MXTAN0213

Interfacing with maXTouch Touchscreen Controllers

INTRODUCTION

This document provides an overview of how to connect to, and use, maXTouch touchscreen controllers in user systems. It is particularly suited to customers using an off-the-shelf module with an embedded operating system rather than a mainstream operating system, such as Android.

This document is relevant to all maXTouch T, T2 and U Series devices.

TABLE OF CONTENTS

1.0 HARDWARE

1.1 Power

Check your module supplier's documentation for the voltage levels used to power the maXTouch device in your application. Be aware that the maXTouch devices can use a different IO voltage to the main power rail, so check carefully that the correct voltages for logic "1" and "0" are being used on both sides.

1.2 I^2C Interface

maXTouch devices normally connect to the host via a Fast-Mode+ I^2C connection, operating at up to 400 kHz (or faster on later devices). See your module supplier's documentation for voltage levels used and whether pull-ups are integrated or need to be provided on the host side.

The host interface uses four wires.

1.2.1 I^2C BUS

This comprises the standard two-wire bus (SCK and SDA), which provides half-duplex bidirectional communications between a host (acting as I^2C master) and one or more slave devices. The maXTouch controller operates as a slave on the bus. Check your supplier's documentation for which I^2C address to use.

1.3 $\overline{\text{CHG}}$

The maXTouch devices use an active-low interrupt line to indicate to the host that there is information ready to send. The $\overline{\text{CHG}}$ pin should be connected to an interrupt-capable pin on the host microcontroller and may be used in edge-triggered or level-triggered mode, depending on desired method of interacting with the MaXTouch device – see later. Polled mode is not recommended for use with maXTouch parts. Always use a weak pull-up (10 kΩ) on the $\overline{\text{CHG}}$ line. Note that the host should never drive this line as an output.

1.4 $\overline{\text{Reset}}$

The maXTouch chip provides an active-low hardware $\overline{\text{RESET}}$ pin to reinitialize the part. Check whether your module makes this available to the end customer. If the $\overline{\text{RESET}}$ pin is used, the host must hold this pin as an output low until all power supplies to the maXTouch chip have stabilized. If the $\overline{\text{RESET}}$ pin is unused, pull this pin high by a 3.3 kΩ resistor to the IO power rail. It is also possible to reinitialize the part using software control (see Section 5.1.2 "Command Processor T6 Object").

2.0 OBJECT PROTOCOL

2.1 Introduction

The Object Protocol provides a single common interface across the Microchip maXTouch controllers. This allows the different features in each controller to be configured in a consistent manner. This makes the future expansion of features and simple product upgrades possible, whilst allowing backwards compatibility for the host driver and application code.

The protocol is designed to control the processing chain in a modular manner. This is achieved by breaking the features of the device into objects that can be controlled individually. Each object represents a certain feature or function of the device, such as a touchscreen. Where appropriate, objects can be disabled or enabled as needed.

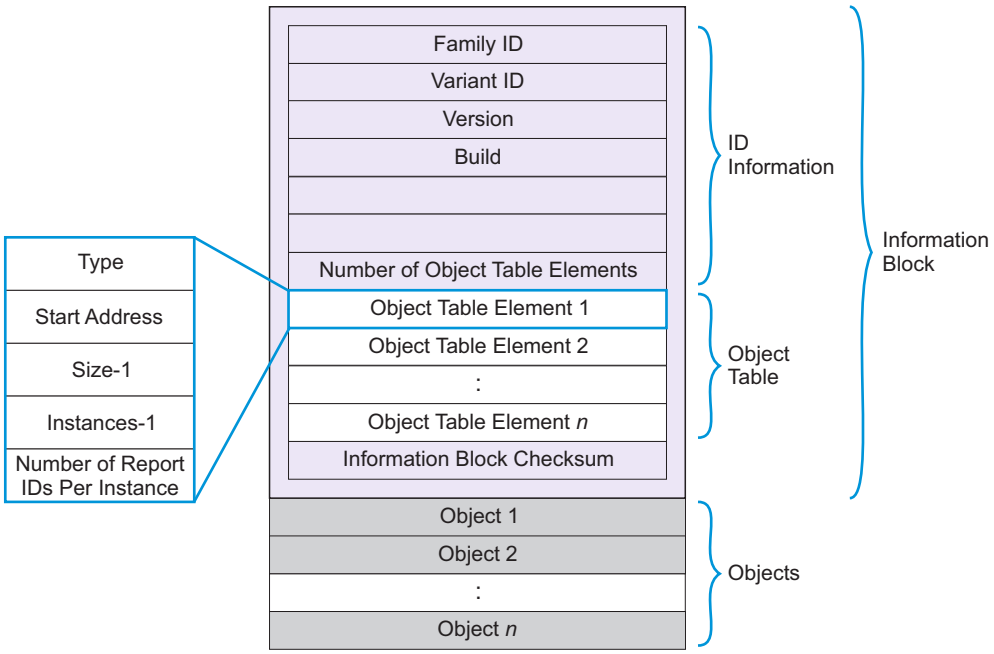
The Object Protocol is designed in such a way that a driver can be “future-proofed”. To this end, two principles or rules are applied whenever new features are added:

- A default value of 0 acts as a “safe” value” with a sensible meaning for any given field (or register). This allows a driver to ignore any fields that it does not understand and set them to zero. This allows for future expansion. For example, if the current version of the driver is aware of only 10 fields in an object but actually finds 15 fields it can safely zero the last 5 fields without adverse effect. This mismatch in the number of fields is typically caused by the chip using a more recent version of some objects in Object Protocol than the driver supports.
- The Object Protocol is designed such that object definitions are expanded by adding new fields to the end of an object. An older driver can therefore still be used with a chip that uses a newer version of the protocol. The driver can simply ignore the additional fields.

2.2 Memory Map Structure

Each object has its own configuration memory. The objects are stacked together to produce an object-based memory map. A generalized structure of this memory map is shown in the following diagram.

FIGURE 1: GENERIC MEMORY MAP STRUCTURE



From the diagram it can be seen that the memory map contains two main sections:

- An Information Block. This documents which objects are contained in the memory map for the device
- The objects themselves

2.2.1 INFORMATION BLOCK

The Information Block allows the host to read information about the layout of objects in the memory map. It contains a list of all the objects in the memory map. This is used by the host driver to determine which objects exist, where they are located in the memory map and their sizes. The host driver can therefore read the device Information Block and gather enough information to be able to communicate with the device.

The Information Block is positioned at the start of the device memory map at address zero. This allows it to be read easily by the host as the first operation. See the following section for details of how to read and write to the device.

The Information Block contains the following three sections:

- ID Information Fields – These include:
  - Standard ID fields that make up the unique identifier for the device
  - The size of the touchscreen matrix the device supports
  - The number of objects in the Object Table
- Object Table – This acts as an “index” to the objects in the memory map. Note that one of the objects may be an Information Block object that holds additional Object Table entries.
- Information Block Checksum – This allows the host to check that the Information Block has been read correctly over the communications interface.

TABLE 1: INFORMATION BLOCK LAYOUT

Byte	Description of Field	Section
0	Family ID – provided for information only	ID Information
1	Variant ID – provided for information only	
2	Version – provided for information only	
3	Build – provided for information only	
4	Matrix X Size – maximum number of X lines available on the device	
5	Matrix Y Size – maximum number of Y lines available on the device	
6	Number of elements in the Object Table – equivalent to the number of object instances (note: some objects have more than one instance)	
7 – 12	Object Table element 1 (6 bytes)	Object Table
13 – 18	Object Table element 2 (6 bytes)	
...	...	
...	Last Object Table element (6 bytes)	
(*end*-2) – *end*	24-bit checksum (3 bytes)	Checksum Field

3.0 I²C COMMUNICATIONS

The maXTouch devices typically use an I²C interface for communication, although some devices also provide USB or SPI interfaces. Check your module documentation for more details.

The I²C interface is used in conjunction with the $\overline{\text{CHG}}$ line. The $\overline{\text{CHG}}$ line going active (low) signifies that a new data packet is available. This provides an interrupt-style interface and allows the device to present data packets when internal changes have occurred.

It is recommended to use the device in I²C mode: for this, the I2CMODE pin should be pulled high if it is brought out to the connector. Alternatively, if auto selection is required, the I2CMODE pin should be left floating and the primary I2C address used.

3.1 I²C Addresses

The maXTouch devices typically support two I²C device addresses that are selected using the ADDSEL line at start up. The two internal I²C device addresses are 0x4A and 0x4B. If ADDSEL is brought out to the connector, then it should be driven low for 0x4A and pulled high for 0x4B – check your device documentation for more details.

The I²C address is shifted left to form the SLA+W or SLA+R address when transmitted over the I²C interface, as shown in the following figure.

FIGURE 2: FORMAT OF AN I2C ADDRESS

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Address: 0b1001_010 or 0b1001_011							Read/write

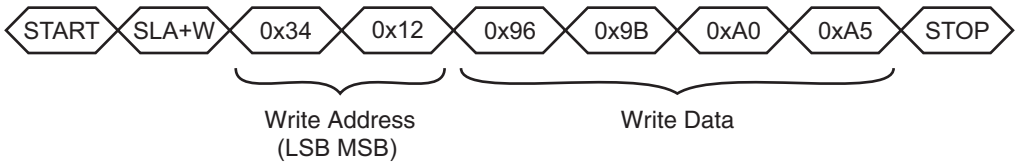
3.2 Writing To the Device

A WRITE cycle to the device consists of a START condition followed by the I²C address of the device (SLA+W, as in Figure 2 above). The next two bytes are the address of the location into which the writing starts. The first byte is the Least Significant Byte (LSByte) of the address, and the second byte is the Most Significant Byte (MSByte). This address is then stored as the address pointer.

Subsequent bytes in a multi-byte transfer form the actual data. These are written to the location of the address pointer, location of the address pointer + 1, location of the address pointer + 2, and so on. The address pointer returns to its starting value when the WRITE cycle STOP condition is detected.

The following figure shows an example of writing four bytes of data to contiguous addresses starting at 0x1234.

FIGURE 3: EXAMPLE OF A FOUR-BYTE WRITE STARTING AT ADDRESS 0X1234

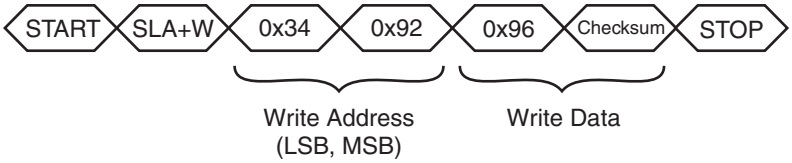


3.3 I²C Writes in Checksum Mode

In I²C checksum mode an 8-bit CRC is added to all I²C writes. The CRC is sent at the end of the data write as the last byte before the STOP condition. All the bytes sent are included in the CRC, including the two address bytes. Any command or data sent to the device is processed even if the CRC fails.

To indicate that a checksum is to be sent in the write, the most significant bit of the MSByte of the write address is set to 1. For example, the I²C command shown in the following figure writes a value of 150 (0x96) to address 0x1234 with a checksum. The address is changed to 0x**9**234 to indicate checksum mode.

FIGURE 4: EXAMPLE OF A WRITE TO ADDRESS 0X1234 WITH A CHECKSUM



3.4 Reading From the Device

Two I^2C bus activities must take place to read from the device. The first activity is an I^2C write to set the address pointer (LSByte then MSByte). The second activity is the actual I^2C read to receive the data. The address pointer returns to its starting value when the read cycle NACK is detected.

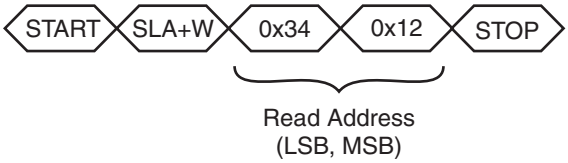
It is not necessary to set the address pointer before every read. The address pointer is updated automatically after every read operation. The address pointer will be correct if the reads occur in order. In particular, when reading multiple messages from the Message Processor T5 object, the address pointer is automatically reset to allow continuous readReading Status Messages with DMA.

The WRITE and READ cycles consist of a START condition followed by the I^2C address of the device (SLA+W or SLA+R respectively). Note that in this mode, checksumming of the data packets is not supported.

