable ADC
    ADCSRA &= ~(1<<ADEN);
    ADMUX = (1<<REFS0) | 0x06;  // select 5V reference and ADC6
    ADCSRB = 0;
    // enable ADC
    ADCSRA = ADC_FREE;
}

// ==================================================================================

/*
  The main loop - we could even insert a while(TRUE) construct here is we had
  to speed it up
*/
void loop()
{
  // Detect button press: the button pulls ADC 6 high.
  // The ADC conversion will be in 0...1023.
  // For speed we test only the high bit.
  // If is is > 2 the values is > 512, i.e. "high".
  if ( ADCH > 2 )
  {
    selectMode( (deviceMode == DEVMODE_SNIFFER) ?
                 DEVMODE_TERMINAL : DEVMODE_SNIFFER );
    delay(100);
    scrollUp();
    delay(100);
    scrollUp();
    delay ( 800 );
    clearArea ( 0,0, LCD_ROWS-1,LCD_COLUMNS-1 );
    idx = 0;
  }

  if ( deviceMode == DEVMODE_SNIFFER )
    handleSniffer();
  else
    handleTerminal();
}

// ==================================================================================
```

Listing 1 HP_IB_Snifter.ino.

```
/*

  This file is part of the HP-Ib Terminal / HP-IB Sniffer application.

  Martin Hepperle, January 2018
*/
#include <LiquidCrystal_I2C.h>


typedef unsigned char byte;
typedef byte boolean;


#define FALSE 0
#define TRUE  1

// a buffer of 512 bytes should be just right for the average HP-85.
#define BUFLEN 600

// HP-IB commands
#define GTL_ 0x01  // Go To Local
#define SDC_ 0x04  // Selected Device Clear
```

```
#define PPC_ 0x05  // Parallel Poll Clear
#define GET_ 0x08  // Group Execute Trigger
#define TCT_ 0x09  // Take Control
#define LLO_ 0x11  // Local Lockout
#define DCL_ 0x14  // Device Clear
#define PPU_ 0x15  // Parallel Poll Unconfigure
#define SPE_ 0x18  // Serial Poll Enable
#define SPD_ 0x19  // Serial Poll Disable
#define UNT_ 0x5F  // Untalk
#define UNL_ 0x3F  // Unlisten

extern int readPointer;
extern int writePointer;
extern byte bufControl[];
extern byte bufData[];

#define LCD_ROWS     4
#define LCD_COLUMNS 20


// we maintain our own screen buffer
extern LiquidCrystal_I2C lcd;  // set the LCD address to 0x27 for a 20 chars and 4
line display
extern char lcdBuffer[]; // including '\0' byte for print() terminator
extern byte idx;


// state machine for terminal emulation
#define STATE_IDLE    0
#define STATE_LISTEN  1

// states
extern byte state;  // we start in idle mode

#define MY_ADDRESS  20   // for Terminal mode, se also string in setup()
```

Listing 2 HP_IB.h.

```
/*
  This file is part of the HP-Ib Terminal / HP-IB Sniffer application.

  HP-IB Sniffer
  =============

  The Sniffer is a passive device which monitors the HP-IB bus.

  In handleSniffer() function it outputs the mnemonics of the command or the value
  of the data byte to the LCD screen. All output is arranged in blocks of four
  characters. The screen scrolls upwards when the last row is filled.

  Martin Hepperle, January 2018
 */

#include "HPIB.h"

#include <string.h>   // memcpy

extern void scrollUp(); // in handleTerminal.cpp

// ==================================================================================

/*
  Output a command in form of its 3 character mnemonics.
 */
void outOP( char * mnemonics)
{
  char * p = mnemonics;
  // Monitor mode:
  // output a block of up to 3 characters (the first 3 characters in mnemonics

  if ( idx == LCD_ROWS*LCD_COLUMNS )
  {
    // beyond bottom row: scroll up and start in last row
    scrollUp();
    idx = (LCD_ROWS-1)*LCD_COLUMNS;
  }

  // row and column
  int r = idx / LCD_COLUMNS;
  int c = idx % LCD_COLUMNS;

  // update screen
  lcd.setCursor(c,r);

