

## Next Generation Intelligent LCDs

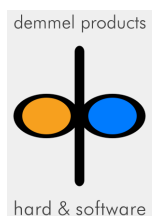
SPI Application Note

Version 1.0

Document Date: May 28, 2008

Copyright © by demmel products 2004 - 2008

Unless otherwise noted, all materials contained in this document are copyrighted by demmel products and may not be used except as provided in these terms and conditions or in the copyright notice (documents and software) or other proprietary notice provided with the relevant materials.



## Preface

This document describes the SPI specific communication with an iLCD controller only. Please read the "iLCD Commands" documentation located on <http://www.demmel.com/download/ilcd/ilcd-commands.pdf> to learn about how to send commands to iLCD panels first.

## Controlling the iLCD Controller via SPI

### Features

The SPI communication protocol used by the iLCD controllers supports miscellaneous settings like selecting the phase (CPHA) and polarity (CPOL) of the clock and choosing the bit order of the data sent and received.

### SPI Data Transfers

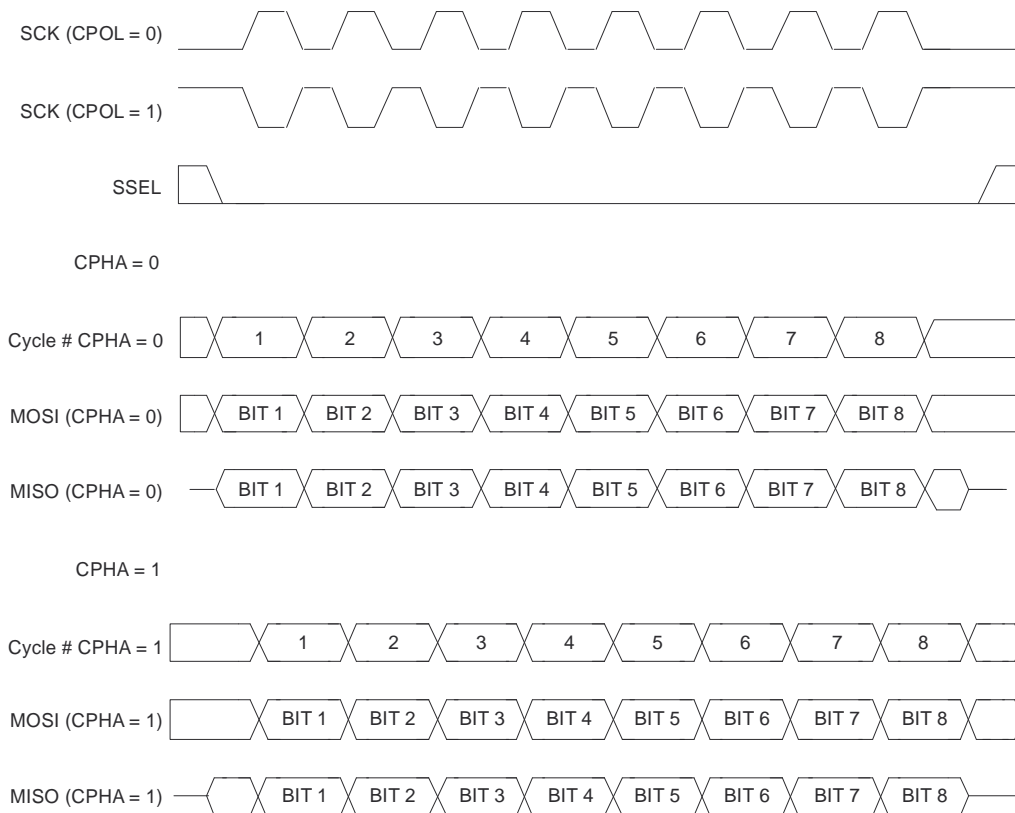


Fig. 1: Possible SPI data transfer formats

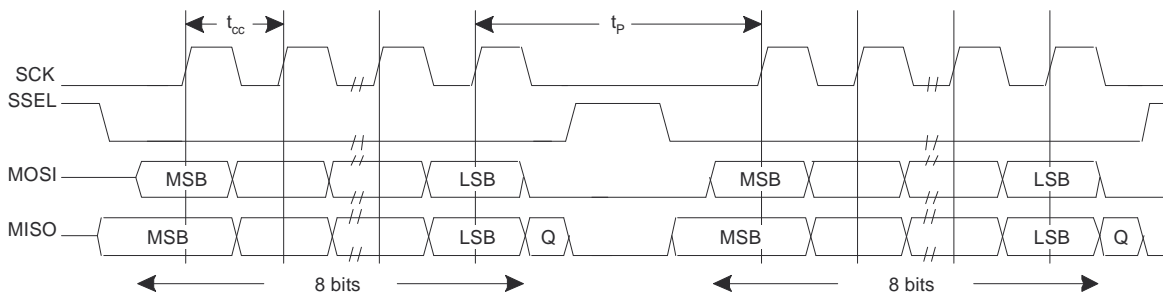


Fig. 2: SPI frame format with CPOL=0, CPHA=0 and MSB first

The SSEL pin does not need to return to high between the byte frames but can stay low as well, the iLCD controller supports both frame formats without any extra provision.

**!** Take care not to overrun the iLCD controller's SPI port by applying a too fast clock speed or data rate, garbled data transfer will be caused otherwise!

According to Fig. 2  $t_{cc}$  (the clock cycle time) can have a value down to  $0.15 \mu s$  giving a SPI clock frequency of up to 6.66 MHz, but  $t_p$  (pause between two consecutive bytes) must be greater than  $4.0 \mu s$  allowing the iLCD controller to remove/insert bytes in time. It is especially important to not make  $t_p$  too short when reading data from the iLCD controller, as this  $4.0 \mu s$  time is needed by the iLCD controller to read data from the output queue and write it to the SPI register. Sending data is not such time-critical, in this case  $t_p$  could be much lower without any problems; a 1 MHz SPI clock could be used here without adding extra pauses.

So if your application's SPI port does not have extra provisions to ensure adequate pauses between consecutive bytes (which is normally not the case), choosing a clock speed of lower than 400 kHz is recommended, as this automatically fits the requirement of  $t_p$  being greater than  $4.0 \mu s$ .