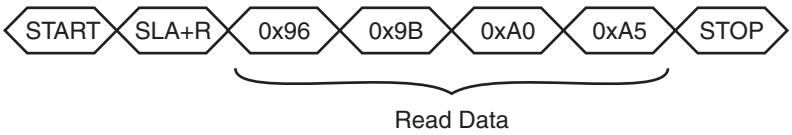
The following figure shows the I^2C commands to read four bytes starting at address 0x1234.

FIGURE 5: EXAMPLE OF A FOUR-BYTE READ STARTING AT ADDRESS 0X1234

Set Address Pointer



Read Data



3.5 Reading Status Messages with DMA

The device facilitates the easy reading of multiple messages using a single continuous read operation. This allows the host hardware to use a direct memory access (DMA) controller for the fast reading of messages, as follows:

1. The host uses a write operation to set the address pointer to the start of the Message Count T44 object, if necessary. If a checksum is required on each message, the most significant bit of the MSByte of the read address must be set to 1.

NOTE	The STOP condition at the end of the read resets the address pointer to its initial location, so it may already be pointing at the Message Count T44 object following a previous message read.

2. The host starts the read operation of the message by sending a START condition.
3. The host reads the Message Count T44 object (one byte) to retrieve a count of the pending messages.
4. The host calculates the number of bytes to read by multiplying the message count by the size of the Message Processor T5 object.

NOTE	The host should have already read the size of the Message Processor T5 object in its initialization code.

5. Note that the size of the Message Processor T5 object as recorded in the Object Table includes a checksum byte. If a checksum has not been requested, one byte should be deducted from the size of the object. That is: number of bytes = count × (size – 1).The host reads the calculated number of message bytes. It is important that the host does not send a STOP condition during the message reads, as this will terminate the continuous read operation and reset the address pointer. No START and STOP conditions must be sent between the messages.
6. The host sends a STOP condition at the end of the read operation after the last message has been read. The NACK condition immediately before the STOP condition resets the address pointer to the start of the Message Count T44 object.

The following figures show examples of using a continuous read operation to read three messages from the device without a checksum and with a checksum.

FIGURE 6: CONTINUOUS MESSAGE READ EXAMPLE – NO CHECKSUM

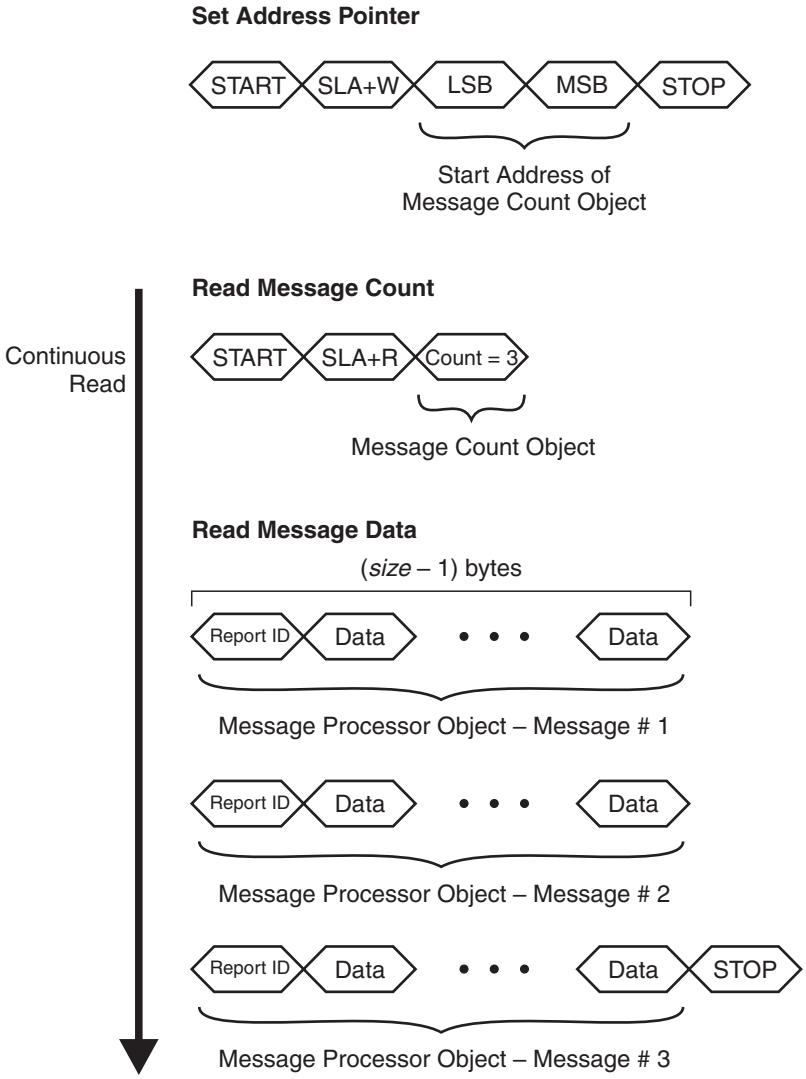
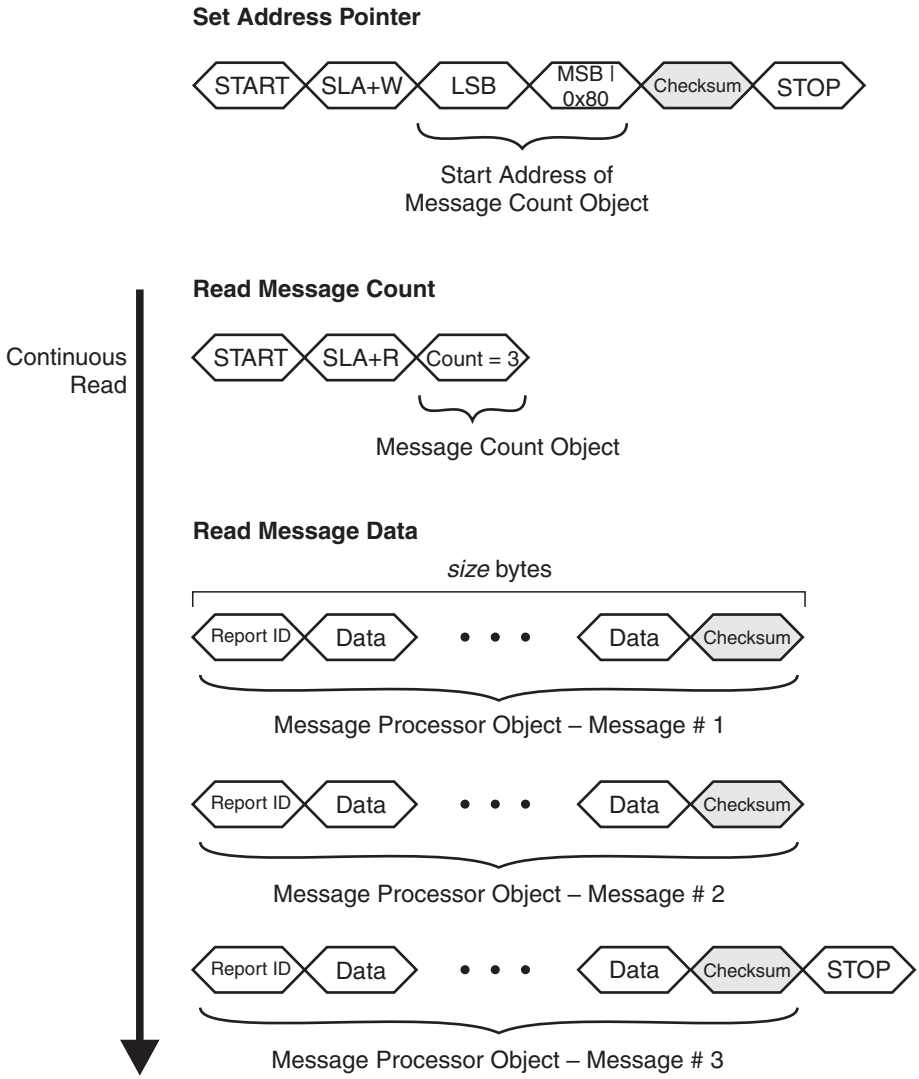


FIGURE 7: CONTINUOUS MESSAGE READ EXAMPLE – I²C CHECKSUM



There are no checksums added on any other I²C reads. An 8-bit CRC can be added, however, to all I²C writes, as described earlier.

An alternative method of reading messages using the $\overline{\text{CHG}}$ line is given in the next section.

3.6 $\overline{\text{CHG}}$ Line

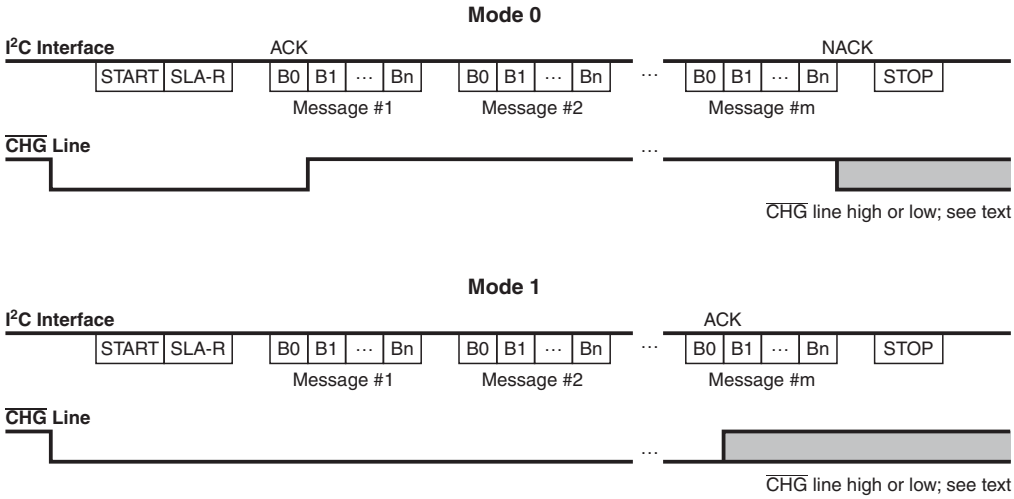
The $\overline{\text{CHG}}$ line is an active-low, open-drain output that is used to alert the host that a new message is available in the Message Processor T5 object. This provides the host with an interrupt-style interface with the potential for fast response times. It reduces the need for wasteful I2C communications.

The $\overline{\text{CHG}}$ line should always be configured as an input on the host during normal usage. This is particularly important after power-up or reset.

A pull-up resistor is required, typically 3.3 kΩ, to VddIO.

The $\overline{\text{CHG}}$ line operates in two modes, as defined by the Communications Configuration T18 object.

FIGURE 8: $\overline{\text{CHG}}$ LINE MODES FOR I2C-COMPATIBLE TRANSFERS



In Mode 0 (Edge-triggered operation):

1. The $\overline{\text{CHG}}$ line goes low to indicate that a message is present.
2. The $\overline{\text{CHG}}$ line goes high when the first byte of the first message (that is, its report ID) has been sent and acknowledged (ACK sent) and the next byte has been prepared in the buffer.
3. The STOP condition at the end of an I^{2}C transfer causes the $\overline{\text{CHG}}$ line to stay high if there are no more messages. Otherwise the $\overline{\text{CHG}}$ line goes low to indicate a further message.

Note that Mode 0 also allows the host to continually read messages by simply continuing to read bytes back without issuing a STOP condition. Messaging reading should end when a report ID of 255 (“invalid message”) is received. Alternatively the host ends the transfer by sending a NACK after receiving the last byte of a message, followed by a STOP condition. If there is another message present, the $\overline{\text{CHG}}$ line goes low again. In this mode the state of the $\overline{\text{CHG}}$ line does not need to be checked during the I^{2}C read.

In Mode 1 (Level-triggered operation):

1. The $\overline{\text{CHG}}$ line goes low to indicate that a message is present.
2. The $\overline{\text{CHG}}$ line remains low while there are further messages to be sent after the current message.
3. The $\overline{\text{CHG}}$ line goes high again only once the first byte of the last message (that is, its report ID) has been sent and acknowledged (ACK sent) and the next byte has been prepared in the output buffer.

