F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2023/24

[0] No proprietary software has been used in producing these slides

Outline

1. Tutorial 1: Using Python and the Linux FS for GPIO Control
2. Tutorial 2: Programming an LED
3. Tutorial 3: Programming a Button input device
4. Tutorial 4: Inline Assembler with gcc
5. **Tutorial 5: Programming an LCD Display**
6. Tutorial 6: Performance Counters on the RPi 2

HERIOT WATT UNIVERSITY

Tutorial 5: Programming an LCD Display

This tutorial will focus on programming a simple output device:

an 16x2 LCD display using an Hitachi HD44780U controller

This will be an exercise of controlling a device slightly more complicated than the LED and button devices so far.
The principles of programming are the same as before.

Overview

We will cover:

1. Connecting an LCD display to the RPi2
2. Low-level interface in assembler (`digitalWrite`)
3. Medium-level interface in C (`lcdClear` etc)
4. Sending characters and strings (`lcdPutChar`, `lcdPuts`)
5. Character data (defining own characters)

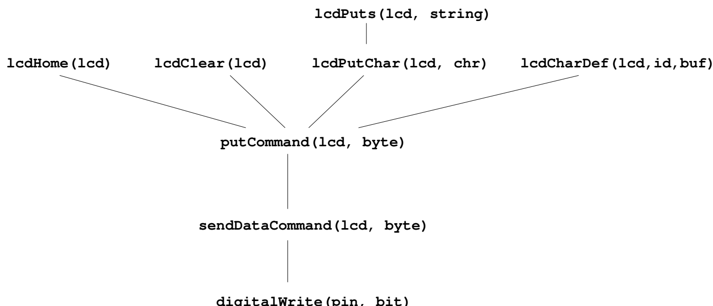
Acknowledgements

The code in this tutorial is mostly taken directly from the **wiringPi** library for the Raspberry Pi, by **Gordon Henderson**.
If you have downloaded the sources, you can look-up examples in the directory `wiringPi/examples` (e.g. `lcd.c`) and the code for the LCD functions in `wiringPi/devLib` (also `lcd.c`)

```
* wiringPi:
* Arduino look-a-like Wiring library for the Raspberry Pi
* Copyright (c) 2012-2015 Gordon Henderson
* Additional code for pwmSetClock by Chris Hall <chris@kchall.plus.com>
*
* Thanks to code samples from Gert Jan van Loo and the
* BCM2835 ARM Peripherals manual, however it's missing
* the clock section /grr/mutter/
***************************************************************
* This file is part of wiringPi:
* https://projects.drogon.net/raspberry-pi/wiringpi/
```

Function dependencies

Here is a simple picture of the dependencies of the API functions:



NB: only the lowest level, `digitalWrite` is in assembler, the rest is in C

LCD commands

We need some constant definitions and boilerplate code:
Here is a list of instructions for the Hitachi HD44780U controller:

```
#define LCD_CLEAR       0x01
#define LCD_HOME        0x02
#define LCD_ENTRY       0x04
#define LCD_CTRL        0x08
#define LCD_CDSHIFT     0x10
#define LCD_FUNC        0x20
#define LCD_CGRAM       0x40
#define LCD_DGRAM       0x80
```

[0] See Table 6 and Figure 11 in the HD4478 Technical Reference

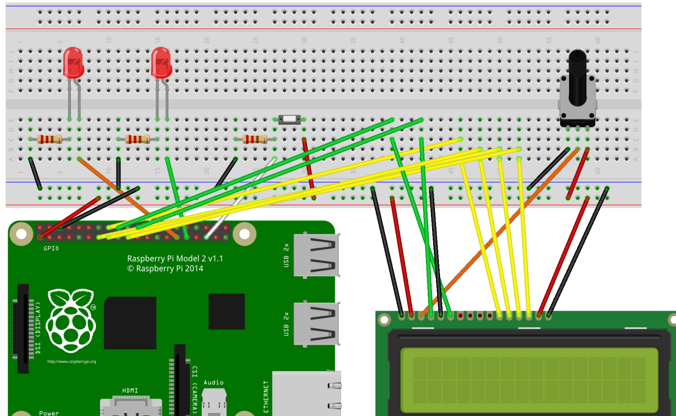
1. The wiring

Pin 19 (White) Button
Pin 13 (Green) green LED
Pin 5 (Orange) red LED

All power wires are red
All ground wires are black

LCD Data (Yellow)
GPIO 23 <-> LCD 11
GPIO 10 <-> LCD 12
GPIO 27 <-> LCD 13
GPIO 22 <-> LCD 14

LCD Control (Green)
GPIO 25 <-> LCD 4
GPIO 24 <-> LCD 6



The wiring: encoded

To encode this wiring in the program we define:

```
#define STRB_PIN 24
#define RS_PIN   25
#define DATA0_PIN 23
#define DATA1_PIN 10
#define DATA2_PIN 27
#define DATA3_PIN 22
```

Data structure for the LCD-connection

The following data structure stores the pin numbers and cursor position:

```
struct lcdDataStruct
{
  int bits, rows, cols ;
  int rsPin, strbPin ;
  int dataPins [8] ;
  int cx, cy ;
} ;
```

