MCU Driver For Maxtouch

A Leading Provider of Smart, Connected and Secure Embedded Control Solutions

Rocup Wan

03/25/2022

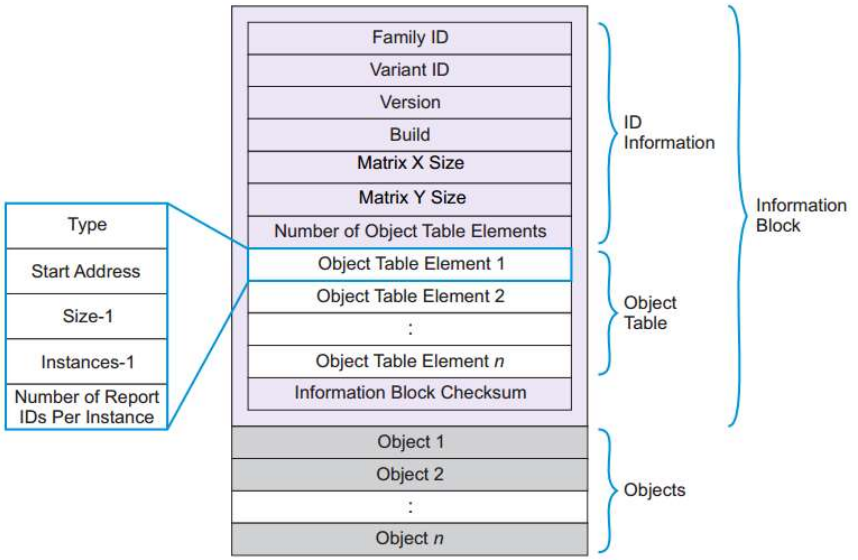
Content

- Maxtouch & MPTT Protocol
- MCU Driver Flow
- Importing Note
- Debugging Note

MICROCHIP

What is MPTT Protocol

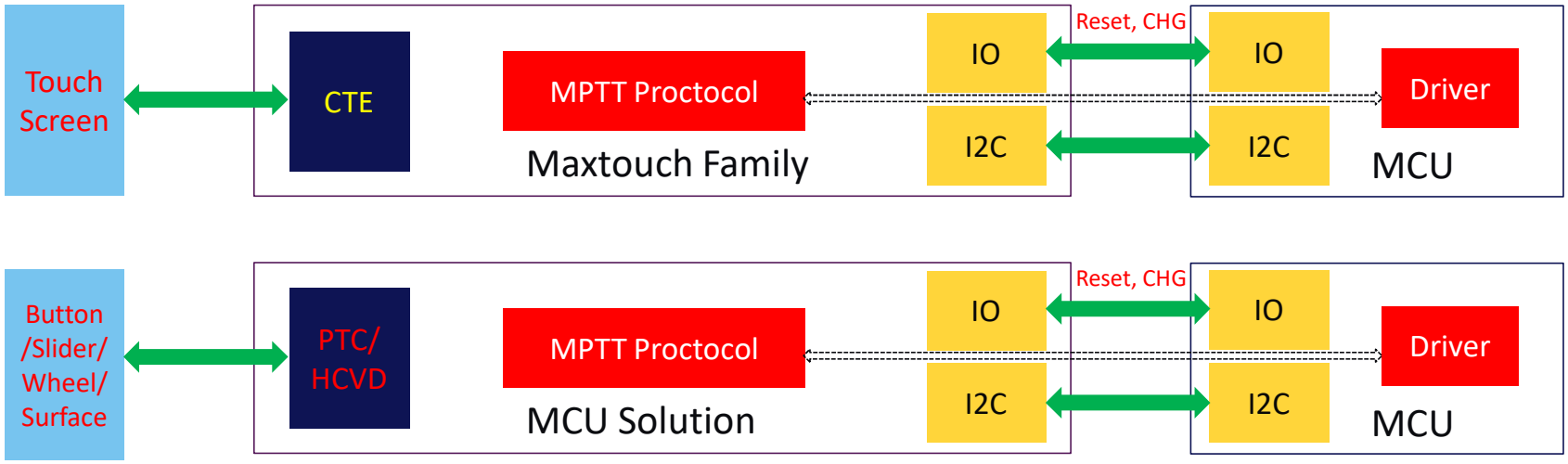
- The protocol is designed to control the processing chain in a modular manner.
- This is achieved by breaking the features of the device into objects that can be controlled individually.
- Each object represents a certain feature or function of the device, such as a touchscreen.



MICROCHIP

Maxtouch and MPTT Protocol

- **The driver for Maxtouch can also be used to all the MCU based solution with MPTT protocol imported.**



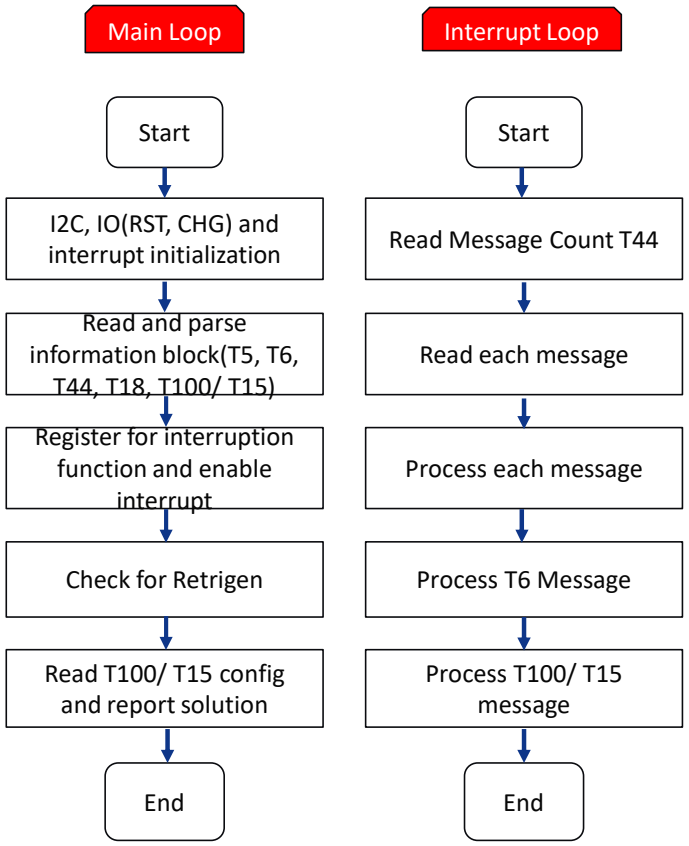
MICROCHIP

MCU Driver Basic Flow

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define TOUCH_KEY           // used for touch keys

// used for touch screen, if there're touch keys only, the macro can be disabled
#define TOUCH_OBJECT
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read T100/ T15 config
- Read and process messages



MICROCHIP

POR Sequence

In POR sequence, Reset line **must** keep low level(< 0.3 * VDDIO) until VDD reached valid voltage (~3.3v, see minimum VDD value in datasheet) and delay 90ns.

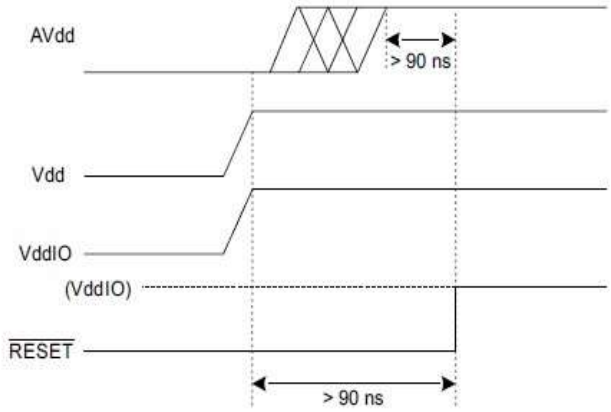
Power-on Reset

There is an internal Power-on Reset (POR) in the device.

If an external reset is to be used the device must be held in $\overline{\text{RESET}}$ (active low) while the digital (Vdd), analog (AVdd) and digital I/O (VddIO) power supplies are powering up. The supplies must have reached their nominal values before the $\overline{\text{RESET}}$ signal is deasserted (that is, goes high). This is shown in Figure 5-1. See Section 11.2 "Recommended Operating Conditions" for nominal values for the power supplies to the device.

A diode from AVDD to VDD is present in the device. If AVDD and VDD are driven from different supplies, the Vdd supply must be powered up earlier than AVdd.

FIGURE 5-1: POWER SEQUENCING ON THE MXT336UD-MAU001



Note: When using external $\overline{\text{RESET}}$ at power-up, VddIO must not be enabled after Vdd

It is recommended that customer designs include the capability for the host to control all the maXTouch power supplies and pull the $\overline{\text{RESET}}$ line low.

After power-up, the device typically takes 122 ms to 362 ms before it is ready to start communications, depending on the configuration.

MICROCHIP

I2C Specification

1. Clock Stretching must be supported by host controller.

2. I2C wave should match open document --- “I2C bus specification and user manual” by NXP.

7.5 SDA and SCL

The I^2C 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^2C device is pulling it down.

The termination resistors should be chosen so that the rise times on SDA and SCL meet the I^2C 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^2C bus with the maXTouch controller.

7.6 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 2 ms and typically less than 350 µs.

11.9 I^2C Specification

Parameter	Value
Address	0x4A
I^2C specification (1)	Revision 6.0
Maximum bus speed (SCL) (2)	1 MHz
Standard Mode (3)	100 kHz
Fast Mode (3)	400 kHz
Fast Mode Plus (3)	1 MHz

Note
1: More detailed information on I^2C operation is available from www.nxp.com/documents/user_manual/UM10204.pdf.
2: In systems with heavily laden I^2C lines, even with minimum pull-up resistor values, bus speed may be limited by capacitive loading to less than the theoretical maximum.
3: The values of pull-up resistors should be chosen to ensure SCL and SDA rise and fall times meet the I^2C specification. The value required will depend on the amount of capacitance loading on the lines.