The iLCD controller acts as a pure slave, so all further reference to "master" apply to the controlling application where "slave" is the iLCD controller.

### SPI Pin Description

Pin Name	Direction for iLCD	Description
SCK	Input	The SPI is a clock signal used to synchronize the transfer of data across the SPI interface. The SPI is always driven by the master and received by the iLCD device. The clock is programmable to be active high or active low. The SPI is only active during a data transfer. Any other time, it is either in its inactive state, or tri-stated.
SSEL	Input	The SPI slave select signal is an active low signal that indicates the iLCD device is currently selected to participate in a data transfer. The SSEL must be low before data transactions begin and normally stays low for the duration of the transaction. If the SSEL signal goes high any time during a data transfer, the transfer is considered to be aborted. In this event, the slave returns to idle, and any data that was received is thrown away.
MISO	Output	The MISO signal is an output signal used to transfer serial data from the iLCD device to the master. When the iLCD device is not selected, it drives the signal high impedance.
MOSI	Input	The MOSI signal is a signal used to transfer serial data from the master to the iLCD device.

In a typical master-slave configuration the slave has no possibility to talk without being asked for data. As there is the necessity to signal unexpected data existence, an extra pin named ALERT is added to the standard pins mentioned above, which is activated (pulled low on default setup) as long as the iLCD controller contains data to be read. By watching the state of this extra pin (usually via an interrupt) your application can react immediately to read any existing data. A different approach is to poll the iLCD controller periodically and to read data when the status read flags existence of data.

#### Note:

DPC10xx iLCD controller: These controllers do not support SPI

DPC20xx and DPC30xx iLCD controller: Please ensure your iLCD controller firmware has a version greater or equal version 1.20 for using the SPI functionality; earlier versions do not support SPI yet.

## **Commands**

As sending/receiving data to/from an iLCD controller is not a typical range of application of an SPI communication, the existing iLCD commands according to the "iLCD Commands" documentation have to be "encapsulated" and some commands have to be provided to talk to an iLCD.