Mode 1 allows the host to continually read the messages until the $\overline{\text{CHG}}$ line goes high, and the state of the $\overline{\text{CHG}}$ line determines whether or not the host should continue receiving messages from the device.

NOTE	The state of the $\overline{\text{CHG}}$ line should be checked only between messages and not between the bytes of a message. The precise point at which the $\overline{\text{CHG}}$ line changes state cannot be predicted and so the state of the $\overline{\text{CHG}}$ line cannot be guaranteed between bytes.

The Communications Configuration T18 object can be used to configure the behavior of the $\overline{\text{CHG}}$ line. In addition to the $\overline{\text{CHG}}$ line operation modes described above, this object allows direct control over the state of the $\overline{\text{CHG}}$ line for test purposes. Refer to section 5 below for more details.

3.7 SDA, SCL

The I^{2}C bus transmits data and clock with SDA and SCL, respectively. These are open-drain. The device can only drive these lines low or leave them open. The termination resistors (Rp) pull the line up to VddIO if no I^{2}C device is pulling it down.

The termination resistors should be chosen so that the rise times on SDA and SCL meet the I^{2}C specifications for the interface speed being used, bearing in mind other loads on the bus. For best latency performance, it is recommended that no other devices share the I^{2}C bus with the maXTouch controller.

3.8 Clock Stretching

The device supports clock stretching in accordance with the I^2C specification. It may also instigate a clock stretch if a communications event happens during a period when the device is busy internally. The maximum clock stretch is approximately 10 to 15 ms.

4.0 INTERFACING WITH THE MAXTOUCH DEVICE

The object protocol requires interrogation of the device and the construction of a corresponding model within the host before any real-time touch messages can be handled.

The driver code should therefore take the following approach:

1. Establish contact with the chip (see Section 4.1 “Establishing Contact”)
2. Read the Information Block to retrieve the ID information.
3. Retrieve the Information Block’s checksum and check it is valid.
4. Read the Object Table.
5. Use the information on the objects retrieved from the Object Table to calculate and store the report IDs for the object instances that generate messages.
6. (Finally) perform the normal running tasks, such as calibrating the chip, reading messages and writing commands.

These steps are described in detail in the following sections.

The example host driver code can also be found in the maXTouch GIT repository on Github (see Appendix A “Sample Reference Code”).

4.1 Establishing Contact

The host should attempt to read any available messages (such as a reset or calibration message) to establish that the device is present and running following power-up or a reset. The host should also check that there are no configuration errors reported.

Normal behavior for the maXTouch chips on power-up is for the $\overline{\text{CHG}}$ line to go low shortly after reset or power-up, indicating that there is a calibration message to be read from the Message Processor T5 object. In this case, if the $\overline{\text{CHG}}$ line does not go low, this indicates that there is a problem with the device.

4.2 Reading the Information Block

The first operation the driver must perform after establishing contact with the device, is to locate the Information Block in the device. This contains useful information about the chip and provides essential information about the objects present in the device. It is stored at the start of the device memory map at address 0 for easy access.

The Information Block can be represented in C code using a structure with the following definition:

```
typedef struct
{
    info_id_t *id; /* Pointer to the struct containing ID Information. */
    object_t *objects;/* Pointer to an array of objects. */
    crc_t *crc;/* Pointer to Information Block Checksum. */

} info_block_t;
```

Note that the example code in the following sections uses a function to read data from the device via the I^2C interface. It is assumed that the function is named `mxt_read_register()` and that it returns an 8-bit status code. It is assumed to have the following prototype:

```
int mxt_read_register(unsigned char *dst_buf, int start_register, int ByteCount)
```

The example host driver code can also be found in the maXTouch GIT repository on Github (see Appendix A “Sample Reference Code”).

4.2.1 READING THE ID INFORMATION

The first seven bytes of the Information Block contain the ID Information.

The ID Information can be represented in C code by the following structure (this structure is used in later examples):

```
typedef struct
{
    uint8_t family_id; /* address 0 */
    uint8_t variant_id; /* address 1 */
    uint8_t version; /* address 2 */
    uint8_t build; /* address 3 */
    uint8_t matrix_x_size; /* address 4 */
```

```
    uint8_t matrix_y_size; /* address 5 */
    uint8_t num_declared_objects; /* address 6 */
} info_id_t;
```

Of a particular interest is the `num_declared_objects` member of the structure, as it determines the number of object table elements present in the object table. This information can be read from the device using the following code:

```
const uint8_t NUM_ID_BYTES = 7;
const uint8_t CRC_LENGTH = 3;
const uint8_t ID_OBJECTS_OFFSET = 6; /* Offset to Object Table */

uint8_t num_declared_objects;

/* Read "num_declared_objects" field of ID Information block */
ret = mxt_read_register((unsigned char *)&num_declared_objects, ID_OBJECTS_OFFSET, 1);
if (ret != 0)
{
    /* perform error-handling code */
}
```

Once the number of objects present in the object table is known, the size of the Information Block can be determined by using following equation:

```
uint16_t no_of_bytes = NUM_ID_BYTES +
               CRC_LENGTH +
               num_declared_objects * sizeof(object_t)
```

Now, the driver code can read the whole Information Block from chip to the host memory, including the 3-byte information block checksum.

```
int memory_offset = 0;
uint8_t *info_block_shadow;

/* Allocate space to read Information Block and Checksum from the chip */
info_block_shadow = (uint8_t *) malloc(no_of_bytes);
if (info_block_shadow == NULL)
{
    /* perform error handling code */
}

/* Read the Information Block from the chip */
ret = mxt_read_register(info_block_shadow, memory_offset, no_of_bytes);
if (ret != 0)
{
    /* perform error-handling code */
}
```

Now, the `info_id_t` structure should be used to memorize the location of ID Information section of the information block within the memory by declaring a “shadow” instance of the structure:

```
 info_block.id = (info_id_t *) info_block_shadow;
```

4.2.2 CHECKING THE CHECKSUM

A checksum for the Information Block should be calculated by the host on start-up when the Information Block is first read. This can then be compared to the stored checksum value at the end of the Information Block. If there is a mismatch, an error has occurred, in which case the device may need to be reset. See Appendix B “Checksum Calculation” for information on how to calculate a 24-bit CRC to validate the Information Block.

4.2.3 STORING THE OBJECT TABLE INFORMATION

Once it has been established that the device is present and running, the driver can build up a list of the objects in the device. This allows the driver to communicate with the device.

To do this, the driver code should read the Object Table held within the Information Block. This contains information on all the objects held within the memory map and indicates which objects exist, where they are located and their size.

This object table is a key feature of the object protocol. It means that the configuration settings can be held in objects rather than as “hard-coded” bytes at specific locations in the memory map. By accessing the information via an object’s address that is retrieved from the chip itself before use, no assumptions are built into driver code about the whereabouts of the configuration data. This makes cross-chip driver code possible.

Each object instance is represented by a 6-byte element in the Object Table.

TABLE 2: FORMAT OF AN ELEMENT IN THE OBJECT TABLE

Byte	Purpose	Notes
0	Type	Identifies the object.
1	Start position LSByte	The driver code should *always* read these bytes to find out where in the memory map the object is located and use this address to communicate with the device. The driver code should never use hard-coded addresses for the objects, as these may change with firmware updates.
2	Start position MSByte	
3	Size – 1	The size (minus 1) of the object in the memory map.
4	Number of instances – 1	The number of instances (minus 1) of the object type. The different instances are arranged consecutively in the memory map.
5	Number of report IDs	A report ID identifies the source object of a message. These are allocated to the object instances in the order they appear in the memory map.

4.2.4 TYPES

Each type of object has a unique type code to identify it, held in byte 0. This is the number after the “T” suffix at the end of the object’s internal name, as given in the object descriptions. For example, the type code for the Command Processor T6 object (GEN_COMMANDPROCESSOR_T6) is 6.

4.2.5 START POSITION

Bytes 1 and 2 of the Object Table element holds the start location of the object in the device memory map (LSByte and MSByte respectively).

The driver code should ALWAYS read these bytes to find out where in the memory map the object is located and use this address to communicate with the object.

This means that driver code can be written without making assumptions about the addresses of the objects. This ensures that the code is “future-proof” and will work correctly following firmware updates to the device. It also makes it possible to write common driver code for communication with any Microchip device that uses this object-based protocol approach.

4.2.6 SIZE

Byte 3 of the Object Table element holds the size in bytes (minus 1) of the object in the memory map. This is stored as Size-1, so it is effectively the offset to the end of the object.

4.2.7 NUMBER OF INSTANCES

Byte 4 of the Object Table element holds the number of instances of the object in the memory map, minus 1. The number of instances can be calculated by adding 1 to this number. The different instances of an object are arranged sequentially in the memory map.

4.3 Calculating the Report IDs

If an object sends messages, it is necessary to identify the messages from the object so that they can be correctly interpreted. A report ID is therefore used to identify the source object of a message returned in the Message Processor object.

An object that sends messages may require several report IDs. For example, in the case of a Multiple Touch Touchscreen T100 object, there is 1 report ID allocated for the screen status message, 1 reserved report ID, and a report ID for each of the touch status messages. Thus the total number of report IDs will be the maximum number of reported touches allowed on the device plus 2.

Report IDs are numbered uniquely and sequentially in the order in which the objects are listed in the Object Table, allowing for the appropriate number of instances for each object.

Note the following:

- A report ID of zero is a reserved value for use by Microchip. Report IDs from a customer's perspective therefore effectively start from 1.
- A report ID value of 255 is reserved to indicate an "invalid message" response.
- If an object has report IDs allocated, each instance of the object will have its own block of report IDs.

The figure below shows an example of how the number of instances, and the number of report IDs per instance, determine the report IDs for a set of objects. In this example there are three objects: Command Processor T6 (single instance), Multiple Touch Touchscreen T100 (single instance) and Unlock Gesture T81 (two instances). Note that the Multiple Touch Touchscreen T100 object requires 12 report IDs, whereas the other three objects only require one each, making a total of 15 report IDs needed.

FIGURE 9: EXAMPLE ASSIGNMENT OF REPORT IDS

Object	Instances	Report IDs
Command Processor T6	1	1
Multiple Touch Touchscreen T100	1	12
Unlock Gesture T81	2	1

Report ID	Object
0	*Reserved*
1	Command Processor T6
2	Multiple Touch Touchscreen T100 Screen Report
3	*Multiple Touch Touchscreen T100 Reserved*
4	Multiple Touch Touchscreen T100 Touch 0
5	Multiple Touch Touchscreen T100 Touch 1
6	Multiple Touch Touchscreen T100 Touch 2
7	Multiple Touch Touchscreen T100 Touch 3
8	Multiple Touch Touchscreen T100 Touch 4
9	Multiple Touch Touchscreen T100 Touch 5
10	Multiple Touch Touchscreen T100 Touch 6
11	Multiple Touch Touchscreen T100 Touch 7
12	Multiple Touch Touchscreen T100 Touch 8
13	Multiple Touch Touchscreen T100 Touch 9
14	Unlock Gesture T81 Instance 0
15	Unlock Gesture T81 Instance 1

NOTE: The objects shown are examples only and may not reflect the objects present on any particular device

The host driver code should build up its own in-memory table of object types and associated report IDs during its initialization. It can do this by parsing the object structure given in the Object Table. This in-memory table can then be used to interpret the messages returned by the device.

A typical algorithm to process the report IDs is as follows:

For each element in the Object Table:

1. Read byte 4 to retrieve the number of instances (remember to add 1 to the value retrieved).
2. Read byte 5 to retrieve the number of report IDs per instance.
3. Multiply the figures retrieved in steps 1 and 2 together, and then add this number of "object type/report ID" pairings to the table being built. The report IDs should have sequential values starting with 1 (the zero value is reserved for use by Microchip).

The code below shows how the driver code can build up a mapping table of object types and associated report IDs by “walking” the object structure given in the Object Table. This mapping table can then be used to interpret the messages returned by the chip.

