AN11249

How to implement the ROM I2C

Rev. 1 — 16 August 2012 Application note

Document information

Info	Content
Keywords	LPC11A02UK; LPC11A04UK; LPC11A11FHN33; LPC11A12FHN33; LPC11A12FBD48; LPC11A13FHI33; LPC11A14FHN33; LPC11A14FBD48; LPC11Axx
Abstract	This application note details how to use the I2C ROM driver in master and slave mode.

Revision history

Rev	Date	Description
1	20120816	Initial version.

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

The LPC11Axx devices have built in I2C driver routines stored in the ROM that facilitate sending and receiving data on the I^2C-bus. The I2C routines support polling and interrupt driven communication in master and slave mode.

The I2C ROM driver supports 7-bit and 10-bit address in master mode and 7-bit address in slave mode.

The inclusions of the I2C drivers allow programmers to quickly and easily create projects to communicate over the I^2C-bus.

2. I2C hardware setup

The LPC11Axx devices have one I2C block that can be outputted to different IO pins depending on the pin-mux configuration. The primary I2C port pins, PIO0_2 and PIO0_3, are true open-drain connections. For other I2C ports, the pin functionality has to be changed to open-drain mode.

The PIO0_2 and PIO0_3 do not have optional internal pull-ups, unlike the other I2C pins. If the PIO0_2 and PIO0_3 are utilized, an external pull-up resistor must be used. If the other I2C pins are used and the internal pull-ups are enabled, external pull-ups are not required. If the internal pull-ups are used, the I2C can achieve a bit rate of 100kb/s with a 20pF load. If a higher speed is desired, then external pull-ups are required.

NOTE - On the WLCSP package, the boot loader configures the open-drain pins (PIO0_2 and PIO0_3) for the Serial Wire Debug (SWD) function.

The steps to initialize I2C are:

1. Enable the peripheral clock to the I2C, IOCON, and GPIO block
2. Setup I2C pins to open-drain operation
   a. Enable internal pull-ups if desired
3. Setup I2C clock frequency

3. I2C ROM driver setup

The I^2C-bus requires a minimum of one master and one slave to operate properly. In either master or slave mode, the mode of communication can either be polling or interrupt based. The I2C ROM driver has a provision to be either a master or slave, as well as being polling driven or interrupt driven.

To use the I2C ROM driver, the application code must initialize parameters required to access the routines in the ROM.

The I2C ROM driver table entry must be defined. This is the point of entry for the I2C ROM drivers. The I2C ROM driver is located at 0x1FFF1FF8.

```
1    #define ROM_DRIVERS_PTR ((ROM *)(*((unsigned int *)0x1FFF1FF8)))
```

The I2C ROM driver utilizes handle pointers for return parameters. It is necessary to define a new type as the I2C handler.

```
2    typedef void* I2C_HANDLE_T;
```

For interrupt driven code, callback functions are utilized. It is only necessary to define this type if interrupts are used.

```
3    typedef void  (*I2C_CALLBK_T) (  uint32_t  err_code, uint32_t  n) ;
```

The I2C ROM driver function calls return an error code. A new Type must be defined to address the different error codes.

```
4    typedef enum {
5      /**\b 0x00000000*/ LPC_OK=0, /**< enum value returned on Success */
6                        ERROR, /**< enum value returned on general failure */
7    ERR_I2C_BASE = 0x00060000,
8    /*0x00060001*/    ERR_I2C_NAK=ERR_I2C_BASE+1,
9    /*0x00060002*/    ERR_I2C_BUFFER_OVERFLOW,
10   /*0x00060003*/    ERR_I2C_BYTE_COUNT_ERR,
11   /*0x00060004*/    ERR_I2C_LOSS_OF_ARBRITRATION,
12   /*0x00060005*/    ERR_I2C_SLAVE_NOT_ADDRESSED,
13   /*0x00060006*/    ERR_I2C_LOSS_OF_ARBRITRATION_NAK_BIT,
14   /*0x00060007*/    ERR_I2C_GENERAL_FAILURE,
15   /*0x00060008*/    ERR_I2C_REGS_SET_TO_DEFAULT
16   } ErrorCode_t;
```