Every command sent to the iLCD controller via SPI starts with an <CID> introducer, which has a hex value of AA. If the data stream sent to the iLCD controller contains the hex byte AA, it must be quoted (that means the AA byte has to be sent twice) in order to not start a new command sequence. When sending commands to the iLCD device, the iLCD command specific command-introducer <CID> must not be inserted when communicating via SPI. Please see the examples below for further explanation.

When reading data from the iLCD controller via SPI, a dummy 0-byte has to be inserted after the command sequence allowing the iLCD controller to internally switch from read to write mode. As SPI is a master-controlled communication protocol, every byte to be read by the master is clocked in by sending out a corresponding null-byte (any other value is allowed as long it's not a <CID> character).

### **Read Status**

```
Out:  AAH 01H 00H 00H
In:   00H 00H 00H ss
```

The "Read Status" command has a command code value of Hex 01. The data byte *ss* read from the iLCD controller has the following meaning:

- Bit 0: If set, some data bytes are available for reading
- Bit 4: Indicates receive register overrun. The iLCD controller could not read the data from the SPI input register before a new byte was clocked in. This happens when in general only then, when the master applies a clock rate > 1 MHz. This bit is automatically reset after the status has been read.
- Bit 5: Indicates iLCD receive data overrun. The iLCD controller's input queue (with a size of 128 byte) was overrun. To avoid receive data overrun one can use the "Get Free Queue Space" before sending data to the iLCD controller. This bit is automatically reset after the status has been read.
- Bit 6: Indicates iLCD transmit data overrun. The iLCD controller's output queue (with a size of 64 byte) was overrun. This usually happens then, when a user misses to read the answers via the "Block Read" command to the previously sent commands or the size of the requested data does not fit into the 64 byte output queue. This bit is automatically reset after the status has been read.
- Bit 7: Indicates "slave abort", happening when the master raises the SSEL pin while transferring a byte. This bit is automatically reset after the status has been read.

### **Get Data Size to Read**

```
Out:  AAH 02H 00H 00H
In:   00H 00H 00H rs
```

The "Get Data Size to Read" command has a command code value of Hex 02. The data byte *rs* read from the iLCD controller describes the number of bytes available in the iLCD's output queue for reading.

### **Get Info**

```
Out:  AAH 03H 00H 00H 00H
In:   00H 00H 00H ss rs
```

The "Get Info" command is a combined command for getting the status and the data size to read via one command. The command code has the value Hex 03. *ss* describes the status according to "Read Status" and *rs* indicates the data size to read according to "Get Data Size to Read".

## Block Read

```
Out:  AAH 04H rc 00H 00H 00H 00H ...
In:   00H 00H 00H 00H r0 r1 r2 ...
```

The "Block Read" command is used to read data according the section "Data Sent By The iLCD Controller" in the "iLCD Commands" documentation. This data is normally sent by the iLCD automatically when communicating via a serial port, but have to be retrieved via a command due to the nature of the SPI protocol.

The "Block Read" command has the command code value Hex 04. *r<sub>c</sub>* is the number of bytes to be read from the iLCD controller and *r<sub>0</sub>*, *r<sub>1</sub>*, *r<sub>2</sub>*, ... are the bytes read back by the master.

If *r<sub>c</sub>* is greater than the number of available bytes stored in the output queue of the iLCD controller, hex 00 bytes are sent after the last available character. Please note, that *r<sub>c</sub>* is corrected to 64 by the iLCD controller if a greater values is requested, as the output queue has a size of 64 bytes only.

There is no need to read the complete iLCD's output queue in one chunk, multiple "Block Reads" can be done as long there is some data in the output buffer; the number of available bytes can be retrieved via the "Get Data Size to Read" or "Get Info" command at any time.

Any read sequence can be aborted by sending a <CID> character (Hex AA), but keep in mind that the next byte returned by the iLCD controller is a data byte which will be lost if not read back my the master.