The code uses the structure below to hold the object type and instance information. An array of these structures can be used to map the object instances to report IDs.

```
typedef struct
{
    uint8_t object_type; /* Object type */
    uint8_t instance;     /* Instance number */
} report_id_map_t;
```

The code is as follows:

```
/* Report ID zero is reserved - start from one */
uint16_t num_report_ids = 1;
uint16_t report_id_count = 1;

uint8_t element_index;
uint8_t instance_index;
uint8_t report_index;
uint16_t no_of_bytes;

object_t element;

/* Calculate the number of report IDs */
for (element_index = 0; element_index < info_block.id->num_declared_objects; element_index++)
{
    element = info_block.objects[element_index];
    num_report_ids += (element.instances + 1) * element.num_report_ids;
}

/* Allocate memory for report ID look-up table */
no_of_bytes = num_report_ids * sizeof(report_id_map_t);
report_id_map = malloc(no_of_bytes);
if (report_id_map == NULL)
{
    /* Perform error handling code */
}

/* Report ID 0 is reserved, so create empty mapping */
report_id_map[0].object_type = 0;
report_id_map[0].instance = 0;

/* Store the object and instance for each report ID */
for (element_index = 0; element_index < info_block.id->num_declared_objects;
     element_index++)
{
    element = info_block.objects[element_index];

    for (instance_index = 0; instance_index < (element.instances + 1); instance_index++)
    {
        for (report_index = 0; report_index < element.num_report_ids; report_index++)
        {
            report_id_map[report_id_count].object_type = element.object_type;
            report_id_map[report_id_count].instance = instance_index;
            report_id_count++;
        }
    }
}
return 0;
```

4.4 Configuring the Device

maXTouch devices are highly configurable, but the full details of this are outside the scope of this guide. However, it is useful to understand the basics of reading and writing object data fields so as to be able to perform simple operations, such as commanding a calibration of the part.

4.4.1 OBJECT CONFIGURATION

The objects are designed such that a default value of zero in their fields is a “safe” value that typically disables functionality. Therefore, before the device can be used, the objects must be configured correctly, and the settings written to the nonvolatile memory using the Command Processor T6 object. For a third-party module this can be assumed to have already been done. Details of the objects that may be required by the module user are given in Section 5.0 “Object Reference”.

The general and debug objects are always present. Some of these objects must be configured correctly for proper device operation; other general and debug objects are used simply to communicate with the device.

4.4.2 RETRIEVING THE ADDRESS OF AN OBJECT

The following structure can be used to represent an Object Table element:

```
struct mxt_object
{
  uint8_t type;                  /*!< Object type ID */
  uint8_t start_pos_lsb;  /*!< LSByte of the start address of the obj config structure */
  uint8_t start_pos_msb;  /*!< MSByte of the start address of the obj config structure */
  uint8_t size_minus_one;        /*!< Byte length of the obj config structure - 1 */
  uint8_t instances_minus_one;   /*!< Number of objects of this obj. type - 1 */
  uint8_t num_report_ids;        /*!< The max number of touches in a screen, etc.*/
} __attribute__((packed));
```

The code described in the section Reading the Information Block reads the information block to the address pointed to by `info_block_shadow`. The driver code should set the `object_t` pointer to the correct offset from the memory pointed to by `info_block_shadow`:

```
info_block.objects = (object_t *) ((uint8_t *)(info_block_shadow) + NUM_ID_BYTES);
```

The following function shows how to retrieve the start address of an object. The function’s prototype is:

```
uint16_t get_object_address(uint8_t object_type, uint8_t instance);
```

The function takes two parameters: the object type specified as the object’s ID, and the instance number of the object. It returns the start address of the object or an error status (`OBJECT_NOT_FOUND`) if an instance of the object is not found.

The code for the function is as follows.

```
#define OBJECT_NOT_FOUND 0
uint16_t get_object_address(uint8_t object_type, uint8_t instance)
{
    uint16_t element_index = 0;
object_t element;
for (element_index = 0; element_index < info_block.id->num_declared_objects; element_index++)
{
    element = info_block.objects[element_index];
    /* Does object type match? */
    if (element.object_type == object_type)
    {
        /* Are there enough instances defined in the firmware? */
        if (element.instances >= instance)
        {
            return get_start_position(element) + ((element.size + 1) * instance);
        }
        else
        {
            /* Perform error handling code */
            return OBJECT_NOT_FOUND;
        }
    }
}
```

4.4.3 WRITING TO A SIMPLE OBJECT

For those objects that consist of 8-bit values only, a simple function can be written to write to the object.

NOTE	Reading multi-byte numbers is little more complicated as numbers are stored in “little endian” configuration (that is, with the least significant byte in the lowest memory location).

The prototype for such as function is:

```
static uint8_t write_simple_object_cfg(uint8_t object_type, uint8_t instance, void *cfg);
```

The code for this function is:

```
uint8_t write_simple_object_cfg(uint8_t object_type, uint8_t instance, void *cfg)
{
    uint16_t object_address;
    uint8_t object_size;

    object_address = get_object_address(object_type, instance);
    object_size = get_object_size(object_type);

    if ((object_size == 0) || (object_address == 0))
    {
        return(CFG_WRITE_FAILED);
    }

    return (write_mem(object_address, object_size, cfg));
}
```

Note how the function is passed an object type, the object instance number and a pointer to a configuration structure (declared as `void*`).

The object type corresponds to the number on the end of the internal name. In this case it is 6, so the object type can be declared as:

```
#define GEN_COMMANDPROCESSOR_T6 6u
```

There is only one instance of the Command Processor T6 object so the instance is zero.

The structure to hold the Command Processor T6 object can be declared as follows:

```
typedef struct
{
    uint8_t reset;       /* Force chip reset    */
    uint8_t backupnv;    /* Force backup to NVM */
    uint8_t calibrate;   /* Force recalibration */
    /* the rest of the bytes are reserved */
    uint8_t reserved1;
    uint8_t reserved2;
    uint8_t reserved3;
    uint8_t reserved2;
} gen_commandprocessor_t6_config_t;
```

A simple function can therefore be written to call the `write_simple_object_cfg()` function above:

```
uint8_t write_power_config(gen_commandprocessor_t6_config_t cfg)
{
    return(write_simple_object_cfg(GEN_COMMANDPROCESSOR_T6, 0, (void *) &cfg));
}
```

4.4.4 READING MESSAGES FROM THE DEVICE

Status information is stored in the Message Processor T5 object in the memory map, so this object must be read to retrieve any status information from the device. The $\overline{\text{CHG}}$ line is asserted whenever a new message is available in the Message Processor T5 object, providing an interrupt-style interface. Note that the host should always use the $\overline{\text{CHG}}$ to be notified of messages; the host should not poll the device for messages.

The following code shows how to read a message from the Message Processor T5 and call a message handler function provided by application to handle it. Note that it calls the `mxt_read_register()` function defined in Section 4.2 “Reading the Information Block”.

This function returns a status message to indicate whether the read was successful or not.

The read may fail if the driver setup was not done correctly or if there is a previous message that is still being read.

```
#define MESSAGE_READ_OK 1u
#define MESSAGE_READ_FAILED 2u

uint8_t get_message(void)
{
    static volatile uint8_t read_in_progress = 0;

    uint8_t ret_val = MESSAGE_READ_FAILED;

    if (read_in_progress == 0)
    {
        read_in_progress = 1;

        if (driver_setup == DRIVER_SETUP_OK)
        {
            if(mxt_read_register(msg, message_processor_address,
                max_message_length) == READ_MEM_OK)
            {
                /* Call the main application's function to handle the message */
                application_message_handler(msg, max_message_length);
                ret_val = MESSAGE_READ_OK;
            }
        }
        read_in_progress = 0;
    }

    else
    {
        /* Previous read still in progress, just skip this one. */
    }

    return (ret_val);
}
```

To request that an 8-bit checksum is generated, the MSBit of the address of the Message Processor T5 object (`message_processor_address`) is set to 1 during the read. This can be read from the last byte returned in the message. See Appendix B "Checksum Calculation" for details on how to calculate the checksum.

4.4.5 CONFIGURATION ERRORS

The driver code must handle configuration errors. These are signaled by the CFGERR bit in the Command Processor T6 and are sent every 200 ms until the configuration error is fixed. The device halts scanning the touchscreen until the error is fixed. Modifying configurations is outside the scope of this document; however, it is useful for the driver to be able to save a reference copy of the configuration, to be able to tell the device to reload its configuration from its own NVM, and to be able to upload a new configuration should the module supplier issue an updated or improved configuration for the module. See Section 5.1.2 "Command Processor T6 Object" for details.

4.5 Configuration Values

The objects are designed such that a value of zero in their fields is a "safe" value and typically a value of zero means that a default value is used.

An object must be configured as required with non-zero values before use. Any unused settings can be left at their default zero values. The settings should also be written to the non-volatile memory using the Command Processor T6 object.

4.5.1 COMPATIBILITY OF OBJECT VERSIONS

The Object Protocol described in this document may contain fields that are not present in the memory map as it is implemented on a particular firmware version of the device. Over time newer versions of the objects in the Object Protocol may gain additional fields to implement new features.

New fields are added to the end of an object to allow for this situation. This preserves the order of the fields between old and new object versions. A driver designed to work with an older version of the device can safely set any unknown fields located at the end of the object to zero. The device will then behave in the same manner as the older version of the device without the field.

The host driver must always use the Object Table to locate the address of each object and the object's current size. It must also zero any fields that it does not intend to use. This ensures that the host driver code is compatible with this object expansion scheme.

4.5.2 BYTE ORDER

The memory map uses a "little-endian" configuration for its bytes, meaning that all multibyte fields lead with the least significant byte (LSByte) at the lowest device memory address.

5.0 OBJECT REFERENCE

This section provides information on some of the commonly used objects. Not all objects are described in this guide, but all the ones that the typical user will need to interact with are detailed here.

NOTE	The descriptions that follow describe the likely interactions with the objects, such as useful commands and the format of the message data. In most cases, modifying the configuration of an object is outside the scope of this document.

5.1 Control, Command and Support Objects

5.1.1 MESSAGE PROCESSOR T5 OBJECT

5.1.1.1 Introduction

The purpose of the Message Processor T5 object is to relay the latest status information to the host. This object contains the message data from those objects in the memory map that generate messages (for example, the touch objects and the Command Processor T6 object). A message is generated whenever an object's status has changed (if it has been configured to do this).

5.1.1.2 Configuration

TABLE 3: CONFIGURATION FOR MESSAGE PROCESSOR T5 (GEN_MESSAGEPROCESSOR_T5)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	REPORTID	Report ID for source object							
1 – *n* (1)	MESSAGE	Message data from source object							
n+1	CHECKSUM (2)	Checksum							

Note 1: The size of the MESSAGE field is set to the length of largest message possible. It is dependent on the objects present in the device at a particular revision. The size should be calculated by subtracting 2 from the size of the Message Processor T5 object retrieved from the Object Table entry. Any unused bytes in a particular message should be treated as reserved bytes.

2: This byte is only transmitted if checksumming is enabled.

REPORTID Field

This field contains the report ID for the message. Messages contain report IDs to allow the host to identify the type of message and its originator. Report IDs are assigned to any object that can send messages.

MESSAGE Field

This field contains the message data for the object generating the message.

The size of the MESSAGE field is fixed to the size of the message data for the largest object. For compatibility with future firmware updates, this should *always* be calculated by subtracting 2 from the size of the object recorded in the Object Table entry for the Message Processor T5 object.

For information on the contents of the MESSAGE field for commonly used objects, see the descriptions for each object elsewhere in this document.

CHECKSUM Field

This field contains the 8-bit checksum for the Message Processor T5 object (that is, for the REPORTID and MESSAGE fields) if a communications checksum is requested.

To request that a checksum is generated, the MSBit of the address of the Message Processor T5 object is set to 1 during a read. For example, if the address of the Message Processor T5 object is 0x0477, specifying the address as 0x**8**477 will generate a checksum for that read.

If the communications checksum feature is not enabled, this byte should not be read.

See Checksum Calculation for details on how to calculate the checksum.

5.1.2 COMMAND PROCESSOR T6 OBJECT

5.1.2.1 Introduction

The Command Processor T6 object allows commands to be sent to the device. This is done by writing an appropriate value to one of its fields.

5.1.2.2 Configuration

TABLE 4: CONFIGURATION FOR COMMAND PROCESSOR T6 (GEN_COMMANDPROCESSOR_T6)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	RESET	Reset							
1	BACKUPNV	Backup settings							
2	CALIBRATE	Calibrate							
3	Reserved	Reserved							
4	Reserved	Reserved							
5	Reserved	Reserved							
6	Reserved	Reserved							

RESET Field

This field forces a reset of the device if a nonzero value is written (note: 0xA5 is a reserved value and must not be used).