The input parameters to the I2C ROM drive have a strict structure that needs to be adhered to. Both the master and slave mode routines utilize these parameters during I2C communication.

```
17   typedef struct i2c_A {            //parms passed to ROM function
18     uint32_t num_bytes_send ;
19     uint32_t num_bytes_rec ;
20     uint8_t *buffer_ptr_send ;
21     uint8_t *buffer_ptr_rec ;
22     I2C_CALLBK_T func_pt;          // callback function
23     uint8_t stop_flag;
24     uint8_t dummy[3] ;         // to make word boundry
25     } I2C_PARAM ;
```

The i2c_get_status() function returns the current status of the I^2C-bus. A typedef enum can be used to indicate the different modes of the I2C.

```
26   typedef enum I2C_mode {
27     IDLE,
28     MASTER_SEND,
29     MASTER_RECEIVE,
30     SLAVE_SEND,
31     SLAVE_RECEIVE  } I2C_MODE_T;
```

The results of an I2C communication are stored in a structure. The structure contains the sent and received bytes.

```
32   typedef struct i2c_R {         // RESULTs struct--results are here when returned
33     uint32_t n_bytes_sent ;
34     uint32_t n_bytes_recd ;
35     } I2C_RESULT ;
```

The I2C ROM driver API is set up as a structure.

```
36   typedef struct  I2CD_API {     // index of all the i2c driver functions
37   void  (*i2c_isr_handler) ( I2C_HANDLE_T*  h_i2c) ;  // ISR  interrupt service
     request
38   // MASTER functions ***
39   ErrorCode_t (*i2c_master_transmit_poll)(  I2C_HANDLE_T* h_i2c, I2C_PARAM* ptp,
40                                                          I2C_RESULT* ptr ) ;
41   ErrorCode_t (*i2c_master_receive_poll)(I2C_HANDLE_T* h_i2c, I2C_PARAM* ptp,
42                                                        I2C_RESULT* ptr ) ;
43   ErrorCode_t (*i2c_master_tx_rx_poll)(I2C_HANDLE_T* h_i2c, I2C_PARAM* ptp,
44                                                      I2C_RESULT* ptr ) ;
45   ErrorCode_t (*i2c_master_transmit_intr)(I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
46   ErrorCode_t (*i2c_master_receive_intr)(I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
47   ErrorCode_t (*i2c_master_tx_rx_intr)( I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
48   // SLAVE functions ***
49   ErrorCode_t (*i2c_slave_receive_poll)( I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
50   ErrorCode_t (*i2c_slave_transmit_poll)( I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
51   ErrorCode_t (*i2c_slave_receive_intr)(  I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
52   ErrorCode_t (*i2c_slave_transmit_intr)(  I2C_HANDLE_T*  h_i2c, I2C_PARAM*  ptp,
     I2C_RESULT* ptr ) ;
53   ErrorCode_t  (*i2c_set_slave_addr)(  I2C_HANDLE_T*  h_i2c,
54                               uint32_t slave_addr_0_3, uint32_t  slave_mask_0_3);
55   // OTHER functions
56   uint32_t  (*i2c_get_mem_size)( void) ;  //"ramsize_in_bytes" memory needed by I2C
     drivers
57   I2C_HANDLE_T*   (*i2c_setup)( uint32_t  i2c_base_addr, uint32_t *start_of_ram ) ;
58   ErrorCode_t   (*i2c_set_bitrate)(I2C_HANDLE_T*  h_i2c,  uint32_t  P_clk_in_hz,
59                               uint32_t  bitrate_in_bps)  ;
60   uint32_t  (*i2c_get_firmware_version)( ) ;
61   I2C_MODE_T  (*i2c_get_status)( I2C_HANDLE_T*  h_i2c ) ;
62    } I2CD_API_T  ;
```

3.1 I2C ROM driver initialization

The I2C ROM drivers require specific parameters passed to it to operate properly. Variables, buffers, functions, etc. must be initialized by the user code before accessing the I2C ROM drivers.

