AVR155: Accessing an \( \text{i}^2\text{C} \) LCD Display using the AVR\textsuperscript{\textregistered} 2-wire Serial Interface

**Features**
- Compatible with Philips’ \( \text{i}^2\text{C} \) protocol
- 2-wire Serial Interface Master Driver for Easy Transmit and Receive Function
- Initialization and Use of a 2 x 16 \( \text{i}^2\text{C} \) LCD Display
- “C” Source Code
- Matches the Most Common \( \text{i}^2\text{C} \) LCD Drivers

1. **Introduction**

The need for a cost effective inter-IC bus for use in consumer, telecommunications and industrial electronics, led to the development of the Philips’ \( \text{i}^2\text{C} \) bus. Today the \( \text{i}^2\text{C} \) bus is implemented in a large number of peripherals and microcontrollers, making it a good choice for low speed applications.

To meet the large amount of ICs with \( \text{i}^2\text{C} \) interface, the AVR TWI family has included the 2-wire Serial Interface (TWI) to its peripherals. The TWI bus can communicate with any \( \text{i}^2\text{C} \) compatible device both as slave and master. More information about \( \text{i}^2\text{C} \) and details about TWI can be found in the Philips \( \text{i}^2\text{C} \) specification and the AVR datasheet.

This application note includes a TWI driver for bus handling and describes how to access a Philips \( \text{i}^2\text{C} \) LCD driver on a Batron LCD display.

**Figure 1-1.** Hardware Connections
2. Theory of Operation

The Batron LCDs is a 2 x 16 alphanumeric display with an I²C user interface and an implemented ASCII table. The LCD is configured and controlled with I²C serial communication from the AVR and its TWI hardware module.

AVR’s TWI is a fully I²C compatible and can access any I²C device following the I²C specification from Philips. This makes it easy to configure and use the Philips I²C LCD driver.

The AVR software is written in C and divided into two main parts. Part one, LCD control, handles the necessary control and data bytes for the LCD while part two, TWI driver, takes care of the I²C bus handling.

The AVR TWI module is basically controlled by the TWINT flag. The flag is used to start all actions to the bus and to flag when an action is done. When the TWINT flag is set, the TWSR contains a status value according to the last action and if it was successful or not. When sending a message to the I²C bus through TWI, the code will pretty much look like a state machine and the next step in the state machine should always be based on the result of the last step, read from the TWSR. See Figure 2-1.

The TWI is very close to the I²C’s signaling and data flow which makes it a very powerful module but also fairly complex to use. To ease the use of TWI, this application note is divided into two sections, a LCD control and a TWI master driver, both written in C.

The TWI master is a communication driver handling all the signaling and data transmission through the TWI module to the I²C bus. It is a general driver and can be used to any master I²C access and will only need a minimum information from the calling function. The driver is realized in its own C file and can be included in any C program requiring a TWI driver.

The LCD control takes care of the LCD specific setup and sends data and control bytes through the TWI master driver to the LCD. It configures the LCD to a dual-line display and display the string “AVR ATmega163 with TWI interface to I²C” as a scrolling text.
2.1 TWI Master Driver

The TWI master driver handles the TWI modules signaling and data flow to and from the I^2^C bus. It only requires a minimum input from the calling function and will return a state according to the result of the action. The driver is released as a polling function which monitor the TWINT flag but can easily be changed to be interrupt driven. (See “Implementation” on page 5.)

The driver needs information about the slaves address and the bytes to send or receive. This is passed on to the function as a table of structs ended by the masters own addressee. See Figure 2-2.
Figure 2-2. Table of Structs for Sending to One Slave

<table>
<thead>
<tr>
<th></th>
<th>SLA+W</th>
<th>Number of Bytes to Send</th>
<th>Pointer to the Bytes to Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Own Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t Care</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t Care</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-2 shows a table for sending data to a slave. SLA+W is the slaves address and is setting it into slave receiver mode. Number of bytes indicates how many bytes the function will send to the slave and the pointer shows were to find the bytes. The function can handle a table with an infinite number of structs but will always be ended if it finds its own address as the slave address. If the table contains more than one package plus the one with it’s own address, it will use repeated start between the packages. If a table as Figure 2-3 is used, it will send a START, SLA+W, nDATA bytes, a repeated START, SLA+W, nDATA bytes and a STOP.

Figure 2-3. Table for Sending to One or Two Slaves

<table>
<thead>
<tr>
<th></th>
<th>SLA+W</th>
<th>Number of Bytes to Send</th>
<th>Pointer to the Bytes to Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SLA+W</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Number of Bytes to Send</td>
<td>Pointer to the Bytes to Send</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Own Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t Care</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t Care</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The function can also handle master receive. The pointer in the struct will then be a pointer to a temporarily buffer for the received bytes. See Figure 2-4.

Figure 2-4. Table for Receiving from One Slave

<table>
<thead>
<tr>
<th></th>
<th>SLA+R</th>
<th>Number of Bytes to Receive</th>
<th>Pointer to a Temporarily Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Own Address</td>
<td>Don’t Care</td>
<td>Don’t Care</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The driver can also handle mixed master receive and transmit with repeated START in between. After the driver is done with the communication, it will return to the calling function with a state value. This value is the result of the transfer and will be 0xFF if success or TWSR if it failed. For a complete list see Table 2-1.

