# Driving the AT42QT2160 QMatrix Sensor IC

1. Introduction

This application note shows how the AT42QT2160-MMU (QT2160) 16-key QMatrix™ Sensor IC can be connected to a microprocessor to provide touch input functionality. Example code is provided to demonstrate how easily the QT2160 can be incorporated into a design. The code can be run "as is" or used as a starting point for QT2160-based projects. The example compiles to under 1 Kbyte of code including all necessary I^{2}C-compatible driver functions.

The example code is written in the C programming language and can easily be adapted for many processor types. See for a listing of the complete program.

Figure 1-1. Circuit Configuration for the Example Project
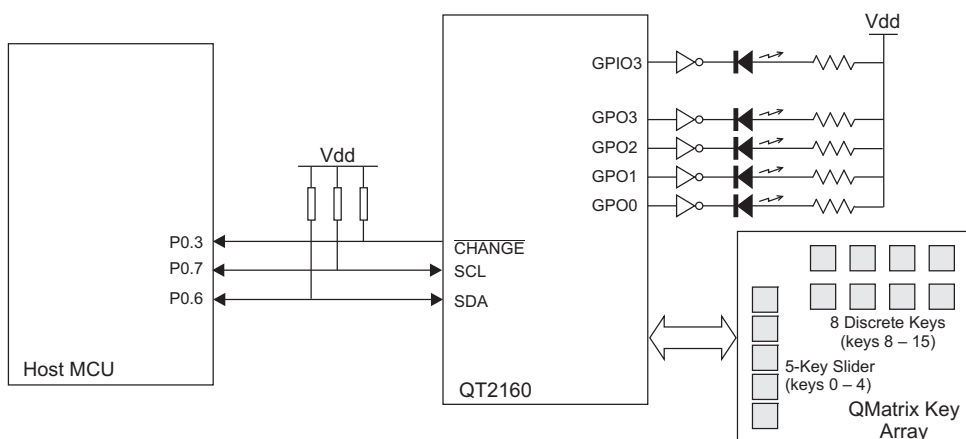
2. Overview of the QT2160

2.1 Introduction

The QT2160 is designed for use with up to 16 keys and a slider (constructed from 2 keys up to 8 keys). There are three dedicated general-purpose input/outputs (GPIO), which can be used as inputs (for example, for mechanical switches) or as driven outputs. There are eight shared general-purpose outputs (GPO). Pulse switch modulation (PWM) control can be applied to all GPIO and GPO pins.

Keys are configured in a matrix format that minimizes the number of required scan lines and device pins. The key electrodes can be designed into a conventional printed circuit board (PCB) or flexible printed circuit board (FPCB) as a copper pattern, or as printed conductive ink on plastic film. The QT2160 can be fine tuned to optimize operation with the electrode design.

Full details of the QT2160 can be found in the QT2160 datasheet, which should be read in conjunction with this application note.

2.2 QT2160 Host Interface

The QT2160 connects to a host controller by means of an I^{2}C-compatible serial interface. This uses a common two-wire connection between the host QT2160 and other I^{2}C-compatible devices in the system. The bus protocol takes care of all addressing functions and bidirectional data transfer. Full details can be found in the Philips document "The I^{2}C-Bus Specification".

In addition, the QT2160 features a $\overline{\text{CHANGE}}$ pin, which can be used either to wake the host in a battery-powered application or as an interrupt signal to inform the host when the QT2160 status has changed. The $\overline{\text{CHANGE}}$ pin can be left unconnected in systems where the QT2160 is polled via the I^{2}C-compatible bus on a regular basis.

2.3 QT2160 Programming Model

All communication with the QT2160 takes place via a set of addressable, 8-bit registers. All read and write operations to the registers are made via the I^{2}C-compatible bus. The protocol for performing reads and writes is based on that used for standard serial EEPROM devices and is fully described in the QT2160 datasheet. The example code presented in this application note includes a suitable I^{2}C-compatible driver.

3. The Example QT2160 Project

3.1 Introduction

The example project shows how to use the host interface to read real-time touch information from the QT2160. It also shows how the discrete output pins of the QT2160 can be simultaneously driven to reflect the touch status.