The parameters used by the I2C ROM are first defined.

```
63   const I2CD_API_T*  pI2cApi ;  //define pointer to type API function addr table
64   I2C_PARAM*  ptop;   // define pointer to param of type 1 structure
65   I2C_PARAM   s1;     // s1 is a structure of type I2C_PARAM
66   I2C_RESULT* ptor;   // define pointer to return values of structure
67   I2C_RESULT  s2;     // s2 is a structure of type I2C_result
68   I2C_HANDLE_T *i2c_handle;
69   ErrorCode_t error_code;
```

```
70   #define RAMBLOCK_H     50  // number of words of memory to reserve for I2C
71   uint32_t I2C_Handle[RAMBLOCK_H]
72   uint8_t I2C_TxBuffer[MAX_BUFF_SIZE];  // ransmit buffer
73   uint8_t I2C_RxBuffer[MAX_BUFF_SIZE];  // receive buffer
```

The entry point into the I2C ROM driver is set up.

```
74   pI2cApi = ROM_DRIVERS_PTR->pI2CD;
```

The amount of SRAM bytes needed by the I2C driver is determined.

```
75   size_in_bytes =  pI2cApi->i2c_get_mem_size();
```

The SRAM can now be initialized to hold the I2C handle.

```
76   Uint32_t I2C_Handle[size_in_bytes];
```

Optionally, the I2C_Handle array can be initialized first with a known value and then check to ensure enough SRAM space is allocated.

```
77     if (RAMBLOCK_H  < (size_in_bytes /4 ) ) {
78        while (1) ;   // here forever if not enough ram for handle
79        }
```

A handle is initialized to the allocated SRAM area for the I2C driver.

```
80   i2c_handle = pI2cApi->i2c_setup(LPC_I2C_BASE, (uint32_t *)&I2C_Handle[0] );
```

The I2C clock frequency is setup. The function call expects the peripheral clock in Hz and requested clock frequency in Hz. For the purpose of this application note, the peripheral clock is set to 48 MHz and the requested clock frequency is 100 kHz.

```
81   error_code = pI2cApi->i2c_set_bitrate((I2C_HANDLE_T*)i2c_handle, 48000000,
     100000);
```

The I2C peripheral is initialized. This includes enabling the clock to the I2C, IOCON, and GPIO domain. The I2C functionality on the GPIO pins are enable, open drain mode is selected, and pull-ups can be enabled if desired.

```
82     LPC_SYSCON->PRESETCTRL |= (0x1<<1);
83     LPC_SYSCON->SYSAHBCLKCTRL |= ((1 << 16) | (1 << 6) | (1<<5));
84     LPC_IOCON->PIO0_2 &= ~0x3F;    /*  I2C I/O config */
85     LPC_IOCON->PIO0_2 |= 0x01;     /* I2C SCL */
86     LPC_IOCON->PIO0_3 &= ~0x3F;
87     LPC_IOCON->PIO0_3 |= 0x01;     /* I2C SDA */
```

The PARM and RESULTS structure are assigned addresses.

```
88     ptop = &s1;    // addr of PARAM struct, s1 is assigned to pointer ptop
89     ptor = &s2;    // addr of RESULT struct, s2 is assigned to pointer ptor
```

4. I2C ROM driver setup for master mode

4.1 Master mode send only using polling

The polling method is usually used during the design/debug phase of a project. This allows quick debugging of the I2C communication.

The transmit buffer is first loaded. For this example, the master is the LPC11Axx and the slave is a SE95 temperature sensor with an address of 0x48. The buffer is set up to send the configure command.

```
90      I2C_TxBuffer[0] = 0x90;   //slave addr in first byte of TX buffer
91      I2C_TxBuffer[1] = 0x01;   //Config command
92      I2C_TxBuffer[2] = 0x00;   //Config data
```

The first byte of the transmit buffer is always the slave address with the R/W bit set to zero. For a 10-bit slave address, the first byte would contain the slave addresses' most significant two bit and the R/W bit. The second byte of the transmit buffer would contain the remaining 8-bit slave address.

