AN1414

# Using HI-TECH C® Compiler to Interface RTCC Devices to 8-Bit PIC® MCUs

*Author: Pinakin K. Makwana*
*Microchip Technology Inc.*

## INTRODUCTION

Microchip's RTCC product line represents a new and unique way to add functionality like Real-Time Clock and Calendar, EEPROM, RAM and Extended Unique Identifier (EUI) to an application. With the small 8-pin packages and $I^2C$™ compatible interface, this device gives the designer added system flexibility. This application note covers details on interfacing RTCC devices with 8-bit PIC® MCUs (PIC16F, PIC18F, etc.). The PIC18 Explorer demo board (DM183032) with the MCP79410 RTCC PICtail™ Plus daughter card (AC164140) are used as hardware.

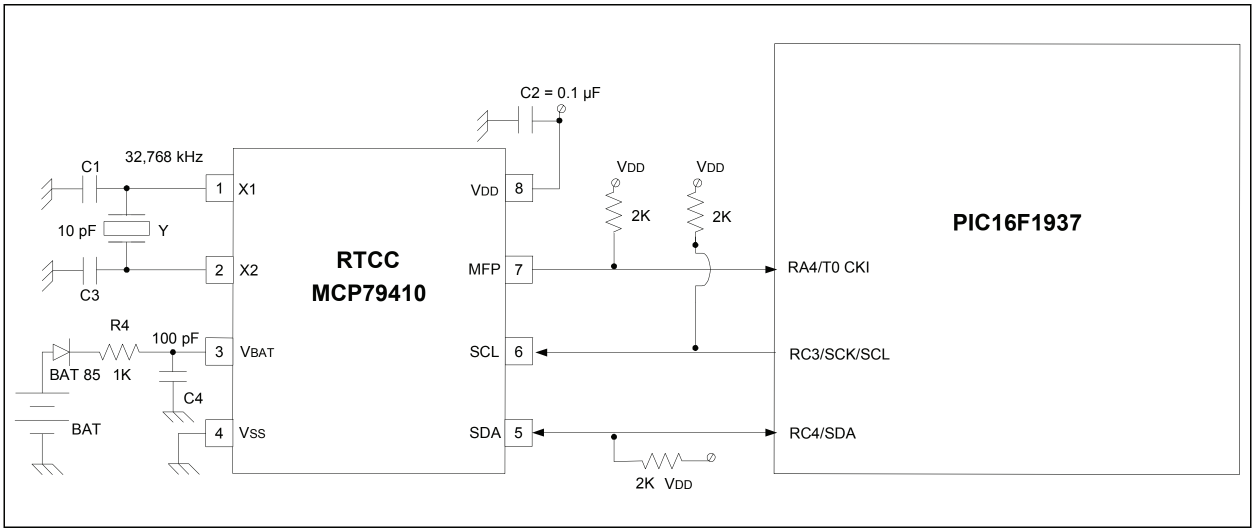
The RTCC product line can work up to 400 kHz speed using $I^2C$ interface. The MSSP module, available on many PIC microcontrollers, provides a very easy-to-use interface for communicating with the MCP7941X series devices. The biggest benefit of using the MSSP module is that the signal timings are handled through hardware rather than software. This allows the firmware to continue executing while communication is handled in the background. This also means that an understanding of the timing specifications associated with the $I^2C$ protocol is not required in order to use the MCP7941X series devices in designs.

This application note is intended to serve as a reference for communicating with Microchip's MCP7941X series RTCC devices with the use of the MSSP module featured on many PIC16 and PIC18 family of devices.

Figure 1 shows the hardware schematic for the interface between Microchip's MCP7941X series devices and the PIC16F1937 PIC microcontroller. The schematic shows the connections necessary between the microcontroller and the RTCC device as tested, and the software was written assuming these connections.

The SDA and SCL pins are open-drain terminals, and therefore, require pull-up resistors to VCC (typically 10 kΩ for 100 kHz, and 2 kΩ for 400 kHz and 1 MHz). The MFP of MCP7941X is also pulled up with a resistor (10kΩ). This pin serves multiple functionality and its various functions will be described later in this document. Figure 1 shows typical hardware connections for the MCP7941X family of RTCC with PIC MCUs.

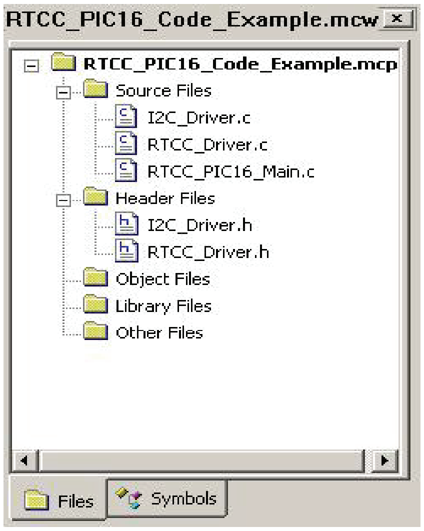
**FIGURE 1: SCHEMATIC**



## FIRMWARE DESCRIPTION

The purpose of the firmware is to facilitate the designer while interfacing the MCP7941X family of devices with an 8-bit PIC microcontroller. The firmware is written using the HI-TECH C compiler. The developed driver for MCP7941X is generic enough to be ported to 8-bit PIC MCUs with little or no code change (provided the PIC device has an on-chip MSSP module). The firmware has multiple files organized as shown in Figure 2.

**FIGURE 2: CODE EXAMPLE**




### I$^2$C_Driver.C

This source file includes low-level APIs to access the MSSP of 8-bit PIC devices. It mainly includes the following functions:

- `InitI2C:` Initializes the MSSP module of the 8-bit PIC device in I$^2$C Master mode. It configures speed of operation for I$^2$C communication. The firmware is set to be used for 100 kHz I$^2$C speed where the system clock of the 8-bit PIC device is assumed to be 10 MHz.
- `CHECK_I2C_IDLE:` It is required for the MSSP module to make sure that the I$^2$C bus is idle before initiating any transactions or events. This API is a blocking function and returns only if the I$^2$C bus is idle. It does not return any value.
- `WRITE_I2C:` This API writes a byte to the addressed slave I$^2$C device connected to the MSSP module.
- `READ_I2C:` This API reads a byte from the addressed slave I$^2$C device connected to the MSSP module.
- `WriteI2CMultipleByte:` This API allows writing multiple bytes to the addressed slave I$^2$C device. This API should be used when a block write is to be performed.
- `ReadI2CMultipleByte:` This API allows reading multiple bytes from the addressed slave I$^2$C device. This API should be used when a block read is to be performed.

### RTCC_Driver.C

This source file includes APIs required to access all functionalities of MCP7941X. Details on each API are provided in this document in the next section.

### RTCC_PIC16_Main.C

This source file shows how to use provided APIs and access MCP7941X.

## ACCESSING MCP7941X FUNCTIONALITIES

It is assumed that the hardware is developed as shown in Figure 1. The MCP7941X family of devices not only have RTCC functionality but also include other features like on-chip SRAM (64 Bytes), on-chip EEPROM (128 bytes) and unique ID as either 48 or 64 bits long.

### Accessing RTCC Functionalities

There are two power sources for MCP7941X where VCC is the default power source. MCP7941X has the capability to maintain time-stamp even in the absence of VCC. It is important to note that communication over $I^2C$ with MCP7941X can be established only during the presence of VCC. Hence, it is the responsibility of the designer to make sure VCC is available to MCP7941X before initiating any transactions over $I^2C$ bus. In other words, when MCP7941X is powered from VBAT, neither a read nor write operation can be performed to MCP7941X.

#### INIT RTCC

This API initializes MCP7941X and configures it to start the oscillator or RTCC. Refer to the MCP7941X data sheet (DS22266) for more details on available options. This API must have been called once before accessing any functionality of RTCC. By default, $I^2C$ is configured for 100 kHz speed where the system clock of the PIC device is assumed to be 10 MHz. The default initialization code generates 1 Hz (non-calibrated) square wave output on the MFP pin of MCP7941X. MCP7941X can also be configured to generate an interrupt when either a alarm0 or alarm1 event occurs. The polarity of the interrupt is also configurable using the ALMxPOL bit. If the end product is expected to remain in storage condition for an extended period of time before its actual usage, then the ST bit (MSB of Seconds register) can be cleared to stop the oscillator, which results in saving battery charge.

#### SETTING DATE AND TIME IN MCP7941X

To ease the access of RTCC, a structure is implemented to read/write time-stamp. The following user-defined structures are created in driver files which are useful to access time-stamp:

- `RTCCTimeDate`: This structure has members for both time and date as detailed below.
  - Day
  - Sec
  - Min
  - Hour
  - Date
  - Month
  - Year
- `RTCCTime`: This structure has members for time only as detailed below.
  - Sec
  - Min
  - Hour
- `RTCCDate`: This structure has members for date only as detailed below.
  - Day
  - Date
  - Month
  - Year

The following APIs are implemented to read or write RTCC time-stamp. Each API follows a similar argument where a pointer to the structure is required to pass while calling.