MICROCHIP

I2C Specification - Read

1. The read address is 16 bit.

2. The read sequence must be complied with the datasheet.

8.4 Reading From the Device

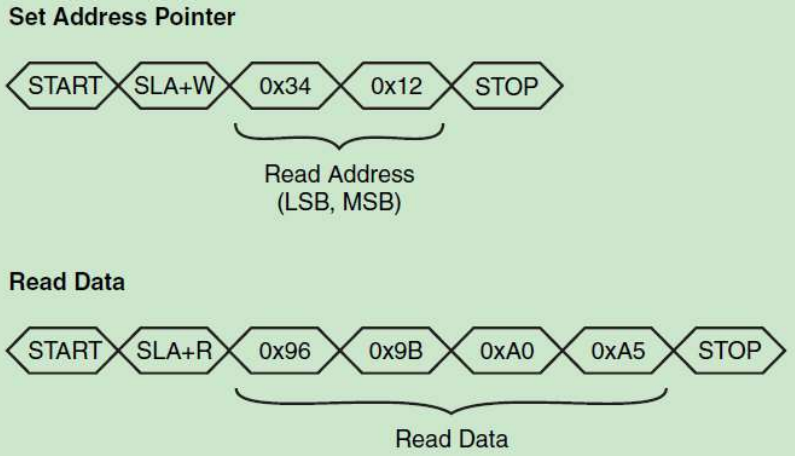
Two I^2C bus activities must take place to read from the device. The first activity is an I^2C write to set the address pointer (LSByte then MSByte). The second activity is the actual I^2C read to receive the data. The address pointer returns to its starting value when the read cycle NACK is detected.

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 reads (see Section 8.5 "Reading 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, calculating a checksum of the data packets is not supported.

Figure 8-3 shows the I^2C commands to read four bytes starting at address 0x1234.

FIGURE 8-3: EXAMPLE OF A FOUR-BYTE READ STARTING AT ADDRESS 0x1234



MICROCHIP

I2C Specification - Write

1. The write address is 16 bit.

2. The write sequence must be complied with the datasheet.

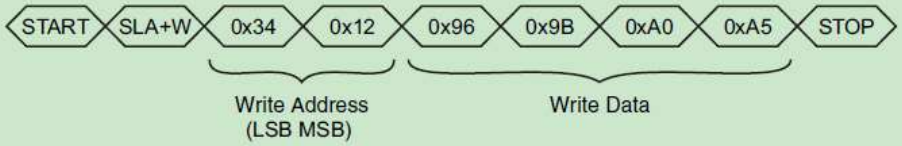
8.2 Writing To the Device

A WRITE cycle to the device consists of a START condition followed by the I^2C address of the device (SLA+W). 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.

Figure 8-1 shows an example of writing four bytes of data to contiguous addresses starting at 0x1234.

FIGURE 8-1: EXAMPLE OF A FOUR-BYTE WRITE STARTING AT ADDRESS 0x1234



Interrupt Assert

Interrupt line is asserting by **low level**. That means if there is any message put in T5 message FIFO (or T44 is not Zero) the Interrupt is asserted.

When host acknowledge the interrupt asserting, the host will initialize the I2C transmitting. In I2C transmitting interval, the CHG could de-assert(Mode 0) and keep asserted(Mode 1), And then decide the next round level by Stop(Mode 0) or last message in T5 FIFO (Mode 1).

(Actual it's not so important to select Mode 0 or 1, we use Mode 0 as default)

For host triggering mode of interrupt handler, we should use Low level to trigger it.

But how about if host want to set **Falling Edge** to trigger interrupt handler: The chip must be configured as **Re-trigger** mode in T18 (see next page)

$\overline{\text{CHG}}$ 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 I^2C communications.

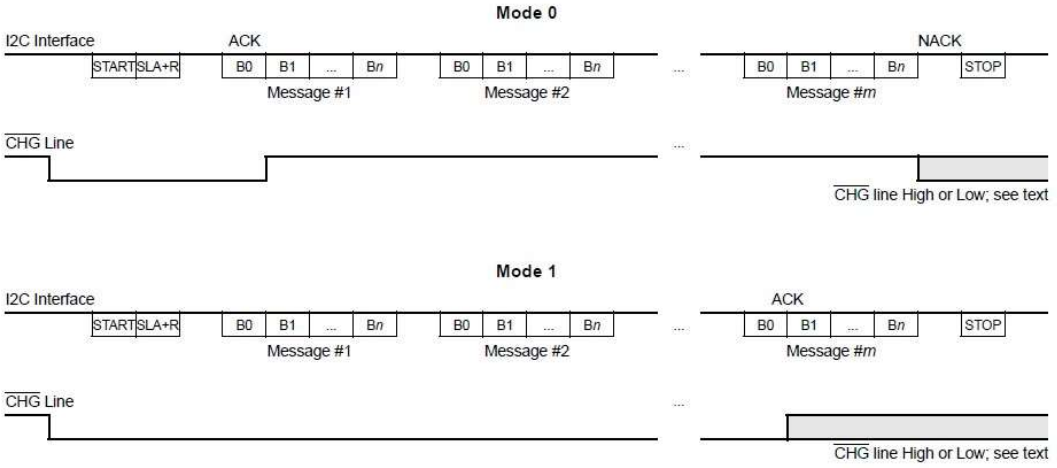
NOTE	The host should always use the $\overline{\text{CHG}}$ line as an indication that a message is ready to be read from the Message Processor T5 object; the host should never poll the device for messages.

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 (see Section 5.0 "Power-up / Reset Requirements").

A pull-up resistor is required to VddIO (see Section 2.0 "Schematic").

The $\overline{\text{CHG}}$ line operates in two modes when it is used with I^2C communications, as defined by the Communications Configuration T18 object.

FIGURE 7-5: $\overline{\text{CHG}}$ LINE MODES FOR I^2C-COMPATIBLE TRANSFERS



Retrigen

When enabled the Retrigen mode, CHG will deassert as each acquisition cycle with 50us interval (high level pulse) .

The T7 will control the acquisition cycle, so decides the Retrigger frequency.

Note, The interrupt is asserted if any message is in T5, the message is not only meaningful for the touch event occurred(T15/100), but also same meaningful for any other object. Specially, when POR completed, there are `reset message` come out for T6. So, when chip boots up normally, the interrupt will toggle to low before your readout the message.

TABLE 6-19: 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	Reserved	RETRIGEN	Reserved		HISPEEDSPI	MODE	Reserved	
1	COMMAND	CHG line command code							

CTRL Field

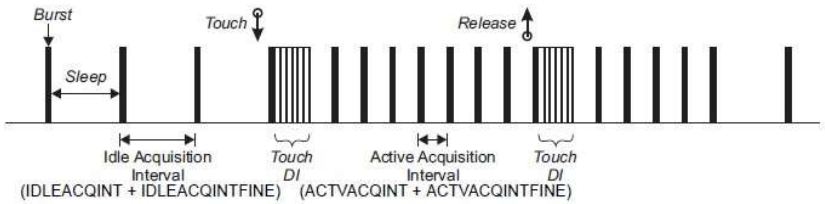
MODE: Selects the CHG line mode for I^2C communications over the I^2C interface. If this bit is set to 0 (the default), the CHG line operates in Mode 0. If this bit is set to 1, the CHG line operates in Mode 1. Refer to the relevant device *Datasheet* for more information on the CHG line modes.

HISPEEDSPI: If this bit is set to 1, debug data is sent over High Speed SPI. If this bit is set to 0 (default), debug data is sent over Normal Speed SPI (the normal speed for Microchip touchscreen products).

RETRIGEN: Enables CHG line host retriggering mode. This provides extra edge-based interrupts to the host on the CHG line. This mode is enabled if this bit is set to 1 and disabled if set to 0. If this mode is enabled, the CHG line is deasserted once every acquisition cycle for 50 µs and then re-asserted. This creates edges on the CHG line for the host in case it missed the CHG line being asserted.

This re-triggering mechanism is performed once every acquisition cycle, if the time elapsed since last time the CHG line was asserted is greater than 10 ms. This means that, if the configured acquisition interval is less than 10 ms then one or more extra acquisition cycles will be necessary to lapse for the CHG line to be re-triggered, as determined by Power Configuration T7 IDLEACQINT and ACTVACQINT (see "IDLEACQINT and ACTVACQINT Fields").

Note that if command code 2 or 3 is set in the COMMAND field, then the command will override the RETRIGEN mode setting.



Driver initialization

MCU initialization contains:

1. Reset pin set to output high.
2. CHG pin set as input interrupt.
3. I2C pin and module initialization.

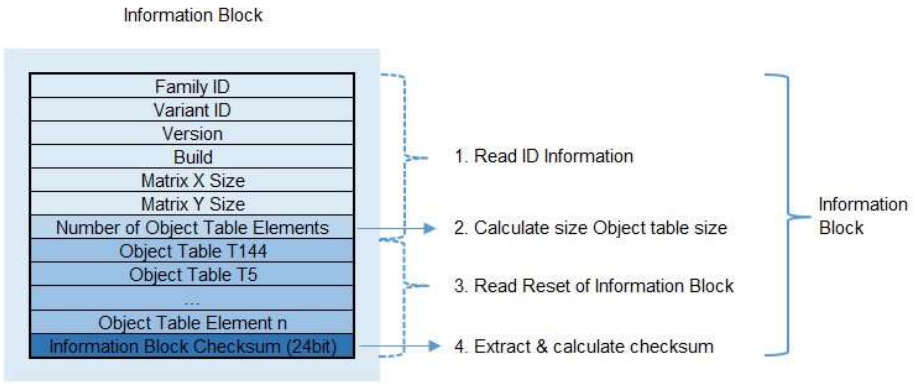
```
void system_init(void)
{
    init_mcu();

    // GPIO on PA19

    gpio_set_pin_level(RST,
                       // <y> Initial level
                       // <id> pad_initial_level
                       // <false"> Low
                       // <true"> High
                       true);

    // Set pin direction to output
    gpio_set_pin_direction(RST, GPIO_DIRECTION_OUT);

    gpio_set_pin_function(RST, GPIO_PIN_FUNCTION_OFF);

    EXTERNAL_IRQ_0_init();

    TIMER_0_init();

    USART_Dbg_init();

    I2C_Mxt_init();
}
```

Read and parse information block

```
static int mxt_read_info_block(struct mxt_data *data)
```

Read out the Information block:

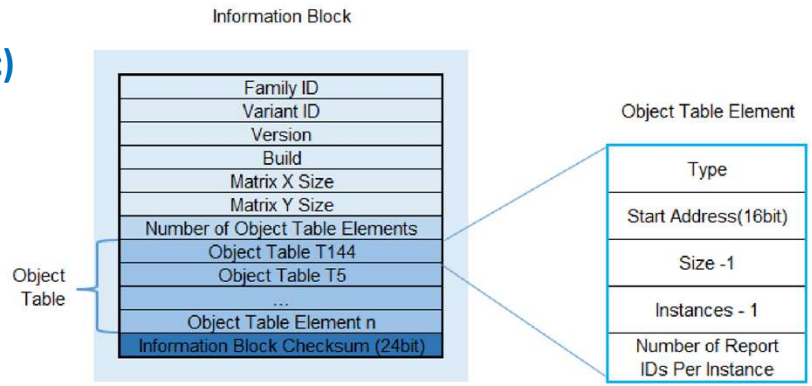
1. Read 7-bytes ID information block starting at address 0
2. Calculate the Information block size: EachElementSize * NumObjects + CrcSize
3. Read rest of info block after id block with offset 7
4. Extract & calculate checksum




```
static int mxt_parse_object_table(struct mxt_data *data, struct mxt_object *object, u8 *ptr_id)
```

Parse the Object Table (Only T5, T6, T7, T18, T100/ T15 for basic)

1. Valid Report IDs start counting from 1
2. Iterate each item in Object Table, record the address, size, and calculate the accumulated Report ID
3. The Report accumulated by: Last RID + NumInstance * NumIDsPerInstance



Register interruption

We need register the interrupt function.
Here the interrupt function is: mxt_interrupt()

The Interrupt trigger mode is Low level and this is initialized in the driver initialization phase.