3.2 Circuit Configuration

shows the circuit used for this example. Representative touch keys are connected to the X/Y drive signals as described in the datasheet. The example uses a five-key slider comprising keys 0 – 4, and eight discrete keys connected as keys 8 – 15. Connection to the host microcontroller uses the two I^{2}C-compatible pins (SCL and SDA) and the $\overline{\text{CHANGE}}$ pin. The example uses LEDs on QT2160 pins GPO0 to GPO3 to indicate touches on keys 8 – 11. It also uses an LED on GPIO3 to indicate the slider position by means of the PWM function.

The following points should be noted:

- The QT2160 I^{2}C-compatible slave address can be selected via pins I2CA1 and I2CA0. The demonstration program uses address 0x0d, which is selected by connecting I2CA1 = I2CA0 = Vss.
- Pull-up resistors are required on the I^{2}C-compatible signals (SCL and SDA). Either discrete pull-up resistors or the host microcontroller's weak pull-ups can be used. Ensure that the pull-up resistors are connected to the same Vdd level as the QT2160.
- The $\overline{\text{CHANGE}}$ pin has open-drain characteristics and requires a pull-up resistor.
- LEDs are connected to the appropriate pins of the QT2160 using inverting buffers as described in the datasheet.

The example code assumes the following port assignments on the host microcontroller:

- SDA is connected to P0.6.
- SCL is connected to P0.7.
- $\overline{\text{CHANGE}}$ is connected to P0.3.

Driving the AT42QT2160 ▬▬▬▬▬▬▬▬

3.3 Interface Timing

Figure 3-1 illustrates the interface timing used in the example. The QT2160 signals any change in its sense status by driving the $\overline{\text{CHANGE}}$ pin low. In this context, sense status includes bytes 2 through 6 of the address map. In response to the QT2160 driving $\overline{\text{CHANGE}}$ low, the host reads all of bytes 2 – 6. The QT2160 releases the $\overline{\text{CHANGE}}$ pin when all changed bytes have been read. Handshaking logic within the QT2160 guarantees that any unread changes in any status byte will cause $\overline{\text{CHANGE}}$ to be driven low, even if this occurs while the device is being read.
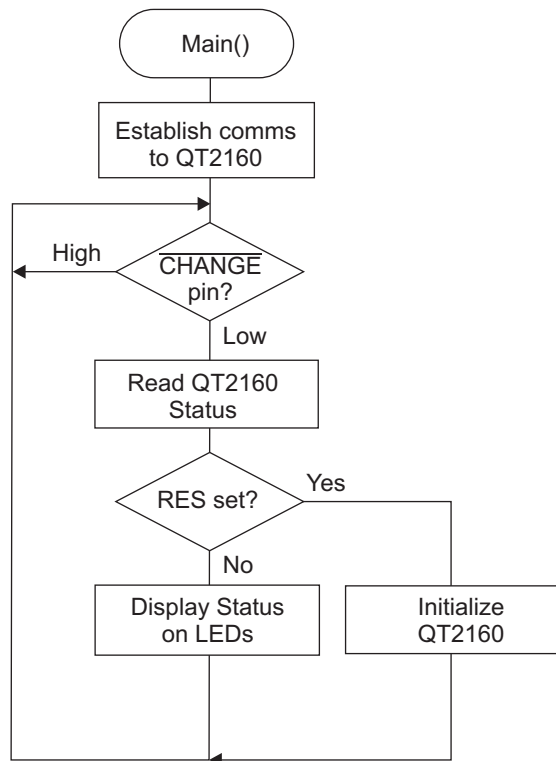
Figure 3-1. QT2160 Interface Timing
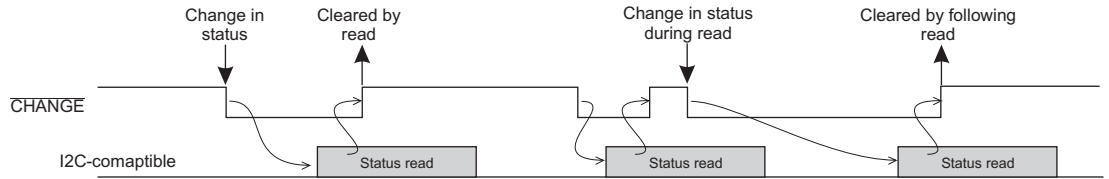
4. Demonstration Program

4.1 The main() Function

A simplified flowchart for the QT2160 demonstration program is shown in Figure 4-1. The program flow is encapsulated in the main() function of the C code (see Section 6 on page 8).