`ReadRTCCTimeDate`

This API can be called whenever time-stamp is to be read. Since this API takes a pointer to the structure, it automatically updates each member of the structure with the updated time-stamp. This API updates time and date.

`WriteRTCCTimeDate`

This API can be called whenever time-stamp is to be updated. A pointer to the structure is passed while calling this API. Each member of the structure should have been updated prior to calling this API. This API will update the time and date of MCP7941X.

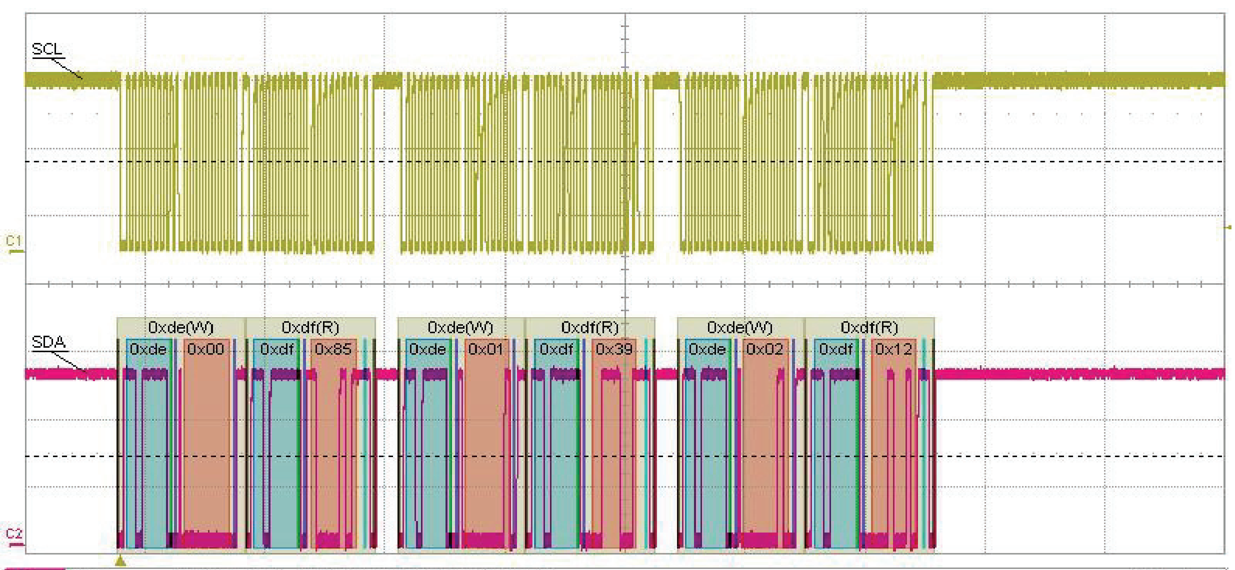
`ReadRTCCTime`

This API function is similar to the `ReadRTCCTimeDate` API except it reads only the time. Figure 3 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 1: READING TIME FROM RTCC**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
RTCCTime      Rtcctime;
void main(void)
{
InitRTCC();
ReadRTCCTime(&Rtcctime);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 3: READING TIME FROM RTCC (HOURS = 0X12, MINUTES = 0X39, SECONDS = 0X05)**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10775 µs | 7 | 0xde | 87 | 1 | 0x00 | |
| 2 | 207.699 µs | 7 | 0xdf | 82 | 1 | 0x85 | |
| 3 | 461.708 µs | 7 | 0xde | 87 | 1 | 0x01 | |
| 4 | 674.513 µs | 7 | 0xdf | 82 | 1 | 0x39 | |
| 5 | 926.922 µs | 7 | 0xde | 87 | 1 | 0x02 | |
| 6 | 1.13973 ms | 7 | 0xdf | 82 | 1 | 0x12 | |

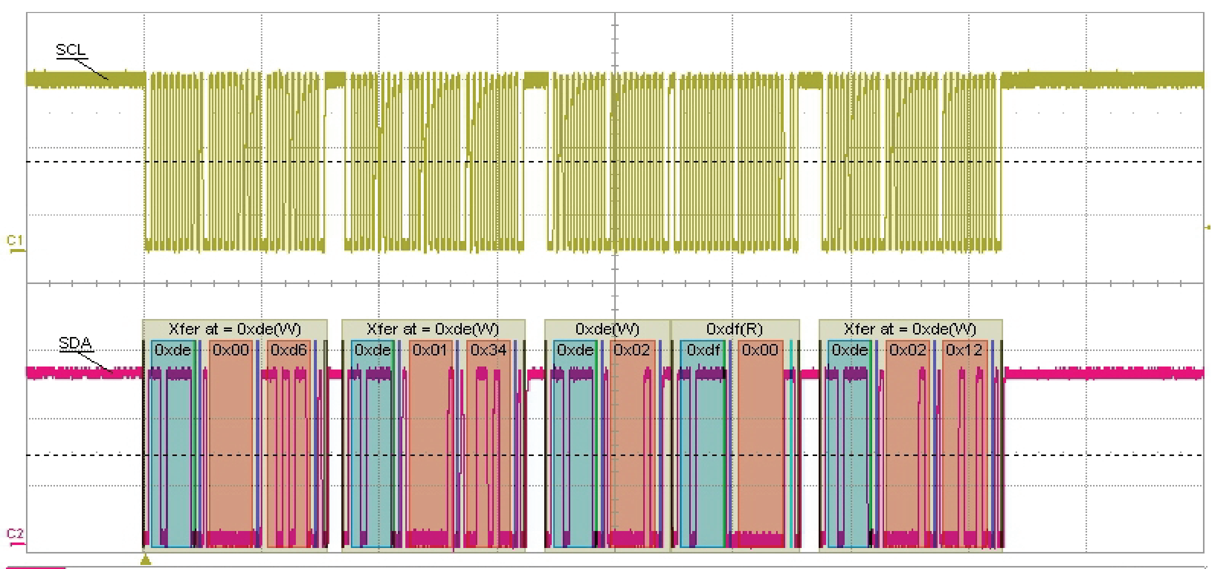
`WriteRTCCTime`

This API function is similar to the `WriteRTCCTimeDate` API except it writes only the time. Figure 4 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 2: WRITE TIME TO RTCC**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
RTCCTime   Rtcctime;
void main(void)
{
InitRTCC();
Rtcctime.Hour = 0x12;//Write 0x12 to hour register
Rtcctime.Min = 0x34;//Write 0x34 to minute register
Rtcctime.Sec = 0x56;//Write 0x56 to second register
WriteRTCCTime(&Rtcctime);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 4: WRITE TIME TO RTCC (HOURS = 0X12, MINUTES = 0X34, SECONDS = 0X56)**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10444 µs | 7 | 0xde | 87 | 2 | 0x00 d6 | |
| 2 | 333.306 µs | 7 | 0xde | 87 | 2 | 0x01 34 | |
| 3 | 676.915 µs | 7 | 0xde | 87 | 1 | 0x02 | |
| 4 | 892.922 µs | 7 | 0xdf | 82 | 1 | 0x00 | |
| 5 | 1.14293 ms | 7 | 0xde | 87 | 2 | 0x02 12 | |

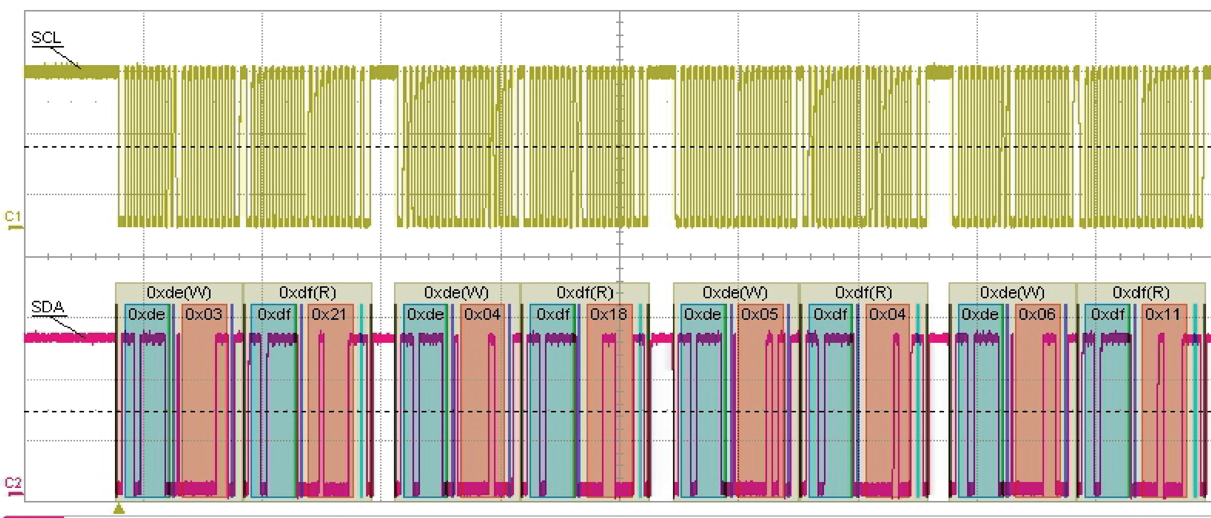
`ReadRTCCDate`

This API function is similar to the `ReadRTCCTime` API except it reads only the date. Figure 5 shows the waveform of the I$^2$C bus when this API is called.