  // output up to 3 characters into cell addressed by idx
  for ( int i=idx ; i<idx+3 ; i++ )
  {
    if ( *p )
    {
      // output character
      lcdBuffer[i] = *p++;
    }
    else
    {
      // fill empty cells
      lcdBuffer[i] = ' ';
    }
    lcd.write(lcdBuffer[i]);
  }

  // advance to next cell block
  idx += 4;

}

// ==================================================================================

void handleSniffer()
{
  if ( readPointer != writePointer )
  {
    byte CTL = bufControl[readPointer];
    byte DIO = bufData[readPointer];
    readPointer = (readPointer+1) % BUFLEN;

    // bits in bufControl
    // 0:EOI
    // 1:IFC
    // 2:NDAC
    // 3:NRFD
    // 4:ATN
    // 5:REN
    // 6:SRQ
    // 7:DAV

    // negative logic for all: low: TRUE
    // EOI - End Or Inquire
    byte EOI = (CTL & 0x01) == 0;
    // IFC - Interface Clear
    byte IFC = (CTL & 0x02) == 0;
    // NDAC - No Data Accepted
    byte NDAC = (CTL & 0x04) == 0;
    // NRFD - Not Ready For Data
    byte NRFD = (CTL & 0x08) == 0;
    // ATN - ATentioN
    byte ATN = (CTL & 0x10) == 0;
    // REN - Remote Enable
    byte REN = (CTL & 0x20) == 0;
    // SRQ - Service Request
    byte SRQ = (CTL & 0x40) == 0;
    // DAV - Dava Valid
    byte DAV = (CTL & 0x80) == 0;
    // lower 5 bits (0...31)
    byte ADDR = DIO & 0x1F;

    // special case of SRQ trigger This is encoded as \0 in both buffers
    if ( CTL == 0xFF && DIO == 0xFF )
    {
      outOP("SRQ");
    }
    else if ( ATN )
    {
      // a command byte
      switch ( DIO )
      {
        case 0x01:   outOP("GTL");  break;
        case 0x04:   outOP("SDC");  break;
        case 0x05:   outOP("PPC");  break;
        case 0x08:   outOP("GET");  break;
        case 0x09:   outOP("TCT");  break;
        case 0x11:   outOP("LLO");  break;
        case 0x14:   outOP("DCL");  break;
        case 0x15:   outOP("PPU");  break;
        case 0x18:   outOP("SPE");  break;
        case 0x19:   outOP("SPD");  break;
        case 0x5F:   outOP("UNT");  break;
        case 0x3F:   outOP("UNL");  break;

        default:
          // mask Listen, Talk, Secondary bits
          byte LTS = DIO & (3<<5);

          if ( LTS == (1<<5) )
          {
            // x01aaaaa
            char out[4];
            out[0] = 'L';
            itoa( (int)ADDR, out+1, 10 );
            outOP(out);
          }
          else if ( LTS == (2<<5)  )
          {
            // x10aaaaa
            char out[4];
            out[0] = 'T';
            itoa( (int)ADDR, out+1, 10 );
            outOP(out);
          }
          else if ( LTS == (3<<5)  )
          {
            // x11aaaaa
            char out[4];
            out[0] = 'S';
            itoa( ADDR, out+1, 10 );
            outOP(out);
          }
          else
          {
            // never happens
          }
          break;
      }
    }
    else
    {
      // a normal data byte (up to 3 digits plus \0
      char out[4];
      itoa( (int)DIO, out, 10 );
      outOP(out);
    }
  }
}

// ==================================================================================
```

Listing 3 handleSniffer.cpp.

```
/*
  This file is part of the HP-IB Terminal / HP-IB Sniffer application.

  HP-IB Terminal
  ==============

  The Terminal is a passive device which monitors the HP-IB bus. It is addressable
  and can be used as a small character output device.