If we want to register the trigger mode as `TRIGGER_FALLING`, be careful we should enable the `Retrigen` bit in the T18 config file.

```
static int mxt_acquire_irq(struct mxt_data *data)
{
    if (!data->irq) {
        register_mxt_irq(&data->irq, mxt_interrupt);
    }

    //enable_irq(data->irq);

    return 0;
}
```

```
void register_mxt_irq(unsigned int *irq, ext_irq_cb_t mxt_interrupt)
{
    ext_irq_register(PIN_PB05, mxt_interrupt);

    *irq = PIN_PB05;
}
```


MICROCHIP

Check for Retrigen

It’s necessary to check for retrigen.

1. If the interrupt is low level retrigen, no need to check.
2. If the interrupt is not low level retrigen, the status of retrigen should be checked.
3. The second case with retrigen disabled will use workaround to read all the possible messages after power on in case that the edge was missing.

```
static int mxt_check_retrigen(struct mxt_data *data)
{
    int error;
    u8 val;
#ifdef SECURITY...
#endif

    /* If it's low level trigger, it will not need retrigger */
    if (check_low_level_trigger() == true) {
        dev_info("Level triggered\n");
        return 0;
    }

    if (data->T18_address) {
         error = mxt_read_reg(data->T18_address + MXT_COMMS_CTRL, 1, &val, data);
        if (error)
            return error;

        if (!(val & MXT_COMMS_RETRIGEN)) {/* wrong in the standard driver ????? */
            dev_info("RETRIGEN enabled\n");
            return 0;
        }
    }
```


MICROCHIP

Read T100/ T15 config

Read T100 Config:
Xmax, Ymax and XY switch (direction).
Read T15 Config:
All the keys' number and instance 0 key number.

```
static int mxt_read_t15_num_keys(struct mxt_data *data)
{
    u8 xsize, ysize;
    u16 num_keys = 0, offset = 0, instance;
    u8 T15_enable;
    int error = 0;

    // maximum instance number is 2
    for (instance = 0; instance < MXT_T15_MAX_INS; instance++) {
        offset = data->T15_size * instance;
        error = mxt_read_reg(data->T15_address + offset + MXT_T15_CTRL,
                             sizeof(T15_enable), &T15_enable, data);
        if (error) {
            dev_err("read T15 instance(%d) CTRL failed\n", instance);
            return error;
        }

        if ((T15_enable & MXT_T15_ENABLE_BIT_MASK) != 0x01 ){
            dev_info("T15 instance(%d) input device not enabled\n", instance);
        } else {
            /* read first T15 size */
            error = mxt_read_reg(data->T15_address + offset + MXT_T15_XSIZE,
                                 sizeof(xsize), &xsize, data);
            if (error) {
```

```
static int mxt_read_t100_config(struct mxt_data *data, u8 instance)
{
    int error;
    u16 range_x, range_y;
    u8 cfg;
    u16 obj_size = 0;

#ifdef SECOND_TOUCH...
#endif

    error = mxt_read_reg(data->tch_obj_info.address.T100 + obj_size + MXT_T100_XRANGE,
                         sizeof(range_x), &range_x, data);
    if (error)
        return error;

    error = mxt_read_reg(data->tch_obj_info.address.T100 + obj_size + MXT_T100_YRANGE,
                         sizeof(range_y), &range_y, data);
    if (error)
        return error;

    error =  mxt_read_reg(data->tch_obj_info.address.T100 + obj_size + MXT_T100_CFG1,
                          1, &cfg, data);
    if (error)
```

Read and process messages

We put mxt_process_messages_t44_t144() to handle the message.

1. Read T44/ T144 message count
2. Loop reading the message from T5 FIFO

mxt_read_and_process_messages()

3. If the RID (message byte[0]) is in target Report IDs range, then start to extract touch information:
    - We call mxt_proc_t<n>_message() to handle the extraction

```
static void mxt_interrupt(void)
{
    struct mxt_data *data = &mxtData;

    data->chg_completion = 1;

    dev_dbg("interrupt\n");

#ifdef BOOTLOADER_PROCESS
    if (data->in_bootloader)
        return;
#endif

#ifndef T44_NONE
    if (data->count_address.T44) {
        return mxt_process_messages_t44_t144(data);
    }
```

```
static int mxt_proc_message(struct mxt_data *data, u8 *message)
{
    u8 report_id = message[0];

    if (report_id == MXT_RPTID_NOMSG
#ifdef SECOND_TOUCH...
#endif
    )
        return 0;

    if (report_id == data->T6_reportid) {
        mxt_proc_t6_messages(data, message);

#ifdef TOUCH_OBJECT

#ifdef T9_OBJECT
    } else if (report_id >= data->tch_obj_info.reportid_min.T9 &&
            report_id <= data->tch_obj_info.reportid_max.T9) {
        mxt_proc_t9_message(data, message);
#else
    } else if (report_id >= data->tch_obj_info.reportid_min.T100 &&
            report_id <= data->tch_obj_info.reportid_max.T100) {
        mxt_proc_t100_message(data, message);
```


MICROCHIP

Process T100 messages

We should lookup the protocol about the T100 message format, that you could call Microchip support or download the MXTAN0213 document directly.

Now we got the T100 message format,
The 1st Report ID of T100 is Screen Status Message
The 2nd Report ID of T100 is Reserved
From 3rd Report ID to maximum of this instance, that's the finger tracking ID, from 1 to maximum.

So we will ignore 1st and 2nd report ID, directly decode from 3rd IDs.

TABLE 4-20: MESSAGE DATA FOR MULTIPLE TOUCH TOUCHSCREEN T100 (TOUCH_MULTITOUCHSCREEN_T100)

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							

- Get TCHSTATUS BIT[7] to decide whether a touch detected
- Get out X/Y position
- Report the even

```
static void mxt_proc_t100_message(struct mxt_data *data, u8 *message)
{
    int id = 0;
    u8 status;
    u8 type = 0;
    u8 event = 0;
    u16 x;
    u16 y;
    bool active = false;

#ifdef SECOND_TOUCH...
#endif

    id = message[0] - data->tch_obj_info.reportid_min.T100 - MXT_RSVD_RPTIDS;

    /* ignore SCRSTATUS events */
    if (id < 0)
        return;

#ifdef SECOND_TOUCH...
#endif

    status = message[1];
    event = status & MXT_T100_EVENT_MASK;
    x = message[2] + (message[3] << 8);
```

The message extracting is finished, all the things down.
We are just waiting for the interrupt occurred now!

MICROCHIP

Process T15 messages

We will lookup the protocol about the T15 message format, that you could call Microchip support or download the MXTAN0213 document directly.

All Keys' event in one instance share one dedicated Report ID. Each instance could support up to 16 keys. We see the format as below:

TABLE 4-7: 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

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.

- Byte[1]: the whole status of any key is detected.
- Byte[2~3]: indicated which key is pressed.

```
static void mxt_proc_t15_messages(struct mxt_data *data, u8 *msg)
{
    int key;
    bool curr_state, new_state;
    unsigned long keystates = msg[2] + (msg[3] << 8);
    int id = msg[0] - data->T15_reportid_min;

    if(id) // the second instance will shift the key states with t
        keystates <<= data->t15_num_keys_inst0;

    for (key = 0; key < data->t15_num_keys_all; key++) {
        curr_state = TEST_BIT(key, &data->t15_keystatus);
        new_state = TEST_BIT(key, &keystates);

        if (!curr_state && new_state) {
            dev_dbg("T15 key press: %u\n", key);
            data->t15_keystatus |= (1U << key);

            report_key(data->t15_keystatus, 1);

        } else if (curr_state && !new_state) {
            dev_dbg("T15 key release: %u\n", key);
            data->t15_keystatus &= ~(1U << key);
```

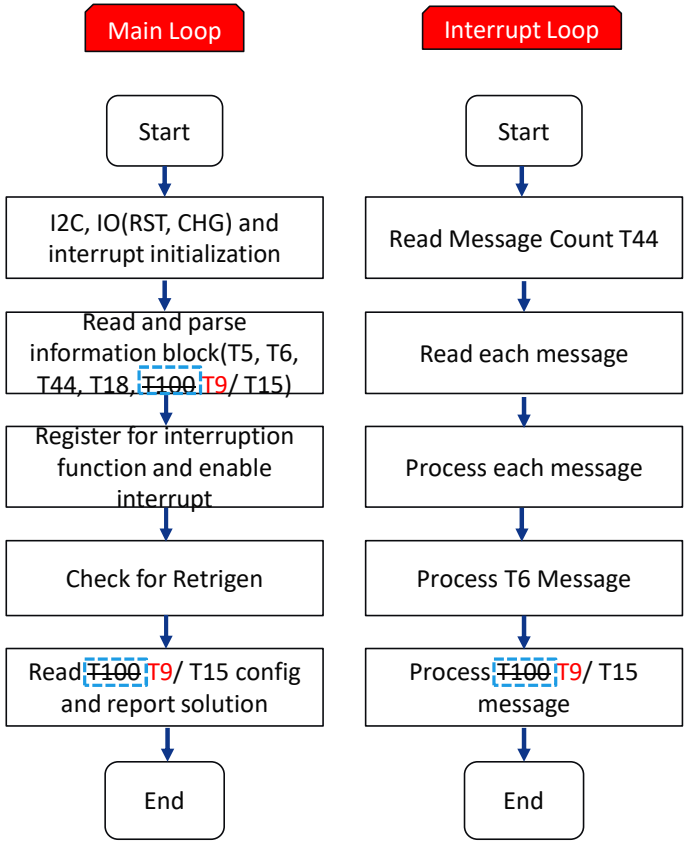

MICROCHIP

MCU Driver Flow With Old Series Maxtouch

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define T9_OBJECT           // used for old IC using T9 for touch screen reporting points
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read ~~T100~~ T9/ T15 config
- Read and process messages



MICROCHIP

Process T9 messages

T9 is the legacy touch screen object used with special purpose(Before S series,
MPTT), we might use it in some condition and wrote the code here

Now we got the T9 message format, Each Report ID means a finger tracking ID.