The number of bytes to transmit and receive are specified. In this case, a total of three bytes are to be transmitted. The number of received bytes is zero. Alternatively, the num_bytes_rec can be omitted.

```
93      ptop->num_bytes_send =3;
94      ptop->num_bytes_rec = 0;
```

The transmit and receive buffer need to be referenced for the I2C ROM driver function call. Since this is a transmit only, the receive buffer is not referenced. Alternatively, the receive buffer pointer can be omitted.

```
95      ptop->buffer_ptr_send = (uint8_t *)&I2C_TxBuffer[0];
96      ptop->buffer_ptr_rec = NULL;
```

The STOP condition is to be sent at the end.

```
97    ptop->stop_flag = 1;
```

The communication is initiated by making a call to the i2c_master_transmit_poll(). When done, the I2C ROM driver will return an error code. A "0" indicates no error was detected.

```
98      error_code = pI2cApi->i2c_master_transmit_poll((I2C_HANDLE_T*)i2c_handle,
99              (I2C_PARAM*)ptop, (I2C_RESULT*)ptor);
```

4.2 Master mode send and receive using polling

If a receive is expected after a transmit, then the i2c_master_tx_rx_poll() can be used. Alternatively, an i2c_master_transmit_poll() can be initiated followed by an i2c_master_receive_poll().

The transmit buffer is first loaded. The buffer is set to send a request for the SE95 ID code. The SE95 has an address of 0x48.

```
100   I2C_TxBuffer[0] = 0x90;    //slave addr in first byte of TX buffer
101   I2C_TxBuffer[1] = 0x05;    //Request SE95 ID
```

The R/W bit is set and loaded into the first byte of the receive buffer.

```
102     I2C_RxBuffer[0] = 0x91;   //SET RW bit
```

The number of bytes to be sent and received are loaded.

```
103     ptop->num_bytes_send =2;
104     ptop->num_bytes_rec = 1;
```

The transmit and receive buffers need to be referenced for the I2C ROM driver function call.

```
105     ptop->buffer_ptr_send = (uint8_t *)&I2C_TxBuffer[0];
106     ptop->buffer_ptr_rec = (uint8_t *)&I2C_RxBuffer[0];
```

The STOP condition is to be sent at the end

```
107     ptop->stop_flag = 1;
```

Initiate the read/write command.

```
108     error_code = pI2cApi->i2c_master_tx_rx_poll((I2C_HANDLE_T*)i2c_handle,
109             (I2C_PARAM*)ptop, (I2C_RESULT*)ptor);
```

4.3 Master mode using interrupt

The setup for the interrupt usage for the I2C ROM driver is similar to the polling function. The difference is the inclusion of a callback function. The callback will be executed once an I2C transaction is done.

To set up the I2C ROM driver for interrupt usage, setup the parameter as in the polling function. Two additional parameters have to be defined; one is the interrupt handler and the other is the callback function.

The I2C interrupt handler is declared. Inside the I2C interrupt, the I2C ROM driver ISR is called.

```
110   void I2C_IRQHandler( void){    // Application Program enables interrupts and
      calls ISR
111         pI2cApi->i2c_isr_handler( (I2C_HANDLE_T*) i2c_handle );
112   }
```

The callback function is used by the I2C ROM driver to determine what to do when data is sent or received.

```
113   void callback_func( void )
```

Declare the call back function. The callback function is called once the I2C ROM driver is finished with an I2C transaction.

```
114     ptop->func_pt = (I2C_CALLBK_T)callback_func;
```

Initiate the write operation. This can be either the i2c_master_transmit_intr(), i2c_master_receive_intr(), or i2c_master_tx_rx_intr();

5. I2C setup in slave mode

The I2C ROM driver supports the I2C slave in either polling or interrupt driven. Unlike the master mode, the I2C ROM driver supports 7-bit addressing only. The i2c_set_slave() function allows four slave addresses to be defined. The four slave addresses are stored in a 32-bit variable. The slave address 0 is the least significant byte and the slave address byte 3 is the most significant byte. The slave address mask is ordered in the same way.

31 25	24	23 17	16	15 9	8	7 1	0
Slave Address 3	GC	Slave Address 2	GC	Slave Address 1	GC	Slave Address 0	GC

To set up an I2C slave address, call the i2c_set_slave() function. For this example, the slave address is set to 0x48 and is loaded for slave address 0. The mask is not set.

```
115   pI2cApi->i2c_set_slave_addr( (I2C_HANDLE_T*)i2c_handle, 0x00000090, 0x00000000) ;
```

Setting the mask allows the slave mode to respond to a series of slave address. For instance, if the slave address 0 is set to 0xB0 and the mask is set to 0xE, then the I2C ROM driver will respond to commands for address 0xB0, 0xB2, 0xB4, 0xB6, 0xB8, 0xBA, 0xBC, and 0xBE.