BACKUPNV Field

This field supports the commands in the following table.

TABLE 5: BACKUPNV COMMANDS

Command	Description
0x55	Backs up the configuration settings to the Non-volatile Memory (NVM).
0x33	Restores the configuration settings from the NVM.

Once the device has processed this command it generates a status message containing the new NVM checksum.

CALIBRATE Field

This field performs a global recalibration on all sensing channels. If all the channels are disabled, no message is generated.

Write value: Nonzero

5.1.2.3 Messages

The message data for the Command Processor T6 object is shown in the following table. This can be read by viewing the contents of the Message Processor T5 object.

TABLE 6: MESSAGE DATA FOR COMMAND PROCESSOR T6 (GEN_COMMANDPROCESSOR_T6)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	RESET	OFL	SIGERR	CAL	CFGERR	COMSERR	Reserved	
2 – 4	CHECKSUM	Configuration settings checksum							

STATUS Field

Reports the current status and flags errors. A bit is set to indicate the corresponding status/error. Note that there may be more than one status/error reported.

CFGERR, CAL, SIGERR and OFL report ongoing status and error conditions, so once these status/error conditions have terminated, a further message is sent with the appropriate bit cleared.

COMSERR and RESET are one-off reports indicating already terminated conditions. These error conditions do not generate a further message with a cleared bit.

COMSERR: There is an error with the communications checksum. This error bit is set when the device is being used in communications checksum mode and there has been a checksum error on the bytes that have been written to the device. Note that if there is a checksum error after a write, then the data will still have been written to the device. It is the host's responsibility to take corrective action.

CFGERR: There is a configuration error in one or more of the enabled objects. The device pauses its processing and generates a status message every 200 ms. Note that the device will stop scanning for touches while the error persists.

NOTE	It is possible to execute a backup command while the device is in this error state.

CAL: The device is calibrating.

SIGERR: There was an error in the acquisition. This error should not normally be seen.

OFL: The acquisition and processing cycle length has overflowed the desired power mode interval. The OFL flag is not updated in Free-run or Deep Sleep modes.

RESET: The device has reset.

CHECKSUM Field

Reports the checksum of the configuration settings held in the non-volatile memory. See Appendix B "Checksum Calculation" for details on how to calculate the checksum.

5.1.3 COMMUNICATIONS CONFIGURATION T18 OBJECT

5.1.3.1 Introduction

The Communications Configuration T18 object specifies additional communications behavior for the device.

5.1.3.2 Configuration

TABLE 7: CONFIGURATION FOR COMMUNICATIONS CONFIGURATION T18 (SPT_COMMSCONFIG_T18)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	CTRL	DISMNTR	Reserved				MODE	Reserved	
1	COMMAND	$\overline{\text{CHG}}$ line command code							

CTRL Field

MODE: Selects the $\overline{\text{CHG}}$ line mode. If this bit is set to 0 (the default), the $\overline{\text{CHG}}$ line operates in Edge-triggered Mode. If this bit is set to 1, the $\overline{\text{CHG}}$ line operates in Level-triggered Mode. See Section 3.6 “CHG Line” for more information on the $\overline{\text{CHG}}$ line modes.

DISMNTR: Disables the bus monitor. This monitors the I^2C-compatible lines. If either line is low for more than 200 ms, the I^2C-compatible hardware is reset. This ensures that a “stuck” I^2C-compatible bus is detected. The bus monitor is disabled if this bit is set to 1 and enabled if it is set to zero. Note that the bus monitor is not active in deep sleep mode.

COMMAND Field

The COMMAND field provides direct control over the $\overline{\text{CHG}}$ line. This overrides the operation of the $\overline{\text{CHG}}$ line controlled by the MODE bit in the CTRL field. Entering one of the command codes listed in the following table alters the state of the $\overline{\text{CHG}}$ line.

TABLE 8: COMMAND CODES

Command	Description
0	No command (default)
1	Return $\overline{\text{CHG}}$ line to normal operation, as determined by the MODE bit of the CTRL field
2	Force the $\overline{\text{CHG}}$ line high (inactive)
3	Force the $\overline{\text{CHG}}$ line low (active)

5.2 Touch Objects

5.2.1 KEY ARRAY T15 OBJECT

NOTE Support for this object is firmware dependent and so this object may not be available on your device.

5.2.1.1 Messages

A Key Array T15 object reports on/off touch information in its message data. Note that Key Array T15 messages will be suppressed if a touch suppression method has been enabled.

The message data can be read by viewing the contents of the Message Processor T5 object.

TABLE 9: MESSAGE DATA FOR KEY ARRAY T15 (TOUCH_KEYARRAY_T15)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	DETECT	Reserved						
2	KEYSTATE	KEY7	KEY6	KEY5	KEY4	KEY3	KEY2	KEY1	KEY0
3		KEY15	KEY14	KEY13	KEY12	KEY11	KEY10	KEY9	KEY8
4		KEY23	KEY22	KEY21	KEY20	KEY19	KEY18	KEY17	KEY16
5		KEY31	KEY30	KEY29	KEY28	KEY27	KEY26	KEY25	KEY24

STATUS Field

Reports the current status of the object.

DETECT: Set if any key is in a touched state.

KEYSTATE Field

Reports the state of each key, one bit per key; 0 = key is untouched, 1 = key is touched. Note that the Object Protocol allows for a maximum of 32 keys per key array, but there may be less than this on your particular device; please check with the module manufacturer. Where there are less than 32 keys allowed, the last bytes will be reserved.

5.2.2 MULTIPLE TOUCH TOUCHSCREEN T100 OBJECT

NOTE Modifying the configuration of the touchscreen is outside the scope of this document.

5.2.2.1 Messages

The Multiple Touch Touchscreen T100 object reports the following screen and touch status information:

- Screen status information, such as the number of nodes affected and the number of reported touches
- Number and details of Finger touches detected
- Any active grip and screen suppression
- Passive and/or Active stylus touches
- Glove touches
- Hovering touches

Note that not all types of touch information will necessarily be present on your particular device.

There are three types of message reported from a Multiple Touch Touchscreen T100. The first report ID from a Multiple Touch Touchscreen T100 is for the global screen status messages that indicate the state of the touchscreen system as a whole. The second report ID is reserved. Subsequent report IDs are for each of the reportable tracked touches.

The message data can be read by viewing the contents of the Message Processor T5 object.

First Report ID – Screen Status Messages

The first report ID from the Multiple Touch Touchscreen T100 object is used to output screen status messages. A screen status message is triggered whenever one of the reported data fields changes its value or, in the case of the auxiliary data, reaches a particular trigger value.

TABLE 10: MESSAGE DATA FOR MULTIPLE TOUCH TOUCHSCREEN T100 (TOUCH_MULTITOUCHSCREEN_T100) – FIRST REPORT ID

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	SCRSTATUS	DETECT	SUP	Reserved					
2 – 9	AUXDATA[]	Auxiliary data							

SCRSTATUS Field

This field indicates the status of the touchscreen.

DETECT: This bit is set to indicate that there is at least one reporting touch on the touchscreen.

SUP: This bit is set to indicate that full screen suppression is in effect.

AUXDATA[] Fields

These fields report extra configurable information about the screen. Values for these fields should be treated as reserved unless advised otherwise by the module manufacturer.

Second Report ID – Reserved

The second report ID is reserved for future use.

Subsequent Report IDs – Touch Status Messages

Subsequent report IDs are used to output the touch status messages. There is a report ID for each of the touches on the device.

TABLE 11: MESSAGE DATA FOR MULTIPLE TOUCH TOUCHSCREEN T100 (TOUCH_MULTITOUCHSCREEN_T100) – SUBSEQUENT REPORT IDS

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	TCHSTATUS	DETECT	TYPE	EVENT					
2	XPOS	X position LSByte							
3		X position MSByte							
4	YPOS	Y position LSByte							
5		Y position MSByte							
6 – 9	AUXDATA[]	Auxiliary data							

A touch status message is triggered when one of the reported data fields reaches a particular trigger value or the touch type changes.

TABLE 12: MESSAGE TRIGGERS – TOUCH STATUS MESSAGES

Field	Effect on Message Generation
TCHSTATUS EVENT	Message generated if there is an event
TCHSTATUS TYPE	Message generated if TYPE changes
AUXDATA[] Fields	No effect on message generation

TCHSTATUS Field

This field indicates the status of the touch.

EVENT: These bits indicate the event that has just occurred for this finger touch.

TABLE 13: EVENT CODES

Event	Name	Description
0	NO EVENT	No specific event has occurred
1	MOVE	The touch position has changed
2	UNSUP	The touch has just been unsuppressed by the touch suppression features of other objects
3	SUP	The touch has been suppressed by the touch suppression features of other objects
4	DOWN	The touch has just come within range of the sensor
5	UP	The touch has just left the range of the sensor
6	UNSUPSUP	Both UNSUP and SUP events have occurred (in either order)
7	UNSUPUP	Both UNSUP and UP events have occurred (in either order)
8	DOWNSUP	Both DOWN and SUP events have occurred (in either order)
9	DOWNUP	Both DOWN and UP events have occurred (in either order)

TYPE: These bits indicates the type of touch that is being reported. A touch status message is generated if the type changes.

TABLE 14: TOUCH TYPES

Event	Name	Description
0	RESERVED	Reserved for future use.
1	FINGER	The touch is considered to be a finger that is contacting the screen.
2	PASSIVE STYLUS	The touch is a passive stylus.
3	ACTIVE STYLUS	The touch is an active stylus (depends on device).

TABLE 14: TOUCH TYPES (CONTINUED)

Event	Name	Description
4	HOVERING FINGER	The touch is a hovering finger (depends on device).
5	GLOVE	The touch is a glove touch.
6	LARGE TOUCH	The touch is a suppressed large touch.

DETECT: This bit is set to 1 to indicate that the touch is reporting and present within the range of the sensor.

XPOS Field

This field reports the X position.

YPOS Field

This field reports the Y position.

AUXDATA[] Fields

These fields hold configurable (type specific) information about the touch. Values for these fields should be treated as reserved unless advised otherwise by the module manufacturer.

5.3 Gesture Objects

5.3.1 ONE-TOUCH GESTURE PROCESSOR T24 OBJECT

NOTE Support for this object is firmware dependent and so this object may not be available on your device.

5.3.1.1 Introduction

The One-touch Gesture Processor T24 object configures the on-chip gesture processing for one-touch gestures (such as taps, double taps, presses, flicks and drags).

5.3.1.2 Messages

The message data for the One-touch Gesture Processor T24 object is shown in the following table. The message data can be read by viewing the contents of the Message Processor T5 object.

TABLE 15: MESSAGE DATA FOR ONE-TOUCH GESTURE PROCESSOR T24 (PROCI_ONETOUCHGESTUREPROCESSOR_T24)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	Reserved				Event			
2	XPOSMSB	X position MSByte							
3	YPOSMSB	Y position MSByte							
4	XYPOSLSB	X position lsbits				Yposition lsbits			
5	DIR	Gesture direction							
6 – 7	DIST	Gesture distance							

STATUS Field

This field reports which single-touch event has occurred.

TABLE 16: EVENTS

Event	Description
0	Reserved
1	Press
2	Release
3	Tap
4	Double-tap
5	Flick
6	Drag
7	Short Press
8	Long Press
9	Repeat Press
10	Tap-and-press
11	Throw
12	Tap-and-touch
13 to 15	Reserved

XPOSMSB, YPOSMSB and XYPOSLSB Fields

These three fields report the X and Y position of the gesture. XPOSMSB/YPOSMSB contains the most significant byte of the position. XYPOSLSB contains the least significant bits of the position.

The position is reported in 12- bit format at the resolution specified by Multiple Touch Touchscreen T100.