Figure 4-1. Simplified Flowchart for the Demonstration Program

During initialization, the program establishes communication with the QT2160 and then sets it up for the required operating mode. In the main loop, the program monitors the $\overline{\text{CHANGE}}$ pin. If the $\overline{\text{CHANGE}}$ pin is low, the program reads the five status bytes. It then writes the state of the discrete keys back to the QT2160 pins GPO0 to GPO3 and writes the slider position back to the PWM driver for GPIO3. The effect of touching the keys and slider is thus displayed on LEDs connected to I/O pins of the QT2160.

4.2 QT2160 Initialization

Initialization occurs at the start of the main() function, as follows:

1. The demonstration program establishes the initial communication with QT2160 by reading its Chip-ID. The QT2160 takes several milliseconds to initialize itself after power-up. During this time, it may not acknowledge (ACK) its slave address. The WriteQtI2c() function repeatedly attempts to send the slave address until the ACK is received.

2. Once the device has been read successfully, the returned chip ID is checked. If the wrong ID is returned, the demonstration program goes no further.

Depending on the construction of the touch keys being used, it may be necessary to adjust the touch-related registers. This should be done in the initialization code.

4.3 Main Loop

Following initialization, the remainder of the main() function consists of a main loop, as follows:

1. The state of the $\overline{\text{CHANGE}}$ pin is sampled.

2. If $\overline{\text{CHANGE}}$ is asserted low, the status registers (addresses 2 – 6) are read into the five-byte QtStatus[] array. Reading these registers restores the $\overline{\text{CHANGE}}$ pin to the inactive (high) state.

3. If the RES bit in the "General Status" register is set, indicating that the QT2160 has been reset, the device is initialized. This will occur after initial power up but may also occur if, for example, the $\overline{\text{RST}}$ pin is pulled low at any time. The demonstration program changes a few registers from their default values. Note that all the QT2160's registers are set to default values every time the device is reset (refer to the QT2160 datasheet for the default values). The demonstration program makes the following changes:

   a. GPIO3 is set for output mode in "GPIO-Direction" (address 73).

   b. GPIO3 is set for PWM mode in "GPIO-PWM" (address 75).

   c. The "RESOLUTION" field is set to 8-bit in Slider-Options (address 21).

   d. Finally, a calibration cycle is initiated.

4. If RES is not set, the touch status is processed. The states of the discrete touch keys 8 – 5 from "Key Status 2" are returned in QtStatus[2]. Key states output by the QT2160 are fully debounced and can be used in an application without further processing. QtStatus[2] is simply written back to "GPO Drive 1" at address 70 and the LEDs indicate the touch state.

5. If the SDET bit in the "General Status" register is set, the slider touch position in QtStatus[3] is written to "PWM Level" at address 76.

4 Driving the AT42QT2160

5. I^{2}C-compatible Driver

5.1 I^{2}C-compatible Communication

I^{2}C-compatible communication is a major aspect of any program involving the QT2160. Some form of I^{2}C-compatible driver must be written to handle read and write operations to the device. Most current microprocessors include a hardware I^{2}C-compatible master function, which could be programmed to efficiently handle these transfers, but the driver code will always be specific to the hardware. The example code in this application note includes a software-based I^{2}C-compatible driver using two port pins of the host microcontroller. It could easily be ported to many processor types.

The I^{2}C-compatible communication sequences used to read and write data to the QT2160 are fully described in the QT2160 datasheet.

5.2 Design Approach

The driver presented here uses bit-banging techniques to manage the I^{2}C-compatible interface. The driver is structured in three layers:

- In the bottom layer, code-macros are used to drive the SCL and SDA pins and to create START and STOP conditions on the I^{2}C-compatible bus.
- In the middle layer, the SendByte() and GetByte() functions sequence the transmission and reception of bytes including handling the ACK bit.
- In the top layer, the WriteQtI2c() and ReadQtI2c() functions can be called by an application to transfer one or more bytes to and from the QT2160. Note that for the sake of clarity, the driver is simplified (for example, it includes no timeouts).