**EXAMPLE 3: READ DATE FROM RTCC**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
RTCCDate      Rtccdate;
void main(void)
{
InitRTCC();
ReadRTCCDate(&Rtccdate);//Read Date from RTCC
while(1)
{
//Other application tasks
}
}
```

**FIGURE 5: READ DATE FROM RTCC (DAY = 0X01, DATE = 0X18, MONTH = 0X04, YEAR = 0X11)**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10697 µs | 7 | 0xde | 87 | 1 | 0x03 | |
| 2 | 207.700 µs | 7 | 0xdf | 82 | 1 | 0x21 | |
| 3 | 462.107 µs | 7 | 0xde | 87 | 1 | 0x04 | |
| 4 | 674.914 µs | 7 | 0xdf | 82 | 1 | 0x18 | |
| 5 | 929.321 µs | 7 | 0xde | 87 | 1 | 0x05 | |
| 6 | 1.14213 ms | 7 | 0xdf | 82 | 1 | 0x04 | |
| 7 | 1.39454 ms | 7 | 0xde | 87 | 1 | 0x06 | |
| 8 | 1.60734 ms | 7 | 0xdf | 82 | 1 | 0x11 | |

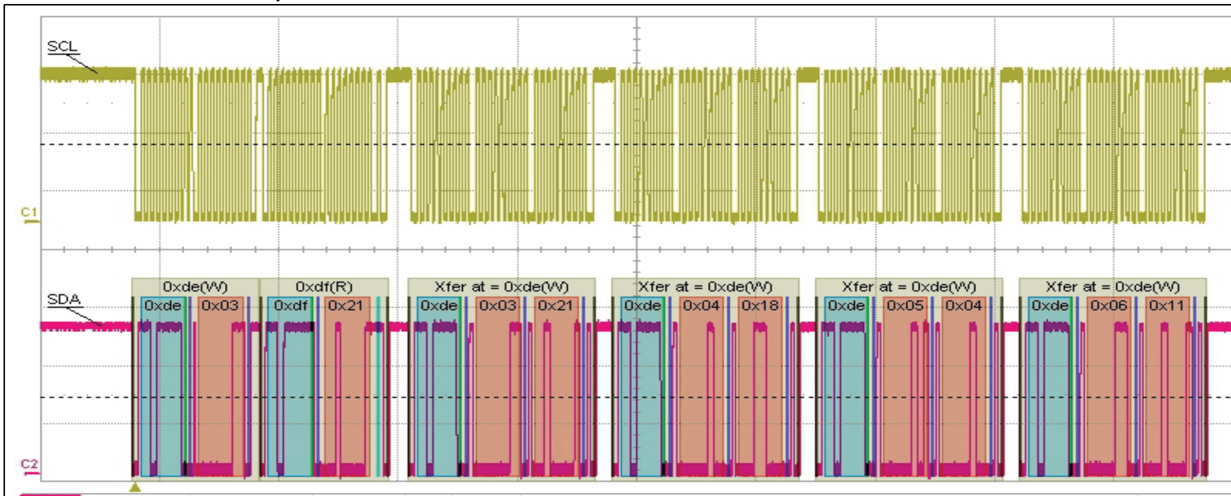
`WriteRTCCDate`

This API function is similar to the `WriteRTCCTime` API except it reads only the date. Figure 6 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 4: WRITING DATE TO RTCC**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
RTCCDate   Rtccdate;
void main(void)
{
InitRTCC();
//Write RTCC with date as Monday,18th April, 2011
Rtccdate.Day = 1;
Rtccdate.Date = 0x18;
Rtccdate.Month = 0x04;
Rtccdate.Year = 0x11;
WriteRTCCDate(&Rtccdate);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 6: WRITING DATE TO RTCC (DAY = 0X01, DATE = 0X18, MONTH = 0X04, YEAR = 0X11)**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10514 µs | 7 | 0xde | 87 | 1 | 0x03 | |
| 2 | 207.700 µs | 7 | 0xdf | 82 | 1 | 0x21 | |
| 3 | 457.307 µs | 7 | 0xde | 87 | 2 | 0x03 21 | |
| 4 | 798.918 µs | 7 | 0xde | 87 | 2 | 0x04 18 | |
| 5 | 1.14053 ms | 7 | 0xde | 87 | 2 | 0x05 04 | |
| 6 | 1.48214 ms | 7 | 0xde | 87 | 2 | 0x06 11 | |

SETTING ALARM IN MCP7941X

MCP7941X has two alarms available which are user configurable. Each alarm is named as '0' and '1'. To access either alarm, an "RTCCAlarm" structure is defined. For each alarm, there is a separate read and write API available.

- ReadRTCCAlarm0
- WriteRTCCAlarm0
- ReadRTCCAlarm1
- WriteRTCCAlarm1
- ReadRTCCAlarm0Flag
- ReadRTCCAlarm1Flag

It is important to note that when the current time-stamp hits the pre-set value of either alarm0 or alarm1, the respective flag is set, which must be cleared by host firmware. To clear these flags, APIs are available which can be called when the alarm event occurs.

## Oscillator Requirement for MCP7941X

The accuracy of the time-keeping function heavily relies on the accuracy of the clock source provided to MCP7941X. MCP7941X has necessary circuits on-chip by which an external crystal (32.768 kHz) can be easily connected. It is recommended to observe the temperature dependency of the crystal when accuracy of the RTCC functionality is very important in an electronic product. The X1 pin of MCP7941X can be used if an external clock source needs to be used. It is recommended to follow a careful PCB layout for the crystal placement. The pin capacitance of X1 and X2 may be required for consideration when higher accuracy from MCP7941X is desired. A reference Gerber file for layout is available from www.microchip.com.

## Battery Back-Up in MCP7941X

A dedicated pin, VBAT, is provided to keep the time function running even in the absence of VCC. A typical CR2032 can be used as battery with MCP7941X. It is important to note that I$^2$C communication is not available when MCP7941X derives its power from the VBAT pin. MCP7941X has the ability to store the time-stamp of VCC failure and VCC restore. Two separate APIs are provided to read VCC failure and restore time.

- ReadRTCCTimeStamp_Vcc_Fail
- ReadRTCCTimeStamp_Vcc_Restore

These time-stamps get overwritten on consecutive failure/restore events. The power-fail time-stamp registers are cleared when the VBAT bit is cleared in software.

## Calibrating MCP7941X

It is required to calibrate MCP7941X due to the inaccuracy of the input clock source. More details on calibration can be found in the MCP7941X data sheet (DS22266). Two APIs are provided to access the calibration registers of MCP7941X.

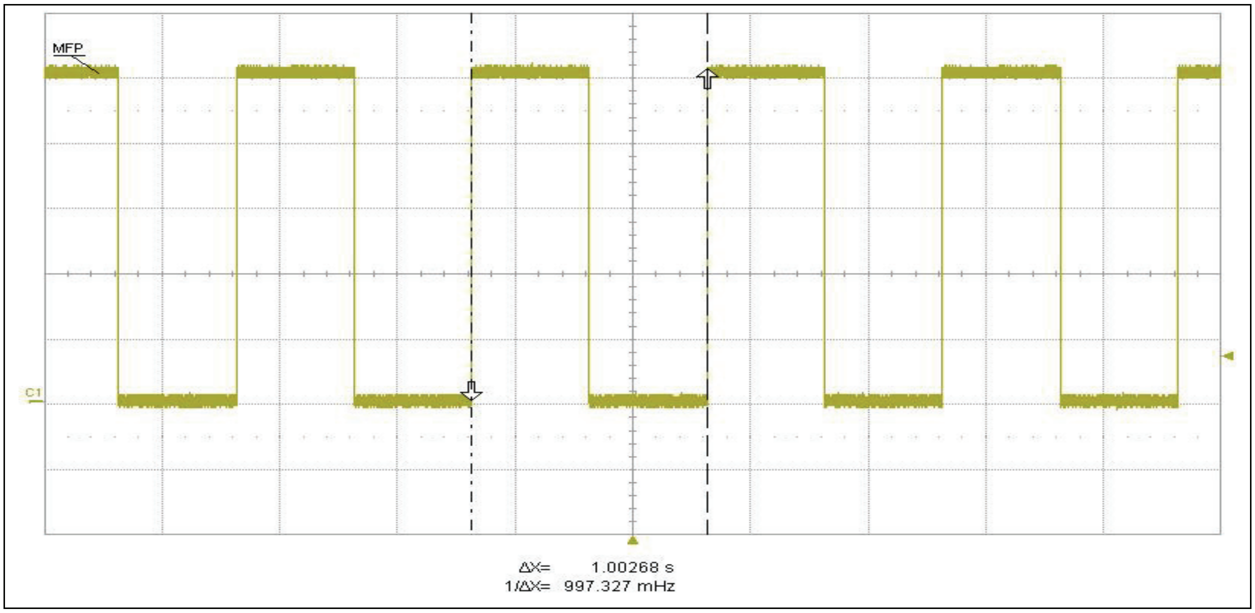
- ReadRTCC_Cal_Reg
- WriteRTCC_Cal_Reg

The "ReadRTCC_Cal_Reg" API reads the calibration value loaded into the CALREG register. "WriteRTCC_Cal_Reg" API write the desired value into the CALREG register.

## Multi-Function Pin (MFP) in MCP7941X

The MFP can be used to generate four different frequencies of square wave. This MFP can also be used to interrupt the PIC device when an alarm event occurs. Figure 7 shows the square wave output on the MFP pin.

**FIGURE 7: OUTPUT ON MFP, NON-CALIBRATED**



## Additional Useful Features of MCP7941X

MCP7941X comes with on-chip SRAM, EEPROM and unique ID. If desired, the unique ID can be EUI-48™ or EUI-64™. SRAM is a volatile memory and preserves its data if VCC fails, provided power from the VBAT pin is available.

## Accessing SRAM in MCP7941X

Following are four basic APIs provided to access on-chip SRAM of MCP7941X. The lower-level APIs take care of appropriate addressing for $I^2C$ bus requirements when one of the following APIs is called.

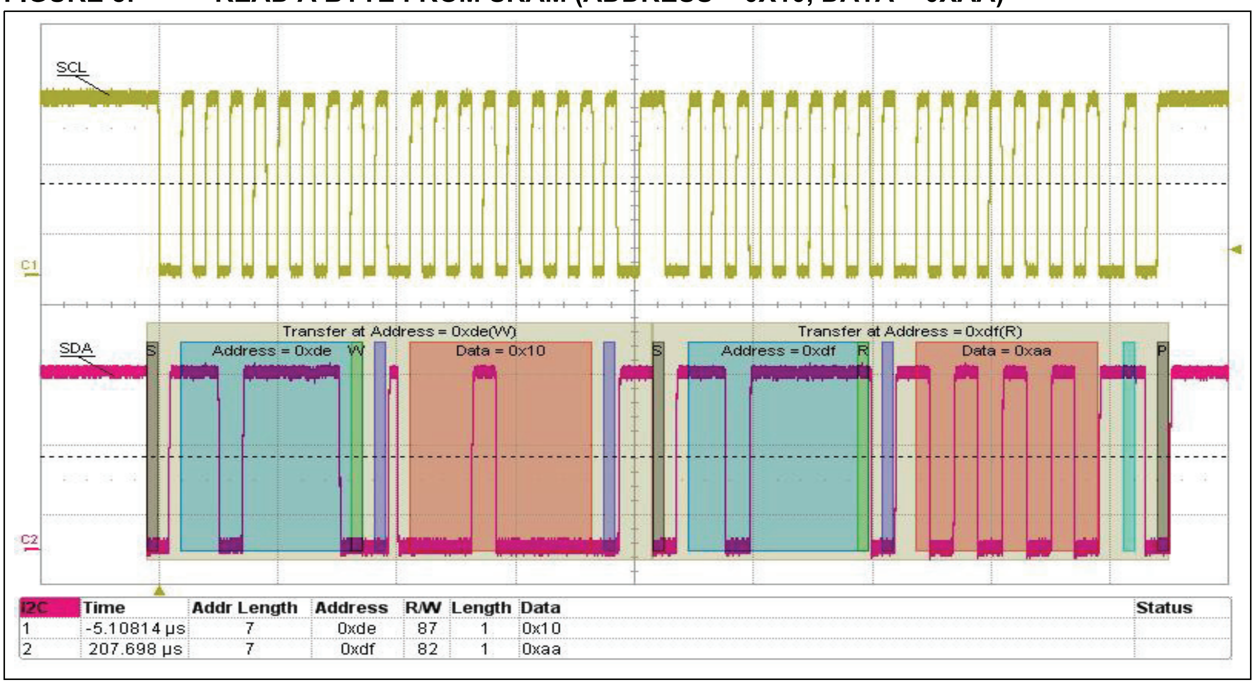
`ReadSRAMByte`

A byte is read and returned when “`ReadSRAMByte`” is called. The user needs to pass the desired SRAM address while calling “`ReadSRAMByte`” API. Figure 8 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 5: READ A BYTE FROM SRAM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
void main(void)
{
unsigned char ReadVal = 0;
InitRTCC();
ReadVal = ReadSRAMByte(0x10);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 8: READ A BYTE FROM SRAM (ADDRESS = 0X10, DATA = 0XAA)**



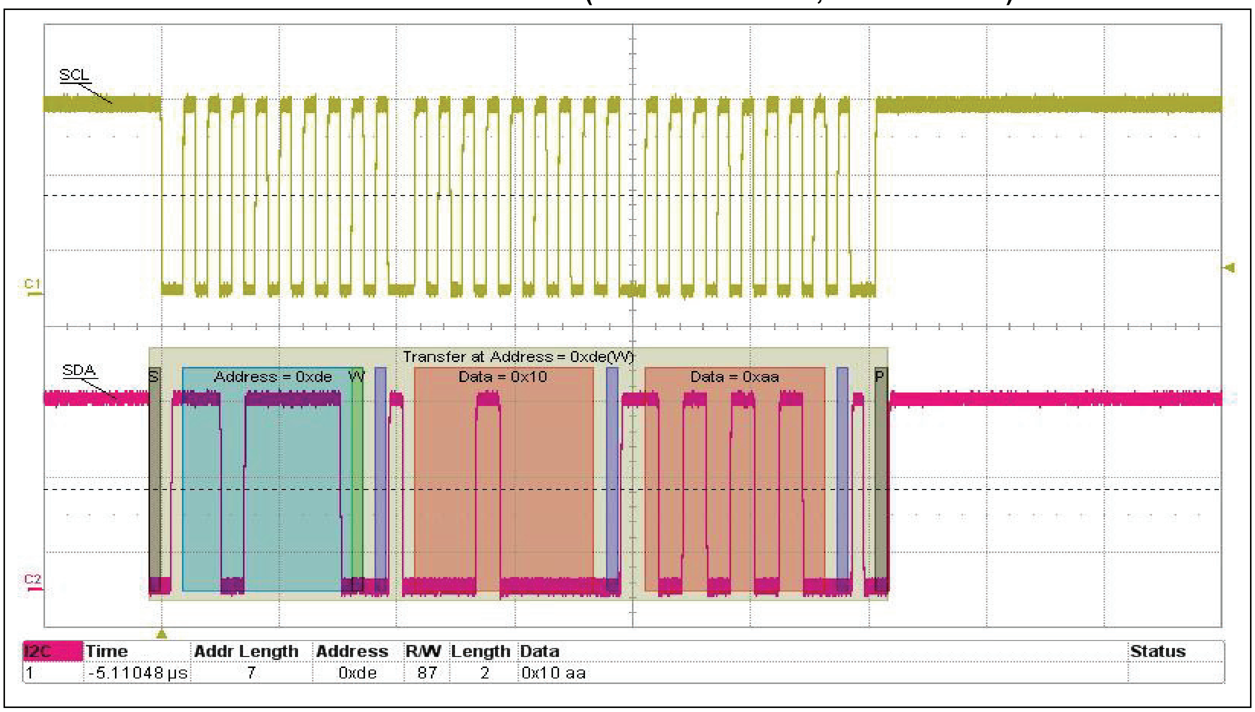
`WriteSRAMByte`

A byte can be written when "`WriteSRAMByte`" is called. The user needs to pass the desired SRAM address and data while calling "`WriteSRAMByte`" API. Figure 9 shows the waveform of the I$^2$C bus when this API is called.

**EXAMPLE 6: WRITE A BYTE FROM SRAM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
void main(void)
{
InitRTCC();
WriteSRAMByte(0x10,0xAA);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 9: WRITE A BYTE FROM SRAM (ADDRESS = 0X10, DATA = 0XAA)**



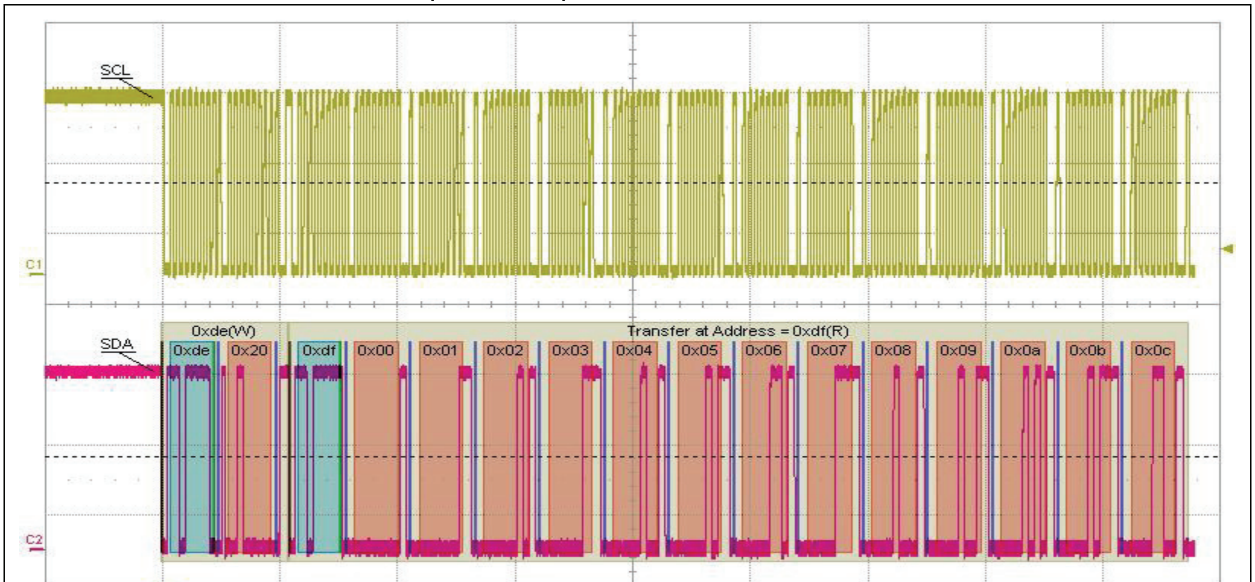
`ReadSRAMBlock`

There is no page in SRAM which is organized as 64 X 8 bits. If required, the entire block of 64 bytes of SRAM can be read using "`ReadSRAMBlock`". The user is required to pass a pointer to an array where the size of the array is required to be 64. This array is updated with the read values. Figure 10 shows the waveform of the $I^2C$ bus when this API is called.

### EXAMPLE 7: READ A BLOCK FROM SRAM

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char Temp_Arr[64];
void main(void)
{
InitRTCC();
ReadSRAMBlock(&Temp_Arr[0]);
while(1)
{
//Other application tasks
}
}
```