Example of an aborted read sequence with status read following:

```
Out:  AAH 04H 40H 00H 00H 00H AAH 01H 00H 00H
In:   00H 00H 00H 00H r0 r1 r2 00H 00H ss
```

## Block Write

```
Out:  AAH 05H tc t0 t1 t2 t3 ...
In:   00H 00H 00H 00H 00H 00H 00H ...
```

The "Block Write" command is used to send data to the iLCD controller. Any data as described starting from section "Command Description" in the "iLCD Commands" documentation can be sent via the "Block Write Command". The "Block Write" command has the command code value Hex 05. *t<sub>c</sub>* indicates the number of bytes to be sent, the maximum value of *t<sub>c</sub>* is 128 (Hex 80), as the input queue of the iLCD controller is 128 bytes. Please keep in mind, that the iLCD controller's internal SPI driver automatically inserts the <CID> character into the input queue when receiving this command, so sending a block of 128 bytes can cause an input queue overflow by one character, even if the input queue was completely empty when issuing this command. Values greater 128 for *t<sub>c</sub>* are automatically corrected by the iLCD controller to 128.

A block write command can be aborted by sending a different command, e.g. a "Read Status" command before the number of bytes determined by *t<sub>c</sub>* are completely sent. When a <CID> character (Hex AA) shall be a part of the data stream to be sent to the iLCD controller, the <CID> character must be quoted by sending the <CID> character twice. Note, that the double <CID> character counts only as one character when calculating *t<sub>c</sub>*.

The <CID> character being part of the iLCD command normally, must be omitted here as it is inserted by the iLCD controller's internal SPI driver automatically.

## Example

The example mentioned below prints the text "-Hi-" to the screen. The iLCD command according to the "iLCD Commands" specification for doing so, is <CID> D T - H i - 00<sub>H</sub>

When sending this command via SPI, the bytes sent and received look as follows, where "44<sub>H</sub> 54<sub>H</sub> 2D<sub>H</sub> 48<sub>H</sub> 69<sub>H</sub> 2D<sub>H</sub>" is the Hex representation of "D T - H i -"

```
Out:  AAH 05H 07H 44H 54H 2DH 48H 69H 2DH 00H
In:   00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
```

Sending data not starting with a <CID> command (e.g. when the iLCD panel is in terminal mode, thus getting simple ASCII code text only) must be done via the "Continue Block Write" command instead.

Please note that iLCD commands must not be sent within one "Block Write" command, they can be split up in as many parts as required by using the "Continue Block Write" subsequently. After an iLCD command has been fully processed, the ACK/NACK (Hex 06/15) response is stored into the iLCD controller's output buffer and must be read via a "Block Read" command then.

Breaking up iLCD commands into several "Block Write" and "Continue Block Write" commands can be necessary to avoid overflowing the iLCD's input queue. If you want to e.g. send a text (after having previously sent a "Control Text" command to align and/or word wrap the following text) with a size of 200 characters, this must be done via consecutive blocks, as this would overflow the input queue of the iLCD controller otherwise. Reading back the number of free bytes of the iLCD's input queue via the "Get Free Queue Space" command between sending multiple blocks is highly appreciated.

### Continue Block Write

```
Out:  AAH 06H tc t0H t1H t2H t3H ...
In:   00H 00H 00H 00H 00H 00H 00H ...
```

The "Continue Block Write" command is used to send any data to the iLCD controller. The "Continue Block Write" command has the command code value Hex 06. t<sub>c</sub> indicates the number of bytes to be sent.

The only difference to the "Block Write" command described above is the absence of auto-inserting a <CID> character into the iLCD's input queue. All other characteristics of this command are the same as for the "Block Write" command.

### Get Free Queue Space

```
Out:  AAH 07H 00H 00H
In:   00H 00H 00H qs
```

The "Get Free Queue Space" command is used to retrieve the number of free bytes in the iLCD controller's input queue. The "Get Free Queue Space" command has the command code value Hex 07.

q<sub>s</sub> contains the number of bytes which can be sent to the iLCD controller without overrunning the controller's input queue at the time when issuing this command. The maximum value q<sub>s</sub> can have, is 128 (hex 80), as the size of the iLCD controller's input queue is 128 byte.