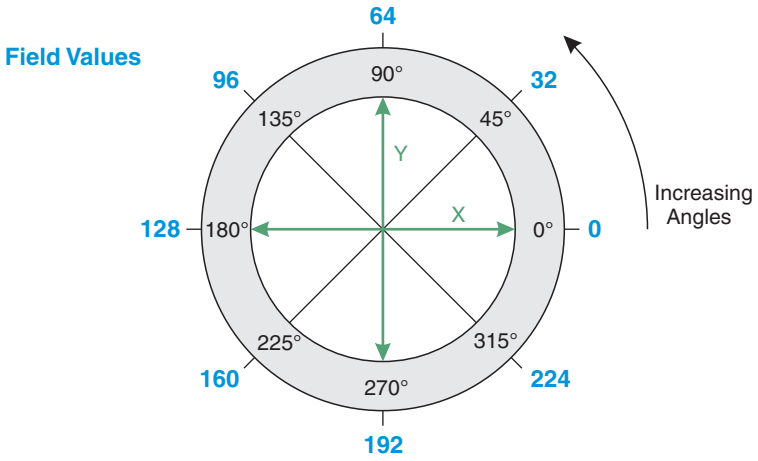
TABLE 17: X AND Y POSITION FORMATS

XPOSMSB								XYPOSLSB							
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**bit 3**	**Bit 2**	**Bit 1**	**bit 0**
X Position Format															
2048	1024	512	256	128	64	32	16	8	4	2	1	Y position lsbits			
Y Position Format															
2048	1024	512	256	128	64	32	16	X position lsbits				8	4	2	1

DIR Field

The DIR field contains the direction of the flick or throw when a flick or throw event is generated. The direction is specified in 1/256 of 360°.

FIGURE 10: DIRECTION AND ANGLE VALUES



Note: The X and Y positions are before any processing of the Multiple Touch touchscreen T100 orientation settings

DIST Field

The DIST field contains the distance of the flick or throw when a flick or throw event is generated.

5.3.2 TWO-TOUCH GESTURE PROCESSOR T27 OBJECT

NOTE Support for this object is firmware dependent and so this object may not be available on your device.

5.3.2.1 Introduction

A Two-touch Gesture Processor T27 object configures the on-chip touchscreen gesture processing for the two-touch gestures: pinch, rotate and stretch.

5.3.2.2 Messages

The message data for a Two-touch Gesture Processor T27 object is shown in the following table. The message data can be read by viewing the contents of the Message Processor T5 object.

TABLE 18: MESSAGE DATA FOR TWO-TOUCH GESTURE PROCESSOR T27 (PROCI_TWOTOUCHGESTUREPROCESSOR_T27)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	STRETCH	ROTATE	PINCH	ROTATEDIR	Reserved			
2	XPOSMSB	Center of gesture X position MSByte							
3	YPOSMSB	Center of gesture Y position MSByte							
4	XYPOSLSB	X position lsbits				Yposition lsbits			
5	ANGLE	Gesture angle							
6	SEP	Gesture separation LSByte							
7		Gesture separation MSByte							

STATUS Field

This field reports which two-touch events have occurred. A two-touch gesture may consist of more than one two-touch gestures. For example, a rotate may be combined with a stretch to create a spiralling zoom effect.

ROTATEDIR: Indicates the direction of the rotate event: 0 = rotation in the direction of increasing angles, 1 = rotation in the direction of decreasing angles. This bit should be ignored if the ROTATE bit is not set.

PINCH: If set to 1, a pinch event has occurred.

ROTATE: If set to 1, a rotate event has occurred.

STRETCH: If set to 1, a stretch event has occurred.

XPOSMSB, YPOSMSB and XYPOSLSB Fields

These three fields report the X and Y position of the center of the gesture. XPOSMSB/YPOSMSB contains the most significant byte of the position. XYPOSLSB contains the least significant bits of the position.

The position is reported in 12- bit format at the resolution specified by Multiple Touch Touchscreen T100.

ANGLE Field

The ANGLE field reports the angle between two touches in a two-touch gesture. The angle between the two touches, and its direction, is determined by the relative X and Y positions of the second touch from the first touch. The angle is reported in 1/256 of 360° in the same way as for the One-touch Gesture Processor T24 DIR field.

SEP Field

The SEP field reports the separation (distance) between the two touches in a two-touch gesture as a 16-bit number. This value could be used by the host, for example, as an indication of the speed of a rotate or stretch gesture.

5.3.3 UNLOCK GESTURE T81 OBJECT

NOTE Support for this object is firmware dependent and so this object may not be available on your device.

5.3.3.1 Introduction

The Unlock Gesture T81 object provides a fully configurable gesture processor. This object provides a message when a gesture satisfies the configuration settings. This feature can then be used in wake up and/or unlock situations with minimal host intervention.

Consult your module manufacturer for details of which unlock gestures have been enabled on your device.

5.3.3.2 Messages

The message data for an Unlock Gesture T81 object is shown in the following table. The message data can be read by viewing the contents of the Message Processor T5 object.

TABLE 19: MESSAGE DATA FOR UNLOCK GESTURE PROCESSOR T81 (PROCI_UNLOCKGESTURE_T81)

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	Reserved							UNLOCK
2	XDELTA	X position difference LSByte							
3		X position difference MSBye							
4	YDELTA	Y position difference LSByte							
5		Y position difference MSByte							

STATUS Field

UNLOCK: Indicates that the unlock gesture is complete.

XDELTA and YDELTA Fields

These fields report the differences between the start and end coordinates of the touch.

APPENDIX A: SAMPLE REFERENCE CODE

The reference code with functions described in this document is part of an open-source utility called mxt-app. The utility can be downloaded from the github repository:

- https://github.com/Microchip-maxtouch/mxt-app/tree/master/src/libmaxtouch

It consists of an application front end to a code library (the *libmaxtouch* library). The utility performs the following operations:

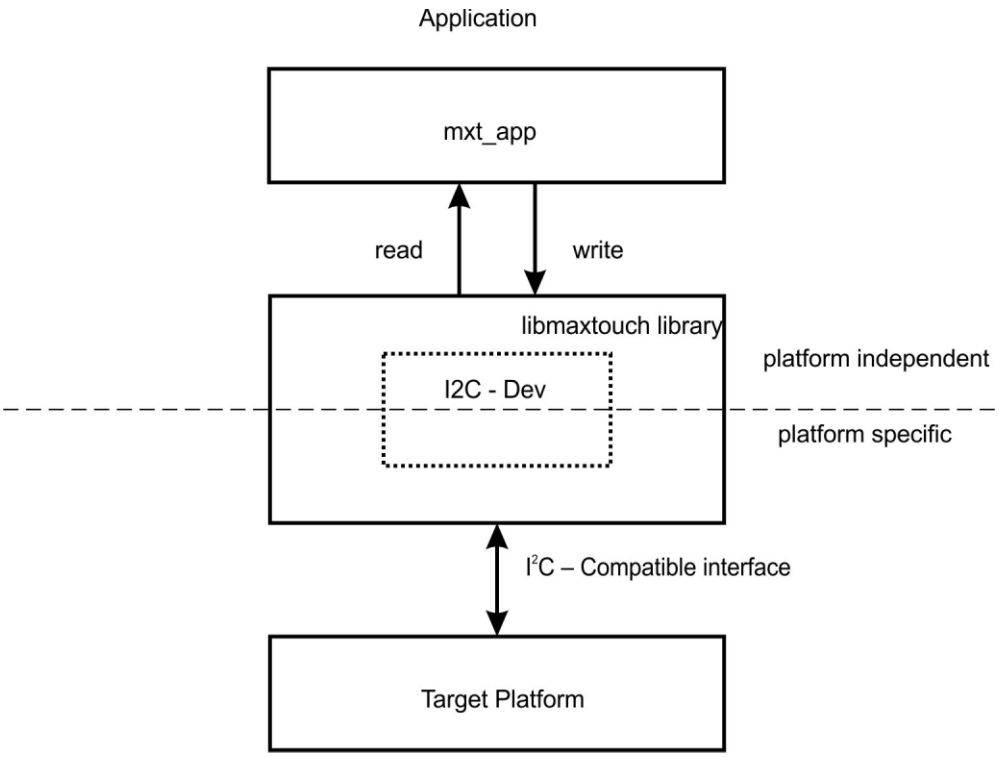
- Loads a configuration file
- Displays the information block
- Reads the bytes in an object from the object table
- Writes the bytes in an object from the object table
- Runs the self-tests
- Back-ups the configuration data
- Resets the chip
- Calibrates the chip

NOTE	The mxt_app utility is written to run on Linux and Android platforms. It is generic C code and should be easily portable to other platforms. The code assumes that an I^2C interface is used for communication. If another interface (for example, SPI) is used, the code will need modification. The utility will work on all maXTouch devices.

A.1 Driver Structure

The following diagram provides a pictorial representation of the various segments of the driver software.

FIGURE 11: DRIVER SOFTWARE BLOCK DIAGRAM



The driver is divided into general and platform specific parts. The platform specific part includes all the functions that directly access the hardware. Thus there should be no need to modify the general part when porting the driver to new platforms.

A.1.1 GENERAL PART

The general part will include all the functions the main application needs for initializing, configuring, and receiving messages from the touch chip. In the example code, the mxt-app application is platform-independent code, whereas the calling libmaxtouch library functions are largely platform independent, with the exceptions described as below.

A.1.2 PLATFORM PART

A.1.2.1 Implementing the Platform Functions

The platform part needs to implement two things:

- The basic I^2C routines to read/write data to and from the chip.
- An interrupt handler that monitors the $\overline{CHG}$ pin and calls the `get_message()` function in the main driver files. This, in turn, should read the message from the chip and call the message handler provided by the main application, with the message and its length as arguments. It may also be possible to monitor this interrupt in the user space by connecting the interrupt through to a GPIO line.

The maXTouch driver operates on an interrupt-based mechanism where maXTouch chip asserts an active-low interrupt ($\overline{CHG}$) whenever there is an event on touch screen.

When implementing the driver for a new platform, it is necessary to create and/or modify the files in the platform directory, as well as the `main.c` file.

The following table lists the functions that needs to be implemented.

TABLE 20: REQUIRED PLATFORM SPECIFIC FUNCTIONS

Function	Description
`void init_changeline(void);`	Initializes the $\overline{CHG}$ line handling.
`void disable_changeline_int(void);`	Disables pin interrupts on $\overline{CHG}$ falling edge (optional).
`void enable_changeline_int(void);`	Enables pin interrupts on $\overline{CHG}$ falling edge (optional).
`uint8_t init_I2C(uint8_t I2C_address);`	Initializes the I^2C interface. Parameter: I^2C address of chip.
`int mxt_read_register (unsigned char *dst_buf, int start_register, int ByteCount);`	Reads data from the chip via the I^2C interface
`int mxt_write_register (unsigned char *dst_buf, int start_register, int ByteCount);`	Writes data to the chip via the I^2C interface.
`uint_8 read_changeline()`	Reads the current state of the $\overline{CHG}$ line.

APPENDIX B: CHECKSUM CALCULATION

B.1 24-bit CRC

B.1.1 INTRODUCTION

The device uses a 24-bit cyclic redundancy check (CRC) in the following places:

- To check the integrity of the main Information Block for the device
- To check the integrity of the configuration settings held in the non-volatile memory

These CRC checksums allow the host to be confident that the memory map layout has been read over the communications bus correctly.

NOTE The C code in this appendix uses the type declarations `uint8_t`, `uint16_t` and `uint32_t` for 8-bit, 16-bit and 32-bit unsigned integers respectively.

B.1.2 24-BIT CRC ALGORITHM

Each checksum is generated by running an algorithm which takes two new bytes of data and combines them with the current checksum to produce a new checksum value.

The sample code below shows how to calculate the checksum for each 16-bit word in a byte stream.

```
uint32_t crc24(uint32_t crc, uint8_t firstbyte, uint8_t secondbyte)
{

    static const uint32_t crcpoly = 0x80001B;
    uint32_t result;
    uint16_t data_word;

    data_word = (uint16_t)firstbyte | (uint16_t)((uint16_t)secondbyte << 8u);
    result = (uint32_t)data_word ^ (uint32_t)(crc << 1u);

    if(result & 0x1000000) // If bit 25 is set
    {
        result ^= crcpoly;// XOR result with crcpoly
    }

    return result;
}
```

The checksum routine is called iteratively to calculate the CRC two bytes (16-bit word) at a time. This means that two bytes must be read from the device before performing each iteration of the checksum. Therefore, if an odd number of bytes is read from the device, the host's checksum code should add a zero byte to the end of the byte stream to make the sequence even. For example, if the following stream of 7 bytes is received from the device:

byte1 — byte2 — byte3 — byte4 — byte5 — byte6 — byte7

The checksum could be calculated as follows:

```
uint32_t CRC = 0;
CRC = crc24(CRC, byte1, byte2)
CRC = crc24(CRC, byte3, byte4)
CRC = crc24(CRC, byte5, byte6)
CRC = crc24(CRC, byte7, 0) /* <- zero added for the last checksum */

CRC = CRC & 0x00FFFFFF; /* <- mask the 32-bit result to 24-bit */
```

An alternative method of calling the CRC calculation function is to use a loop, as shown in the next section.