### FIGURE 10: READ A BLOCK (64 BYTES) FROM SRAM, FIRST 13 BYTES SHOWN




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10699 µs | 7 | 0xde | 87 | 1 | 0x20 | |
| 2 | 210.900 µs | 7 | 0xdf | 82 | 13 | 0x00 01 02 03 04 05 06 07 08 09 0a 0b 0c | |

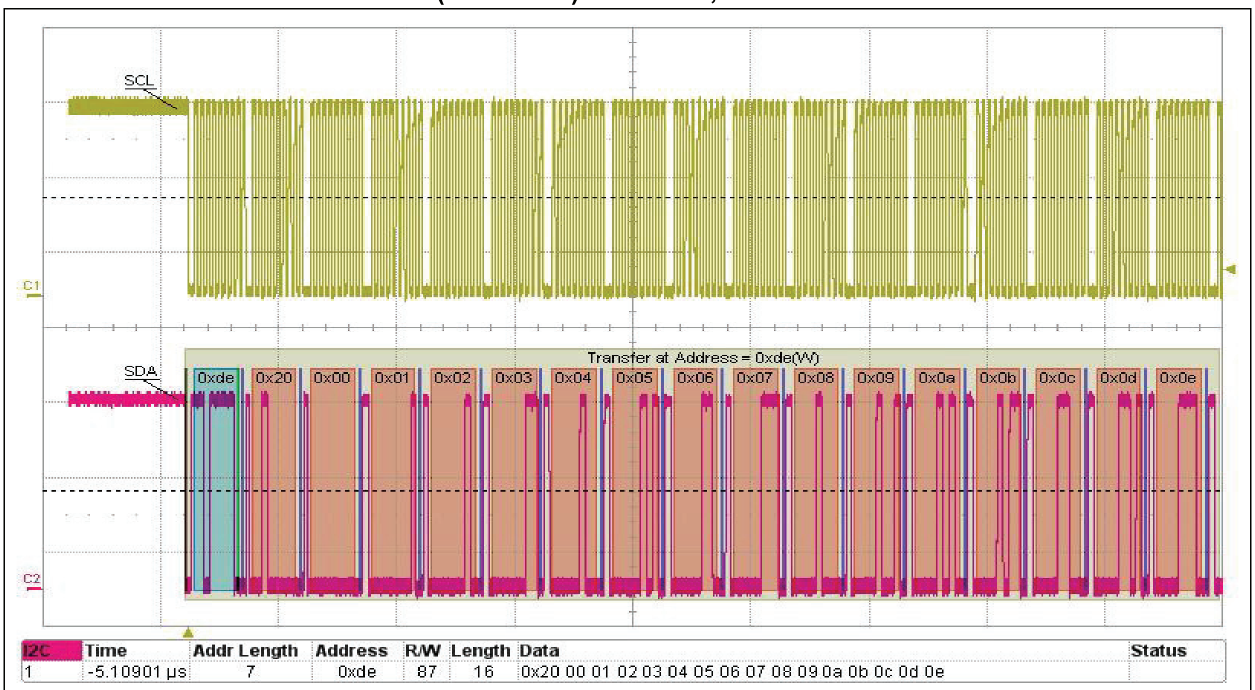
`WriteSRAMBlock`

The entire 64 bytes of SRAM can be written using "`WriteSRAMBlock`" API. The user is required to pass a pointer to an array where the size of the array is required to be 64. Figure 11 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 8: WRITE A BLOCK (64 BYTES) TO SRAM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char Temp_Arr[64];
void main(void)
{
InitRTCC();
WriteSRAMBlock(&Temp_Arr[0]);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 11: WRITE A BLOCK (64 BYTES) TO SRAM, FIRST 15 BYTES SHOWN**



### Accessing EEPROM in MCP7941X

MCP7941X has on-chip 1 Kbits of EEPROM organized as 128 X 8 bits. Following are four basic APIs provided to access EEPROM. The lower-level APIs take care of the appropriate addressing for the I$^2$C bus requirement when one of the following APIs is called.

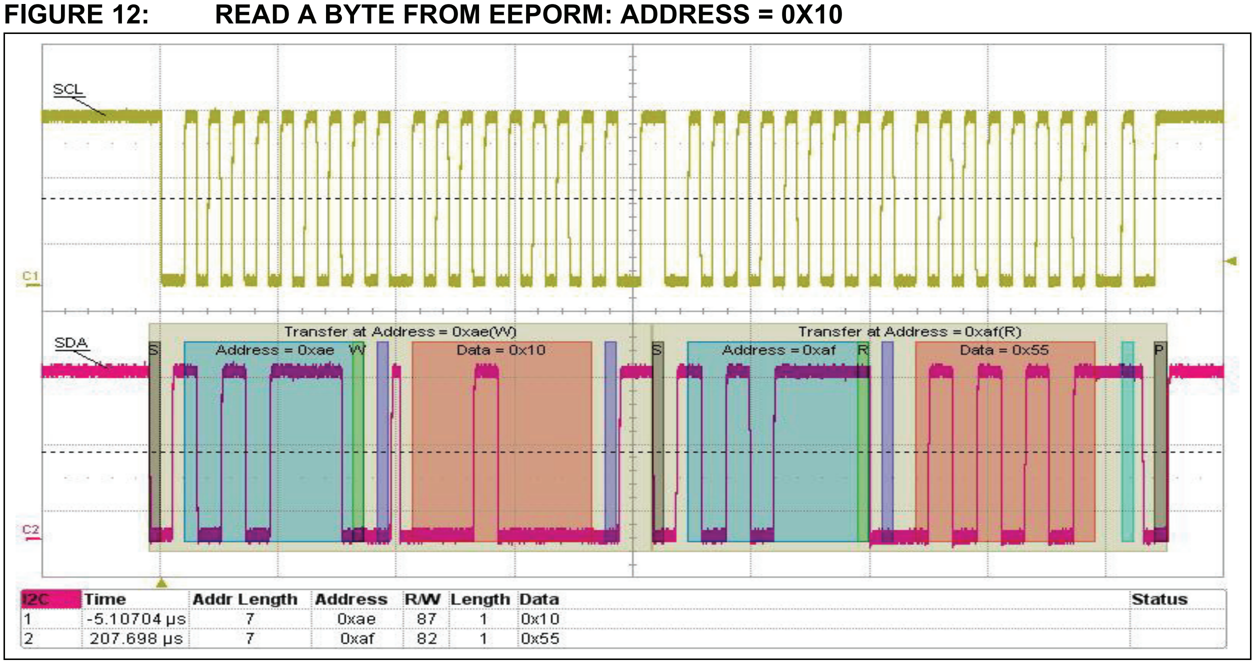
`ReadEEPROMByte`

A byte of EEPROM can be read using “`ReadEEPROM-Byte`” API. The user needs to pass the desired EEPROM address while calling “`ReadEEPROMByte`” API. Figure 12 shows the waveform of the I$^2$C bus when this API is called.