  The implemented escape sequences follow HP and some ANSI sequences as follows:
  (many of the HP sequences equal VT52, ANSI sequences equal VT100)

  cursor up by one row (will stop at row 0)
  HP: ESC A
  VT: ESC [ n A
  where n: omitted == 1, else: number of rows to move

  cursor down by one row (will stop at row LCD_ROWS-1)
  HP: ESC B
  VT: ESC [ n B
  where n: omitted == 1, else: number of rows to move

  cursor right by one column (will stop at column LCD_COLS-1)
  HP: ESC C
  VT: ESC [ n C
  where n: omitted == 1, else: number of columns to move

  cursor left by one column (will stop at column 0)
  HP: ESC D
  VT: ESC [ n D
  where n: omitted == 1, else: number of columns to move

  home
  HP: ESC H

  erase screen and home cursor
  HP: ESC J

  erase part of the screen
  VT: ESC [ n J
  where n: omitted or 0: from current (inclusive) to end of screen
                      1: from start of screen to current (inclusive) column
                      2: from start of screen to end of screen (complete screen)

  erase in current line
  HP: ESC K
  erase part of current line
  VT: ESC [ n K
  where n: omitted or 0: from current (inclusive) to last column
                      1: from first to current (inclusive) column
                      2: from first to last column (complete line)

  insert blank line at current line
  HP: ESC L
  delete current line and move following lines up
  HP: ESC M

  delete current character and move following characters left
  HP: ESC P
  VT: ESC [ n P
  where n: omitted == 1, else: number of characters to delete

  insert mode on
  HP: ESC Q

  insert mode off
  HP: ESC R

  roll up
  HP: ESC S

  roll down
  HP: ESC T

  save curent cursor position
  HP: ESC 7
  VT: ESC [ s

  restore cursor position from position saved by ESC 7
  HP: ESC 8
  VT: ESC [ u

  absolute cursor positioning
  HP: ESC &a col c row R
  VT: ESC [ row ; col H
  VT: ESC [ row ; col f
  where:
  col = 1 to n ASCII digits: column [0...LCD_COLS-1]
  row = 1 to n ASCII digits: row [0...LCD_ROWS-1]


  Characters with codes < 32 are control codes and are generally replaced by
  a space character, except for:
  7  bell:             displays a bell symbol
  8  backspace:        move insertion point left and replace character by a space.
                       Next character will overwrite the blanked character.
  10 line feed:        move insertion point down by one line.
                       Scrolls upwards if at bottom line.
  12 form feed:        clear display and home cursor
  13 carriage return:  move insertion point to first column on same line

  Martin Hepperle, January 2018

*/

#include "HPIB.h"

// states
byte state = STATE_IDLE;  // we start in idle mode
byte inESCape = FALSE;    // we start not inside an escape sequence
byte insertMode = FALSE;  // we start not in insert mode
byte idxSave;

extern void repaintLCDScreen();
extern byte parseInteger ( char ** ppCol, int * pResult );
extern void cursorLeft();
extern void cursorRight();
extern void cursorUp();
extern void cursorDown();
extern void insertLine();
extern void deleteLine();
extern void insertCharacter( char newChar );
extern void deleteCharacter();
extern void scrollUp();
extern void scrollDown();
extern void clearArea( int row_1, int col_1, int row_2, int col_2 );

// ==================================================================================

/*
 Parse an integer number starting at *ppCol and return the result in *pResult.
 Returns FALSE if no digist are found. In this case *pResult = 0.
 If digits are found, *ppCol is updated and points to the first non-digit character.
 */
byte parseInteger ( char ** ppCol, int * pResult )
{
  int c = 0;
  char * pCol = *ppCol;
  boolean ret = FALSE;

  while ( *pCol >= '0' && *pCol <= '9' )
  {
    c = c*10;
    c = c + (*pCol - '0');
    pCol++;
    ret = TRUE;
  }

  // return column of next non-digit character
  *ppCol = pCol;

  // return number
  *pResult = c;

  return ret;
}

// ==================================================================================

/*
 Move the cursor LEFT by one column, until the first column is reached.
 Does not wrap to the previous line.
 */
void cursorLeft()
{
  int first = idx / LCD_COLUMNS;
  // starting index of this line
  first = first * LCD_COLUMNS;
  if ( idx > first ) idx--;
}

// ==================================================================================

/*
 Move the cursor RIGHT by one column, until the last column is reached.
 Does not wrap to the previous line.
 */
void cursorRight()
{
  int last = idx / LCD_COLUMNS;
  // last index in this line
  last = last * LCD_COLUMNS + LCD_COLUMNS-1;
  if ( idx < last ) idx++;
}