The following table shows an example block of 32 bytes and the CRCs these generate. The bytes consist of 31 data bytes plus an additional zero byte to even up the count. You can use the data in this table to verify the validity of any coded CRC routine.

TABLE 21: EXAMPLE CRC CALCULATIONS FOR 24-BIT CRC

First Byte	Second Byte	CRC Calculation Result
0x00	0xFF	0x00FF00 (Intermediate CRC – partial calculation)
0x11	0xEE	0x011011 (Intermediate CRC – partial calculation)
0x22	0xDD	0x02FD00 (Intermediate CRC – partial calculation)
0x33	0xCC	0x053633 (Intermediate CRC – partial calculation)
0x44	0xBB	0x0AD722 (Intermediate CRC – partial calculation)
0x55	0xAA	0x150411 (Intermediate CRC – partial calculation)
0x66	0x99	0x2A9144 (Intermediate CRC – partial calculation)
0x77	0x88	0x55AAFF (Intermediate CRC – partial calculation)
0x88	0x77	0xAB2276 (Intermediate CRC – partial calculation)
0x99	0x66	0xD6226E (Intermediate CRC – partial calculation)
0xAA	0x55	0x2C116D (Intermediate CRC – partial calculation)
0xBB	0x44	0x586661 (Intermediate CRC – partial calculation)
0xCC	0x33	0xB0FF0E (Intermediate CRC – partial calculation)
0xDD	0x22	0xE1DCDA (Intermediate CRC – partial calculation)
0xEE	0x11	0x43A841 (Intermediate CRC – partial calculation)
0xFF	0x00	0x87507D (Expected CRC – final calculation)

B.1.3 INFORMATION BLOCK CHECKSUM

The checksum for the Information Block should be calculated by the host on start-up when the Information Block is first read. This should be compared to the checksum value at the end of the Information Block. If there is a mismatch, an error has occurred.

The following code shows how to use the example `crc24()` function given in the previous section to calculate the CRC checksum for the Information Block.

```
uint32_t crc = 0; /* Calculated CRC */
uint16_t crc_area_size;/* Size of data for CRC calculation */
uint8_t *mem;/* Data buffer */
uint8_t i;
uint8_t status;

/* 7 bytes of version data, 6 * NUM_OF_OBJECTS bytes of object table. */
crc_area_size = ID_INFORMATION_SIZE +
            info_block->info_id.num_declared_objects *
            OBJECT_TABLE_ELEMENT_SIZE;

mem = (uint8_t *) malloc(crc_area_size);
if (mem == NULL)
{
    /* Handle error */
}

/*
 * Read the data using a function written for this purpose
 * Here, it is assumed that the function is named read_mem()
 * and that it returns a status code with the value READ_MEM_OK.
 * It is assumed to have the following prototype:
 * uint8_t read_mem( uint16_t Address, uint8_t ByteCount, uint8_t *Data )
*/
status = read_mem(0, crc_area_size, mem);

if (status != READ_MEM_OK)
{
    /* Handle error */
}
```

```
/*
 * Call the CRC function crc24() iteratively to calculate the CRC,
 * passing it two bytes at a time.
*/
i = 0;
while (i < (crc_area_size - 1))
{
    crc = crc24(crc, *(mem + i), *(mem + i + 1));
    i += 2;
}

/*
 * Call the crc24() with the final byte,
 * plus an extra 0 value byte to make the sequence even.
*/
crc = crc24(crc, *(mem + i), 0);

free(mem);

/* Final result */
crc = (crc & 0x00FFFFFF); /* <- mask the 32-bit result to 24-bit */
```

B.1.4 CONFIGURATION CHECKSUM

The configuration checksum checks the integrity of the memory map when the device non-volatile memory is written to. Typically this is when the device is initially set in the factory or during subsequent firmware upgrades. The host should perform a CRC on the entire contents of the memory map from the address of the Dynamic Configuration Container T71 object onwards. This CRC should then be compared with the expected checksum.

B.2 8-bit CRC

B.2.1 INTRODUCTION

An 8-bit CRC can be used in the following places:

- During communications with the host to check that data has been transmitted over the communications bus correctly
- To check the integrity of data in messages from the Message Processor T5 object when receiving messages
- To check the integrity of data output in messages

B.2.2 8-BIT CRC ALGORITHM

The sample code below shows how to calculate the 8-bit checksum.

```
uint8_t crc8(unsigned char crc, unsigned char data)
{
    static const uint8_t crcpoly = 0x8C;
    uint8_t index;
    uint8_t fb;
    index = 8;

    do
    {
        fb = (crc ^ data) & 0x01;
        data >>= 1;
        crc >>= 1;
        if (fb)
            crc ^= crcpoly;
    } while (--index);

    return crc;
}
```

B.2.3 COMMUNICATIONS CHECKSUM MODE

B.2.3.1 Introduction

In communications checksum mode an 8-bit CRC is added to all data transmissions. The CRC is sent during I^2C communications (for example, at the end of the data as the last byte before the STOP condition). The CRC is calculated on all the bytes sent, including the address bytes.

To indicate that a checksum is to be included, the most significant bit of the MSByte of the address is set to 1. In the following examples, therefore, the start address of 0x1234 is sent as 0x9234.

The following sections give examples of I2C write and read operations.

B.2.3.2 I^2C Write Example

This example sets the address pointer to 0x1234. The address is sent to the device with the most significant bit of the MSByte set to 1 (that is, 0x9234). This indicates I^2C communications checksum mode. The example then writes five bytes to the device: four data bytes plus a checksum byte.

FIGURE 12: EXAMPLE I^2C WRITE WITH CHECKSUM

		Address LSByte	Address MSByte	Data	Data	Data	Data	CRC	
START	SLA-W	0x34	0x92	0x96	0x9B	0xA0	0xA5	0x7A	STOP

The example I^2C command in the diagram above writes the four bytes to contiguous addresses:

0x96 to address 0x1234
0x9B to address 0x1235
0xA0 to address 0x1236
0xA5 to address 0x1237

The I^2C command sends a checksum (in this case, 0x7A) as the last byte before the STOP condition. The device compares this checksum with its own checksum calculated from the data sent. If the two checksums do not match, the Command Processor T6 object in the device generates a COMSERR message.

NOTE	The order of messages is not guaranteed so messages may be out of order. Other intervening messages may be received before this one.

The following table shows the sequence of the bytes in this example and the intermediate calculations of the CRC.

TABLE 22: CRC CALCULATIONS FOR EXAMPLE I^2C WRITE

Bytes Sent by Host		**CRC Calculations in the Device**	
0x34	Address LSByte	0xDF	Intermediate CRCs (partial calculations)
0x92	Address MSByte	0xBB	
0x96	Data	0xDE	
0x9B		0x79	
0xA0		0xCB	
0xA5		0x7A*	Expected CRC (final calculation)
0x7A*	Actual CRC		

* These two values should match

B.2.3.3 I^2C Read Example

A checksum can be added to I^2C reads of the Message Processor T5 object. This contains a checksum as its last byte when I^2C communications checksum mode is enabled in the Message Processor T5 object. No other I^2C reads can have a checksum.

The following example reads the message data from the Message Processor T5 object in the device. It is assumed in this example that the address of the Message Processor T5 object is 0x1234. The address is sent to the device with most significant bit of the MSByte set to 1 (that is, 0x9234). This indicates I^2C communications checksum mode.

FIGURE 13: EXAMPLE I²C READ WITH CHECKSUM

Set Address Pointer

			Address LSByte	Address MSByte	CRC	
START	SLA-W		0x34	0x92	0xBB	STOP

Read Data

		Report ID	Data	Data	Data	Data	Data	Data	Data	CRC	
START	SLA-R	0x01	0x9B	0xA0	0xA5	0xAA	0xAF	0xB4	0xB9	0xA8	STOP

		Report ID	Data	Data	Data	Data	Data	Data	Data	Data	CRC	
START	SLA-R	0x01	0x9B	0xA0	0xA5	0xAA	0xAF	0xB4	0xB9	0xCC	0x04	STOP

The example consists of two operations. The first operation is an I²C write of the address to the device. The second operation is an I²C read of the message data.

The sequence of the initial write bytes with the intermediate calculations of the CRC is shown in the following table.

TABLE 23: CRC CALCULATIONS FOR EXAMPLE I²C WRITE

Bytes Sent by Host		CRC Calculations in the Device	
0x34	Address LSByte	0xDF	Intermediate CRC (partial calculation)
0x92	Address MSByte	0xBB*	Expected CRC (final calculation)
0xBB*	Actual CRC		

* These two values should match

An extra byte that holds the checksum (0xBB in this case) of the start address of the Message Processor T5 object is sent as the last byte before the STOP condition. This prevents a COMSERR message being generated by the Command Processor T6 object.

The following table gives the sequence of the read bytes with the intermediate calculations of the CRC.

TABLE 24: CRC CALCULATIONS FOR EXAMPLE I²C READ OF MESSAGE DATA

Bytes Sent by Host		CRC Calculations in the Device	
0x01	Report ID	0x5E	Intermediate CRCs (partial calculations)
0x9B	Data	0xF5	
0xA0		0xE4	
0xA5		0x18	
0xAA		0x8E	
0xAF		0x7D	
0xB4		0x56	
0xB9		0xA8	
0xCC		0x04*	Expected CRC (final calculation)
0x04*	Actual CRC		

* These two values should match

B.2.4 READING THE CRC FOR THE MESSAGE PROCESSOR T5 DATA

The following code shows how to use the example `crc8()` function given earlier to calculate the CRC checksum for the data in the Message Processor T5 object.

```
uint8_t crc = 0;
uint8_t data_in;

for(i=0; i<MESSAGEPROCESSOR_SIZE; i++)
{
    data_in = read_byte();
    crc = crc8(crc, data_in);

}
if(crc == 0)
{
    /* CRC is OK - do something appropriate */
    crc_pass();
}
else
{
    /* CRC failed - handle error */
    crc_fail();
}
```

APPENDIX C: SELF TEST OBJECT

The self Test T25 object can be used in a third-party manufacturer's factory for module testing. This appendix gives details of the object.

C.1 Self Test T25 (SPT_SELFTEST_T25)

C.1.1 INTRODUCTION

The Self Test T25 object runs self-test routines in the device to find faults in the sense lines and electrodes. The Self Test T25 object runs a series of test sequences. As soon as the first failure is found, the test run stops and the object reports a message. See Section C.1.3 "Messages" for details of the test messages.

Self Test T25 object can run three types of tests:

- On-demand tests – run by sending a command to the Self Test T25 object
- Start-up tests – run following a device reset
- Periodic tests – run at set intervals while the device is running to monitor the integrity of the device

These are described in the following sections.

C.1.1.1 On-demand Tests

The following tests can be run:

- Analog power test. This tests that AVdd power is present.
- Pin fault test. This tests the sense pins on the device. This allows low- resistance shorts (line-to-line, line-to-power and line- to- GND) to be detected, as well as resistive line-to-line shorts. This test also tests for shorts on the driven shield line.

NOTE	The pin fault test should not be confused with the open pin fault test that is run as part of the initial pin fault test on device reset (see Section C.1.1.2 "Initial Pin-Fault Test").

- PTC pin fault test. This tests the PTC pins on the device (where present) so that low- resistance shorts (line-to-line, line-to-power and line-to-GND) can be detected, as well as resistive line-to-line shorts.
- Signal limit tests. These test the signals from each of the touch objects on the device. These tests are run per touch object. An error indicates a fault with the sensor (for example, a broken line or similar).