**EXAMPLE 9: READING A BYTE FROM EEPROM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & IESO_OFF &
FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
void main(void)
{
unsigned char ReadVal = 0;
InitRTCC();
//Read EEPROM address 0x10 and store the read value to “ReadVal”
ReadVal = ReadEEPROMByte(0x10);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 12: READ A BYTE FROM EEPORM: ADDRESS = 0X10**



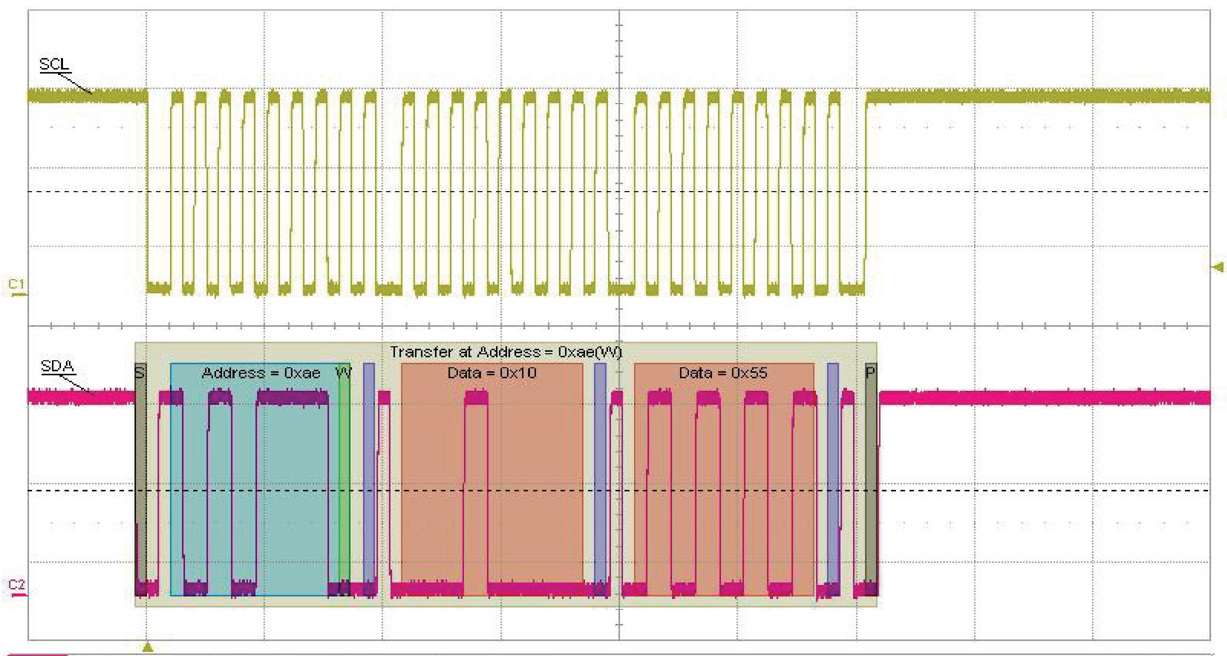
WriteEEPROMByte

A byte can be written when "WriteEEPROMByte" is called. The user needs to pass the desired EEPROM address and data while calling "WriteEEPROMByte" API. Figure 13A shows the waveform of the $I^2C$ bus when this API is called. This API does not check whether EEPROM is idle or not. If a consecutive write is required using this API, then EEPROMBusy API must be called between two calls of WriteEEPROMByte. Figure 13B shows the waveform of the $I^2C$ bus when EEPROMBusy API is called.