### Table 2-1. Return Values from the TWI Master Driver Function

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Bus Error due to illegal START or STOP conditions</td>
</tr>
<tr>
<td>0x20</td>
<td>Slave address +W transmitted and NACK is received</td>
</tr>
<tr>
<td>0x30</td>
<td>Data byte transmitted and NACK is received</td>
</tr>
<tr>
<td>0x38</td>
<td>Arbitration Lost</td>
</tr>
<tr>
<td>0x48</td>
<td>Slave address +R transmitted and NACK is received</td>
</tr>
<tr>
<td>0x58</td>
<td>Data byte received and NACK is transmitted</td>
</tr>
<tr>
<td>0xFF</td>
<td>Buffer transmit OK/ buffer receive OK</td>
</tr>
</tbody>
</table>

#### 2.2 LCD_control function

The LCD code gives an example on how to initialize the LCD, clear the display and make a scroll text. For more specific details and all the features of the display, please see the Batron LCD datasheet and Philips 2119 I²C LCD driver datasheet.

#### 3. Implementation

The code is written to be as self explaining as possible and for more details about the program please read the comments in the code.

The functions are not interrupt driven as a polled version is a less complex way of doing a TWI driver. It is fairly easy to convert it to a interrupt driven version and could be done by moving the entire TWI driver function into the TWINT interrupt routine. Change the function to be a state machine and define a global variable to keep control of the state machine. You will also need a global table for the data to send or as a buffer for receiving.

The code is written in C divided into two files, TWI_driver.c and LCD_control.c. Both files have separate header files and are written for IAR C-compiler Version 1.50C/WIN. They are written in ANSI C and it should be simple to change them for other compilers. I/O and linker files are included in the code package added to the application note.

#### 3.1 TWI Driver Functions

The TWI driver handles all the communication against the I²C bus for the calling function. For more details about the function please see the comments in the code.
3.1.1 **char Init_TWI (void)**
Setup for the AVR TWI module. Initialize the AVR slave address, sets the baudrate for the communication and enable the module.

3.1.2 **void Wait_TWI_int (void)**
Loop until the TWI interrupt flag (TWINT) get set.

3.1.3 **unsigned char Send_start (void)**
Apply a START condition to the bus, waits until the condition is done and returns 0xFF if success. If a error occur it will return TWSR.

3.1.4 **void Send_stop (void)**
Apply a STOP condition to the bus and return immediately.

3.1.5 **unsigned char Send_to_TWI (unsigned char *tx_frame, unsigned char *rx_frame)**
The main driver function which handles all the transfer activity and calls the other functions for applying START/STOP conditions, transmit/receive data and transmit/receive address bytes. Return 0xFF if success, TWSR if error.

3.1.6 **unsigned char Send_byte (unsigned char data)**
Send a data byte to the slave and wait for a ACK/NACK. Return 0xFF if ACK, TWSR if error.
3.1.7 **unsigned char Send_adr (unsigned char adr)**

Send a slave address to the bus and wait for a ACK/NACK. Return 0xFF if success, TWSR if error.

3.1.8 **unsigned char Get_byte (unsigned char *rx_frame)**

Waits until you get a TWINT flag set and read out the TWI data registry. The read value is saved in SRAM at "*rx_frame". Return 0xFF if success, TWSR if error.

3.1.9 **void Delay_ms (char mS)**

A simple delay function. The delay is decided by the variable “mS” which decides number of mS delay.

3.1.10 **void Reset_TWI (void)**

Reset the TWI module and release the bus. Used to free the bus if a error occur.

3.2 **LCD Control Functions**

The LCD control takes care of the LCD specific setup and sends data and control bytes through the TWI master driver to the LCD. It configures the LCD to a dual-line display and display the string “AVR ATmega163 with TWI interface to I²C” as a scrolling text on the top line.

3.2.1 **char setup(void)**

Setup function for the AVR.

3.2.2 **unsigned char Init_DSP (void)**

Setup function for the display. Configure it to two lines with 40 characters each.

3.2.3 **unsigned char Clear_DSP (void)**

The display has a internal clear display function which writes 0x20 to the entire memory area. At this version of the driver 0x20 is not a blank character, this function will write blank characters to the entire display instead of 0x20.
3.2.4  unsigned char Write_String1 (void)
Writes the string “AVR ATmega16 with TWI interface to I^2C” to the display starting at line one, position 0.

3.2.5  unsigned char Shift_left (void)
Shifts part of the display memory that is viewable once to the left and makes a scrolling effect. Shifts one time every half second.

3.2.6  unsigned char Read_read_controlbyte (void)
Read out the displays read control byte including the busy flag and display address counter. Used to check if the display is busy or ready for new commands.
3.3 Hardware

The LCD is connected to the AVR through a I\^2C interface as shown in Figure 2-2.

4. References

Philips “The I\^2C-bus Specification Version 2.0” - (December 1998)
Philips “PCF2119x LCD controllers/drivers” datasheet - (March 1999)
Batron “BT21605V-SRE-I\^2C-COG” LCD module datasheet
Atmel “ATmega163” datasheet