The following sections describe the driver and should be read in conjunction with the code listing.

5.3 Macros

Table 5-1 describes the various code macros used to drive the SCL and SDA pins. Any macro that changes the state of a pin includes a fixed 5 μs delay following the edge. The 5 μs delay ensures the 100 kHz I^{2}C-compatible specification is met and results in a bus clock rate of around 66 kHz.

Table 5-1.   Code Macros for the I^{2}C-compatible Bus Functions

Macro	Description
I2cDelay	Generates a program-delay to guarantee I^{2}C-compatible timing requirements. The constant BUS_DELAY should be adjusted to produce a 5 μs delay.
SetHiSCL	Floats the SCL pin. SCL is then pulled high by the pull-up resistor. Note: the QT2160 may extend the low clock-phase. This means that after floating the pin, the driver waits until the high state is achieved before continuing.
SetLoSCL	Drives the SCL pin low. There is no need to wait for the pin to achieve the low state.
SetHiSDA	Floats the SDA pin. SDA is then pulled high by the pull-up resistor. After floating the pin, the driver waits until the high state is achieved before continuing.

Table 5-1. Code Macros for the I2C-compatible Bus Functions (Continued)

Macro	Description
SetLoSDA	Drives the SDA pin low. There is no need to wait for the pin to achieve the low state.
FloatSDA	This simply floats the SDA pin so that the QT2160 can drive SDA during data-reads or ACK cycles.
SendSTART	Outputs the START condition
SendSTOP	Outputs the STOP condition
SendCLOCK	Pulses the SCL pin high-low

5.4 SendByte()

The SendByte() function transmits a single byte onto the I2C-compatible bus. The byte is supplied as an input parameter. SendByte() returns I2C_OK if the byte is acknowledged or I2C_FAIL if a NACK is returned.
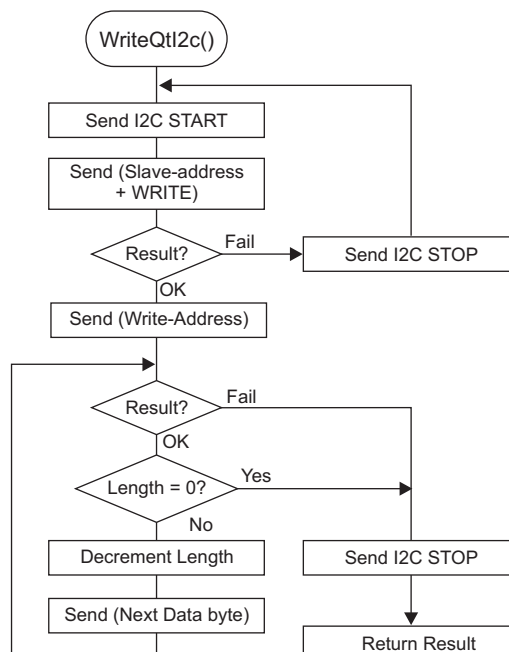
5.5 GetByte()

The GetByte() function receives a single byte and terminates it with either ACK or NACK. NACK should be specified for the last byte of an I2C-compatible read transfer. GetByte() returns the received byte.

5.6 WriteQtI2c()

The flowchart for the WriteQtI2c() function is shown in Figure 5-1.