2. Low-level Assembler interface

This code is essentially the same as in the blinking LED example, i.e. we want to "send" one bit to a pin that's an argument to the interface:

- Set the mode of the pin to output (before calling the function)
- Identify the register and bit to write to
- Write one bit (1) into this location
- It is recommended that you use inline assembler to implement this function

2. Low-level Assembler interface

For CW2 you need to complete the code in `lcdBinary.c`:

```
void digitalWrite (uint32_t *gpio, int pin, int value) {
 asm volatile(/* inline assembler version of setting/clearing LED to
     ouput" */
  "\tLDR R1, %[gpio]\n"
  /* COMPLETE inline assembler code  */
 : /* COMPLETE Output operands */
 : [gpio] "m" (gpio)
   /* COMPLETE Input operands */
 : "R1" /* COMPLETE Clobbers */ );
}
```

The above assembler code needs to implement the functionality of this partial C code (see `tut_led.c` in Tutorial LED):

```
  int off = (value == LOW) ? 10 : 7; // register number for GPSET/GPCLR
  *(gpio + off) = 1 << (pinACT & 31) ;
```

2. Low-level Assembler interface

For CW2 you need to complete the code in `lcdBinary.c`:

```
void digitalWrite (uint32_t *gpio, int pin, int value) {
 asm volatile(/* inline assembler version of setting/clearing LED to
     ouput" */
  "\tLDR␣R1,␣%[gpio]\n"
  /* COMPLETE inline assembler code  */
 : /* COMPLETE Output operands */
 : [gpio] "m" (gpio)
   /* COMPLETE Input operands */
 : "R1" /* COMPLETE Clobbers */ );
}
```

The above assembler code needs to implement the functionality of this partial C code (see `tut_led.c` in Tutorial LED):

```
  int off = (value == LOW) ? 10 : 7; // register number for GPSET/GPCLR
  *(gpio + off) = 1 << (pinACT & 31) ;
```

3. Medium-level interface

Sending data uses `digitalWrite` to send bits over the 4 pins:

```
void sendDataCmd (const struct lcdDataStruct *lcd, unsigned char data)
{
  unsigned char          i, d4 ;

  d4 = (myData >> 4) & 0x0F;
  for (i = 0 ; i < 4 ; ++i)
  {
    digitalWrite (lcd->dataPins [i], (d4 & 1)) ;
    d4 >>= 1 ;
  }
  strobe (lcd) ;

  d4 = myData & 0x0F ;
  for (i = 0 ; i < 4 ; ++i)
  {
    digitalWrite (lcd->dataPins [i], (d4 & 1)) ;
    d4 >>= 1 ;
  }
  strobe (lcd) ;
}
```

Sending a command

Sending a command works like sending a byte, except that we only need 4 bits to encode the command, and therefore only one loop in the body:

```
void lcdPut4Command (const struct lcdDataStruct *lcd, unsigned char
    command) {
  register unsigned char myCommand = command ;
  register unsigned char i ;

  digitalWrite (lcd->rsPin,   0) ;

  for (i = 0 ; i < 4 ; ++i)  {
    digitalWrite (lcd->dataPins [i], (myCommand & 1)) ;
    myCommand >>= 1 ;
  }
  strobe (lcd) ;
}
```

Move cursor home

Now that we can send a command, we can create instances for each of the commands that are specified for the HD44780U controller:

```
void lcdHome (struct lcdDataStruct *lcd) {
  lcdPutCommand (lcd, LCD_HOME) ;
  lcd->cx = lcd->cy = 0 ;
  delay (5) ;
}
```

4. Sending characters and strings

Sending a character involves, sending the char as a byte, moving to the next position, and updating the position on the LCD display:

```
void lcdPutchar (struct lcdDataStruct *lcd, unsigned
    char data) {
  digitalWrite (lcd->rsPin, 1) ;
  sendDataCmd  (lcd, data) ;

  if (++lcd->cx == lcd->cols)  {
    lcd->cx = 0 ;
    if (++lcd->cy == lcd->rows)
      lcd->cy = 0 ;

    lcdPutCommand (lcd, lcd->cx + (LCD_DGRAM | (lcd
       ->cy>0 ? 0x40 : 0x00)   /* rowOff [lcd->cy]
       */  )) ;
  }
}
```

Writing strings

Once we can send characters, we only need a loop on top of it to send entire strings:

```
void lcdPuts (struct lcdDataStruct *lcd, const char
    *string) {
  while (*string)
    lcdPutchar (lcd, *string++) ;
}
```

5. Putting things together

In the main function we:

- Memory-map the GPIO address into user space (`gpio`)
- Configure an `lcd` data structure with the pin numbers for our wiring
- Initialise the connection to this `lcd`
- Initialise the display using `lcdClear()` and `lcdHome()`
- Write “Hello World” using `lcdPuts`

See the `lcd-hello.c` sample program.

5. Putting things together

In the main function we:

- Memory-map the GPIO address into user space (`gpio`)
- Configure an `lcd` data structure with the pin numbers for our wiring
- Initialise the connection to this `lcd`
- Initialise the display using `lcdClear()` and `lcdHome()`
- Write “Hello World” using `lcdPuts`

See the `lcd-hello.c` sample program.