General call is supported in the I2C ROM driver. To enable this, set the least significant bit of the slave address. When general call is enabled, the device will monitor the address of 0x00.

5.1 I2C receive polling

The polling technique is useful during the development and debugging phase, but is inefficient for practical use. The polling routines should be used during the early stage of development to ensure the I^2C-bus is operating correctly.

Set up the receive buffer. The receive buffer should be equal to or more than the total of bytes expected to received.

```
116   ptop->num_bytes_rec = MAX_BUF_SIZE;         /* max buffer size */
```

Setup the receive buffer pointer.

```
117     ptop->buffer_ptr_rec = (uint8_t *)&I2C_RxBuffer[0];
```

Initiate a read.

```
118     error_code = pI2cApi->i2c_slave_receive_poll((I2C_HANDLE_T*)i2c_handle,
119             (I2C_PARAM*)ptop, (I2C_RESULT*)ptor);
```

5.2 I2C write polling

The slave write is usually called after a slave read. A slave read would determine if the R/W bit is set. If the bit is set, then a write would be initiated.

For this example, the LPC11Axx is emulating an SE95 temperature sensor. Assuming a command to return the SE95 ID code is received, the transmit buffer will be loaded with the SE95 ID, 0xA1.

```
120         I2C_TxBuffer[0] = 0xA1;
```

Declare number of bytes to be transmitted.

```
121        ptop->num_bytes_send =1;
```

Declare the transmit buffer pointer.

```
122        ptop->buffer_ptr_send = (uint8_t *)&I2C_TxBuffer[0];
```

Initiate a write.

```
123        error_code = pI2cApi->i2c_slave_transmit_poll((I2C_HANDLE_T*)i2c_handle,
124             (I2C_PARAM*)ptop, (I2C_RESULT*)ptor);
```

5.3 I2C receive interrupt initialization

To enable an interrupt for the I2C ROM driver, the I2C interrupt must first be declared. Inside the I2C interrupt, the I2C ROM driver ISR handler must be declared. Callback functions are used in the I2C ROM driver ISR and should be defined.

Declare interrupt handler

```
125   void I2C_IRQHandler( void){    // Application Program enables interrupts and
      calls ISR
126     pI2cApi->i2c_isr_handler( (I2C_HANDLE_T*) i2c_handle );
127   }
```

Define callback functions

```
128   void slave_rx_callback ( void );
129   void slave_tx_callback ( void );
```

Enable I2C interrupt

```
130   NVIC_EnableIRQ(I2C_IRQn);
```

5.3.1.1 I2C receive/transmit using interrupt

The receive function is usually the first function call to make. The slave does not act until there is a command from the master. When a command is received from the master, the data is stored in the receive buffer. The I2C ROM driver then calls the callback function to determine what to do with the data.

The setup for the interrupt is the same as the polling with the addition of the callback function being set.

```
131     ptop->func_pt = (I2C_CALLBK_T)slave_rx_callback;
```

Initiate slave read mode

```
132     error_code = pI2cApi->i2c_slave_receive_intr((I2C_HANDLE_T*)i2c_handle,
133             (I2C_PARAM*)ptop, (I2C_RESULT*)ptor);
```

Initiate slave write mode

```
134        error_code = pI2cApi->i2c_slave_transmit_intr((I2C_HANDLE_T*)i2c_handle,
135             (I2C_PARAM*)ptop, (I2C_RESULT*)ptor);
```

6. Conclusion

The I2C ROM driver allows a quick and easy way to communicate to external peripherals using the I^2C-bus. The I2C ROM driver removes the complexity associated with initializing the I^2C-bus and allows the system designer more time to work on the system.

Example code for the I2C ROM driver can be downloaded from http://www.lpcware.com. The example code demonstrates the usage of the I2C ROM driver in the master and slave mode. In the master mode, the LPC11A14 is used to communicate with a SE95 temperature sensor. In slave mode, the LPC11A14 emulates an SE95 temperature sensor. All examples can be set to polling or interrupt driven.

The example code can be used in the Keil MDK, IAR, and LPCXpresso IDE.

7. Legal information

7.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

7.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

7.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

8. Contents

For more information, visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 16 August 2012
Document identifier: AN11249