Figure 5-1. Flowchart for the WriteQtI2c() Function

Driving the AT42QT2160

The WriteQtI2c() function writes one or more bytes to the QT2160 as specified in the datasheet. It accepts four input parameters:
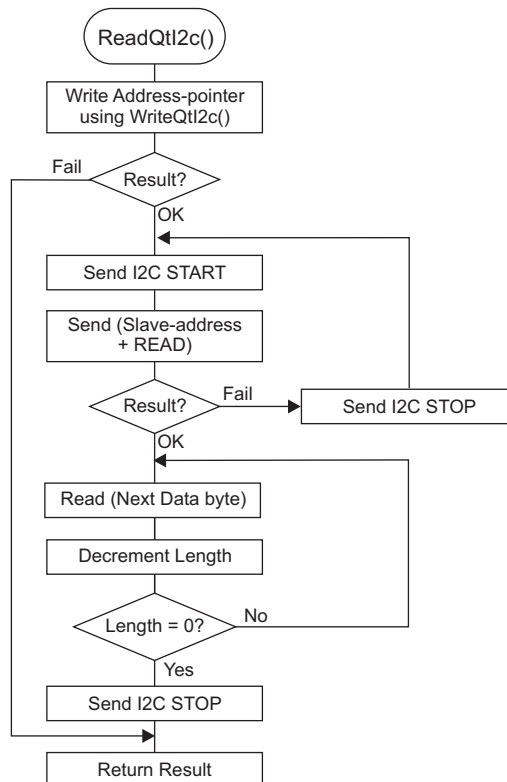
- SlaveAddress: The QT2160 I^{2}C-compatible address as selected at pins I2CA1 and I2CA0.
- WriteAddress: The address of the first QT2160 register to be written.
- WriteLength: The number of bytes to be written.
- WritePtr: A pointer to the first byte to be written. The pointer should point to a byte array.

The function returns I2C_OK if the entire write transfer is acknowledged (ACK), or I2C_FAIL if a NACK is returned.

5.7   ReadQtI2c()

The flowchart for ReadQtI2c() is shown in Figure 5-2.

Figure 5-2.   Flowchart for the ReadQtI2c() Function

The ReadQtI2c() function reads one or more bytes to the QT2160 as specified in the datasheet. It accepts four input parameters:

- SlaveAddress: The QT2160 I^{2}C-compatible address as selected at pins I2CA1 and I2CA0.
- ReadAddress: The address of the first QT2160 register to be read.
- ReadLength: The number of bytes to be read.
- ReadPtr: A pointer to a byte-array where the read data should be saved.

The function returns I2C_OK if the entire read transfer is completed correctly.

Note that the WriteQtI2c() function is called to set the address-pointer within the QT2160.