// ==================================================================================

/*
 Move the cursor UP by one row, until the top margin is reached.
 Does not wrap.
 */
```

```
void cursorUp()
{
  if ( idx >= LCD_COLUMNS ) idx -= LCD_COLUMNS;
}

// ===================================================================================

/*
 Move the cursor DOWN by one row, until the bottom line is reached.
 Does not wrap.
 */
void cursorDown()
{
  if ( idx < (LCD_ROWS-1)*LCD_COLUMNS ) idx += LCD_COLUMNS;
}

// ===================================================================================

/*
 Deletes the character at the cursor position and moves trailing characters left.
 Does not wrap to next line.
 */
void deleteCharacter()
{
  // row and column
  int r = idx / LCD_COLUMNS;
  int c = idx % LCD_COLUMNS;

  // last index in this line
  int last = r * LCD_COLUMNS + LCD_COLUMNS-1;

  int i = idx;

  // update screen
  lcd.setCursor(c,r);
  while ( i < last )
  {
      lcdBuffer[i] = lcdBuffer[i+1];
      lcd.write(lcdBuffer[i]);
      i++;
  }
  // blank trailing character
  lcdBuffer[last] = ' ';
  lcd.write(' ');
}

// ===================================================================================

/*
 Inserts the character at the cursor position and moves trailing characters right.
 Does not wrap to next line.
 */
void insertCharacter( char newChar )
{
  // row and column
  int r = idx / LCD_COLUMNS;
  int c = idx % LCD_COLUMNS;

  // last index in this line
  int i = r * LCD_COLUMNS + LCD_COLUMNS-1;

  // move trailing characters right
  while ( i > idx )
  {
      lcdBuffer[i] = lcdBuffer[i-1];
      i--;
  }
  // insert new character
  lcdBuffer[idx] = newChar;
  // update screen
  // last index in this line
  i = r * LCD_COLUMNS + LCD_COLUMNS-1;
  lcd.setCursor(c,r);
  int k=idx;
  while ( k < i )
  {
    lcd.write(lcdBuffer[k]);
    k++;
  }
}

// ==================================================================================

/*
 Clear the area from column 'row_1/col_1' to 'row_2/col_2' (inclusive).
 */
void clearArea( int row_1, int col_1, int row_2, int col_2 )
{
  if ( row_1 >= LCD_ROWS ) row_1 = LCD_ROWS-1;
  if ( row_2 >= LCD_ROWS ) row_2 = LCD_ROWS-1;
  if ( col_1 >= LCD_COLUMNS ) col_1 = LCD_COLUMNS-1;
  if ( col_2 >= LCD_COLUMNS ) col_2 = LCD_COLUMNS-1;

  int from = row_1*LCD_COLUMNS + col_1;
  int to   = row_2*LCD_COLUMNS + col_2;

  for ( int i=from ; i<=to ; i++)
    lcdBuffer[i] = ' ';

  // update LCD screen
  repaintLCDScreen();
}

// ==================================================================================

/*
 Scroll the contents of the display buffer up and insert a blank line at the bottom.
 Refreshes the display.
 */
void scrollUp()
{
  // move 3 lines up
  memcpy ( lcdBuffer, lcdBuffer+LCD_COLUMNS, (LCD_ROWS-1)*LCD_COLUMNS );
  // blank last line
  memset ( lcdBuffer+(LCD_ROWS-1)*LCD_COLUMNS, ' ', LCD_COLUMNS);

  // update LCD screen
  repaintLCDScreen();
}

// ==================================================================================

/*
 Scroll the contents of the display buffer down and insert a blank line at the top.
 Refreshes the display.
 */
void scrollDown()
{
  // first move 3 down
   char * src = lcdBuffer + (LCD_ROWS-1)*LCD_COLUMNS;
   char * dst = src + LCD_COLUMNS;

   for ( int i=0 ; i<(LCD_ROWS-1)*LCD_COLUMNS ; i++ )
   {
     *(--dst) = *(--src);
   }
  // blank first line
  memset ( lcdBuffer, ' ', LCD_COLUMNS);

  // update LCD screen
  repaintLCDScreen();
}

// ===================================================================================

