Texas Instruments

bq803xx ROM API v 3.0

User's Guide

October 2005 PMP Portable Power

SLUU225

bq803xx ROM API v 3.0

User's Guide

Literature Number: SLUU225
October 2005

TEXAS INSTRUMENTS

Contents

Read This First

The bq802xx contains 6K of mask ROM code, consisting of boot-ROM code and library routines. The boot-ROM code executes at reset and detects whether the bq802xx is configured to boot into the application program in flash memory. If not, the boot ROM makes available a set of SMBus-accessible routines for flash programming and verification, and reading or writing the data memory space (including hardware registers). The ROM also contains library routines, which can be called from applications programs running in flash memory. This document describes the method of accessing library routines, the library services available, and the boot-ROM routines available at system reset.

This document describes

- Use of library services
- Boot ROM routines available at system reset

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface similar to a typewriter's. Examples use a **bold version** of the special typeface for emphasis; interactive displays use a **bold version** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).
- Here is a sample program listing:

```
0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006               .even
```

Here is an example of a system prompt and a command that you might enter:

```
C: csr -a /user/ti/simuboard/utilities
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect
  "section name", address

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.
- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here is an example of an instruction that has an optional parameter:

  LALK
  16-bit constant [, shift]

The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they are not optional).

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here is an example of a list:

  ```
  { * | *+ | *- }
  ```

  This provides three choices: *, *+, or *-.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte
  value₁ [, ... , valueₙ]

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

FCC Warning

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

Interrupt Vectors and Hooks

This chapter describes the operation of the interrupt vectors and hooks in the bq802xx ROM.

Topic **Page**

1.1 Introduction

The reset and interrupt vectors of the bq802xx are populated with JUMP instructions. They are defined in the assembly support file crt0.s and are arranged in flash program memory as follows:

```
0x0000:  jump main          ; flash "reset" vector
0x0001:  jump xinHandler    ; external interrupt handler
0x0002:  jump pinHandler    ; peripheral interrupt handler
0x0003:  jump cinHandler    ; communications interrupt handler
0x0004:  jump smbWaitIntr   ; "wait" for next smb event
```

The operation of the three interrupt vectors can be modified by symbols defined at assembly time. The xin and pin interrupts can be redirected by defining an assembler symbol, ROM_INT, to use interrupt prologue and epilogue code (stacking and restoring registers, RETI instruction). To conserve flash program memory, this code is in ROM. In this case, the vectors to the user-provided interrupt service routine bodies are at 0x0007 and 0x0008:

```
0x0007: jump xinHandler ; external interrupt handler body
0x0008: jump cinHandler ; communications interrupt han body
```

The interrupt service routines bodies can then be written as C functions, which return with RETS, or in assembler. Besides saving program memory space, this eliminates the danger of writing a C interrupt handler which saves data on the stack before the registers can be saved.

The cinHandler and smbWaitIntr vectors can also be redirected using the assembler symbol SCHED_CIN. This uses the communications interrupt handler and scheduler provided in ROM. The ROM communications interrupt handler simply sets the communications process (process 0) to ACTIVE, then calls the scheduler. All the work is done by the process code. See the document *Gas Gauge Example with AFE* for instructions and examples for configuring the compiler and assembler. The ROM scheduler theory of operation is described in the document *Scheduler Operation*.

The ROM also provides routines to perform i2c accesses to an external device, such as a serial EEPROM. This is a software-driven serial access, which does not use the SMBus engine in the bq802xx. The user must provide low-level i/o access to the pins selected for i2c access.

```
;----------------------------------------------------------
; hooks to user-provided i2c routines
;----------------------------------------------------------
0x0009:    jump  i2c_clockhi
0x000a:    jump  i2c_clocklo
0x000b:    jump  i2c_wait_clockhi
0x000c:    jump  i2c_datahi
0x000d:    jump  i2c_datalo
0x000e:    jump  i2c_datain
0x000f:    jump  i2c_wait_quarter_bit
```

If you are programming in assembly language, you must ensure that the vectors from i2c_clockhi to i2c_wait_quarter_bit jump to subroutines that ultimately return with a RETS instruction. Also, i2c_wait_clockhi and i2c_datain returns a value in r2. See the section on i2c library routines for details.

The vector main_init is the reset vector. Control is transferred here by the boot ROM when it finds the flash integrity word defined as 0x155454 at address 0x0005.

```
.ifdef INTEGRITY
            .byte 0x00,0x15,0x54,0x54; flash integrity word good
.else
            .byte 0x00,0x3f,0xff,0xff; flash integrity word bad
.endif
```

The integrity word should be undefined (anything except 0x155454) while you are developing code, so that a power-on reset causes the bq802xx to return to the boot ROM. From boot ROM you can erase and reprogram the part. If you do set the integrity word to 0x155454, the part jumps from boot ROM to flash at reset.

It may be necessary to program the integrity word and boot to flash for testing. In this case you should provide a function that allows you to return to the boot ROM by calling the library function flash_execute()-without this function you must invoke the hardware fail-safe feature to return to boot ROM to reprogram the part.

To invoke the hardware fail-safe feature you should tie ra3 and ra7 together. The hardware fail-safe signals the boot-ROM code to ignore the integrity word and continue to execute from boot ROM.

The security word at address location 0x0006 is used to prevent unauthorized access and is undefined when it is 0x3fffff. Any other value in that location is considered defined and disables the hardware fail-safe feature. TI recommends that the security word be used with caution and only on production code. During development,. leave the security word undefined.

In addition, the following RAM locations may be used by the ROM code to exchange information with flash program code:

```
; RAM locations
smb_ctl           = 0x0000
smb_errno         = 0x0001
i2c_errno         = 0x0002
process_list      = 0x0003, 0x0004
process_ptr       = 0x0005, 0x0006
num_proc          = 0x0007
halt_mode         =    0x0008
peek_poke address = 0x0009, 0x000a
```

The locations of these variables must remain constant in order for the ROM code to use them, so other variables must not be allocated on top of them, if the ROM library SMBus or i2c routines are used.

For C programs, the configuration for the vector and RAM allocation, and other initializations, are controlled by the cstart file crt0.s. See the readme file in the support files for detailed instructions for configuring the cstart file.

1.2 Making Calls to the ROM

In order to use the ROM library routines, you must make a software call (CALLS) to the listed entry point for the routine, and pass parameters and retrieve return values in accordance with the function prototypes listed in this document. The entry points to the library routines are contained in the library files included in the development environment. The source code refers to the library functions by name and the linker provides the physical address. Inclusion of the appropriate header files in C programs, or declaration of the function name as a global in assembly programs, provides the compiler or assembler with the symbolic reference.

In general, the ROM routines are called from C programs. In this case, all that is necessary is to conform to the C function prototypes. For assembly language programs, the calls to the ROM library routines must pass parameters and retrieve function return values exactly as a C program would. For this, follow the parameter-passing conventions used by the compiler: In mixed C/assembly programs, it is important to remember to preserve the stack pointer, i3, and registers i2 and ip, across the assembly subroutine call, because the C compiler expects them to remain intact.

1.3 C Function Parameter Passing

The C compiler uses the index register i3 as a stack pointer and the registers r0,r1,r2,r3 to carry out the exchange of parameters. Some examples :

extern char as_byte(char u);

The parameter **u** is carried in by **r3** and the return value by **r2**

extern char as_byte(char i, char u);

The parameter **i** is carried out by **r3** and **u** by **r2** and the return value by **r2**

extern int as_byte (char i, char u);

The parameter **i** is carried out by **r3** and **u** by **r2** and the return value by **r2** and **r3** (**r2**=lsb and **r3**=ms**b**)

extern int as_getbit(short x, short i);

The parameter **x** is carried out by **r3,r2** (**r2**=lsb and **r3**=ms**b**) and **i** by **r1,r0** (**r0**=lsb and **r1**=ms**b**) **a**nd the return value by **r2** and **r3** (**r2**=lsb and **r3**=ms**b**)

extern int as_getbit(long x, short i);

The parameter **x** is carried out by **r3,r2,r1,r0** (**r**0=lsb and **r**3=msb) and **i** by the stack **(i3,0) and (i3,1)** and the return value by **r2** and **r3**.

Stack depths as reported for the individual functions are the depth of stack used after the routine is entered. Some parameters are passed to the routine on the stack, and others are passed in registers, but the previous contents of these registers may need to be saved on the stack. These would all be saved by the calling function before the call to the library function. In addition, ip is used for the call, and must be preserved by the calling routine, but may already have been saved due to a previous call in the calling routine. These variable stack uses must be added to the reported stack depth to gauge accurately the effect on the stack of the library call.

Chapter 2

SLUU225 – October 2005

ROM Library Functions

This chapter describes the ROM Library Functions of the bq802xx.

Topic **Page**

2.1 SMBus Routines

The SMBus ROM routines provide easy access to the SMBus engine in the bq802xx. These routines send or receive multiple bytes over the SMBus. Because the SMBus hardware handles the clocking of individual bits in or out, CPU action is normally required only when each byte is to be transferred to or from the SMBus hardware. In order to avoid needlessly tying up the CPU, the ROM routines can be made to relinquish the CPU while they are waiting for the SMBus hardware to finish clocking a byte in or out. If the user sets the SMB_FLASH bit in the smb_ctl byte in RAM, these routines jump to the user's flash code through the smbWaitIntr vector in flash program memory when waiting for more data. The user's code may then perform other processing, usually by yielding to the scheduler, and return to the SMBus ROM code when the next SMBus event occurs. If the user does not set the SMB_FLASH bit, the SMBus ROM code uses polling, and thus retains control of the CPU until the entire SMBus transaction is complete.

The SMBus ROM routines share two RAM locations with the user's flash routines, to exchange status and configuration information. They contain bit flags for control and status as follows:

smb_ctl at address 0x00 contains configuration information for SMB

enum Smb_Ctl {

SMB_FLASH	= 0x01, //yield to flash
SMB_PECEN	= 0x02, //use PEC in master mode
RESERVED	= 0x04 //reserved
RESERVED2	0x08 //reserved
I2C_NO_ACK	= 0x10 //I2C routines do not require an ACK
SMB_PEC_DET	= 0x20 //SMB routines return error if PEC not used

};

smb_errno at address 0x01 contains error code of last SMBus transaction

enum Smb_Err {

SMB_OK,
SMB_Busy,
SMB_Reserved,
SMB_Unsupported,
SMB_AccessDenied,
SMB_Overflow,
SMB_BadSize,
SMB_UnknownError

};

Not all of these error codes are used by the ROM code. smb_errno should be set to SMB_OK (zero) by the application program before calling the SMBus ROM routine. The SMbus routine returns 1 if there is no error; otherwise, it returns 0.

When SMB_FLASH is set, the ROM code jumps to flash through a vector every time it must wait for further SMBus activity. It jumps to smbWaitIntr(), provided by the user, to yield to the scheduler. This allows other useful work to be done while waiting for the next SMBus event. This is the anticipated normal mode of operation. See the document *Scheduler Operation* for further explanation.

When SMB_PECEN is set, master mode transactions uses a PEC (packet error checking) byte. In slave mode, the PEC is appended to the transmission if the master requests it by sending an ACK after the last data byte, and can be checked if the master sends it. The PEC is generated and checked by the SMBus hardware.

When SMB_PEC_DET is set, the smbSlave functions indicates an error by returning zero if a PEC was not used in the slave transaction. In the SMBus specification, the slave device is required to behave the same independent of whether a PEC is used or not; so, this bit is for users who wish to operate without complete conformance to the SMBus specification.

When I2C_NO_ACK is set, the I2C routines returns no error even if data has not been acknowledged by the slave device.

In slave mode, the SMBus engine acknowledges its own address, set in the SMBus target register at 0x8006. When the command word arrives from the bus master, the SMBus engine sets SMSTA_DRDY or SMBSTA_DREG true and generates an interrupt (if enabled). At this point, it is the responsibility of the application code to take the appropriate action. See the document Gas Gauge Example for further details.

In order to avoid ambiguity in the following descriptions, the description of the *SMBus protocol as read and write* are always from the perspective of the bus master, i.e., they are master read and master write. The function names of the SMBus ROM library routines reflect the direction from the perspective of the bq802xx in its role as bus master or slave. Thus a master send word (smbMasterWrWord) is an SMBus write word protocol, but a slave send word (smbSlaveSndWord) is an SMBus read word protocol. Consequently, although sending a word with smbMasterWrWord() necessarily implies receiving it somewhere else with smbSlaveRcvWord(), the transaction's protocol is called a write word protocol, because the master is writing.

The bq802xx can act as either the master or the slave in an SMBus transaction. This transaction can fail in one of several ways:

cause of failure	smb_erro	master	slave
loss of arbitration	unchanged	X	
SMBus is busy	unchanged	X	X
SMBus transaction times out	unchanged	X	X
no ACKnowledgment	unchanged	X	X
packet error check fails	SMB_UnknownError	X	X
unexpected SMBSTA_DRDY	SMB_AccessDenied		X
block size too large	SMB_BadSize		X
command not found	SMB_Unsupported		X

smbMasterRdWord *smb Master Read Word*

function prototype int smbMasterRdWord (unsigned char address, unsigned char command, int *data);

description This function is used for SMBus Read Word protocol. smbMasterRdWord sends a slave address, a command byte and then reads a word (two bytes, lsb first) from the selected slave when called. It yields to the scheduler between bytes.

Input:	smbMasterRdWord has 3 inputs
address	device address of slave
command	command byte for slave
data	pointer to storage for word to be read from slave.
Output	smbMasterRdWord has 2 outputs
function return:	0 = fail (busy, timeout, no acknowledgment, packet error check fail)
	1 = success
side effects:	global variable smb_errno contains code for error: SMB_UnknownError if a PEC is detected.
Stack depth:	13

example

```
unsigned char address, command;
int target;
int *data;
     data = (int *)
     address = SLAVEADDRESS;
     command = RETURN_WORD;
     .
     .
     status=smbMasterRdWord(address,command,data);
     if(!status) {
         //do error-handling
     }
     //now word has been read from slave
```

smbMasterWrWord *smb Master WriteWord*

function prototype

int smbMasterWrWord (unsigned char address, unsigned char command, int data);

description

This function is used for SMBus Write Word protocol. smbMasterWrWord sends a slave address, a command byte and then a word (two bytes, lsb first) to the selected slave when called. It yields to the scheduler between bytes.

Input:	smbMasterWrWord has 3 inputs
address	device address of slave
command	command byte for slave
data	the word to be read from slave over SMBus.
Output	smbMasterRdWord has 2 outputs
function return:	0 = fail (busy, timeout, no acknowledgment, packet error check fail)
	1 = success
side effects:	none
Stack depth:	12

example:

```
unsigned char address, command;
int data;
     address = SLAVEADDRESS;
     command = DO_THIS;
     data = somevalue;
     .
     .
     status=smbMasterWrWord(address,command,data);
     if(!status) {
         //do error-handling
     }
     //now word has been read from slave
```

smbMasterRdBlock *smb Master Read Block*

function prototype: int smbMasterRdBlock (unsigned char address, unsigned char command, unsigned char *byte_cnt, unsigned char *block);

description This function is used for SMBus Read Block protocol. smbMasterRdBlock sends a slave address, a command byte, a maximum block length, and then reads a block length, followed by a block of up to *byte_cnt bytes from the selected slave into a RAM buffer when called. If the slave attempts to return more than *byte_cnt bytes, it fails with an SMB_BadSize error. Otherwise, *byte_cnt contains the number of bytes actually read. It yields to the scheduler between bytes.

Input:	smbMasterRdBlock has 4 inputs
address:	device address of slave
command:	command byte for slave
byte_cnt:	pointer to max block length in bytes
block:	pointer to storage for block to be read from slave.
Output:	smbMasterRdBlock has 2 outputs
function return:	0 = fail (busy, timeout, no acknowledgment, packet error check fail)
	1 = success
	*byte_cnt contains number of bytes actually read
side effects:	global variable smb_errno contains code for error: SMB_UnknownError if a PEC error is detected.
Stack depth:	14

example

```
unsigned char address, command; byte_cnt;
unsigned char *block;
     address = SLAVEADDRESS;
     command = RETURN_BLOCK;
     byte_cnt = LENGTH;
     block = (unsigned char *)MY_BUFFER;
     .
     .
     status=smbMasterRdBlock(address,command,& byte_cnt, block);
     if(!status) {
         //do error-handling
     }
     //now word has been read into ram buffer
```

smbMasterWrBlock *smb Master Write Block*

function prototype: int smbMasterWrBlock (unsigned char address, unsigned char command, unsigned char byte_cnt, unsigned char *block);

description This function is used for SMBus Write Block protocol. smbMasterWrBlock sends a slave address, a command byte, a block length, and then a block of data from a RAM buffer to the selected slave when called. It yields to the scheduler between bytes.

Input:	smbMasterWrBlock has 4 inputs
address:	device address of slave
command:	command byte for slave
byte_cnt:	block length in bytes
block:	pointer to the block to be read from slave
Output:	smbMasterRdBlock has 2 outputs
function return:	0 = fail (busy, timeout, no acknowledgment)
	1 = success
side effects:	none
Stack depth:	13

example