6. Source Code

```
/*==============================================================================

Project:                        AT42QT2160-MMU Example Code

==============================================================================*/

typedef unsigned char uint8_t;

enum {          /* QT2160 registers */
    QT_CHIP_ID = 0,         QT_CODE_VERSION,        QT_GENERAL_STATUS,      QT_KEY_STATUS_1,
    QT_KEY_STATUS_2,        QT_SLIDER_POSITION,     QT_GPIO_READ,           QT_CALIBRATE = 10,
    QT_RESET,               QT_LP_MODE,             QT_AWAKE_TIMEOUT = 14,  QT_NEG_DRIFT_COMP,
    QT_POS_DRIFT_COMP,      QT_DI_LIMIT,            QT_NEG_RECAL_DELAY,     QT_DRIFT_HOLD_TIME,
    QT_SLIDER_CONTROL,      QT_SLIDER_OPTIONS,      QT_KEY_CONTROL,         QT_KEY_NTHR = 38,
    QT_KEY_BL = 54,         QT_GPO_DRIVE = 70,      QT_GPIO_DRIVE,          QT_GPIO_DIR = 73,
    QT_GPO_PWM,             QT_GPIO_PWM,            QT_PWM_LEVEL,           QT_GPIO_WAKE,
    QT_COMM_CHG_KEYS_1, QT_COMM_CHG_KEYS_2
    };

#define QT2160_ID           0x11  /* Chip ID (read from device-address 0) */
#define QT_STATUS_SDET      0x01  /* bitmask for Slider-detect bit at device-address 2 */
#define QT_STATUS_RES       0x80  /* bitmask for Reset bit at device-address 2 */
#define QT_GPIO_1           0x04  /* bitmask for GPIO1 */
#define QT_GPIO_2           0x08  /* bitmask for GPIO2 */
#define QT_GPIO_3           0x10  /* bitmask for GPIO3 */
#define SLIDER_8BIT_RES     0x00  /* 8-bit resolution (written to device-address 21) */

#define CHANGE_ASSERTED_LOW   0     /* asserted state for /CHANGE pin */
#define QT2160_I2C_ADDRESS    0x0d  /* I2C address used by demo. I2CA1 = I2CA0 = 0 */

uint8_t QtStatus[5];      /* application storage for QT2160 device-status */
uint8_t QtData;           /* data buffer used for single-byte read and write transfers */

/* Port-assignment for QT2160-interface pins */
sbit SCL        = P0 ^ 7;
sbit SDA        = P0 ^ 6;
sbit CHANGE_PIN = P0 ^ 3;

/* I2C driver - Function prototypes */
uint8_t WriteQtI2c ( uint8_t SlaveAddress,
    uint8_t WriteAddress, uint8_t WriteLength, uint8_t *WritePtr );
uint8_t ReadQtI2c ( uint8_t SlaveAddress,
    uint8_t ReadAddress, uint8_t ReadLength, uint8_t *ReadPtr );
```

```
/*==============================================================================
   Function:      main()
==============================================================================*/
void main ( void )
{

/*---------------------------- Initialisation ----------------------------------*/
/* (1) Establish communication with the QT2160 touch-sensor */
   /* wait for successful transfer at the QT2160's address - Read Chip-ID */
   while ( !ReadQtI2c (QT2160_I2C_ADDRESS, QT_CHIP_ID, 1, &QtData) );

/* (2) Check that the responding device is a QT2160! */
   while ( QtData != QT2160_ID );

/*---------------------- End of Initialisation ----------------------------------*/

/*---------------------- Main Loop ----------------------------------------------*/
   while (1) {

      if ( CHANGE_PIN == CHANGE_ASSERTED_LOW )     /* test /CHANGE pin */
      {
         /* If /CHANGE is asserted, read all status-bytes */
         ReadQtI2c ( QT2160_I2C_ADDRESS, QT_GENERAL_STATUS, 5, QtStatus );
         /* reading these registers will restore /CHANGE pin to the inactive (hi) state */

         /* Has device just reset? */
         if ( QtStatus[0] & QT_STATUS_RES )
         {
            /* After any reset, configure device for demo requirements:
            The demo uses power-up default values for all touch-related registers.
            All keys are enabled by default and the slider is set to use 5-keys.    */

            QtData = QT_GPIO_3;
            /* Set GPIO3 for output mode */
            WriteQtI2c ( QT2160_I2C_ADDRESS, QT_GPIO_DIR, 1, &QtData );
            /* Configure GPIO3 for PWM mode */
            WriteQtI2c ( QT2160_I2C_ADDRESS, QT_GPIO_PWM, 1, &QtData );
            /* Configure Slider for 8-bit resolution */
            QtData = SLIDER_8BIT_RES;
            WriteQtI2c ( QT2160_I2C_ADDRESS, QT_SLIDER_OPTIONS, 1, &QtData );
            /* Send calibrate command */
            QtData = 1;
            WriteQtI2c ( QT2160_I2C_ADDRESS, QT_CALIBRATE, 1, &QtData );
         }
```

```
        else
        {
            /* write key-states back to the QT2160 LEDs */
            WriteQtI2c ( QT2160_I2C_ADDRESS, QT_GPO_DRIVE, 1, &QtStatus[2] );

            /* Update the PWM level if the slider is touched */
            if (QtStatus[0] & QT_STATUS_SDET)
                WriteQtI2c ( QT2160_I2C_ADDRESS, QT_PWM_LEVEL, 1, &QtStatus[3] );

        }  /* End if ( QtStatus[0] & QT_STATUS_RES ) */

    } /* End if ( CHANGE_PIN.. ) */


  } /* End while (1) */
/*---------------------- End of Main Loop ----------------------------------*/
}




/*--------------------------- I2C Driver -----------------------------------*/
/* The following code is the I2C driver */