To run a test, the CMD field is set to the code of the test to be run. The Self Test T25 object then immediately runs the test once and then stops. At the end of the test, the CMD field is cleared. If reporting is enabled, the Self Test T25 object also sends a report message with the result of the test.

If an error is found, the calibration command in the Command Processor T6 object should be used once the error has been cleared. Note that if the Analog Power Test fails, it will also be necessary to perform a power cycle on the product otherwise the device may still report a pin fault issue.

C.1.1.2 Initial Pin-Fault Test

In addition to the above tests, the device runs an initial pin fault test (followed by an optional open pin fault test).The initial pin fault test is run whenever the device is reset. The initial pin fault test runs a simple test to check whether any high voltage X drive lines are shorted to ground, another X line or any Y line. This detects shorts that would cause damage to the device if a high voltage is run through the X line.

If there is no problem and the initial pin fault test succeeds, the device continues to its acquisition and communications routines as normal. If the test fails, a message is sent but the device does not enter its acquisition routines. This allows the user to reset the device in the event of any problems during development. Note that the initial pin fault test is run, and the pass or failure message sent, *regardless of whether Self Test T25 is enabled or reporting is enabled*.

C.1.1.3 Periodic Tests

NOTE	Support for this feature depends on the firmware and so periodic tests may not be available on your device.

If your device allows it, the Self Test T25 object can perform a periodic test while the device is running to monitor the integrity of the device. The Self Test T25 object reports on success or failure of the test, or both, depending on its configuration.

The following periodic tests can be run:

- Periodic Power Test
- Signal Limit Test

See Section C.1.1.1 “On-demand Tests” for details on both these tests.

If reporting is enabled, the Self Test T25 object sends a report message with the result of the test.

In addition to reporting its results in the message data, the Periodic Test can also report its results by using a GPIO pin:

- If your device is configured to report either a healthy or failure report (but not both), the GPIO pin is pulsed if the test has detected that the device is healthy or faulty, as appropriate
- If both a healthy and failure report is requested, the GPIO is pulsed if the device is healthy and set to a continuous level if the device is faulty to distinguish between the two reports.

C.1.2 CONFIGURATION

TABLE 25: CONFIGURATION FOR SELF TEST T25

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	CTRL	Reserved						RPTEN	ENABLE
1	CMD	Test code of test to run							
2 – *n*	Reserved	All other bytes reserved							

Note: *n* = number of touch objects, assigned in the following order:
All Multiple Touch Touchscreen T100 objects
All Key Array T15 objects
All PTC Key Set T97 objects

CTRL Field

ENABLE: Enables the object. The object is enabled if set to 1, and disabled if set to 0.

RPTEN: Allows the object to send status messages to the host through the Message Processor T5 object. Reporting is enabled if set to 1, and disabled if set to 0.

CMD Field

This field is used to send test commands to the device. Valid test commands are listed in the table below.

TABLE 26: TEST COMMANDS

Command	Description
0x00	The CMD field is set to 0x00 after test completed
0x01	Performs an immediate check for the presence of the AVdd power line. Note that this test is also automatically run every 200 ms if the object is enabled.
0x12	Runs the pin fault test
0x17	Runs the signal limit test
0x18	Runs the PTC key pin fault test (if PTC keys are present on the device)
0xFE	Run all the tests in the order above, except for the open pin test.

C.1.3 MESSAGES

The Self Test T25 object reports the test results in its message data. The message data for the Self Test T25 object is shown in the following table. This can be read by viewing the contents of the Message Processor T5 object.

TABLE 27: MESSAGES FOR SELF TEST T25

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	Result code							
2 – 6	INFO	Result data							

STATUS Field

This field contains a result code that indicates the success or failure of the test. Valid codes are given in the following table.

TABLE 28: STATUS RESULT CODES

Code	Test Result
0xFE	All tests passed.
0xFD	The test code supplied in the CMD field is not associated with a valid test.
0xFC	The test could not be completed due to an unrelated fault (for example, an internal communications problem).
0x01	AVdd is not present. This failure is reported to the host every 200 ms.
0x12	The test failed because of a pin fault. The INFO fields indicate the first pin fault that was detected. Note that if the initial pin fault test fails, then the Self Test T25 object will generate a message with this result code on reset.
0x14	The test failed because of an open pin fault. Note that if the open pin fault test fails (run following the initial pin fault test), then the Self Test T25 object will generate a message with this result code on reset.
0x17	The test failed because of a signal limit fault.
0x18	The test failed because of a PTC pin fault. The INFO fields indicate the first pin fault that was detected.

INFO Field

This field contains the result data of the test. The actual data depends on which test was run, as detailed below. Note that if a test does not generate data, the INFO field consists solely of reserved bytes.

INFO Field – Analog Power Fault

The INFO field consists of reserved bytes only.

NOTE	If the Analog Power Test fails, it will be necessary to perform a power cycle on the product otherwise the device may still report a pin fault issue.

INFO Field – Pin Fault

If the result was a pin fault, the INFO field has the data format shown in the following table.

TABLE 29: TEST RESULT DATA FOR A PIN FAULT

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2	SEQ_NUM	Test sequence number							
3	X_PIN	Failing pin indication							
4	Y_PIN	Failing pin indication							

SEQ_NUM Field

The test sequence number of the test in which a fault is found. The sequence numbers and their meanings are listed in the table below.

TABLE 30: SEQUENCE NUMBERS FOR PIN FAULT TEST

Code	Test Result
0x01	**Driven Ground** This test detects shorts to a power rail (that is, they fail high). All pins are driven low to Ground. Note that the detection of X/Y shorts is dependent on the supply voltage levels.
0x02	**Driven High** This test detects shorts to Ground (that is, they fail low). This test can detect resistive shorts, but only up to ~200 kΩ. All pins are pulled high.
0x03	**Walking 1** This test is similar to test sequence 01, but uses the pull-up resistors to detect resistive shorts (but only up to ~200 kΩ). All the pins are set to zero. Then, starting with the first pin, each pin in turn is pulled high (set to 1), leaving all the other pins set to zero. The effect of this operation is that a 1 bit walks along the pins.
0x04	**Walking 0** This test is similar to test sequence 02, but uses the pull-up resistors to detect resistive shorts (but only up to ~200 kΩ). All the pins are set to 1. Then, starting with the first pin, each pin in turn is cleared (set to 0), leaving all the other pins set to 1. The effect of this operation is that a "0" bit "walks" along the pins.
0x07	**Initial High Voltage** This test detects an initial high voltage pin fault. Specifically, it detects shorts between high voltage drive lines and GND, X or Y lines that would damage the device if a high voltage acquisition were performed. All pins are driven low and then the high voltage drive lines are all pulled high one after another.

X_PIN and Y_PIN Fields

These fields indicate which pins have failed. A non zero value in X_PIN indicates the X line number plus 1 of a failed X sense pin. A non zero value in Y_PIN indicates the Y line number plus 1 of a failed Y sense pin. For example, if X5 has failed, X_PIN will read 6. Similarly, if Y3 has failed, Y_PIN will read 4.

If more than one pin fails, the test reports the first failed pin detected.

TABLE 31: PIN FAILURE RESULTS

Code		Test Result
X	Non zero	A Y sense pin or GKEYY pin (if present on the device) has failed. Y_PIN will be set to the number of the pin plus 1 (with Generic Key pins reported following the sense pins).
Non zero	X	An X sense pin or GKEYX pin (if present on the device) has failed. X_PIN will be set to the number of the pin plus 1 (with Generic Key pins reported following the sense pins).
0	0	If both X_PIN and Y_PIN are zero, the driven shield line has failed.

INFO Field – Open Pin Fault

If the result was an open pin fault following the initial pin fault test, the INFO field has the data format shown in the following table.

TABLE 32: TEST RESULT DATA FOR AN OPEN PIN FAULT

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2	SEQ_NUM	Test sequence number							
3	X_PIN	Failing pin indication							
4	Y_PIN	Failing pin indication							

SEQ_NUM Field

The test sequence number of the test in which a fault is found. This is always 0x01.

X_PIN and Y_PIN Fields

These fields indicate which pins have failed. See the table below for details. If more than one pin fails, the test reports the first failed pin detected.

TABLE 33: PIN FAILURE RESULTS

Code		Test Result
X	Non zero	A Y sense pin has failed. Y_PIN will be set to the number of the pin plus 1. For example, if Y3 has failed, Y_PIN will read 4.
Non zero	X	An X sense pin has failed. X_PIN will be set to the number of the pin plus 1. For example, if X5 has failed, X_PIN will read 6.
0	0	If both X_PIN and Y_PIN are zero, one of the following has failed if they are present on the device: • The Driven Shield line • A PTC Key line • A Generic Key line

INFO Field – Signal Limit Error

If the result was a signal limit error, the INFO field has the data format shown in the following table.

TABLE 34: TEST RESULT DATA FOR A SIGNAL LIMIT ERROR

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2	TYPE_NUM	Touch object type number							
3	TYPE_INSTANCE	Touch object type instance							

TYPE_NUM Field

The type number of the touch object for which a signal limit error was found.

TYPE_INSTANCE Field

The instance number of the touch object for which a signal limit error was found.

INFO Field – PTC Pin Fault

If the result was a PTC pin fault, the INFO field has the data format shown in the following table.

TABLE 35: TEST RESULT DATA FOR A PTC PIN FAULT

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2	SEQ_NUM	Test sequence number							
3	PTC_PIN	Failing pin indication							

SEQ_NUM Field

The test sequence number of the test in which a fault is found. The sequence numbers and their meanings are listed in the following table.

TABLE 36: SEQUENCE NUMBERS FOR PTC PIN FAULT TEST

Code	Test Result
0x01	**Driven Ground** This test detects shorts to a power rail (that is, they fail high). All pins are driven low to Ground. Note that the detection of X/Y shorts is dependent on the supply voltage levels.
0x02	**Driven High** This test detects shorts to Ground (that is, they fail low). All pins are pulled high.
0x03	**Walking 1** This test is similar to test sequence 01, but uses the pull-up resistors to detect resistive shorts (but only up to ~30 kΩ). All the pins are set to zero. Then, starting with the first pin, each pin in turn is pulled high (set to 1), leaving all the other pins set to zero. The effect of this operation is that a 1 bit walks along the pins.

PTC_PIN Field

This field indicates the pin that has failed. PTC_PIN will be set to the number of the PTCXY pin plus 1. If more than one pin fails, the test reports the first failed pin detected.

APPENDIX D: HANDLING CONFIGURATIONS

D.1 Configuration File concept

maXTouch devices are highly customizable to meet different customer needs. Configuration data is held in a `.xcfg` file and can be read from or written to the chip NVM to modify its behavior in order to address differing application requirements. Details of the tuning process are beyond the scope of this document, but it may be that customer devices need to upload a new `.xcfg` file.

D.2 Uploading and Backing Up Configurations

The `.xcfg` data format is text-based and provides a simple mapping to the individual object table elements. Refer to the following document for more information:

- Application Note: MXT0201 – *Object Server Configuration File Format*

After parsing the information block and object table, as described earlier in this document, the new `.xcfg` data values are simply written byte-for-byte to the corresponding object elements (in increasing address order) in the object table memory.

Once the write is complete, you will need to execute a Command Processor T6 BACKUP command to flash the modified parameters into the maXTouch non-volatile memory (NVM).

D.3 Downloading Configurations

Configuration data can also be read back via the I^2C bus and a `.xcfg` file reconstructed from it (refer to MXT0201 for details). The `.xcfg` file is in a text-based format so any commercially-available file-compare utility can be used to check whether the new `.xcfg` file has been uploaded correctly.

APPENDIX E: REVISION HISTORY

Revision A (October 2015) – Atmel edition

Final Atmel revision

Revision A (April 2017) – Microchip edition

Reformatted and updated – Microchip edition

This revision incorporates the following updates:

- Updated to Microchip application note format:
  - Revision History moved to this appendix
  - Back cover updated
- Changes to content:
  - Appendix C: Self Test T25 object updated to include periodic tests
- New documentation number assigned

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949

Trademarks

ISBN: 978-1-5224-1576-3

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110

Canada - Toronto
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon

Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Dongguan
Tel: 86-769-8702-9880

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

ASIA/PACIFIC

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7828

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Dusseldorf
Tel: 49-2129-3766400

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Venice
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Poland - Warsaw
Tel: 48-22-3325737

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820