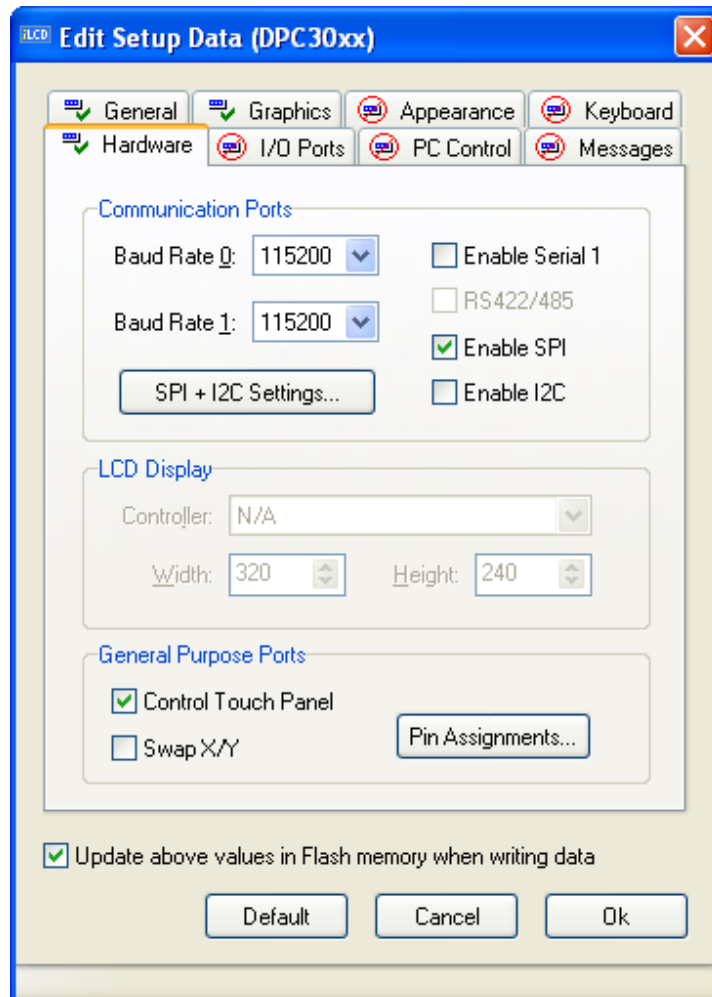
Please keep in mind that the iLCD controller's input queue fill condition is decreased even while sending new commands to the iLCD controller is carried out, so the situation may be completely different when sending this command again.

If all of your commands sent to the iLCD controller have a length of less or equal 127 bytes and issuing a new command is not done before the <ACK> response character is successfully retrieved from the iLCD controller, there is no need to use the "Get Free Queue Space" command. "Get Free Queue Space" is rather to be used when iLCD commands with a length of more than 127 bytes are sent via consecutive "Block Write" and "Continue Block Write" commands.