/*
 Write the given string to the screen buffer.
 Does not refresh the display.
 */
void outputText (int r, int c, char const * text )
{
  if ( r >= LCD_ROWS )    r = LCD_ROWS-1;
  if ( c >= LCD_COLUMNS ) c = LCD_COLUMNS-1;

  int start = r*LCD_COLUMNS + c;
  int len = strlen(text);
  if ( start+len > LCD_ROWS*LCD_COLUMNS ) len = LCD_ROWS*LCD_COLUMNS - start;

  memcpy ( lcdBuffer+start, text, len );
}

// ===================================================================================

/*
 The 20 x 4 LCD has an interleaved memory scheme.

 Therefore we output our linear screen buffer in that way.
 For other displays this routine has to be rewritten.
 */

void repaintLCDScreen()
{
  byte i;

  // column, row
  lcd.setCursor(0,0);

  //  display        memory
  //  0 ... 19        0 ... 19   20 ... 39
  // 40 ... 59       40 ... 59   60 ... 79
  // 20 ... 39
  // 60 ... 79
  char * p = lcdBuffer;  for ( i=0 ; i<LCD_COLUMNS ; i++ )  lcd.write(*p++);
  p +=   LCD_COLUMNS;    for ( i=0 ; i<LCD_COLUMNS ; i++ )  lcd.write(*p++);
  p -= 2*LCD_COLUMNS;    for ( i=0 ; i<LCD_COLUMNS ; i++ )  lcd.write(*p++);
  p +=   LCD_COLUMNS;    for ( i=0 ; i<LCD_COLUMNS ; i++ )  lcd.write(*p++);
}

// ===================================================================================

/*
 Delete the current line and move following lines up.
 */
void deleteLine()
{
  int l = idx / LCD_COLUMNS;
  // starting index of this line
  l = l * LCD_COLUMNS;

  // move following lines up
  while ( l < (LCD_ROWS-1)*LCD_COLUMNS )
  {
    memcpy ( lcdBuffer+l, lcdBuffer+l+LCD_COLUMNS, LCD_COLUMNS );
    l = l + LCD_COLUMNS;
  }
  // blank last line
  memset ( lcdBuffer+(LCD_ROWS-1)*LCD_COLUMNS, ' ', LCD_COLUMNS);

  repaintLCDScreen();
}

// ==================================================================================

/*
 Delete the current line and move following lines up.
 */
void insertLine()
{
  int ins = idx / LCD_COLUMNS;
  // starting index of this line
  ins = ins * LCD_COLUMNS;

  int dst = (LCD_ROWS-1)*LCD_COLUMNS;

  while ( dst > ins )
  {
    int src = dst - LCD_COLUMNS;
    memcpy ( lcdBuffer+dst, lcdBuffer+src, LCD_COLUMNS );
    dst = src;
  }
  memset ( lcdBuffer+ins, ' ', LCD_COLUMNS );

  repaintLCDScreen();
}

// ==================================================================================

void handleTerminal()
{
  // these are used to collect the data for cursor positioning
  static char escBuffer[32];
  static int escIdx = 0; // if 0: we are NOT in a cursor positioning escape sequence

  if ( readPointer != writePointer )
  {
    byte PB = bufControl[readPointer];
    byte DIO = bufData[readPointer];
    readPointer = (readPointer+1) % BUFLEN;

    if ( PB == 0xFF && DIO == 0xFF )
    {
      // SRQ: do nothing with it
      return;
    }

    // bits in PB:
    // 0:EOI
    // 1:IFC
    // 2:NDAC
    // 3:NRFD
    // 4:ATN
    // 5:REN
    // 6:SRQ
    // 7:DAV

    // negative logioc for all: low: TRUE
    // EOI - End Or Inquire
    byte EOI = (PB & 0x01) == 0;
    // IFC - Interface Clear
    byte IFC = (PB & 0x02) == 0;
    // NDAC - No Data Accepted
    byte NDAC = (PB & 0x04) == 0;
    // NRFD - Not Ready For Data
    byte NRFD = (PB & 0x08) == 0;
    // ATN - ATentioN
    byte ATN = (PB & 0x10) == 0;
    // REN - Remote Enable
    byte REN = (PB & 0x20) == 0;
    // SRQ - Service Request
    byte SRQ = (PB & 0x40) == 0;
    // DAV - Dava Valid
    byte DAV = (PB & 0x80) == 0;
    // lo5 5 bits
    byte ADDR = DIO & 0x1F;