**EXAMPLE 10: WRITE A BYTE TO EEPROM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
void main(void)
{
InitRTCC();
//Write 0x55 to EEPROM address 0x10
WriteEEPROMByte(0x10,0x55);
while(1)
{
//Other application tasks
}
}
```

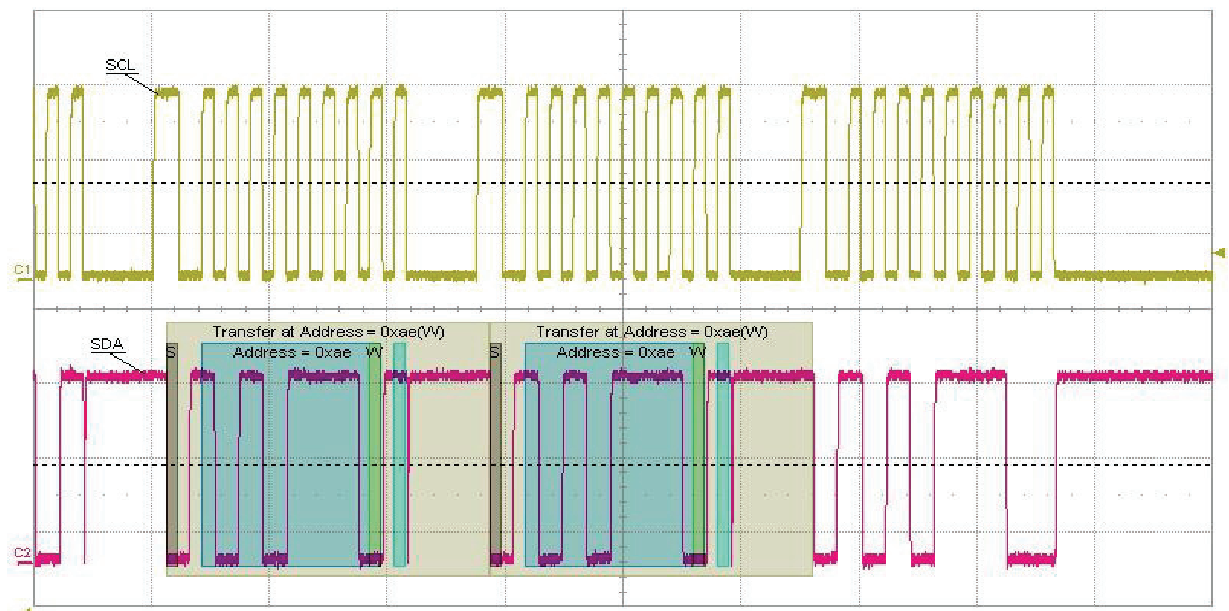
**FIGURE 13A: WRITE A BYTE TO EEPROM: ADDRESS = 0X10, DATA = 0X55**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10905 µs | 7 | 0xae | 87 | 2 | 0x10 55 | |

**FIGURE 13B: POLLING FOR ACKNOWLEDGEMENT FROM EEPROM**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | 3.91061 ms | 7 | 0xae | 87 | 0 | | |
| 2 | 4.04782 ms | 7 | 0xae | 87 | 0 | | |

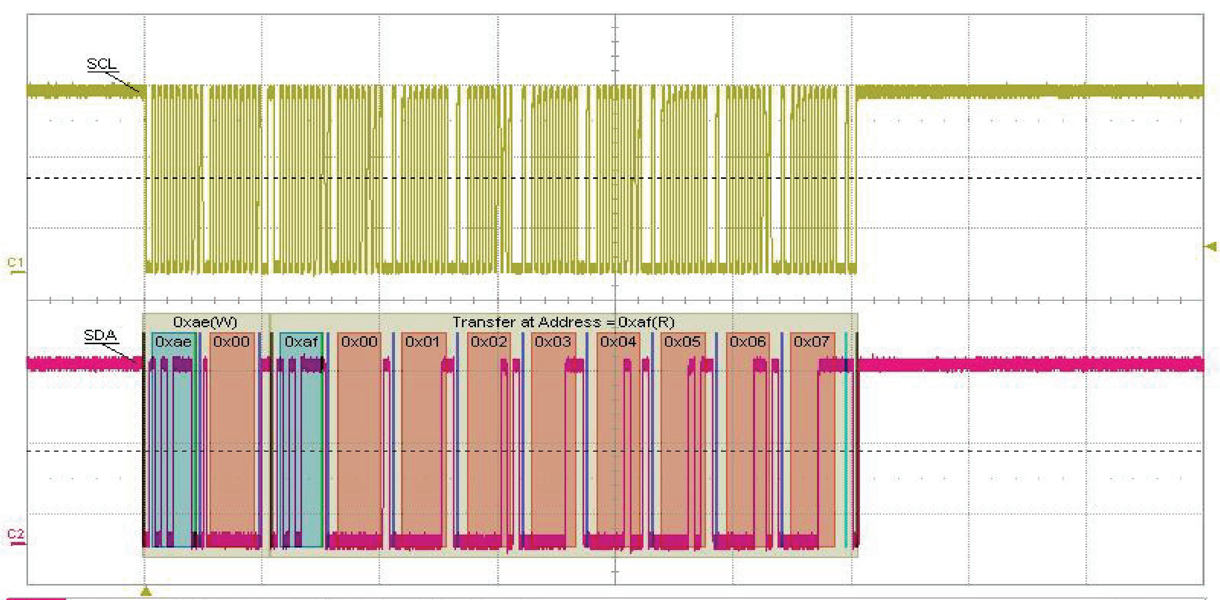
`ReadEEPROMPage`

The entire 128 bytes of EEPROM is divided into pages and each page has 8 bytes. "`ReadEEPROMPage`" API can be used to read the entire page of the EEPROM. The user is required to pass a pointer to an array capable of holding 8 members with the starting address of a read operation. Figure 14 shows the waveform of the I$^2$C bus when this API is called.

**EXAMPLE 11: READ A PAGE FROM EEPROM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char ReadBuf[8];
void main(void)
{
InitRTCC();
//Read a page from EEPROM starting from address 0x00
ReadEEPROMPage(&ReadBuf[0],0x00);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 14: READ A PAGE FROM EEPROM (STARTING ADDRESS = 0X00)**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10772 µs | 7 | 0xae | 87 | 1 | 0x00 | |
| 2 | 211.698 µs | 7 | 0xaf | 82 | 8 | 0x00 01 02 03 04 05 06 07 | |

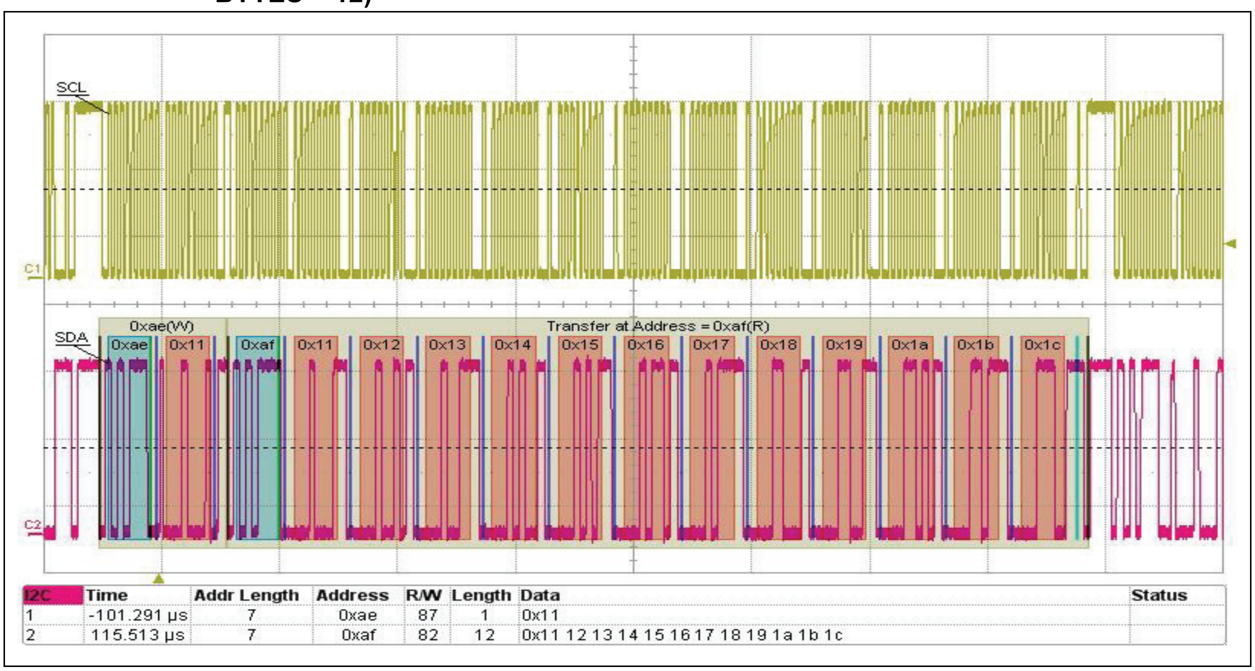
`ReadEEPROMArray`

The entire 128 bytes of EEPROM can be read in a single operation using “`ReadEEPROMArray`”. The user is required to pass three different arguments while calling this API. The user is required to pass a pointer to an array capable of holding required read values with the starting address for read operation and the number of values to be read. Figure 15 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 12: SEQUENTIAL READ FROM EEPROM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char ReadBuf[12];
void main(void)
{
InitRTCC();
//Read 12 bytes from EEPROM starting from address 0x11
ReadEEPROMArray(&ReadBuf[0],0x11,12);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 15: SEQUENTIAL READ FROM EEPROM (STARTING ADDRESS = 0X11, NUMBER OF BYTES = 12)**



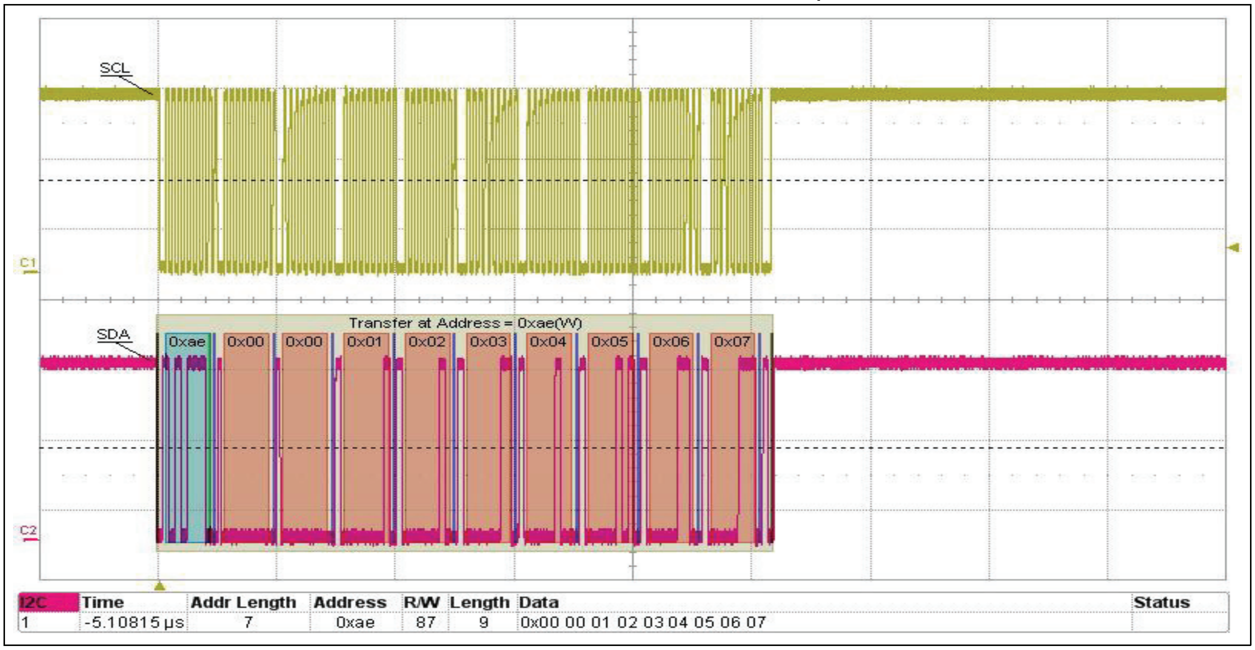
`WriteEEPROMPage`