```
unsigned char address, command; byte_cnt;
unsigned char *block;
     address = SLAVEADDRESS;
     command = RETURN_BLOCK;
     byte_cnt = LENGTH;
     block = (unsigned char *)MY_BUFFER_OF_DATA;
     .
     .
     status=smbMasterWrBlock(address,command, byte_cnt, block);
     if(!status) {
         //do error-handling
     }
     //now word has been sent from ram buffer to slave
```

smbSlaveCmd — *smb Slave Command*

function prototype: int smbSlave Cmd (unsigned char cmd, unsigned char size, unsigned *table)());

description

This function is used to execute a command via a table lookup in a user-defined jump table in flash program memory. smbSlaveCmd ACKs the command word, enables the bus free interrupt and executes a command in the command table when called, after it has determined that the host's command word is in the user's command table. If the command is not found, the command is NACKed.

Input:	smbSlaveCmd has 3 inputs
cmd:	an index to command table for command to be executed
size:	command table size
table:	pointer to command table
Output:	smbSlaveCmd has 2 outputs
function return:	0 = fail command not in table
	other = success, return value determined by selected command
side effects:	global variable smb_errno contains code for error: SMB_Unsupported.
Stack depth:	5 plus stack depth of called function

example:

```
extern unsigned char (*MY_COMMAND_TABLE[]) ();
     .
     .
     if(SMB->sta & SMB_DATA_RDY) {
         cmd = SMB->da;
         if (cmd >= FIRST_COMMAND && cmd <= LAST COMMAND) {;
            status=smbSlaveCmd(cmd,TABLE_SIZE,;
                MY_COMMAND_TABLE);
         }
     if(!status) {
     }
     //now command successfully executed
```

smbSlaveRcvWord *smb Slave Receive Word*

function prototype: int smbSlave RcvWord (int*data);

description This function is used for SMBus Write Word protocol. smbSlaveRcvWord receives a word (two bytes, lsb first) from the host (master) when called, after the user program has determined from the host's command word that a slave receive is required. It yields to the scheduler between bytes.

Input:	smbSlaveRcvWord has 1 input
data:	a pointer to storage for the word to be received from SMBus
Output:	smbSlaveRcvWord has 2 outputs
function return:	0 = fail (timeout, no acknowledgment, packet error check fail)
	1 = success
side effects:	global variable smb_errno contains code for error: SMB_UnknownErrorif a PEC error is detected
Stack depth:	13

example:

```
int data;
     if(SMB->sta & SMBSTA_DRDY) {
         if ( (cmd=SMB->da) == ReadThisWord ) {;
            smb_ACK ();
            status=smbSlaveRcvWord(&data);
         }
     if(!status) {
         //do error-handling {
     }
     //now word has been read and is stored in data
```

smbSlaveSndWord *smb Slave Send Word*

function prototype: int smbSlaveSndWord (int data);

description

This function is used for SMBus Read Word protocol. smbSlaveSndWord sends a word to the host (master) when called, after the user program has determined from the host's command word that a slave send is required.

Input:	smbSlaveSndWord has 1 input
data:	the word to be sent over SMBus
Output:	smbSlaveSndWord has 2 outputs
function return:	0 = fail (timeout, no acknowledgment)
	1 = success
side effects:	global variable smb_errno contains code for error: SMB_AccessDenied if the Master tries to read data.
Stack depth:	17

example:

```
unsigned char data
     data = somevalue;
     .
     .
     if (SMB->sta & SMBSTA_DRDY)
        if( (cmd=SMB->da == SendThisWord ) {
          smb_Ack ();
          status=smbSlaveSndWord (data);
        }
     if (!status) {
         //do error-handling {
     }
     //now byte has been sent
```

smbSlaveSndBlock *smb Slave Send Block*

function prototype: int smbSlaveSndBlock (unsigned char byte_cnt, unsigned char *block);

description This function is used for SMBus Read Block protocol. smbSlaveSndBlock sends a block length, followed by a block of bytes, to the host (master) when called, after the user program has determined from the host's command word that a slave block send is required. It yields to the scheduler between bytes.

Input:	smbSlaveSndBlock has 2 inputs
byte_cnt:	the number of bytes in block
block:	a pointer to the block to be sent over SMBus.
Output:	smbSlaveSndBlock has 2 outputs
function return:	0 = fail (timeout, no acknowledgment)
	1 = success
side effects:	global variable smb_errno contains code for error: SMB_AccessDenied if the Master tries to read data.
Stack depth:	19

example:

```
unsigned char *block;
unsigned char len;
len = BLOCKLEN;
//fill block with data to send
     .
     .
     if (SMB->sta & SMBSTA_DRDY)
        if( (cmd=SMB->da == SendThisBlock ) {
          smb_Ack ();
          status=smbSlaveSndBlock (byte_cnt, block);
        }
     if (!status) {
         //do error-handling {
     }
     //now block has been sent
```

smbSlaveRcvBlock *smb Slave Receive Block*

function prototype: int smbSlaveRcvBlock (unsigned char *byte_cnt, unsigned char *block);

description This function is used for SMBus Write Block protocol. smbSlaveRcvBlock receives a block length, followed by a block of bytes from the host (master) when called, after the user program has determined from the host's command word that a slave block receive is required. It yields to the scheduler between bytes.

Input:	smbSlaveRcvBlock has 2 inputs
block:	a pointer to storage for the block to be read from SMBus.
byte_cnt:	a pointer to the maximum number of bytes in block
Output:	smbSlaveRcvBlock has 3 outputs
function return:	0 = fail (timeout, no acknowledgment, bad size, packet error check fail)
	1 = success
*block:	contains bytes received
side effects:	global variable smb_errno contains code for error: SMB_BadSize or SMB_UnknownError. byte_cnt contains the number of bytes actually received
Stack depth:	21

example:

```
unsigned char *block;
unsigned char len;
len = BLOCKLEN;
//fill block with data
     .
     .
     if (SMB->sta & SMBSTA_RDY)
        if( (cmd=SMB->da == SendThisBlock ) {
          smb_Ack ();
          status=smbSlaveRcvBlock (&len,block);
        }
     if (!status) {
         //do error-handling {
     }
     //now block has been read and is stored at block
```

smbSlaveWord — *smb Slave Word*

function prototype: int smbSlaveWord (unsigned char *datardy, int *data);

description

This function is used for SMBus Read Word and SMBus Write Word protocol. It sends or receives a word depending on whether the Master requests a read or a write. Because the data direction is unknown at the time of the call, valid data should be set up beforehand. The datardy flag indicates whether the data was read by the host or overwritten. This would typically be used to access a variable the master would read but also sometimes update. It yields to the scheduler between bytes.

Input:	smbSlaveSlaveWord has 2 inputs
datardy:	a pointer to a flag indicating read/write direction
data:	a pointer to the word to be read or written
Output:	smbSlaveWord has 3 outputs
function return:	0 = fail (timeout, no acknowledgment, packet error check fail)
	1 = success
*block:	contains word sent or received
side effects:	global variable smb_errno contains code for error: SMB_UnknownError if a PEC error is detected. datardy indicates host read (0) or host write (1)
Stack depth:	19

example:

```
   unsigned char read_write;
   unsigned int *data;
   data = readwritedata; // point to target data);
//Sends or receives word depending on Master
//read_write = 1 if receive
     .
       .
       if (SMB->sta & SMBSTA_DRDY)
         if( (cmd=SMB->da == ReadorWriteThisWord ) {
           smb_Ack ();
           status=smbSlaveWord (&read_write, data);
         }
       if (!status) {
           //do error-handling {
       }
      //now word has been either read or written as master requested
      if (read_write) {    // if data was sent by host
        //read_write data has changed, take appropriate action
      }
```

smbSlaveBlock *smb Slave Block*

function prototype: int smbSlaveBlock (unsigned char *datardy, unsigned char *byte_cnt, unsigned char max_cnt, unsigned char *block);

description This function is used for SMBus Read Block and SMBus Write Block protocol. It sends or receives a block depending on whether the Master requests a read or a write. Because the data direction is unknown at the time of the call, valid data should be set up beforehand. The datardy flag indicates whether the data was read by the host or overwritten. byte_cnt is set to the number of bytes to be sent if the master performs a read, max_cnt is the maximum number of bytes which can be received if the master performs a write. This function would typically be used to access a block of data the master would read but also sometimes update. It yields to the scheduler between bytes.

Input:	smbSlaveSlaveBlock has 4 inputs
datardy:	a pointer to a flag indicating read/write direction
byte_cnt:	a pointer to the number of bytes to be sent
max_cnt:	the maximum number of bytes to be received
block:	a pointer to the data block to be sent or received
Output:	smbSlaveBlock has 6 outputs
function return:	0 = fail (timeout, no acknowledgment, packet error check fail)
	1 = success
*datardy:	indicates direction (0=read, 1=write)
*byte_cnt:	contains number of bytes received if host write
*block:	contains word sent if host write
side effects:	global variable smb_errno contains code for error: SMB_UnknownError if a PEC error is detected. datardy indicates host read (0) or host write (1)
Stack depth:	20

example:

```
   unsigned char read_write;
   unsigned char byte_cnt;
   unsigned char max_cnt;
   unsigned char *block;
   byte_cnt = max_cnt = DATA_BLOCK_SIZE;        //;
   block = (unsigned char *) &readwritedata
// point to target data
// Sends or receives block depending on Master
// read_write = 1 if receive
     .
       .
       if (SMB->sta & SMBSTA_DRDY)
         if( (cmd=SMB->da == ReadorWriteThisBlock ) {
           smb_Ack ();
status=smbSlaveBlock (&read_write, &byte_cnt, max_cnt, block);
         }
       if (!status) {
           //do error-handling {
       }
      //now block has been either read or written
```