Table 5-6. Message Data for TOUCH_MULTITOUCHSCREEN_T9

Byte	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	STATUS	DETECT	PRESS	RELEASE	MOVE	VECTOR	AMP	SUPPRESS	UNGRIP
2	XPOSMSB	X position MSByte							
3	YPOSMSB	Y position MSByte							
4	XYPOSLSB	X position lsbits				Y position lsbits			
5	TCHAREA	Size of touch							
6	TCHAMPLITUDE	Touch amplitude (sum of measured deltas)							
7	TCHVECTOR	Component 1				Component 2			

Note: The format for the XYPOSLSB fields depend on the resolution (10-bit or 12-bit); see page 34.

- Get STATUS BIT[7] to decide whether a touch detected
- Get out X/Y position
- Report the even

Note for different resolution of X/Y configured, the X/Y position may use different LSB decode:

Table 5-7. X 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
10-bit Format															
512	256	128	64	32	16	8	4	2	1	N/A		Y position lsbits			
12-bit Format															
2048	1024	512	256	128	64	32	16	8	4	2	1	Y position lsbits			

Table 5-8. Y Position Formats

YPOSMSB								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
10-bit Format															
512	256	128	64	32	16	8	4	X position lsbits					2	1	N/A
12-bit Format															
2048	1024	512	256	128	64	32	16	X position lsbits					8	4	2	1

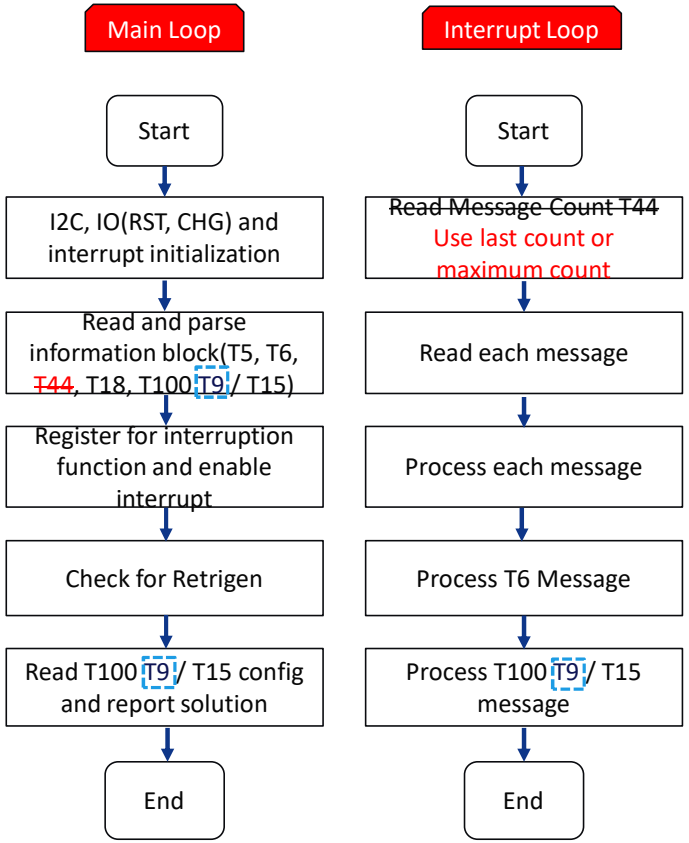
```
static void mxt_proc_t9_message(struct mxt_data *data, u8 *message)
{
    int id;
    int x;
    int y;
    u8 status;
    u8 type = 0;

    id = message[0] - data->tch_obj_info.reportid_min.T9;
    status = message[1];
    x = (message[2] << 4) | ((message[4] >> 4) & 0xf);
    y = (message[3] << 4) | ((message[4] & 0xf));

    /* Handle 10/12 bit switching */
    if (data->max_x < 1024)
        x >>= 2;
    if (data->max_y < 1024)
        y >>= 2;
```

MCU Driver Flow With Maxtouch without T44

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define T44_NONE              // used for old IC without T44
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config
- Read and process messages (read last count number of messages)



Read and process messages

We put mxt_process_messages() to handle the message.

1. Use last message count.
2. Loop reading the message from T5 FIFO mxt_read_and_process_messages()
3. If the RID (message byte[0]) is in target Report IDs range, then start to extract touch information:
    - We call mxt_proc_t<n>_message() to handle the extraction

```
#ifdef T44_NONE
static void mxt_process_messages(struct mxt_data *data)
{
    int total_handled, num_handled;
    u8 count = data->last_message_count;

    if (count < 1 || count > data->max_reportid)
        count = 1;

    /* include final invalid message */
    total_handled = mxt_read_and_process_messages(data, count + 1);
    if (total_handled < 0)
        return;
    /* if there were invalid messages, then we are done */
    else if (total_handled <= count)
        goto update_count;

    /* keep reading two msgs until one is invalid or reportid limit */
    do {
        num_handled = mxt_read_and_process_messages(data, 2);
        if (num_handled < 0)
            return;

        total_handled += num_handled;
```

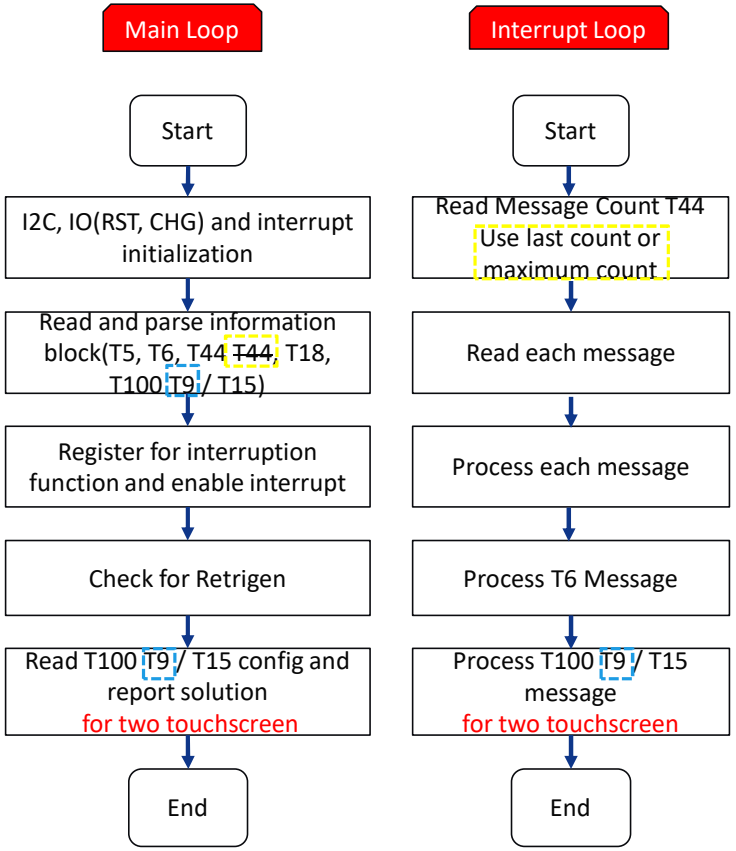

MICROCHIP

MCU Driver Flow With Second Touch

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define SECOND_TOUCH        // used for the second touch instance
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config for two touchscreen
- Read and process messages ([read last count number of messages] T100 report two touchscreen)



MICROCHIP

Read T100 config for two touchscreen

Read T100 Config for two touchscreen:
Xmax, Ymax and XY switch
for each touchsreen

```
static int mxt_initialize_input_device(struct mxt_data *data, bool primary)
{
    int error = 0;

#ifdef T9_OBJECT
    error = mxt_read_t9_resolution(data);
    if (error)
        dev_warn("Failed to initialize T9 resolution\n");
#else
    if (primary) {
        error = mxt_read_t100_config(data, 1);
    }
#ifdef SECOND_TOUCH
    else {
        error = mxt_read_t100_config(data, 2);
    }
#endif
    if (error)
        dev_warn("Failed to read T100 config\n");
#endif

    report_resolution(data->max_x, data->max_y);

    return error;
}
```

```
static int mxt_read_t100_config(struct mxt_data *data, u8 instance)
{
    int error;
    u16 range_x, range_y;
    u8 cfg;
    u16 obj_size = 0;

#ifdef SECOND_TOUCH...
#endif

    error = mxt_read_reg(data->tch_obj_info.address.T100 + obj_size + MXT_T100_XRANGE,
                         sizeof(range_x), &range_x, data);
    if (error)
        return error;

    error = mxt_read_reg(data->tch_obj_info.address.T100 + obj_size + MXT_T100_YRANGE,
                         sizeof(range_y), &range_y, data);
    if (error)
        return error;

    error =  mxt_read_reg(data->tch_obj_info.address.T100 + obj_size + MXT_T100_CFG1,
                          1, &cfg, data);
    if (error)
```


MICROCHIP

Process T100 messages for two touchsreen

We will parse the messages to get the coordinate of two touch screens.