Since the entire 128 bytes of EEPROM is divided into pages, in a single operation a maximum of 8 bytes can be written. The user is required to pass a pointer to an array capable of holding 8 values and, as a second argument to this API, the address value is passed from where EEPROM will be written. It is the firmware's responsibility to pass the second argument carefully or data may be overwritten because when the page boundary is crossed, the internal address counter rolls over to the beginning of the page. If this API is used to write more than 8 bytes consecutively, then it is required to make sure that the internal write operation is completed. This is done using "`EEPROMBusy`" API. `EEPROMBusy` API returns '`0`' if EEPROM is ready for the next operation, otherwise '`1`'. Figure 16 shows the waveform of the $I^2C$ bus when `WriteEEPROMPage` API is called.

**EXAMPLE 13: WRITING A PAGE TO EEPROM**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char WriteBuf[8];
void main(void)
{
InitRTCC();
//Write a page to EEPROM starting from address 0x00
WriteEEPROMPage(&WriteBuf[0],0x00);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 16: WRITING A PAGE TO EEPROM (STARTING ADDRESS = 0X00 WITH DATA AS 0X00, 0X01, 0X02, 0X03, 0X04, 0X05, 0X06, 0X07)**



## Accessing Unique ID in MCP7941X

MCP7941X provides 8 bytes of nonvolatile memory, which is part of the EEPROM, as well as additional memory. If required, these 8 bytes can be EUI-48 or EUI-64 to serve as MAC addresses of the product. Alternatively, it can also be used as a unique serial number for the product. The unique ID locations can be read similar to EEPROM. However, to avoid spurious writes to the unique ID locations, it must be unlocked before performing a write operation. Two APIs are provided to access unique ID locations.

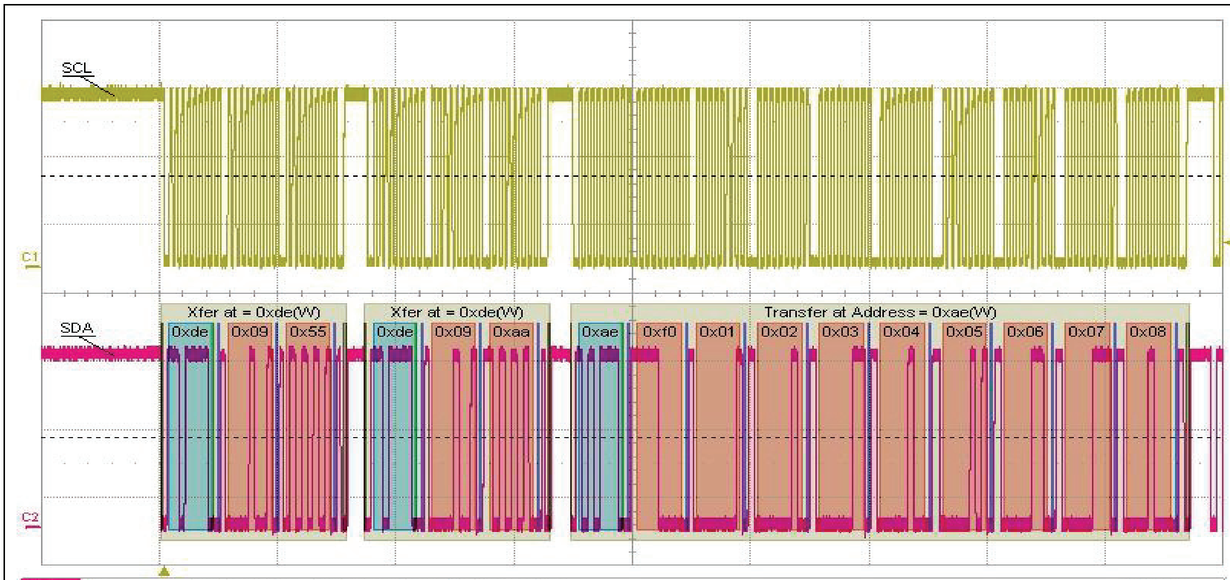
`WriteUniqueID`

This API takes care of unlocking the sequence so the user is required to pass a pointer to an array capable of holding 8 values. The array should contain the required numbers stored prior to calling this API. Figure 17 shows the waveform of the $I^2C$ bus when this API is called.

**EXAMPLE 14: WRITING TO UNIQUE ID LOCATIONS**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char WriteBuf[8];
void main(void)
{
InitRTCC();
//Write Unique ID locations
WriteUniqueID(&WriteBuf[0]);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 17: WRITING TO UNIQUE ID LOCATIONS**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10242 µs | 7 | 0xde | 87 | 2 | 0x09 55 | |
| 2 | 339.307 µs | 7 | 0xde | 87 | 2 | 0x09 aa | |
| 3 | 687.317 µs | 7 | 0xae | 87 | 9 | 0xf0 01 02 03 04 05 06 07 08 | |

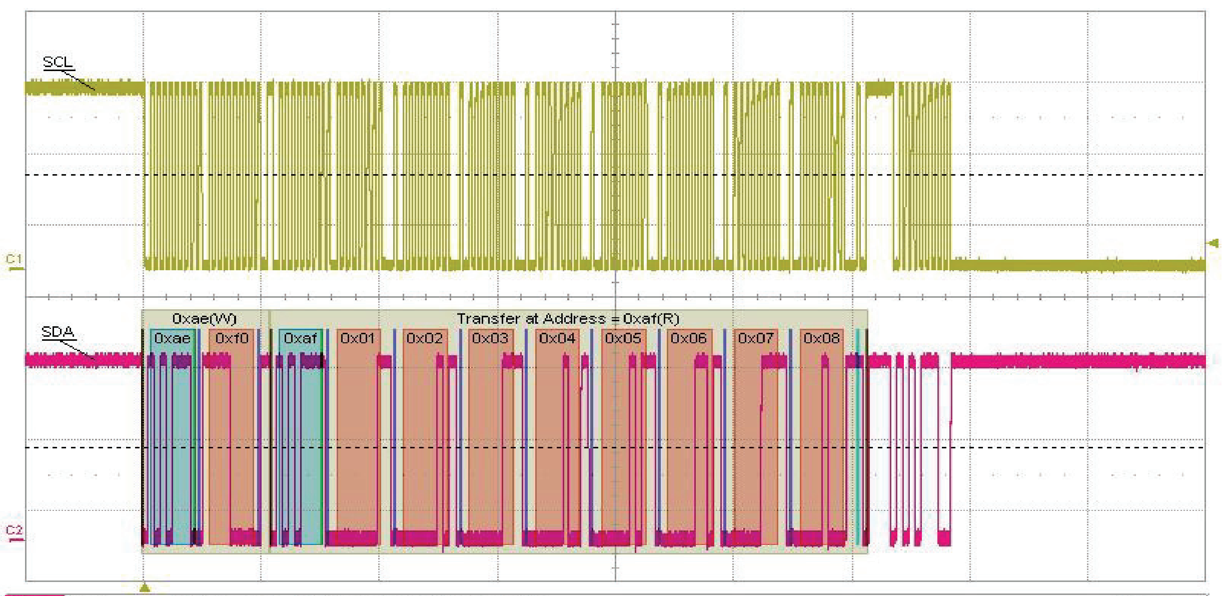
`ReadUniqueID`

This API reads unique ID locations and the user is required to pass a pointer to an array capable of holding 8 values. Figure 18 shows the waveform of the I$^2$C bus when this API is called.

**EXAMPLE 15: READING FROM UNIQUE ID LOCATIONS**