```
if (read_write) {    // if data was sent by host
  //block data has changed, take appropriate action
}
```

smbSlaveSndWordNoWait *smb Slave Send Word No Wait*

function prototype: int smbSlaveSndWordNoWait (data);

description This function is used for SMBus Read Word protocol. It sends when an unexpected master read occurs (not immediately preceded by a command word) so that SMBus action is required immediately, not after waiting for the next SMBus event. In practice, the slave would know what to send based on a previous command from the master. It yields to the scheduler between bytes.

Input:	smbSlaveSndWordNoWait has 3 inputs
data:	the word to be sent
size:	??
table:	??
Output:	smbSlaveSndWordNoWait has 3 outputs
function return:	0 = fail (timeout, no acknowledgment)
	1 = success
side effects:	none
Stack depth:	13

example:

```
.
.
if (SMB->sta & SMBSTA_DREQ) { //master wants data right now
      status=smbSlaveSndWordNoWait (standby_data);
   }
if (!status) {
    //do error-handling {
}
//now word has been sent to master
```

smbSlaveSndBlockNoWait *smb Slave Send Block No Wait*

function prototype: int smbSlaveSndBlockNoWait (unsigned char byte_cnt, unsigned char *block);

description This function is used for SMBus Read Block protocol. smbSlaveSndBlockNoWait sends a block length followed by a block of bytes. It sends when an unexpected master read occurs (not immediately preceded by a command word) so that SMBus action is required immediately, not after waiting for the next SMBus event. In practice, the slave would know what to send based on some earlier command from the master. It yields to the scheduler between bytes.

Input:	smbSlaveSndBlockNoWait has 3 inputs
byte_cmd:	the block length
block:	a pointer to the block to be sent
table:	??
Output:	smbSlaveSndBlockNoWait has 3 outputs
function return:	0 = fail (timeout, no acknowledgment)
	1 = success
side effects:	none
Stack depth:	14

example:

```
unsigned char *block;
unsigned char byte_cnt;
block = (unsigned char *) MYDATABLOCK;
byte_cnt = MYBLOCKLEN;
  .
  .
  if (SMB->sta & SMBSTA_DREQ) { //give master the block now
      status=smbSlaveSndBlockNoWait (byte_cnt, block);
   }
if (!status) {
    //do error-handling {
}
//now block has been sent as master requested
```

smbACK — *smb Acknowledgment*

function prototype: void smb_ACK(void);

description This function writes a 1 to the SMBACK register, causing the SMBus engine to generate an ACK on the SMBus. This acknowledgment allows the SMBus transaction to continue. Conversely, withholding it (sending a NACK or allowing a timeout) aborts the SMBus transaction. It is called by the ROM during mulitbyte transactions, but also by the user when a received command from the host is determined to be valid, allowing the host to continue.

Input:	none
Output:	smb_ACK has none
function return:	none
side effects:	none
Stack depth:	0

example:

```
unsigned char cmd;
  .
  .
   if (SMB->sta & SMBSTA_DRDY) {
       cmd = SMB->da;
       if (cmd < FIRST_COMMAND && cmd > LAST_COMMAND)
    {
         smb_NACK(); //command is not valid so abort
    }
    else {
         smb_ACK(); //command is not valid
        //take proper action for command
    }
  .
  .
```

smbNACK *smb NACK*

function prototype: void smb_NACK(void);

description This function writes a 0 to the SMBACK register, causing the SMBus engine to abort the current transaction, or in the case of bus master transactions, to signal the slave to send no further data. It is called by the ROM in case of error or to terminate mulitbyte transactions, but also by the user when a received command from the host is determined to be invalid, signaling the host to abort the transaction.

Input:	none
Output:	smb_ACK has none
function return:	none
side effects:	none
Stack depth:	0

example:

```
unsigned char cmd;
 .
 .
  if (SMB->sta & SMBSTA_DRDY) {
      cmd = SMB->da;
      if (cmd < FIRST_COMMAND && cmd > LAST_COMMAND) {
        smb_NACK(); //command is not valid so abort
   }
   else {
        smb_ACK(); //command is valid
       //take proper action for command
   }
 .
 .
```

smbSetBFI — *smb Set Bus FreeI*

function prototype: void smb_SetBFI(void);

description

This function activates the Bus Free interrupt for the SMBus engine, allowing the SMBus engine to wake the SMB process when the bus becomes free. It is used internally by the ROM code and can also be used by application programs when suspending a process to ensure the process wakes again when one of the possible outcomes is an idle SMBus.

Input:	none
Output:	smb_ACK has none
function return:	none
side effects:	none
Stack depth:	0

example:

```
unsigned char cmd;
 .
 .
  if (SMB->sta & SMBSTA_DRDY)
    if ( (cmd=SMB->da) == ExecuteCommand ) {
      SMB->pec = SMBPEC_PEC_CHK;
      smbSetBFI();    //wake up if bus becomes free
      status=smbCheckPecSlave();
      if (status) {
        smb_ACK();
    do_command();
      }
      else {
        smb_NACK();
        smbWaitBusFree();
       //do error-handling
      }
   }
 }
```

smbWaitBusFree *smb Wait Bus Free*

function prototype: int smbWaitBusFree(char status);

description

This function waits for the SMBus to become free by suspending its process while waiting for SMBus interrupts, clearing unwanted SMBus interrupts by NACKing. It returns when the SMBus is free. The bus free interrupt must be enabled before calling this function, using the smbSetBFI function. This function is used internally by ROM code to clear a failed transaction, and it could also be used by an application program.

Input:	Status-error status of current SMBus transaction
Output:	smb WaitBusFree has 2 outputs
function return:	0 = failed (either input status is zero or an interrupt occurred which was not BUS_FREE)
	1 = success (input status is one and first interrupt is BUS_FREE)
side effects:	clears SMBCTL_BFI_EN
Stack depth:	5

example:

```
  .
  .
 if (SMB->sta & SMBSTA_DRDY)
   if ( (cmd=SMB->da) == ExecuteCommand ) {
     SMB->pec = SMBPEC_PEC_CHK;
     smbSetBFI();    //wake up if bus becomes free
   status=smbCheckPecSlave();
   if (status) {
       smb_ACK();
       do_command();
       }
       else {
          smb_NACK();
          smbWaitBusFree();
          //do error-handling
       }
    }
 }
```

smbCheckPecSlave *smb Check PEC Slave*

function prototype: int smbCheckPecSlave(void);

description This function checks the Packet Error-checking Code sent by the master. It is used when performing a slave SMBus write command transaction to verify the correctness of the Packet Error-checking Code sent by the master. This guards against executing a garbled command. smbCheckPecSlave returns pec okay if the PEC is correct, or if the master does not send a PEC, but fails if the PEC is incorrect, or if the master is actually sending other data, but the command byte has been garbled into a command-only code.

Input:	none
Output:	smbCheckPecSlave has 1 output
function return:	error code (0=fail, 1=PEC okay)
side effects:	none
Stack depth:	9

example:

```
      .
      .
 if (SMB->sta & SMBSTA_DRDY)
   if ( (cmd=SMB->da) == ExecuteCommand ) {
     SMB->pec = SMBPEC_PEC_CHK;
     smbSetBFI();    //wake up if bus becomes free
  status=smbCheckPecSlave();
  if (status) {
      smb_ACK();
      do_command();
       }
       else {
          smb_NACK();
          smbWaitBusFree();
          //do error-handling
       }
    }
 }
```

2.2 Flash Memory Access Routines

There are two sections of flash memory in the bq803xx, reflecting the Harvard architecture of the CPU core. The program flash is a 24k × 22 array starting at address 0x0000. All instructions are 22 bits long. The 8-bit data memory space consists of 2048 bytes of flash data memory at address 0x4000. The top 64 bytes of data memory are reserved and can only be read. Both of these memory spaces are mapped to their respective CPU address spaces and thus can be read directly by the CPU over its program or data memory bus in normal operation. Writing to flash memory requires access through special hardware registers. Because this access requires removing the flash from CPU memory space, no writes to flash program memory can be performed directly from code running in flash program memory. These writes must instead be performed by code running in ROM. The ROM library routines provide read, write, and erase functions for flash program memory and flash data memory. The smallest unit that can be erased in data flash is a row of 64 bytes, in program flash two rows (64 instruction words), and both can be mass erased. The erased state for both is all ones.

2.3 Flash Program Memory Routines

These routines are used to store integers into the 22-bit flash program memory locations. They provide additional nonvolatile storage (beyond the 2016 bytes of the flash data memory), which can be used when the flash program memory is not filled with code. They cannot be used to write code to the flash program memory, because they only access the low 16 bits of the flash program word.

Interrupts are disabled during the execution of these routines, because any attempt to execute flash program code while the flash program memory is not mapped to the CPU address space would be disastrous.

FlashRdRow — *Flash Read Row*

function prototype: void FlashRdRow(unsigned int xadr, unsigned char yadr,unsigned char cnt, int *data);

description

This function reads integers from the low 16 bits of the words in a row of flash program memory into a RAM buffer. If yadr + cnt exceeds row size, the read wraps to the beginning of the row.

Input:	FlashRdRow has 4 inputs
xadr:	the flash row address
yadr:	the flash column address
cnt:	the number of integers to read from flash
Output:	FlashRdRow has 2 outputs
function return:	none
side effects:	none
Stack depth:	1

example:

```
unsigned char *buffer;
unsigned int xadr;
unsigned int yadr;
unsigned char cnt;
buffer = (unsigned char *) MYBUFFER;
cnt = 32;  //the whole row
xadr = FLASH_DATA_ROW;  //the row set aside for data storage
xadr = FLASH_DATA_COLUMN;    //the start address in the row
.
.
FlashRdRow (xadr,yadr,cnt,buffer);//read flash data into ram buffer
//done
```

FlashEraseRow *Flash Erase Row*

function prototype: void FlashEraseRow(unsigned int xadr);

description This function erases two rows of flash program memory. The low bit of the input parameter xadr is ignored; the even/odd row pair is erased. The erased state is all ones.

Input:	FlashEraseRow has 1 input
xadr:	the flash row start address
Output:	FlashEraseRow has 2 outputs
function return:	none
side effects:	none
Stack depth:	2

example:

```
unsigned int xadr;
xadr = FLASH_DATA_ROW;  //the row pair to erase
xadr = FLASH_DATA_COLUMN;    //the start address in the row
.
.
FlashEraseRow(xadr); //erase the even/odd row pair
//now flash row is ready for new data
//done
```

FlashProgRow *Flash Prog Row*

function prototype: void FlashProgRow(unsigned int xadr, unsigned char yadr,unsigned char cnt, int *data);

description This function stores integers from a ram buffer into the low 16 bits of a row of flash program memory. If yadr + cnt exceeds row size, the write wraps to the beginning of the row.

Input:	FlashProgRow has 4 inputs
xadr:	the flash row address
yadr:	the flash column address
cnt:	the number of integers to write to flash
data:	a pointer to the ram buffer area
Output:	FlashProgRow has 2 outputs
function return:	none
side effects:	none
Stack depth:	2

example:

```
unsigned char *buffer;
unsigned int xadr;
unsigned int yadr;
unsigned char cnt;
buffer = (int *) MYBUFFER;
cnt = 32;  //the whole row (twice)
xadr = FLASH_DATA_ROW;  //the start row for data storage
yadr = FLASH_DATA_COLUMN;    //the start column (column 0)
.
.
FlashEraseRow (xadr);//erase the rows first
FlashProgRow (xadr, cnt,buffer);//write ram buffer into flash
buffer += 32;      //write second half of buffer
FlashProgRow (xadr, cnt,buffer);//write ram buffer into flash
//done
```

FlashChecksum *Flash Checksum*

function prototype: long FlashChecksum();

description This function returns the checksum of the instruction flash.

Input:	takes no arguments
Output:	FlashChecksum has 2 outputs
function return:	long integer value of the checksum
side effects:	none
Stack depth:	11

example:

```
unsigned long Csum;
Csum=FlashChecksum();
```

2.4 Flash Data Memory Routines

These routines can be used to erase and write to flash data memory. Note that the writes can be block writes within a page of memory, 64 bytes, but the smallest unit that can be erased is an entire row of memory. In practice, this means that if the target area is not known to be erased, the entire row must be preserved in a buffer, the necessary bytes written to that buffer, then the flash data memory row erased and rewritten with the buffer contents. If the block to be written crosses a row boundary, this process must be done twice. The function FdataWrBlock handles this necessary preservation; so, all that is required when using it is to set up the block of data to be written.

FlashProgRow *Flash Prog Row*

function prototype: void FdataProgRow(unsigned char xadr, unsigned char yadr, unsigned char cnt, unsigned char *data)

description This function stores bytes from a buffer into a selected row of flash data memory, starting at a specified column. If yadr + cnt exceeds row size, the write wraps to the beginning of the row. Erasure of the row, if necessary, must be done in a separate operation, as well as preservation of contents of the row not included in the write.

Input:	FdataProgRow has 4 inputs
xadr:	the flash row address
yadr:	the starting column in the row
cnt:	the number of bytes to write to flash
data:	a pointer to the buffer area
Output:	FdataProgRow has 2 outputs
function return:	none
side effects:	none
Stack depth:	5

example:

```
unsigned char *buffer;
unsigned char xadr, yadr;
unsigned char cnt;
buffer = (unsigned char *) MYBUFFER; //data to write
cnt = 32;  //number of bytes to write
xadr = FLASH_DATA_ROW;  //the row set aside for data storage
yadr = FLASH_DATA_COL;    //the starting column for the write
.
.
FlashEraseRow (xadr);//erase the row first
FlashProgRow (xadr, yadr, cnt, buffer);//write ram buffer to flash
//done
```

FdataProgWord *Fdata Prog Word*

function prototype: void FdataProgWord(unsigned char *addr, unsigned char data)

description This function writes one byte to a selected flash data memory location in the range 0x4000–0x47bf. Writes outside the range are ignored. Bits can only be written to zero, so the target byte should contain all ones (erased).

Input:	FdataProgWord has 2 inputs
addr:	a pointer to the data flash location to be written
data:	the byte to be written
Output:	FdataProgWord has 2 outputs
function return:	none
side effects:	none
Stack depth:	4

example:

```
unsigned char data;
unsigned char *addr;
data = my_data_byte;
//setup data byte
FlashProgRow (addr, data);//write data to flash
//done
```

FdataEraseRow *Fdata Erase Row*

function prototype: void FdataEraseRow(unsigned char xadr)

description

This function erases 64 bytes starting at the selected row. Note that one row is 32 bytes. An xadr greater than 0x40 wraps to the beginning of flash data memory. Note that the low bit of the row address is ignored. Reserved flash cannot be erased.

Input:	FdataEraseRow has 1 input
xadr:	the flash row address
Output:	FdataEraseRow has 2 outputs
function return:	none
side effects:	none
Stack depth:	3

example:

```
  unsigned char *buffer;
  unsigned char xadr,yadr;
  unsigned char cnt;
  cnt = 12;   //number of bytes to write
  xadr - FLASH_DATA_ROW;  //the row set aside for data storage
  yadr - FLASH_DATA_COL;  //the starting column for the write
 .
 .
  FdataEraseRow (xadr);//erase the row first
  FdataProgRow (xadr,yadr,cnt,buffer);//write ram buffer to flash
//done
```

FdataMass Erase *Fdata Mass Erase*

function prototype: void FdataMassErase(void)

description This function erases all of flash data memory.

Input:	none
Output:	FdataMassErase has 2 outputs
function return:	none
side effects:	none
Stack depth:	7

example:

```
  unsigned char *buffer;
  unsigned char xadr,yadr;
  unsigned char cnt;
  //get ready to put new info into flash data memory
  .
  .
 //but first erase the whole thing;
  FdataMassErase();
//done
//continue
```

2.4.1 Math Library Routines

Math routines accessible by function call

Calls to these routines are not generated automatically by the C compiler. They are special-purpose math routines useful in some of the calculations commonly used in battery management.

accumulate *accumulate*

function prototype: void Accumulate(Accum *accum, double val)

description Adds a double on the stack to an extended-precision (48 bit) integer pointed to by accum. This extended-precision data type, called Accum, is used to hold the accumulated charge in battery gas-gauging applications.