And the report will use ID to distinguish which screen to report.

```
#ifdef SECOND_TOUCH
        if (id >= MXT_MIN_RPTID_SEC) {
            /* report type event and coordinate */
            report_state(type, event);
            report_coordinate(x, y);

            if (id == MXT_MIN_RPTID_SEC)
            {
                report_single_touch();
            }
        } else
#endif
        {
            /* report type event and coordinate */
            report_state(type, event);
            report_coordinate(x, y);
        }
    } else {
        /* report type and event */
    #ifdef SECOND_TOUCH

        if (id >= MXT_MIN_RPTID_SEC) {
```

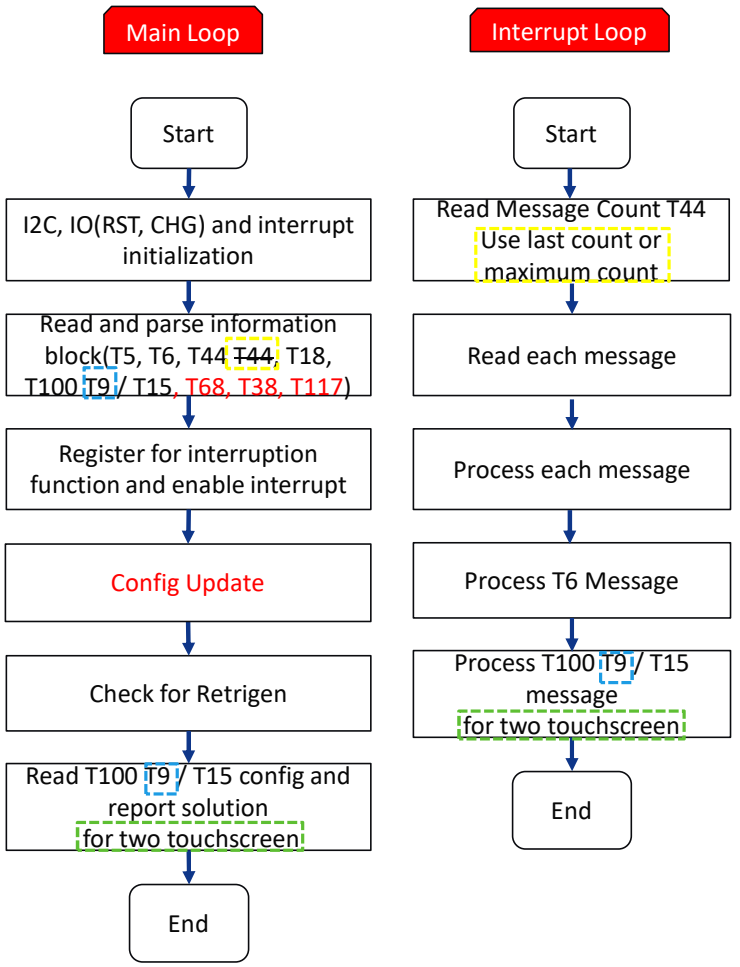

MICROCHIP

MCU Driver Flow With Config Update

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define CONFIG_UPGRADE      // used for upgrade config
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config [for two touchscreen]
- Read and process messages ([read last count number of messages] [T100 report two touchscreen])
- Config Update



Config Update

- In mxt_initialize(), there is default calling of the config update with the length of the config data array that's inverted from raw config file, the `mxt_update_cfg()` is called and the chip will be flashed with the new config data array.

```
#ifdef CONFIG_UPGRADE
    if (data->cfg_length) {
        error = mxt_update_cfg(data);
        if (error)
            dev_warn("Error %d updating config\n", error);
    }
#endif
```

```
/* write config to ram and calculate the checksum */
for (int i = 0; i < data->cfg_length; ) {

    if (cfg[i] == 117) {            /* T117 may be the first object */
        addr = data->T117_address;
    } else if (cfg[i] == 68) {      /* T68 may be the first object */
        addr = data->T68_address;
    } else if (cfg[i] == 38) {      /* T38 may be the first object */
        addr = data->T38_address;
    }    /* other objects' address is following T38's address in sequence */

    type = cfg[i++];
    i++;   /* skip the instance byte */

    size = cfg[i++];

    writed_len = 0;
    do {// if more than MXT_MAX_BLOCK_WRITE bytes, write MXT_MAX_BLOCK_WRITE bytes one time
        if ((size - writed_len) >= MXT_MAX_BLOCK_WRITE)
            writing_len = MXT_MAX_BLOCK_WRITE;
        else
            writing_len = size - writed_len;

        error = mxt_write_reg(addr + writed_len, writing_len, &cfg[i + writed_len], data);
```

```
uint8_t file_device_info[] = {0xA4, 0x19, 0x10, 0xAA, 0x20, 0x14, 0x28};
uint32_t file_block_info_crc = 0xE356C3;
uint32_t file_cfg_crc = 0x82A64E;
uint8_t file_cfg_data[] = {
    0x0044, 0x0000, 0x0049, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x0026, 0x0000, 0x0040, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,
    0x0047, 0x0000, 0x00C8, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
    0x006E, 0x0000, 0x0028, 0x11, 0x00, 0x59, 0x07, 0x51, 0x09, 0x19, 0x09,
```

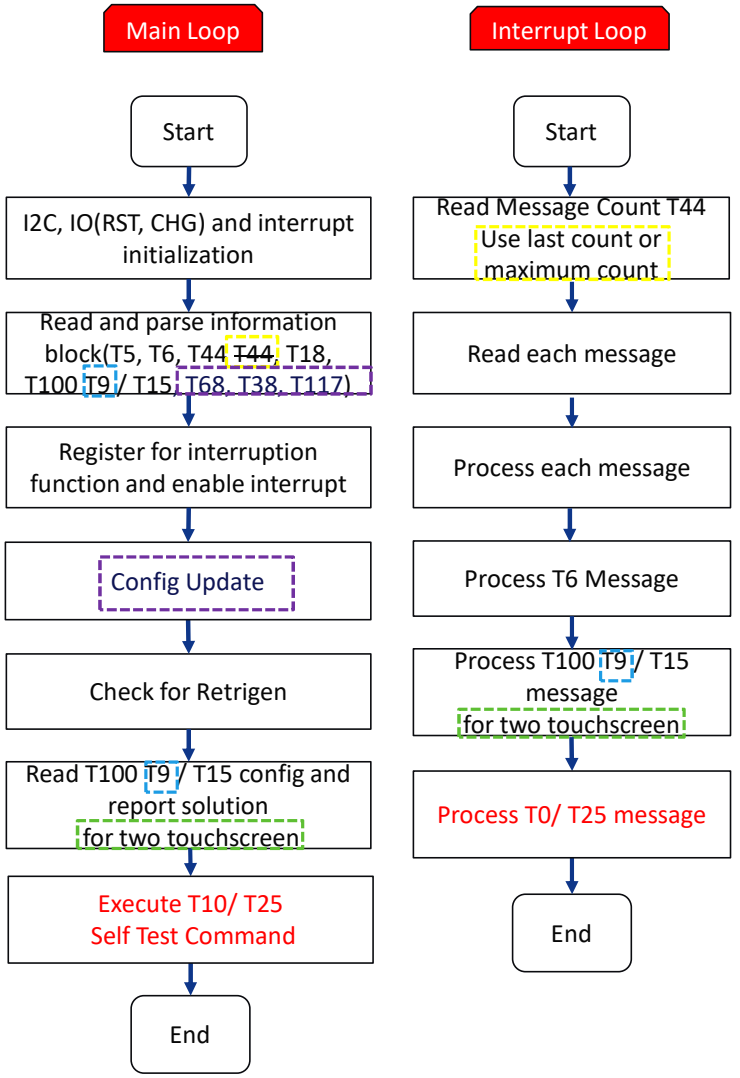

MICROCHIP

MCU Driver Flow With Self Test

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
// used for self test in the host including pin fault, signal limit and avdd present
// T10 and T25 are all implemented in code and will be selected by the object information,
// but you can select one of them according to your IC type to save code size
//#define SELF_TEST
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config [for two touchscreen]
- Read and process messages ([read last count number of messages] [T100 report two touchscreen])
- Config Update
- Execute T10/ T25 Self Test Command
- Process T0/ T25 message



Execute T10/ T25 Self Test Command

We have T25/T10 selftest interface of `selftest`:

- It will accept a test command to execute the test and feedback result by readout the same interface.
- The cmd input is set as below for T25(Non-HA) or T10(HA):
- We should set all the appropriate parameter in chip config before issue the test command.
- Please refer the protocol for more test details

Command of T25 (Normal we will issue `0xFE` for the command):

TABLE 6-11: TEST COMMANDS

Command	Description
0x00	The CMD field is set to 0x00 after test completed
0x01	Test for AVdd power present
0x12	Run the pin fault test
0x17	Run the signal limit test
0xFE	Run all the tests

Command of T10 (Normal we will issue `0x3E` for the command):

TABLE 6-4: TEST COMMANDS

Command	Description
0x00	No command
0x31	Run the Power test
0X32	Run the pin fault test
0x33	Run the signal limit test
0x3E	Run all the tests in the order above
All other values	Reserved; no effect

MICROCHIP

Process T10/ T25 message

- Result of T25 (We will get 0xFE if passed) :

TABLE 6-15: 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.
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 (see Table 6-16). 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.
0x17	The test failed because of a signal limit fault.

- Result of T10 (We will get 0x31 if passed):

TABLE 6-6: STATUS FIELD

Result Code	Meaning	INFO[] Field Data
0x31	All on-demand tests have passed	Not used; the INFO field consists of reserved bytes only
0x32	An on demand test has failed	See Section "INFO Field – Test Failed"
0x3F	The test code supplied in the CMD field is not associated with a valid test	See Section "INFO Field – Invalid Command Code"
0x11	All POST tests have completed successfully	Not used; the INFO field consists of reserved bytes only
0x12	A POST test has failed	See Section "INFO Field – Test Failed"
0x21	All BIST tests have completed successfully	Not used; the INFO field consists of reserved bytes only
0x22	A BIST test has failed	See Section "INFO Field – Test Failed"
0x23	BIST test cycle overrun	Not used; the INFO field consists of reserved bytes only

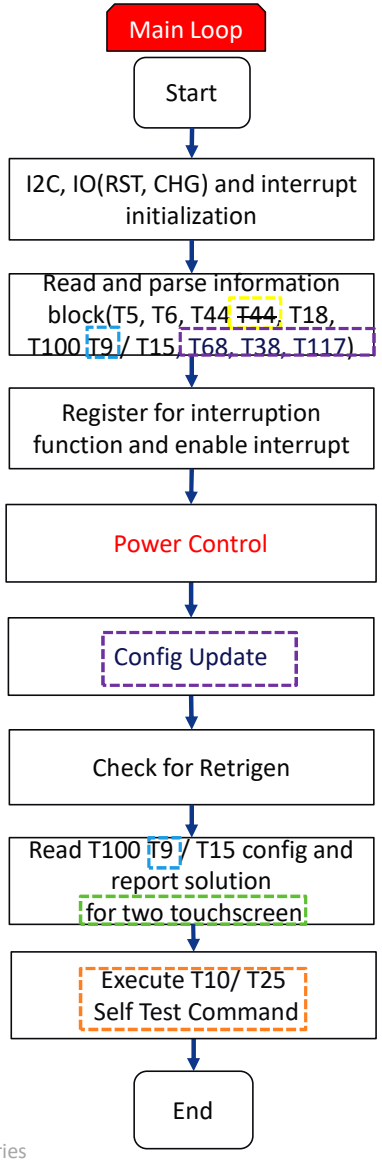
MICROCHIP

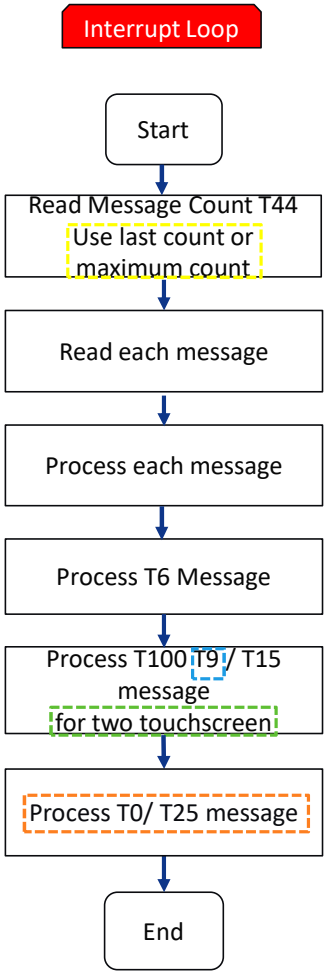
MCU Driver Flow With Power Control

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define POWER_CONTROL      // use T7 for power control
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config [for two touchscreen]
- Read and process messages ([read last count number of messages] [T100 report two touchscreen])
- Config Update
- Execute T10/ T25 Self Test Command
- Process T0/ T25 message
- Power Control