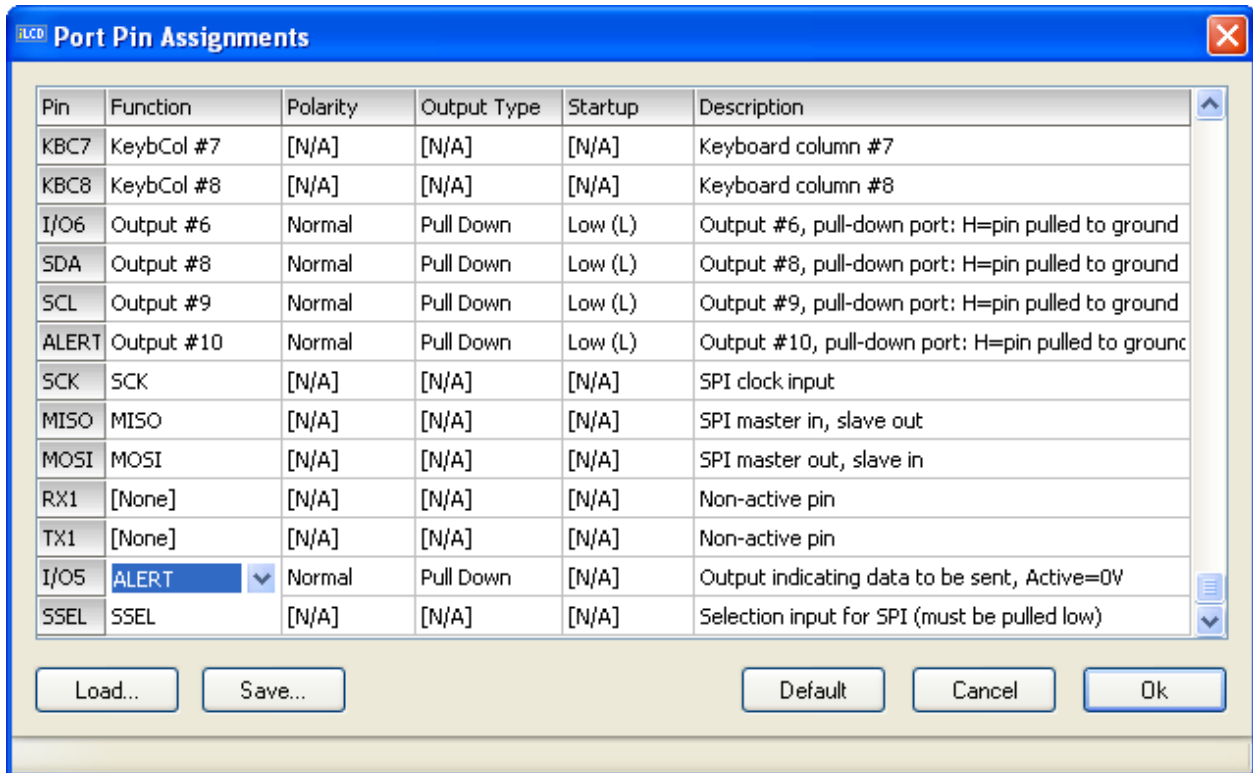
## **SPI on iLCD in Real-Life**

### **Setting Up the iLCD Panel for SPI Communication**

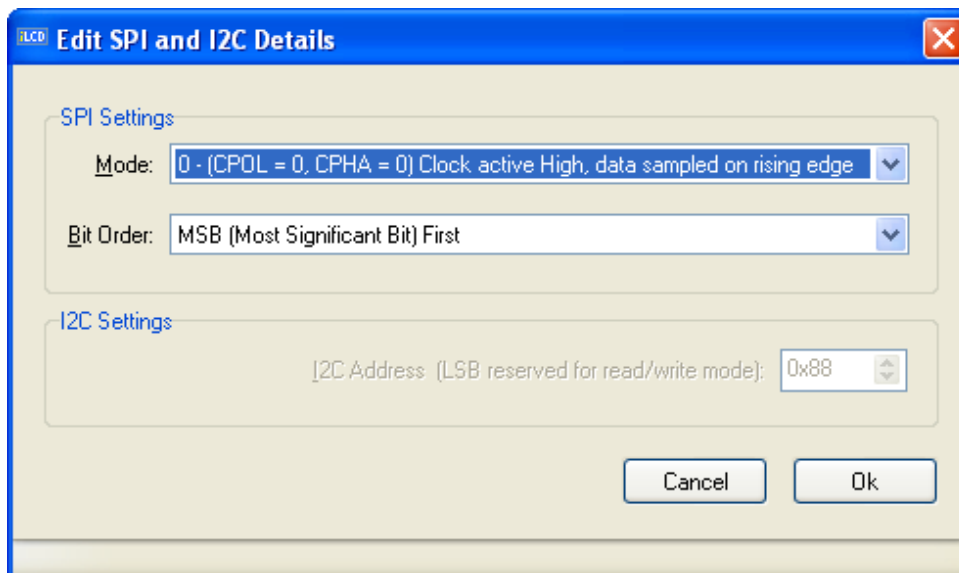
After starting the iLCD Setup software go to the "Setup" screen, load your preferred data file (e.g. "Touch Demo 320x240 (DPC20xx).lcdp-flash") press "Edit→Setup Data...", locate the "Hardware" tab and check the "Enable SPI" checkbox. This automatically enables all pins used for the SPI communication.



Enabling the ALERT function on the “Port Pin Assignments” screen enables the controlling application to be informed when some data is available to be read.



Setting up the SPI parameters to be used can be done by pressing the “SPI + I2C Settings...” button on the “Edit Setup Data” screen:



After setting all necessary parameters ensure that the "Update above values in Flash memory when writing data" checkbox on the “Edit Setup Data” screen is set and press "Ok".

Write the data to the iLCD panel by using the "Write Flash..." button finally.

## Sending/Receiving Data via SPI

To clarify the SPI communication with iLCD via SPI, we've made some screenshots showing the pin states in detail.