```
typedef struct {
        unsigned char[6];
  } Accum;
```

Input:	accumulate has 2 inputs
accum:	pointer to accumulator
val:	value to be added
Output:	value is added to Accum
Stack depth:	1

example:

```
    accum total_charge;
    double charge_increment;
    charge_increment = get_charge();  //pick up charge increment
    Accumulate (&total_charge, charge_increment); //add to total
    .
    .
  //done
```

exp — *double exp (double d)*

function prototype: double exp (double d)

description Returns a double that is 2=2.718 to power defined by input parameter.

Input:	accumulate has 2 inputs
accum:	d: pointer to accumulator
Output:	result of raising e to said power.
Stack depth:	13

example:

```
double mVolts, dTemp;
double c1 = 1.24;
mVolts = getAD(TS1) ;
dTemp = C1* exp(mVolts);
```

log — *double log (double f)*

function prototype: double log (double f)

description Returns a double that is the natural logarithm of the input parameter.

Input:	f: value of which to compute the logarithm.
Output:	result as a double of computing the natural logarithm of the input parameter.
Stack depth:	20

example:

```
double edv;
double temp;
temp = ReadAD();
edv = log(temp);
```

long abs_long

long abs_long (long li)

function prototype: long abs_long (long li)

description returns the absolute value of a long integer (4-byte).

Input:	long integer to be converted
Output:	long integer result
Stack depth:	0

example:

```
long liln, liOut;
liOut = abs_long(liln);
```

int abs_int(int In)

function prototype: int abs_int(int In)

description

Returns an integer that is the absolute value of the input integer.

Input:	integer to be converted
Output:	integer result of absolute value of input.
Stack depth:	0

example:

```
int iln, iOut;
iOut = abs_int(iln);
```

int round (double x) *int round (double x)*

function prototype: int round (double x)

description Returns the rounded value of a signed double to a signed integer. If x is greater than MAX_INT, then MAX_INT is returned. If x is less than MIN_INT is returned.

Input:	x value to be converted to an integer.
Output:	Result of conversion.
Stack depth:	6

example:

```
int i;
double f;
i = round (f);
```

int AB_div_C — *int AB_div_C (int a, int b, int c)*

function prototype: int AB_div_C (int a, int b, int c)

description Returns the result of multiplying a by b and then dividing by c. The routine uses intermediate long values to preserve the precision of the math.

Input:	a, b, c signed integer operands.
Output:	integer result of (a*b)/c.
Stack depth:	0

example:

```
int iTemp;
int imVolt;
int iDeg;
int ipVolts;
iTemp = AB_div_C (imVolt, iDeg, ipVolts);
```

unsigned int unsigned_AB_div_C *(unsigned int a, unsigned int b, unsigned int c)*

function prototype: unsigned int unsigned_AB_div_C (unsigned int a, unsigned int b, unsigned int c)

description Returns the result as an unsigned integer of (a*b)/c where all operands are unsigned integers. Internally, the function uses long math to retain precision.

Input:	a, b, c unsigned math operands.
Output:	result of (a*b)/c.
Stack depth:	13

example:

```
unsigned int iTemp;
unsigned int imVolt;
unsigned int iDeg;
unsigned int ipVolts;
iTemp = unsigned_AB_div_C (imVolt, iDeg, ipVolts);
```

unsigned int iRoot *unsigned int iRoot*

function prototype: unsigned int iRoot (iRoot(iRootFuncPtr func, unsigned int x1, unsigned int x2, int eq_val, void*ptr)

description Find the integer root of the function 'func' between the x1 and x2 bounds.

Input: func - pointer to the function of the form: f(x, void*ptr), where the pointer is typically to function coefficients, if needed.

x1 - unsigned integer lower bound for root

x2 - unsigned integer upper bound for root

eq_val - integer function equivalence, that is, the equation solved is of the form: f(x) - eq_val = 0

ptr - see definition of the function above.

Output: unsigned integer root.

Stack depth: 16 + stack used by 'func'

example:

```
static int tquad (unsigned int x, void *coef)
{ // linear function

return ( (int)(4 *x) - 100);
}
int testiRoot(void)
{
int i=0;
i = iRoot(tquad, 10, 50, 0, (void *)0);
return i;
}
```

unsigned int calculate_percent *(unsigned int x, unsigned int max)*

function prototype: unsigned int calculate_percent (unsigned int x, unsigned int max)

description Returns (x/max) * 100 preserving integer precision.

Input:	x - a partial amount
	max - the maximum amount
Output:	percent of x in max.
Stack depth:	17

example:

```
unsigned int testcalculatepercent(void)
{
unsigned int p=30, x = 100;

return calclate_percent(p,x);
}
```

unsigned int calculate_percent_of *(PercentWord p, unsigned int max)*

function prototype: unsigned int calculate_percent_of(PercentWord p, unsigned int max)

description Returns the result of computing p percent of x, a total: (p*x)/100.

Input:	p - percentage of total value
	x - total or full value.
Output:	Percent of total value
Stack depth:	9

example:

```
int testcalculateperof(void
{
int p=30, x = 90;

return calculate_percent_of(p,x);
}
```

void sha1_mac *(const uchar *AuthKey, unsigned long *sha1_digest)*

function prototype: void sha1_mac(const uchar *AuthKey, unsigned long *sha1_digest)

description This function is used to provide a response to a challenge for authentication of the part. Host systems can use this to determine if a proper part has been added to the system.

Input: AuthKey - points to an array of value representing an Authorization key.

sha1_digest - initially an array of value representing the challenge code.

Output: sha1_digest - is written over to contain the proper response to the initial challenge.

Stack depth: 89

example:

```
int testSha1(void)
{
unsigned int AuthKey[8];
unsigned char sha1_digest[20];
sha1_digest[0] = 0x61;
sha1_digest[1] = 0x62;
sha1_digest[2] = 0x63;
sha1_digest[3] = 0x20;
sha1_digest[4] = 0x20;
sha1_digest[5] = 0x20;
sha1_digest[6] = 0x20;
sha1_digest[7] = 0x20;
sha1_digest[8] = 0x20;
sha1_digest[9] = 0x20;
sha1_digest[10] = 0x20;
sha1_digest[11] = 0x20;
sha1_digest[12] = 0x20;
sha1_digest[13] = 0x20;
sha1_digest[14] = 0x20;
sha1_digest[15] = 0x20;
sha1_digest[16] = 0x20;
sha1_digest[17] = 0x20;
sha1_digest[18] = 0x20;
sha1_digest[19] = 0x20;
AuthKey[0] = 0x0123;
AuthKey[1] = 0x4567;
AuthKey[2] = 0x89ab;
AuthKey[3] = 0xcdef;
AuthKey[4] = 0xfedc;
AuthKey[5] = 0xba98;
AuthKey[6] = 0x7654;
AuthKey[7] = 0x3210;
sha1_mac( (unsigned char *)AuthKey, sha1_digest);
sha1_mac( (unsigned char *)AuthKey, sha1_digest);
smbSlaveSndBlock(20, sha1_digest);
return1;
```

2.5 Math Routines Called by the Compiler

The C compiler automatically generates calls to these routines to implement basic arithmetic functions. They can also be called from assembly language code, using the CALL instruction, as long as the C compiler's parameter-passing conventions are observed. Stack handling precautions must be observed: stack parameter passing uses big-endian ordering on the stack. This means the msb of a parameter is at the lower memory address. Parameters are pushed on the stack in the order given. The stack depths reported are the pushes within the routine, so any pushes required to save registers or pass parameters must be added.

mulhi3

description This function multiplies two signed integers.

Input:	r3:r2:	int a
	r1:r0:	int b
Output:	r1:r0:	int (a * b)
Stack depth:	0	

mulhisi3

description This function multiplies two signed integers to a long.

Input:	r3:r2:	int a
	r1:r0:	int b
Output:	r3:r2:r1:r0:	long int (a * b)
Stack depth:	4	

mulsi3

description This function multiplies two longs to a long. The result is truncated to a long.

Input:	stack:	long int a
	stack:	long int b
Output:	r3:r2:r1:r0:	long int (a * b)
Stack depth:	4	

umulhisi3

description This function multiplies two unsigned ints to long.

Input:	r3:r2:	unsigned int a
	r1:r0:	unsigned int b
Output:	r3:r2:r1:r0:	long int (a * b)
Stack depth:	2	

mulsf3

description This function multiplies two, 4-byte doubles.

Input:	r3:r2:r1:r0:	double a
	stack:	double b
Output:	r3:r2:r1:r0:	double (a * b)
Stack depth:	5	

addsf3

description This function adds two (floating point), 4-byte doubles.

Input:	r3:r2:r1:r0:	double a
	stack:	double b
Output:	r3:r2:r1:r0:	double (a * b)
Stack depth:	2	

floatqisf2

description This function converts a signed or unsigned char to a 4-byte double.

Input:	0:	char to convert
	Z:	set if converting from unsigned char
Output:	r3:r2:r1:r0:	input char converted to double
Stack depth:	0	

floathisf2

description This function converts a signed integer to a 4-byte double.

Input:	r2:r1:	the signed int to convert
Output:	r3:r2:r1:r0:	the converted double
Stack depth:	0	

floatsisf2

description This function converts a long to a 4-byte double.

Input:	r3:r2:r1:r0:	the long int to convert
Output:	r3:r2:r1:r0:	the converted double
Stack depth:	2	

fix_truncsfhi2

description This function converts a double to a signed integer. It does not round to the nearest integer; it truncates.

Input:	r3:r2:r1:r0:	the double to convert
Output:	r1:r0:	the converted result
Stack depth:	0	

fixuns_truncsfhi2

description This function converts a double to an unsigned integer. It does not round to the nearest integer; it truncates.

Input:	r3:r2:r1:r0:	the double to convert
Output:	r1:r0:	the converted result
Stack depth:	0	

fix_truncsfhi2

description This function converts a double to an signed integer.

Input:	r3:r2:r1:r0:	the double to convert
Output:	r3:r2:r1:r0:	the converted result
Stack depth:	2	

fixuns_truncsfsi2

description This function converts a double to an unsigned long integer. It does not round to the nearest integer, it truncates.

Input:	r3:r2:r1:r0:	the double to convert
Output:	r3:r2:r1:r0:	the converted result
Stack depth:	1	

divmodhi4

description This function divides two signed integers. Returns the quotient a/b and the remainder.

Input:	r1:r0:	int a
	stack:	int b
Output:	r1:r0:	quotient of (int a) / (int b)
	r3:r2:	remainder
Stack depth:	3	

udivmodhi4

description This function divides two unsigned integers. Returns the quotient a/b and the remainder.

Input:	r1:r0:	int a
	stack:	int b
Output:	r1:r0:	quotient of (unsigned int a) / (unsigned int b)
	r3:r2:	remainder
Stack depth:	1	

divmodsi4

description This function divides two signed long integers. Returns the quotient a/b and the remainder.

Input:	r3:r2:r1:r0:	long int a
	stack:	long int b
Output:	r3:r2:r1:r0:	quotient of (long int a) / (long int b)
	stack:	long remainder
Stack depth:	7	

divsf3

description This function divides two doubles. Returns the quotient

Input:	r3:r2:r1:r0:	double a
	stack:	double b
Output:	r3:r2:r1:r0:	quotient of (double a) / (double b)
Stack depth:	4	

2.6 I2C Functions

These functions implement a software-driven i2c bus on the bq803xx, in addition to the SMBus engine provided in hardware. This bus resides on pins determined by the user, who must also provide support functions to manipulate the chosen pins. The user functions set clock and data pin states, read the states, generate timing delays, and set timeouts for clock stretches. Because the I/O access is provided in the user functions, the user determines whether they are polled or interrupt-driven, and whether they yield to the scheduler.

The user must provide the following functions to support the higher-level functions in the library ROM:

```
extern void i2c_clockhi(void);                 // release the clock pin to high
extern void i2c_clocklo(void);                 / set the clock pin low
extern unsigned char i2c_wait_clockhi(void);   //release clock pin and
                                               //wait for clock hi, up to limit return 0 if clock is not high
extern void i2c_datahi(void);                  // release data pin to high
extern void i2c_datahi(void);                  // release data pin to high
extern void i2c_datalo(void);                  // set data pin low
extern unsigned char i2c_datain(void);         //read data pin into bit 0
extern void i2c_wait_quarter_bit(void);        // 1/4 of clock period
```

See the section on *interrupt vectors and hooks* for further information about linking these support functions to the ROM i2c code.

In addition, the user's code must initialize the i2c bus by setting clock and data lines high and optionally providing power to the i2c device, and optionally removing power after the transaction. These initialization routines are provided by the user and called in the user's code. The ROM library routines assume the bus has been properly initialized. These are declared (as a reminder) in i2c.h as:

```
extern void i2c_power_up(void);
extern void i2c_power_down(void);
```

Note that the reported stack depths depend on the stack depths of the user-provided, low-level functions for accessing clock and data pins and providing timing. These depths vary, depending on the implementation by the user. You must add the reported stack depth to the stack depth of the listed user-provided, low-level function that has the greatest stack depth.

All of the i2c library functions return either a zero for failure or a 1 for success. In addition, the global error variable i2c_errno is set to one of the following values:

```
enum i2c_errors {
         ERR_NACKED = 1,
         ERR_TIMEOUT,
         ERR_SHORT,      //bus is shortedERR_COMPARE
         }
```

I2CReadBlock *I2C Read Block*

function prototype: unsigned char I2CReadBlock (unsigned char addr, unsigned char cmd, unsigned char cnt, unsigned char *data);

description This function reads a block on a selected i2c peripheral into a RAM buffer.

Input:	I2CReadBlock has 4 input
addr:	the address of the i2c peripheral (bits 7..1)
cmd:	the command understood by the peripheral
cnt:	the length of the data block to be read
data:	a pointer to storage for the block received.
Output:	I2CReadBlock has 2 outputs
function return:	1 = success
	0 = fail
side effects:	global variable i2c_errno contains code for error: ERR_NACKED, ERR_TIMEOUT, or ERR_SHORT
Stack depth:	**12 plus the max stack depth of:** **datahi()** **datalo()** **clockhi()** **clocklo()** **waitclockhi()** **waitquartersecond()** **datain()**

example:

```
unsigned char *block;
unsigned char len;
unsigned char status;
len = BLOCKLEN;
//allocate ram buffer block
   .
   .
status=I2CReadBlock(EE,READ_BLK,byte_cnt, block);
if(!status) {
      //do error-handling
}
  //now block has been read
```

I2CWriteBlock *I2C Write Block*

function prototype: unsigned char I2CWriteBlock (unsigned char addr, unsigned char cmd, unsigned char cnt, unsigned char *data);

description This function writes a block from a buffer to a selected i2c peripheral.

Input:	I2CWriteBlock has 4 input
addr:	the address of the i2c peripheral (bits 7..1)
cmd:	the command understood by the peripheral
cnt:	the length of the data block to be written
data:	a pointer to storage for the block sent.
Output:	I2CWriteBlock has 2 outputs
function return:	1 = success
	0 = fail
side effects:	global variable i2c_errno contains code for error: ERR_NACKED, ERR_TIMEOUT, or ERR_SHORT
Stack depth:	**14 plus the max stack depth of:** **datahi()** **datalo()** **clockhi()** **clocklo()** **waitclockhi()** **waitquartersecond()** **datain()**

example:

```
unsigned char *block;
unsigned char len;
unsigned char status;
len = BLOCKLEN;
//allocate and fill block with data to send
   .
   .
status=I2CWriteBlock(EE,READ_BLK,byte_cnt, block);
if(!status) {
      //do error-handling
}
  //now block has been sent
```

I2CDeviceAvail — *I2C Device Avail*

function prototype: unsigned char I2CDeviceAval(unsigned char addr, unsigned int wait);

description This function attempts to get an address acknowledgment from a selected device, to determine whether the device is present on the i2c bus. It continues until the device acknowledges or the specified retry count is exceeded.

Input:	I2CDeviceAvail has 2 input
addr:	the address of the i2c peripheral (bits 7..1)
wait:	number of times to try to address peripheral
Output:	I2CDeviceAvail has 2 outputs
function return:	1 = success
	0 = fail
side effects:	global variable i2c_errno contains code for error: ERR_NACKED, ERR_TIMEOUT, or ERR_SHORT
Stack depth:	**10 plus the max stack depth of:** **datahi()** **datalo()** **clockhi()** **clocklo()** **waitclockhi()** **waitquartersecond()** **datain()**

example:

```
unsigned char *block;
unsigned char len;
unsigned char status;
len = BLOCKLEN;
//allocate and fill block with data to send
  .
  .
//TEST WHETHER DEVICE IS PRESENT FIRST:
if (I2CDeviceAvail(EE,MY_TIMEOUT){
status=I2CWriteBlock(EE,READ_BLK,byte_cnt, block);
if(!status) {
      //do error-handling
 }
}
   //now block has been sent
 else
   //can't find device
```

I2CCompareBlock *I2C Compare Block*

function prototype: unsigned char I2CCompareBlock(unsigned char addr, unsigned char cmd, unsigned char cnt, unsigned char *data);

description This function compares a block of data in memory with a block of data read from a selected i2c device.

Input:	I2CCompareBlock has 4 input
addr:	the address of the i2c peripheral (bits 7..1)
cmd:	the command understood by the peripheral
cnt:	the length of the data block to be compared
data:	a pointer to storage for the block to be compared
Output:	I2CCompareBlock has 2 outputs
function return:	1 = success
	0 = fail
side effects:	global variable i2c_errno contains code for error: ERR_NACKED, ERR_TIMEOUT, ERR_SHORT, or ERR_COMPARE
Stack depth:	**15 plus the max stack depth of:** **datahi()** **datalo()** **clockhi()** **clocklo()** **waitclockhi()** **waitquartersecond()** **datain()**

example:

```
unsigned char *block;
unsigned char len;
unsigned char status;
len = BLOCKLEN;
//allocate and fill block with data to send
  .
  .
status=I2CWriteBlock(EE,READ_BLK,byte_cnt, block);
if(!status) {
      //do error-handling, block not written
 }
}
   //now block has been sent, verify the write:
status=I2CCompareBlock(EE,READ_BLK,byte_cnt, block);
if(!status) {
      //do error-handling, block does not compare
}
   //now block has been sent, and verified
```

boot-ROM Routines

This chapter describes the boot-ROM routines for the bq803xx.

Topic **Page**

3.1 boot-ROM Routines

These routines are available immediately after system reset, when control is not transferred to the program in flash memory (i.e., during development). They are accessible via the SMBus, by sending commands to the bq803xx at address 0x16. These routines program, read, and erase flash, as well as read and write RAM and the registers of hardware peripherals. They are implemented as a jump table in ROM called when an SMBus command is detected by the boot-ROM code. Routines 0x01-0x07 are used to program and erase the instruction flash memory. Routines 0x0e-0x12 program and erase data flash memory.

3.1.1 Smb_FlashWrAddr

SMBus protocol: — write block[3]

SMBus command: — 0x00

description: — This function writes a block of three bytes containing the row and column addresses for a subsequent read from flash program memory. The first two bytes are row (lsb/msb); the third byte is the column address.

3.1.2 Smb_FlashRdWord

SMBus protocol: — read block[3]

SMBus command: — 0x01

description: — This function reads a complete 22-bit flash memory word from the address previously set by Smb_FlashWrAddr. The result is read as a 3-byte block, lsb first. It increments the column address.

3.1.3 Smb_FlashRdRow

SMBus protocol: — read block[96]

SMBus command : — 0x02

description: — This function reads a complete row of 32, 22-bit flash memory words (96 bytes, greater than allowed by the SMBus spec) from the row address previously set by Smb_FlashWrAddr. Each 22-bit word is returned in 3 bytes, lsb first.

3.1.4 Smb_FlashRowCheckSum

SMBus protocol: — read block[4]

SMBus command: — 0x03

description: — This function reads the 4-byte checksum (lsb..msb) of a row of 32, 22-bit flash memory words at the row address previously set by Smb_FlashWrAddr.

3.1.5 Smb_FlashProgWord

SMBus protocol: — write block[6]

SMBus command : — 0x04

description: — This function writes a 22-bit word to the specified row and column address. The block sent is a 6-byte block, consisting of the row (lsb/msb) and column addresses, then the 22-bit word to be programmed as a 3-byte block, lsb first.

3.1.6 Smb_FlashProgRow

SMBus protocol: — write block

SMBus command : — 0x05[98]

description: — This function writes a complete row of 32, 22-bit words to the row address (lsb/msb) set by the first 2 bytes of the block sent. This is then followed by 32 words to be written. Each 22-bit word is sent as 3 bytes, lsb first.

3.1.7 Smb_FlashEraseRow

SMBus protocol: — write word

SMBus command : — 0x06

description: — This function erases 2 rows of 32, 22-bit words at the row address contained in the word written (lsb/msb). Note that address is a 32-word wide row address, but that 64 words are erased starting from that address.

3.1.8 Smb_FlashMassErase

SMBus protocol: — write word

SMBus command : — 0x07

description: — This function erases the complete flash program memory. The word written must be 0x83de.

3.1.9 FlashExecute

SMBus protocol: — send command

SMBus command : — 0x08

description: — This function transfers execution to the flash program memory by mapping the flash program memory into the CPU address space and then jumping to the flash reset vector.

3.1.10 SetAddr

SMBus protocol: — write word

SMBus command : — 0x09

description: — This function writes the 16-bit address (lsb/msb) for a subsequent read or write to RAM or I/O space

3.1.11 PokeByte

SMBus protocol: — write word

SMBus command : — 0x0a

description: — This function writes a single byte to RAM or I/O space at the address previously set by SetAddr. The byte written is the lsb of the word sent over SMBus.

3.1.12 PeekByte

SMBus protocol: — read word

SMBus command : — 0x0b

description: — This function reads a single byte of RAM or I/O space from the address previously set by SetAddr and returns it as the lsb of the word read from SMBus.

3.1.13 ReadRAMBlk

SMBus protocol: — read block[32]

SMBus command : — 0x0c

description: — This function reads 32 bytes of RAM or I/O space from the address previously set by SetAddr.

3.1.14 Version

SMBus protocol: — read word

SMBus command : — 0x0d

description: — This function returns the ROM version number (lsb/msb). Major revision number is in msb, minor revision number is in lsb.

3.1.15 Smb_FdataChecksum

SMBus protocol: — read word

SMBus command : — 0x0e

description: — This function returns the checksum for the data flash memory from 0x4000 to 0x47e0 (it does not include the 32 reserved data flash memory locations) in lsb/msb order.

3.1.16 Smb_FdataProgWord

SMBus protocol: — write block[3]

SMBus command : — 0x0f

description: — This function programs one byte of flash data memory. The block consists of the memory address (lsb/msb) and the data to be written. It cannot be used to program the reserved bytes.

3.1.17 Smb_FdataProgRow

SMBus protocol: — write block[33]

SMBus command : — 0x10

description: — This function programs an entire row of 32 bytes of flash data memory. The block consists of the memory row address and 32 bytes of data to be written. If the row programmed is the last row, the reserved bytes are not affected.

3.1.18 Smb_FdataEraseRow

SMBus protocol: — write word

SMBus command : — 0x11

description: — This function erases 2 rows (64 bytes) of flash data memory. The word sent contains the memory row address in the lsb. If the row erased is the last row, the reserved bytes are not affected.

3.1.19 Smb_FdataMassErase

SMBus protocol: — write word

SMBus command : — 0x12

description: — This function erases the entire flash data memory. The word written must be 0x83de. The reserved bytes are not affected.

Appendix A
SLUU225–October 2005

ROM Entry Points

4004	SMB ROM functions
4005	smbMasterWrWord
4006	smbMasterRdWord
4007	smbMasterRdBlock
4008	smbMasterWrBlock
4009	smbSlaveCmd
400a	smbSlaveRcvWord
400b	smbSlaveSndWord
400c	smbSlaveSndBlock
400d	smbSlaveRcvBlock
400e	smbSlaveWord
400f	smbSlaveBlock
4010	smbSlaveSndWordNoWait
4011	smbSlaveSndBlockNoWait
4012	smb_ACK
4013	smb_NACK
4014	FlashRdRow
4015	FlashProgRow
4016	FlashEraseRow
4017	SetAddr
4018	PokeByte
4019	PeekByte
401a	ReadRAMBlk
401b	mulhi3
401c	mulhisi3
401d	umulhisi3
401e(1)	mulsi3
401f	mulsf3
4020	divmodhi4
4021	udivmodhi4
4022	divmodsi4
4023	divsf3
4024	addsf3
4025	floatqisf2
4026	floathisf2
4027	fix_truncsfhi2
4028	fixuns_truncsfhi2
4029	accumulate

(1) The mulsi3 in v. 1.4 ROM does not work correctly. It cannot be called by its absolute address, but should instead be called by name. The development tools links the call to the library copy, which is placed in flash memory.

402a	exp
402b	log
402c	fix_truncsfsi2
402d	fixuns_truncsfsi2
402e	floatsisf2
4031	Reserved
4032	Reserved
4033	Reserved
4034	Reserved
4035	Reserved
4036	Reserved
4037	FdataProgRow
4038	FdataProgWord
4039	FdataEraseRow
403a	FdataMassErase
403b	I2CReadBlock
403c	I2CWriteBlock
403d	I2CDeviceAvail
403e	I2CCompareBlock
403f	Reserved
4040	Reserved
4041	smbCheckPecSlave
4042	smbSetBFI
4043	smbWaitBusFree
ROM entry points	
8004	rom_execute
8005	smbMasterWrWord
8006	smbMasterRdWord
8007	smbMasterRdBlock
8008	smbMasterWrBlock
8009	smbSlaveCmd
800a	smbSlaveRcvWord
800b	smbSlaveSndWord
800c	smbSlaveSndBlock
800d	smbSlaveRcvBlock
800e	smbSlaveWord
800f	smbSlaveBlock
8010	smbSlaveSndWordNoWait
8011	smbSlaveSndBlockNoWait
8012	smb_ACK
8013	smb_NACK
;FLASH functions	
8014	FlashRdRow
8015	FlashProgRow
8016	FlashEraseRow
;Peek Poke	
8017	SetAddr
8018	PokeByte
8019	PeekByte

801a	ReadRAMBlk
;Math functions, Table entry	
801b	mulhi3
801c	mulhisi3
801d	umulhisi3
801e	mulsi3
801f	mulsf3
8020	divmodhi4
8021	udivmodhi4
8022	divmodsi4
8023	divsf3
8024	addsf3
8025	floatqisf2
8026	floathisf2
8027	fix_truncsfhi2
8028	fixuns_truncsfhi2
8029	accumulate
802a	exp
802b	log
802c	fix_truncsfsi2
802d	fixuns_truncsfsi2
802e	floatsisf2
;FDATA functions	
8037	FdataProgRow
8038	FdataProgWord
8039	FdataEraseRow
803a	FdataMassErase
;I2C functions	
803b	I2CReadBlock
803c	I2CWriteBlock
803d	I2CDeviceAvail
803e	I2CCompareBlock
;More SMB, Fdata functions	
803f	reserved
8040	reserved
8041	smbCheckPecSlave
8042	smbSetBFI
8043	smbWaitBusFree
8044	FlashChecksum
8045	FlashMassErase
;More math functions	
;built-ins	
8046	umodqi3
8047	udivmodsi4
8048	cmpsf3
8049	udivmodqi4
804a	sqrt
;gauge math	

804b	abs_long
804c	abs_int
804d	round
804e	AB_div_C
804f	unsigned_AB_div_C
8050	poly
8051	mul_shift16
8052	umul_shift16
8053	round_shift16
8054	iRoot
8055	calculate_percent
8056	calculate_percent_of
;new bq8030 FDATA function	
8057	copy
8058	FdataFullErase
8059	sha1_mac

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265