/* ----------- I2C Driver Defines -------------- */
#define ACK      0
#define NACK     1
#define I2C_OK    1
#define I2C_FAIL  0
#define READ_FLAG 0x01
#define BUS_DELAY 23


/*----------- I2C Code Macros ------------------ */
#define I2cDelay  for (i = 0; i < BUS_DELAY; i++)
#define SetHiSCL  {SCL = 1; while (!SCL); I2cDelay;}
#define SetLoSCL  {SCL = 0; I2cDelay;}
#define SetHiSDA  {SDA = 1; while (!SDA); I2cDelay;}
#define SetLoSDA  {SDA = 0; I2cDelay;}
#define FloatSDA  {SDA = 1;}
#define SendSTART {SetLoSDA; SetLoSCL;}
#define SendSTOP  {SetLoSDA; SetHiSCL; SetHiSDA;}
#define SendCLOCK {SetHiSCL; SetLoSCL;}
/*------------------------------------------------ */
```

```
/*==============================================================================
       Function:       SendByte()
       Input:          Byte to send
       Output:         I2C_OK if device ACKs, I2C_FAIL if device NACKs
==============================================================================*/

uint8_t SendByte ( uint8_t TxByte )
{
   uint8_t i,b, Result = I2C_OK;

   for (b = 0; b < 8; b++)
   {
      if ( TxByte & 0x80 )
         SetHiSDA
      else
         SetLoSDA
      SendCLOCK
      TxByte <<= 1;  /* shift out data byte */
   }

   FloatSDA /* prepare to receive ACK bit */
   SetHiSCL /* read ACK bit */
   if ( SDA == 1 )
      Result = I2C_FAIL;
   SetLoSCL

   return Result;
}


/*==============================================================================
       Function:       GetByte()
       Input:          State of ACK bit to send
       Output:         Received byte
==============================================================================*/

uint8_t GetByte ( uint8_t AckBit )
{
   uint8_t i,b, RxByte = 0;

   FloatSDA

   for (b = 0; b < 8; b++)
   {
      RxByte <<= 1;  /* shift in data byte */
      SetHiSCL;
```

```c
        if ( SDA )
            RxByte |= 1;
        SetLoSCL;
    }

    if ( AckBit )  /* send ACK bit */
        SetLoSDA
    else
        SetHiSDA
    SendCLOCK


    return RxByte;
}



/*===============================================================================
        Function:       WriteQtI2c() Executes multi-byte write to QT-device
        Input:          SlaveAddress = Device address on the I2C bus
                        WriteAddress = Register address
                        WriteLength = Number of bytes to write
                        WritePtr = Pointer to byte array containing write-data
        Output:         I2C_OK if transfer completes, I2C_FAIL if device NACKs
===============================================================================*/

uint8_t WriteQtI2c ( uint8_t SlaveAddress,
    uint8_t WriteAddress, uint8_t WriteLength, uint8_t *WritePtr )
{
    uint8_t i, Result;

    do {  /* attempt to address device until ACK is received */
        SendSTART;
        if ( (Result = SendByte ( SlaveAddress << 1 )) != I2C_OK )
            SendSTOP;
    } while ( Result != I2C_OK );

    /* write address-pointer to device */
    Result = SendByte ( WriteAddress );

    while ( (Result == I2C_OK) && WriteLength-- )
        Result = SendByte ( *WritePtr++ );

    SendSTOP;   /* terminate transfer */

    return Result;
}
```

```
/*==============================================================================
      Function:       ReadQtI2c() Executes multi-byte read from QT-device
      Input:          SlaveAddress = Device address on the I2C bus
                      ReadAddress = Register address
                      ReadLength = Number of bytes to read
                      ReadPtr = Pointer to byte array for read-data
      Output:         I2C_OK if transfer completes, I2C_FAIL if device NACKs
==============================================================================*/

uint8_t ReadQtI2c ( uint8_t SlaveAddress,
    uint8_t ReadAddress, uint8_t ReadLength, uint8_t *ReadPtr )
{
    uint8_t i, Result;

    /* write address-pointer to device */
    Result = WriteQtI2c ( SlaveAddress, ReadAddress, 0, 0 );

    if ( Result == I2C_OK )
    {
        do {  /* attempt to address device until ACK is received */
            SendSTART;
            if ( (Result = SendByte ( (SlaveAddress << 1) + READ_FLAG) ) != I2C_OK )
                SendSTOP;
        } while ( Result != I2C_OK );

        do {
            *ReadPtr++ = GetByte ( --ReadLength );
        } while ( ReadLength );

        SendSTOP; /* terminate transfer */
    }

    return Result;
}
/*------------------------End of I2C Driver -------------------------------------*/
```

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Touch Technology Division
1 Mitchell Point
Ensign Way
Hamble
Southampton
Hampshire SO31 4RF
United Kingdom
Tel: (44) 23-8056-5600
Fax: (44) 23-8045-3939

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
qprox.support@atmel.com

Sales Contact
qprox.sales@atmel.com

Literature Requests
www.atmel.com/literature