MICROCHIP

Power Control

`mxt_init_t7_power_cfg()`
set T7 power only to make IC run in normal mode if the latest T7 config is invalid （active and idle is all 0 means IC is in sleep mode） . We will make a copy if T7 config is verified.

```
static int mxt_init_t7_power_cfg(struct mxt_data *data)
{
    int error;
    bool retry = false;

recheck:
    error = mxt_read_reg(data->T7_address, sizeof(data->t7_cfg), &data->t7_cfg, data);
    if (error)
        return error;

    if (data->t7_cfg.active == 0 || data->t7_cfg.idle == 0) {
        if (!retry) {
            dev_dbg("T7 cfg zero, retry\n");
            retry = true;
            goto recheck;
        } else {
            dev_dbg("T7 cfg zero after reset, overriding\n");
            data->t7_cfg.active = 20;
            data->t7_cfg.idle = 100;
            return mxt_set_t7_power_cfg(data, MXT_POWER_CFG_RUN);
        }
    }
```

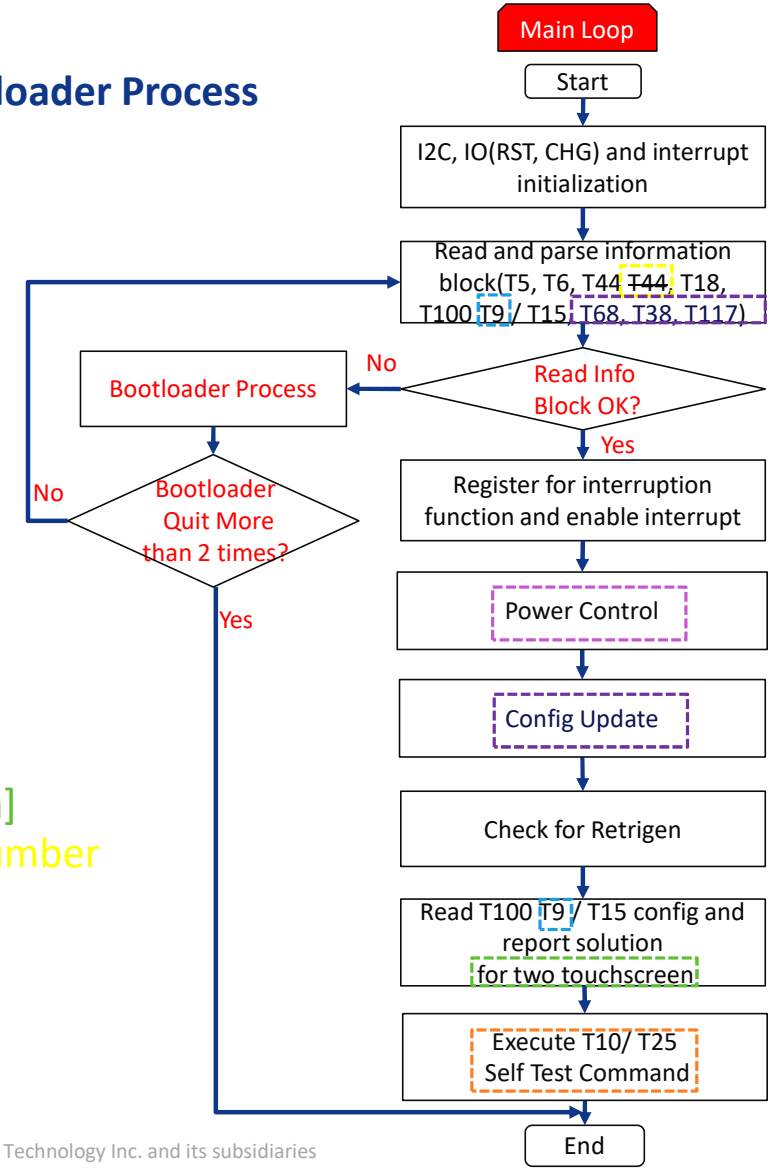

MICROCHIP

MCU Driver Flow With Bootloader Process

- **Hardware Specification**
- POR Sequence
- I2C Specification
- Interrupt Assert
- Retrigen

```
//#define BOOTLOADER_PROCESS  // used for quit bootloader after power up
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Bootloader Process
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config [for two touchscreen]
- Read and process messages ([read last count number of messages] [T100 report two touchscreen])
- Config Update
- Execute T10/ T25 Self Test Command
- Process T0/ T25 message
- Power Control



MICROCHIP

Bootloader Process

Bootloader mode is **only** for flash the firmware content. Because the chip has firmware inside original and we don't need this action normally. But we should know

what's bootloader mode and how to avoid to enter it.

We have 3 methods to get into bootloader mode:

1. Send T6 reset command with dedicated parameter(0xA5)
2. Stretched CHG line for more than 50ms when Reset line de-assert.
3. Toggling Reset line for more than 10 times without I2C communication.

So you know how to entered the boot loader mode now, while you know how to avoid enter it by mistake.

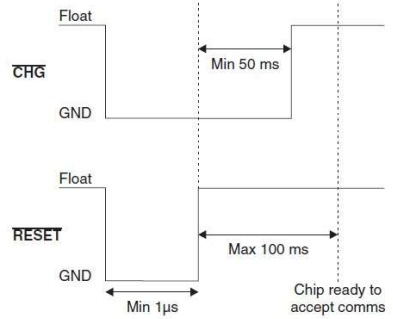
2.2 Command Processor Force-flash Sequence

Write 0xA5 to the Command Processor Object's RESET field to enter the bootloader mode. Refer to the *Protocol Guide* for your device for information on how to do this.

2.3 $\overline{\text{CHG}}$ and $\overline{\text{RESET}}$ Force-flash Sequence

With this sequence the $\overline{\text{CHG}}$ line is held low while the chip is powered up after a reset (see Figure 2-1).

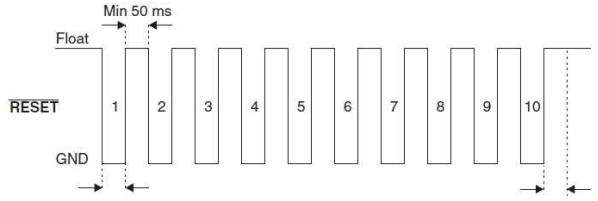
Figure 2-1. $\overline{\text{CHG}}$ and $\overline{\text{RESET}}$ Force-flash Sequence



2.4 $\overline{\text{RESET}}$ Toggling Force-flash Sequence

With this sequence the $\overline{\text{RESET}}$ line is asserted ten times in a row without communicating via the I^2C-compatible bus between the resets.

Figure 2-2. $\overline{\text{RESET}}$ Toggling Force-flash Sequence



Bootloader Quit

For exiting the bootloader mode, you could re-POR or send exit command through I2C.

We use send exit command through I2C for 2 times with different bootloader address in the driver. If it cannot quit the bootloader mode after these 2 times, the driver will return with error.

3. Solution

If the condition (that is, the application I^2C address does not respond) is detected, it can be corrected by WRITING a single byte value (recommendation: 0x00) to the bootloader I^2C address. The bootloader will then restart the application and communicate can continue as normal on the application I^2C address.

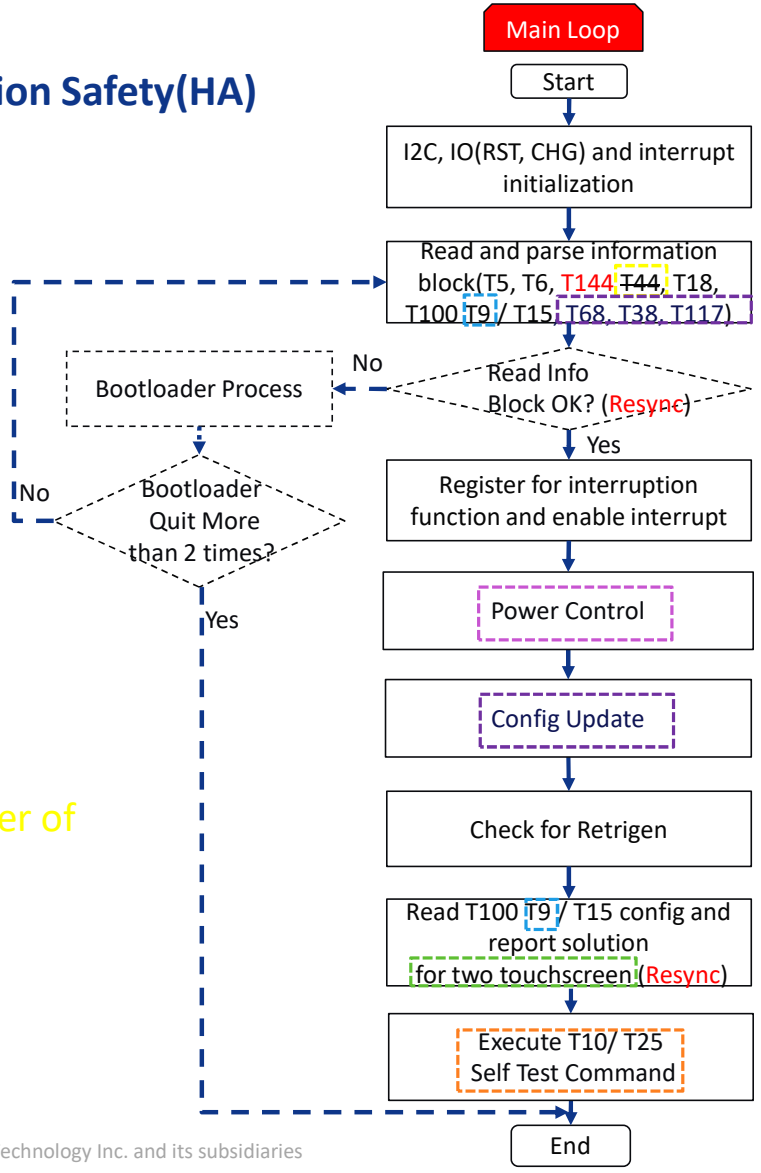
Note: Do not write the byte values 0xDC or 0xAA to the bootloader. These bytes have the special meaning of unlocking the bootloader for downloading a new application, and should be avoided.

MCU Driver Flow With Function Safety(HA)

- **Hardware Specification**
- POR Sequence
- I2C Specification (HA, Function Safety)
- Interrupt Assert
- Retrigen