    if ( ATN )
    {
      // ATN line is asserted (low): this is a command
      switch ( DIO )
      {
        case GTL_:                        break;  // Go To Local
        case SDC_:                        break;  // Selected Device Clear
        case PPC_:                        break;  // Parallel Poll Clear
        case GET_:                        break;  // Group Execute Trigger
        case TCT_:                        break;  // Take Control
        case LLO_:                        break;  // Local Lockout
        case DCL_:                        break;  // Device Clear
        case PPU_:                        break;  // Parallel Poll Unconfigure
        case SPE_:                        break;  // Serial Poll Enable
        case SPD_:                        break;  // Serial Poll Disable
        case UNT_:                        break;  // Untalk
        case UNL_:  state = STATE_IDLE;   break;  // Unlisten

        default:
          // mask Listen, Talk, Secondary bits
          byte LTS = DIO & (3<<5);

          if ( LTS == (1<<5) )
          {
            // x01aaaaa
            if ( ADDR == MY_ADDRESS )
            {
              state = STATE_LISTEN;
             }
          }
          else if ( LTS == (2<<5)  )
          {
            // x10aaaaa
          }
          else if ( LTS == (3<<5)  )
          {
            // x11aaaaa
          }
          else
          {
            // this never happens
          }
          break;
      }
    }
    else
    {
      // ATN is high: "normal" data

      if ( state == STATE_LISTEN )
      {
        // I am listening

        if ( escIdx > 0 )
        {
          // accumulating parameters for ESC ... sequence
          if ( escIdx < sizeof(escBuffer) )
            escBuffer[escIdx++] = DIO;
          else
            escIdx = 0;  // overflow, forget sequence

          // test for end of sequence
          if ( escBuffer[0] == '&' )
          {
            // in ESC & sequence

            if ( DIO == 'R')
            {
              // end of sequence indicated by capital "R"
              escIdx = 0; // end of escape sequence

              // escBuffer = "&a col c row R" (column and row are 0-based)
              //        e.g. "&a19c0R"
              if ( escBuffer[1] == 'a' )
              {
                // skip '&a'
                char * pCol = escBuffer+2;
                int r, c;
                // read digits for column (0...19)

                if ( parseInteger ( &pCol, &c ) )
                {
                  if ( *pCol == 'c'  )
                  {
                     // skip 'c'
                    pCol++;
                    // read digits for row (0...3)
                    if ( parseInteger ( &pCol, &r ) )
                    {
                      idx = r * LCD_COLUMNS + c;
                    }
                  }
                }
              }
            }
          }
          else if ( escBuffer[0] == '[' )
          {
            // VT100/ANSI escape sequence

            if ( (DIO >= 'A' && DIO <= 'D') || DIO == 'P' )
            {
              // in ESC [ n A/B/C/D/P sequence
              escIdx = 0; // end of escape sequence
              // skip '['
              char * pCol = escBuffer+1;
              int n;

              if ( ! parseInteger ( &pCol, &n ) )
              {
                // if no number was given: default to 1
                n = 1;
              }

              // action!
              while ( n-- )
              {
                if (      DIO == 'A')
                  cursorUp();
                else if ( DIO == 'B')
                  cursorDown();
                else if ( DIO == 'C')
                  cursorRight();
                else if ( DIO == 'D')
                  cursorLeft();
                else if ( DIO == 'P')
                  deleteCharacter();
              }
            }
            else if ( DIO == 'H' || DIO == 'f'  )
            {
              // ESC [ row ; col H
              // ESC [ row ; col f
              escIdx = 0; // end of escape sequence
              // position cursor at row,column (1-based)
              // skip '['
              char * pCol = escBuffer+1;
              int r, c;