### Sending a Block of Data

Screenshots 1, 2 and 3 show how to send the command to draw the text "-Hi-" on the screen.

Screenshot 1 shows the details of the SSEL, SCK, MOSI and MISO pins, "SPI -> iLCD" shows the hex bytes sent to the iLCD and "SPI <- iLCD" the hex bytes received from the iLCD.

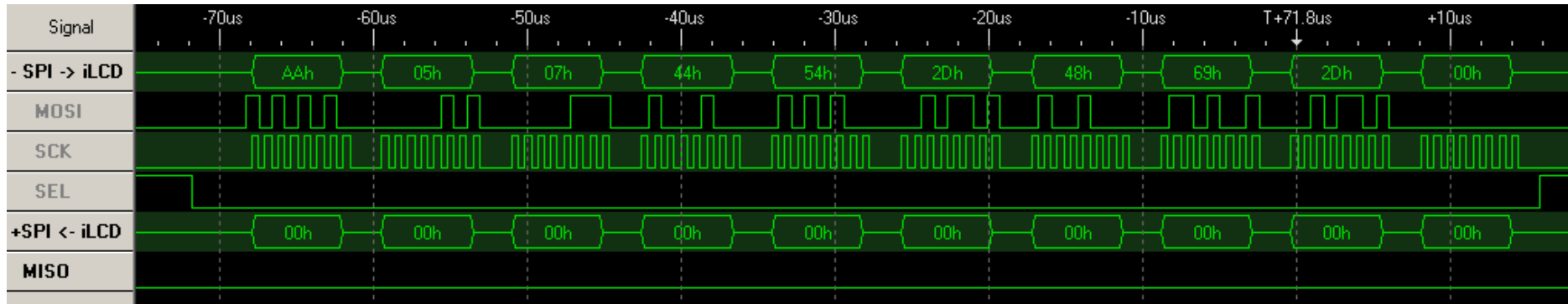
Screenshot 2 has the same contents as screenshot 1 and additionally shows the ALERT signal. The magnification is lower on this screen shot to see the ALERT pin becoming active when the iLCD has processed the command and offers data to be retrieved (the <ACK>, Hex 06 value, answering any successfully processed command).

Screenshot 3 shows the same as screenshot 1, but the SPI interpretation is shown in ASCII in this picture.

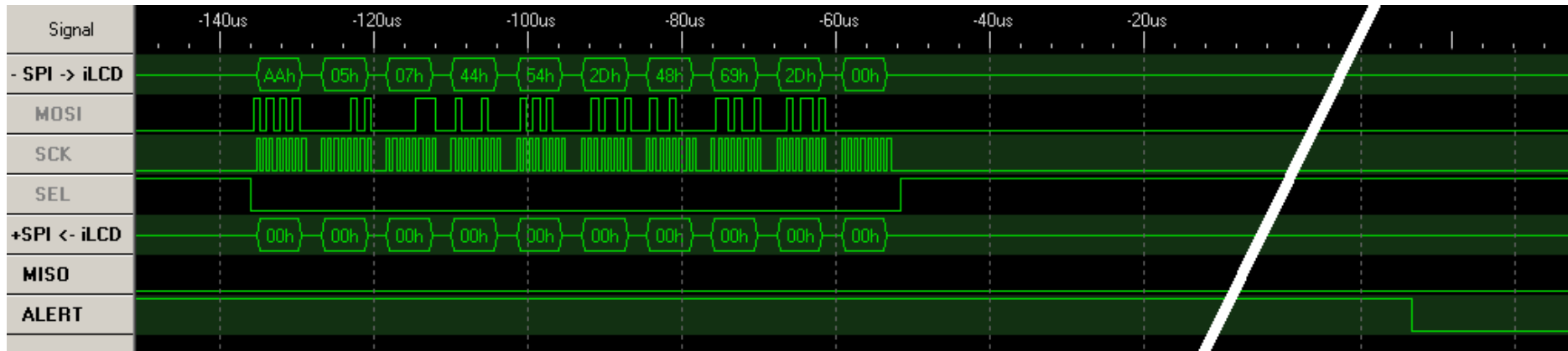
### Reading a Block of Data

Screenshot 4 shows how to read back data from the iLCD (in this case an <ACK>). After sending the "Block Read" command (Hex 04) with the succeeding dummy null-byte the next byte outputted by the master clocks in the reply from the iLCD controller synchronously.