```
//#define SECURITY            // used for function safety
```

- **Driver Specification**
- Driver initialization
- Read and parse information block
- Bootloader Process
- Register interruption
- Check for Retrigen
- Read T100[T9]/ T15 config [for two touchscreen]
- Read and process messages ([read last count number of messages] [T100 report two touchscreen])
- Config Update
- Execute T10/ T25 Self Test Command
- Process T0/ T25 message
- Power Control
- Resync(HA)



MICROCHIP

I2C Specification – Write Normal Object

The object writing operation finishes in one transmit cycle, the `Seqnum(W)` and `CRC` is required in data content.

The Touch controller asserting the ACK mostly means package is received and self `Seqnum(W)` will be incremented by 1 (No matter whether the Seqnum(W) and CRC are matched in package data).

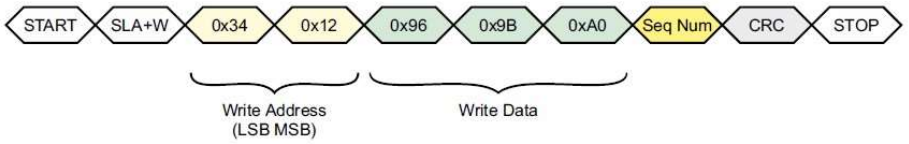
7.2 Writing To the Device

An I²C WRITE cycle consists of the following bytes:

START	1 bit	I²C START condition
SLA+W	1 byte	I²C address of the device (see Section 7.1 "I2C Address")
Address (LSByte, MSByte)	2 bytes	Address of the location at which the data writing starts. This address is stored as the address pointer.
Data	0 .. 11 bytes	The actual data to be written. The data is written to the device, starting at the location of the address pointer. The address pointer returns to its starting value when the I²C STOP condition is detected. Note that a maximum of 11 bytes of data can be written in any one transaction.
Sequence number	1 byte	The sequence number for this write. The sequence number must start at 0 for the first write after power-up/reset and incremented by 1 for each subsequent write. When the sequence number reaches 255, it is reset to 0.
CRC	1 byte	An 8-bit CRC that includes all the bytes that have been sent, including the two address bytes, but not the SLA+W byte. If the device detects an error in the CRC during a write transfer, a COMSERR fault is reported by the Command Processor T6 object.
STOP	1 bit	I²C STOP condition

Figure 7-1 shows an example of writing three bytes of data to contiguous addresses starting at 0x1234.

FIGURE 7-1: EXAMPLE OF A THREE-BYTE WRITE STARTING AT ADDRESS 0x1234



MICROCHIP

I2C Specification – Read Normal Object

The object reading operation finishes in two transmit cycles, the `Seqnum(W)` and `CRC` is required in first setting address package.

<First cycle>

- The Touch controller asserting the ACK mostly means package is received and self `Seqnum(W)` will be incremented by 1 (No matter whether the Seqnum(W) and CRC are matched in package data).
- If the Seqnum(W) and CRC are both matched, slave will allocate the address pointer to current address of the package.

<Second cycle>

- The Slave will report the data without `Seqnum(W)` and `CRC` packed. Be noted only the 1st cycle is verified successfully, the data in 2nd cycle will be valid, otherwise the data is unspecified.

7.3 Reading From the Device

Two I²C bus activities must take place to read from the device. The first activity is an I²C write to set the address pointer (LSByte then MSByte). The second activity is the actual I²C read to receive the data. The address pointer returns to its starting value when the read cycle NACK or STOP is detected.

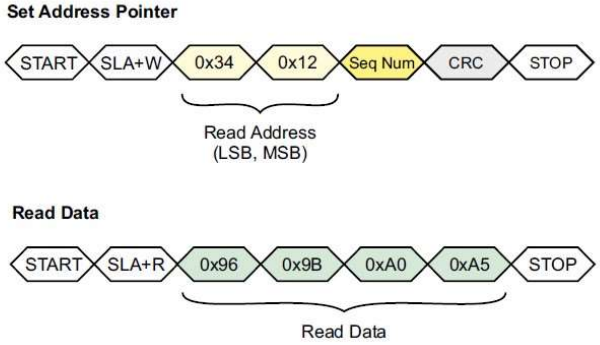
It is not necessary to set the address pointer before every read. The address pointer is updated automatically after every read operation. The address pointer will be correct if the reads occur in order. In particular, when reading multiple messages from the Message Processor T5 object, the address pointer is automatically reset to the address of the Message Processor T5 object, in order to allow continuous reads (see Section 7.3.3 "Reading Status Messages with DMA").

The WRITE and READ cycles consist of a START condition followed by the I²C address of the device (SLA+W or SLA+R respectively).

NOTE Note that only certain read operations include a CRC of the data packets (see Section 7.3.1 "checksums for Read Transactions").

Figure 7-2 shows the I²C commands to read four bytes starting at address 0x1234.

FIGURE 7-2: EXAMPLE OF A FOUR-BYTE READ STARTING AT ADDRESS 0x1234



NOTE At least one data byte must be read during an I²C READ transaction; it is illegal to abort the transaction with an I²C STOP condition without reading any data.

I2C Specification – Read Single Message

The object reading operation finishes in two transmit cycles, the `Seqnum(W)` and `CRC` is required in first setting address package. The 2nd response data will be packed by `Sequm(R)` and `CRC` information.

<First cycle>

•Same as before.

<Second cycle>

•The Slave will report the data with `Seqnum(R)` and `CRC` packed.

Noted only the 1st cycle is verified successfully, the data in 2nd cycle will be valid, otherwise the data is un-specified.

The `Sequm(R)` is different with the `Sequm(W)`, it's the message's reading cycle sequence number, and it will be omitted by host normally (useless) .

Note:

Sequm(W): The write cycle Sequm

Sequm(R): The reading cycle Sequm

If Sequm is written without suffix () literally , we might consider it as Sequm(W) since it's the only sequence number used in the host driver.

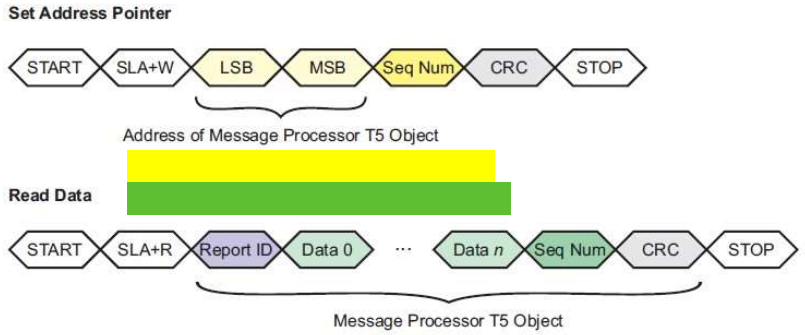
7.3.2 READING A MESSAGE FROM THE MESSAGE PROCESSOR T5 OBJECT

An I²C read of the Message Processor T5 object contains the following bytes:

START	1 bit	I²C START condition
SLA+R	1 byte	I²C address of the device (see Section 7.1 "I2C Address")
Report ID	1 byte	Message report ID
Data	1 or more bytes	The message data (size = size of Message Processor T5 MESSAGE field)
Sequence number	1 byte	The Message Processor T5 sequence number for this read. The sequence number starts at 0 for the first write after power-up/reset and is incremented by 1 for each subsequent read, wrapping round when it reaches 255.
CRC	1 byte	An 8-bit CRC for the Message Processor T5 report ID, message data and sequence number
STOP	1 bit	I²C STOP condition

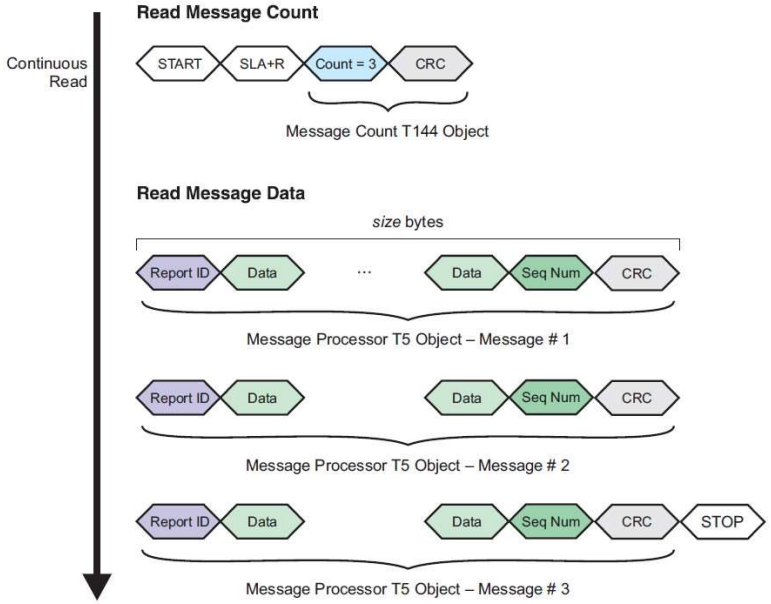
Figure 7-3 shows an example read from the Message Processor T5 object. To read multiple messages using Direct Memory Access, see Section 7.3.3 "Reading Status Messages with DMA".

FIGURE 7-3: EXAMPLE READ FROM MESSAGE PROCESSOR T5



MICROCHIP

I2C Specification – Reading Multi Messages



7.3.3 READING STATUS MESSAGES WITH DMA

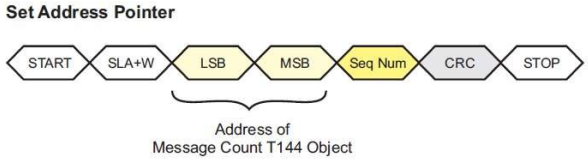
The device facilitates the easy reading of multiple messages using a single continuous read operation. This allows the host hardware to use a Direct Memory Access (DMA) controller for the fast reading of messages, as follows:

1. The host uses a write operation to set the address pointer to the start of the Message Count T144 object, if necessary. Note that 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 T144 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 T144 object (two bytes) to retrieve a count of the pending messages, plus the 8-bit CRC.
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 that the host should have already read the size of the Message Processor T5 object in its initialization code.

   Note that the size of the Message Processor T5 object as recorded in the Object Table includes the sequence number and checksum bytes.
5. 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 T144 object.

Figure 7-4 shows an example of using a continuous read operation to read three messages from the device.

FIGURE 7-4: CONTINUOUS READ EXAMPLE



The object reading operation finishes in two transmit cycles, the `Seqnum(W)` and `CRC` is required in first setting address package.
The 2nd response data will have 2 parts:

1. T144 Object data
• It's same as the normal `Reading the Normal Object` operation. The CRC you saw it's an object content of T144, not the packed content.
2. T5 Message data:
• It's a bundle of messages which are same as former T5 content format of each.

Note: Normally, we will access T144 first to get out of the message count, and then using single message reading to get out the messages one by one instead of the multi-packages reading.

MICROCHIP

Resync(HA)

(Algorithm)

For HA silicon, there are extra stuffed information that named `Seqnum` in each frame (See `I2C Specification` Chapter). If the Seqnum is mis-matched between host and touch controller, the touch controller will not execute the command sent from host or can't respond correct data to host. That will damage the data communication between them.

So how could we do if Seqnum is occasional missed, can we retrieve it back specified? Unfortunately, the official document don't talk about it in details, we should achieve it in our experience.

The key point is when Seqnum is lost, mostly the CRC in each package is incorrect, then we must consider the Resync algorithm to communicate later.

We mostly consider the Resync at 2 scenarios:

- First bootup when the information block CRC verified failed.
- In interrupt processing, the CRC of T44 or T5 are incorrect.

How could we think of the Resync algorithm to make communication workable?
We should know the facts from the test result first:

- We can verify the response data by the checksum to decide whether a Seqnum is correct.
- There are 3 scenarios that data respond with checksum byte:
  1) Read Information Block
  2) T144 data
  3) T5 data
- If the Seqnum is mismatched, the address pointer of the firmware is stuck somewhere, the feedback data will be consistent each time.
- The correct Seqnum can conclude a correct data, but correct data is **not** sufficient to conclude a correct Seqnum, because the address pointer might be occasional same as you want to set this time.
- With all above information, we will design the algorithm by reading Information Block with address pointer offset (twice read) to verify the result:
- Read out the 7 byte id information with address offset 0, then verified id.
- Read out left pieces of Information block with offset 7, then verified the CRC again.
- If the information block verified rightly, we consider it's correct Seqnum retrieved.

But you may ask why we chose the Information block instead of T144/T5 which will be shorted data transferred each time. Here we considered the T144 and T5 data is variable and can't reuse the exist ID information we may have in initialization stage (We consider the Interrupt data broken in priority). But you could think about to use 1T44 to speed up the Resync algorithm.

Now, we will assume the 3 scenarios to retrieve the Seqnum:

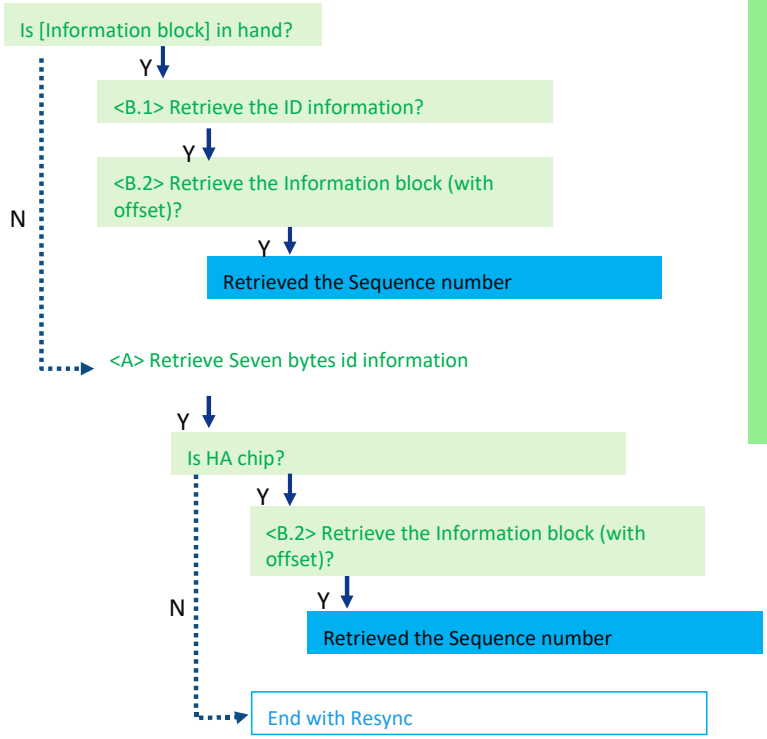
1) The chip may occasionally reset (By ESD/BOD or some other things), the Seqnum is mostly Zero and nearby.
   - We search from Seqnum 2
2) The Seqnum is somewhere unknown.
   - We search from Zero for 256 times
3) The Seqnum is ZERO by Hardware reset.
   - We search from Zero with 3 times loop

Resync(HA)

Round 0: Search from with Seqnum 2

Round 1: We search from 0 for 256 times

Round 2: The Seqnum is ZERO by Hardware reset