              // read digits for row (0...3)
              if ( parseInteger ( &pCol, &r ) )
              {
                if ( *pCol == ';'  )
                {
                   // skip ';'
                  pCol++;
                  // read digits for column (0...19)
                  if ( parseInteger ( &pCol, &c ) )
                  {
                    // adjust for 0-base
                    r--;
                    c--;
                    idx = r * LCD_COLUMNS + c;
                  }
                }
              }
            }
            else if ( DIO == 'J' )
            {
              // erase in display
              // ESC [ n J
              escIdx = 0; // end of escape sequence
              // erase part of line
              // skip '['
              char * pCol = escBuffer+1;
              // row and column
              int r = idx / LCD_COLUMNS;
              int c = idx % LCD_COLUMNS;

              // read digit (or zero)
              int n;
              parseInteger ( &pCol, &n );

              // |-----X--------------|
              if ( n == 0 )
              {
                // from current (inclusive) to end of screen
                // |-----               |
                clearArea( r, c, LCD_ROWS-1, LCD_COLUMNS-1 );
              }
              else if ( n == 1 )
              {
                // from start of screen to current (inclusive) position
                // |      --------------|
                clearArea( 0, 0, r, c );
              }
              else if ( n == 2 )
              {
                // |                    |
                // from start of screen to end of screen
                clearArea( 0, 0, LCD_ROWS-1, LCD_COLUMNS-1 );
              }
            }
            else if ( DIO == 'K' )
            {
              // ESC [ n K
              escIdx = 0; // end of escape sequence
              // erase part of line
              // skip '['
              char * pCol = escBuffer+1;
              // row and column
              int r = idx / LCD_COLUMNS;
              int c = idx % LCD_COLUMNS;

              // read digit (or zero)
              int n;
              parseInteger ( &pCol, &n );

              // |-----X--------------|
              if ( n == 0 )
              {
                // from current (inclusive) to last column
                // |-----              |
                clearArea( r, c, r, LCD_COLUMNS-1 );
              }
              else if ( n == 1 )
              {
                // from start to current (inclusive) column
                // |      --------------|
                clearArea( r, 0, r, c );
              }
              else if ( n == 2 )
              {
                // |                    |
                // from start to last column
                clearArea( r, 0, r, LCD_COLUMNS-1 );
              }
            }
            else if ( DIO == 's' )
            {
              // ESC [ s save current cursor position
              escIdx = 0; // end of escape sequence
              idxSave = idx;
            }
            else if ( DIO == 'u' )
            {
              // ESC [ u unsave current cursor position
              escIdx = 0; // end of escape sequence
              idx = idxSave;
            }
          }
        }
        else if ( inESCape )
        {
          // escape sequence was started, check character after ESC

          inESCape = FALSE;

          switch ( DIO )
          {
            case 'A':  // ESC A = CURSOR UP
              cursorUp();
              break;

            case 'B':  // ESC B = CURSOR down
              cursorDown();
              break;

            case 'C':  // ESC C = CURSOR RIGHT
              cursorRight();
              break;

            case 'D':  // ESC D = CURSOR LEFT
              cursorLeft();
              break;
            case 'H':  // ESC H = HOME
              idx = 0;
              break;

            case 'J':  // ESC J: CLEAR SCREEN and home cursor
              clearArea( 0, 0, LCD_ROWS-1, LCD_COLUMNS-1 );
              idx = 0;
              break;

            case 'K':  // ESC K = CLEAR (current) LINE
              clearArea( idx/LCD_COLUMNS, 0, idx/LCD_COLUMNS, LCD_COLUMNS-1 );
              break;

            case 'L':  // ESC L = insert at (current) LINE
              insertLine();
              break;

            case 'M':  // ESC L = delete (current) LINE
              deleteLine();
              break;

            case 'P':  // ESC P = delete one character
              deleteCharacter();
              break;

            case 'Q':  // ESC Q = insert mode on
              insertMode = TRUE;
              break;

            case 'R':  // ESC R = insert mode off
              insertMode = FALSE;
              break;

            case 'S':  // ESC S = roll up
              scrollUp();
              break;

            case 'T':  // ESC T = roll down
              scrollDown();
              break;

            case '7':  // ESC 7 = save current position
              idxSave = idx;
              break;

            case '8':  // ESC 8 = restore current position
              idx = idxSave;
              break;

            case '&':  // ESC & ... R  start of absolute positioning sequence
            case '[':  // ESC [ ...   start of ANSI/VT100 sequence
                       // both collect more characters until a terminator is found
              escBuffer[0] = DIO;
              escIdx++; // indicates that we are in a multi-character escape sequence
              break;

            default:
            break;
          }
        }
        else if ( DIO == 27 ) // ESC
        {
          // start a (single character or multi-character) escape sequence
          inESCape = TRUE;
        }
        else
        {
          // regular character output
```