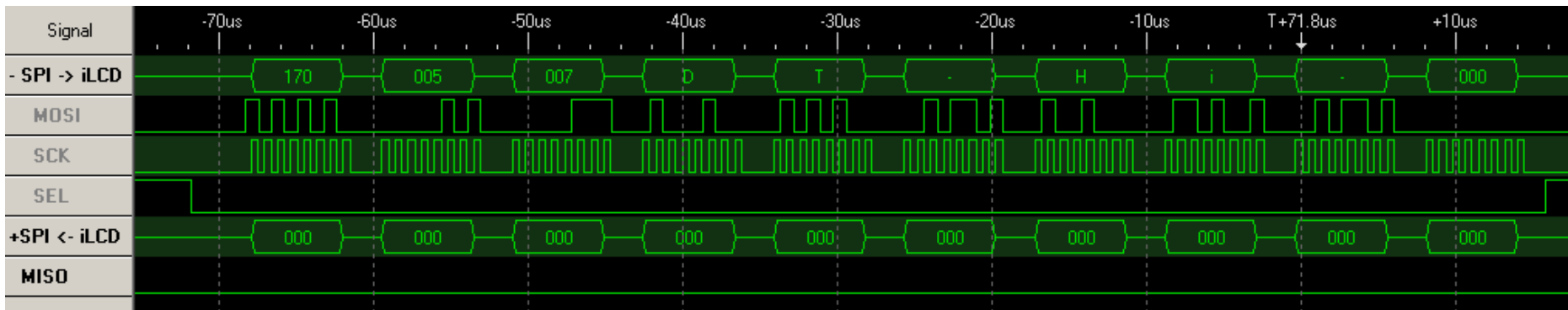
Screenshot 1: Sending Block of Data (Hex-Interpreted)



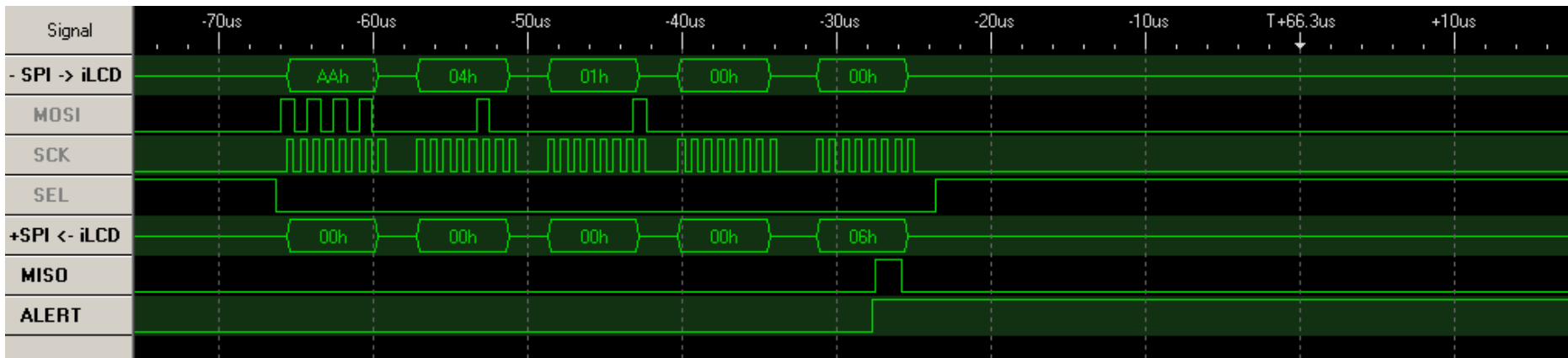
Screenshot 2: Sending Block of Data and Watching the ALERT pin



Screenshot 3: Sending Block of Data (Ascii-Interpreted)



Screenshot 4: Reading Block of Data (ACK)



**Revision History**

Date	Rev. #	Revision Details
May 28, 2008	1.0	First issue

If you find any errors in this document, please contact demmel products at [support@demmel.com](mailto:support@demmel.com)