```
/* Start check the Seq Num */
for ( round = 0; round < 3 && insync != true; round ++ ){
    // 'I' Round, use overflow search or hardware reset
    if (round == 0) {
        // <Round.0>: Assumed Seqnum change to `0` with unknown reason, so current is 2
        count = 1;  // one time's trying is enough
        mxt_update_seq_num_lock(data, true, 2); // CRC missed + ID info, so the Info block will be 3

        dev_info("Resync Round <I.0>: Assumed the seq round to 2\n");
    }else if (round == 1) {
        // <Round.1>: use the overlfow method to retrieve the seq num
        count = 256;
        // start with 0
        mxt_update_seq_num_lock(data, true, 0);
        dev_info("Resync Round <I.1>: Using overflow to search seq\n");
    } else {
        // <Round.2>: use Hardware reset to retrieve the seq number, set seq to 0
        count = 1;
        dev_info("Resync Round <I.2>: Using Hardware reset to sync\n");

        mxt_hard_reset(data);    // if hardware reset is not available, Round 2 can be omitted
    }
```


MICROCHIP

Importing Note - File Introduction

atmel_mxt_ts.c : main driver process which contains initialization, reading and parsing information block, message handling and reporting, config updating and so on

atmel_mxt_ts.h : macro and structure definition used for main driver process

debug_info.c : debug output for the driver which need to modify with the specified MCU

mcu_interface.c : config array, sleep and wait implementation , I2C read and write implementation, interrupt related implementation and coordinate, key reporting implementation

MICROCHIP

Importing Note – Importing Steps

Step 1: Copy the five documents to your MCU code project, you can put them into the new folder of maxtouch.

Step2: Search the key word “Modify Note”, and modify at the position where “Modify Note” was and as the instruction below the key word. Most of the modification will be occurred in the file of mcu_interface.c.

Step3: After all were modified, you should call mxt_initialize() in your main loop. Then you can build the project and test with the debug info.

Importing Note – Macros

TOUCH_OBJECT is a must for maxtouch Family and TOUCH_KEY is optional.

If only touch keys existed in MCU based solution, TOUCK_OBJECT can be commented.

Other macros are all optional and you can add based on your requirements.

The config update function is very important for maintain the host machine. If there's no update function, you need to break the machine to do upgrade. You can add some conditions to judge whether it need to upgrade.

```
 If the IC you used is an old part, T44 maybe not existed, you need to define T44_NONE;
 and T100 maybe not existed, you need to define T9_OBJECT.
 ********************************************************************************/
// used for touch screen, if there're touch keys only, the macro can be disabled
#define TOUCH_OBJECT
//#define T9_OBJECT           // used for old IC using T9 for touch screen reporting points
//#define TOUCH_KEY           // used for touch keys
//#define T44_NONE            // used for old IC without T44
//#define SECOND_TOUCH        // used for the second touch instance
//#define SECURITY            // used for function safety
// used for self test in the host including pin fault, signal limit and avdd present
// T10 and T25 are all implemented in code and will be selected by the object information,
// but you can select one of them according to your IC type to save code size
//#define SELF_TEST
//#define CONFIG_UPGRADE      // used for upgrade config
//#define BOOTLOADER_PROCESS  // used for quit bootloader after power up
//#define POWER_CONTROL      // use T7 for power control

//#define DEBUG_INFO
```


MICROCHIP

Debug Note - Information Block

1. If the IC is in normal mode and the I2C read function is normal, you will get the correct information through mxt_read_info_block(). You will see the debug info of all the object information.

Tip1: I2C read will be abnormal if the I2C read function don't use 16 bit register in the IC or I2C pin don't have pull-up.

Tip2: The I2C reading reports error if the IC is in bootloader mode because of hardware error.

Tip3: If you get the info of information block crc error, you must check the I2C transfer to make it completely right.

Tip4: If HA parts were used, I2C transfer with sequence and checksum will be used and debug will become more difficult.

Debug Note - Interrupt and Report

2. Please make sure the external interrupt is normal. And then you can get the messages after power on. If the config is normal, you will get messages when you touch the screen.

Tip1: If the interrupt is abnormal, you should check the level of CHG pin. The CHG should be low after power on and when there's message to report in the IC.

Tip2: You should check the I2C read function if the CHG pin is always low. In the normal mode, the CHG pin should be high after the messages are read by the MCU.

Debug Note – Config Update

Tip1: I2C write function may be no effect. You can do an easy test to make sure the writing is OK. You should write the device address, 16bit reg, data together as the datasheet instructed.

Tip2: The config checksum get from power on will indicate whether you update successfully.

Tip3: There’re 2 tools can be used to convert .raw config file to config array data. One is python scripts and another is object-server v5.0.0.0 You can select File->Load Config to load the raw config file and then select File->Save Config to save Raw to Array(*.txt) format to get the array data.