```
        // current position
        int r = idx / LCD_COLUMNS;
        int c = idx % LCD_COLUMNS;

        // translate certain characters
        switch ( DIO )
        {
          case 0:
          case 1:
          case 2:
          case 3:
          case 4:
          case 5:
          case 6:
            // 7:      ^J BEL -> handled below
            // 8:      ^H BS  -> handled below
          case 9:  // ^I TAB
            // 10:     ^J LF  -> handled below
          case 11: // ^K
            // 12:     ^L FF  -> handled below
            // 13:     ^M CR  -> handled below
          case 14: // ^N
          case 15: // ^O
          case 16: // ^P
          case 17: // ^Q resume output
          case 18: // ^R
          case 19: // ^S stop output
          case 20: // ^T
          case 21: // ^U
          case 22: // ^V
          case 23: // ^W
          case 24: // ^X
          case 25: // ^Y
          case 26: // ^Z
          case 27: // ESC
          case 28: //
          case 29: //
          case 30: //
            // convert these control characters to blank
            DIO = 32;
            break;

          case 31: //
            // convert to filled rectangle
            DIO = 255;
            break;

          case 7:  // ^G bell, replace by user defined symbol 0: bell
            DIO = 0;
            break;

          case 92:
            // replace Yen symbol by user defined symbol 1: backslash
            DIO = 1;
            break;

          case 126:
            // replace by user defined symbol 2: tilde
            DIO = 2;
            break;

          case 127:
            // replace by user defined symbol 3: rubout
            DIO = 3;
            break;
        }

        // now handle some special characters
```

```
          if ( DIO == 8 )
          {
            // backspace
            if ( idx == LCD_ROWS*LCD_COLUMNS )
            {
              // behind last character on display
              idx--;
              c = LCD_COLUMNS-1;
              r = LCD_ROWS-1;
            }
            else
            {
              // backspace
              // move left and replace character by space character
              cursorLeft();
              c--;
            }
            DIO = 32;
            lcdBuffer[idx] = DIO;
            // update cell on display
            lcd.setCursor(c,r);
            lcd.write((char)DIO);
          }
          else if ( DIO == 12 )
          {
            // form feed: clear and home cursor
            clearArea ( 0,0, LCD_ROWS-1,LCD_COLUMNS-1 );
            idx = 0;
          }
          else if ( DIO == 13 )
          {
            // carriage return: back to start of row
            idx = r*LCD_COLUMNS;
          }
          else if ( DIO == 10 )
          {
            // line feed
            if ( r >= LCD_ROWS-1 )
            {
              scrollUp();
              idx = (LCD_ROWS-1)*LCD_COLUMNS + c;
            }
            else
            {
              cursorDown();
            }
          }
          else
          {
            // any other character

            if ( idx >= LCD_ROWS*LCD_COLUMNS )
            {
              // beyond bottom row: scroll up and start in last row
              scrollUp();
              idx = (LCD_ROWS-1)*LCD_COLUMNS;
              c = 0;
              r = LCD_ROWS-1;
            }

            if ( insertMode )
            {
              insertCharacter( DIO );
            }
            else // NOT in insert mode
            {
              lcdBuffer[idx] = DIO;

              // update cell on display
              lcd.setCursor(c,r);
              lcd.write((char)DIO);
            }

            // cursor movement with line wrap
            idx++;
            // We do now want to scroll at this point when a character is written
            // to the last valid position at the bottom right edge of the screen.

            // Therefore idx may end up behind the visible screen, at
            // LCD_ROWS*LCD_COLUMNS.
            // This oveflow is taken into account when the next character is handled.
          }
        }
      }
    }
  }
}

// ======================================================================================
```

Listing 4 handleTerminal.cpp.