```
#include<htc.h>
#include "RTCC_Driver.h"
__CONFIG(FOSC_HS & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
IESO_OFF & FCMEN_OFF);
__CONFIG(VCAPEN_OFF & PLLEN_OFF & LVP_OFF);
unsigned char ReadBuf[8];
void main(void)
{
InitRTCC();
//Read Unique ID locations
ReadUniqueID(&ReadBuf[0]);
while(1)
{
//Other application tasks
}
}
```

**FIGURE 18: READING FROM UNIQUE ID LOCATIONS**




| I2C | Time | Addr Length | Address | R/W | Length | Data | Status |
|---|---|---|---|---|---|---|---|
| 1 | -5.10300 µs | 7 | 0xae | 87 | 1 | 0xf0 | |
| 2 | 211.705 µs | 7 | 0xaf | 82 | 8 | 0x01 02 03 04 05 06 07 08 | |

## PORTING THE FIRMWARE TO OTHER PIC16/PIC18 FAMILY OF PIC DEVICES

The firmware available with this application should work out of the box if the PIC18 Explorer demo board, with PIC16F1937 PIM and MCP79410 RTCC PICtail Plus daughter card, is used as the hardware. The firmware is written in 'C' language for ease of portability. The same firmware can be recompiled for other PIC16/PIC18 devices with little or no change in the firmware. It may be required to change the SSPADD value if the desired $I^2C$ clock rate is not 100 kHz, or the system clock to PIC16 is not 10 MHz. Refer to the MSSP module section in the respective PIC16/PIC18 data sheet to determine what value should be loaded into the SSPADD register for communicating with MCP7941X. It may be required to change the Configuration register settings done for PIC devices to meet the target application and/or hardware requirement. The firmware is also made available in MPLAB® X, which is a new IDE platform from Microchip.

## CONCLUSION

MCP7941X is a feature-rich Real-Time Clock Calendar. This application note presents how to use Microchip's $I^2C$ RTCC, MCP79410. This application note is written around the PIC18 Explorer demo board and RTCC PICtail Plus daughter card. The firmware is generic and may be optimized for code size or speed, as required by the application.

## ADDITIONAL RECOMMENDED READING

- MCP7941X Data Sheet (DS22266)
- AN1365, Recommended Usage of Microchip Serial RTCC Devices (DS01365)
- TB3065, Enabling Intelligent Automation Using the MCP7941X $I^2C$™ RTCC (DS93065)

## APPENDIX A: REVISION HISTORY

### Revision A (11/2011)

Initial Release.

**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.



QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2009

**Trademarks**



Printed on recycled paper.

ISBN: 978-1-61341-769-0

*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.*

# Worldwide Sales and Service

## 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

**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**
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

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

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

**Santa Clara**
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

**Toronto**
Mississauga, Ontario,
Canada
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: 852-2401-1200
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 - Hangzhou**
Tel: 86-571-2819-3187
Fax: 86-571-2819-3189

**China - Hong Kong SAR**
Tel: 852-2401-1200
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: 86-755-8203-2660
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

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

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

## ASIA/PACIFIC

**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: 91-20-2566-1512
Fax: 91-20-2566-1513

**Japan - Yokohama**
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

**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: 886-7-536-4818
Fax: 886-7-330-9305

**Taiwan - Taipei**
Tel: 886-2-2500-6610
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 - Munich**
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

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

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

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

**UK - Wokingham**
Tel: 44-118-921-5869
Fax: 44-118-921-5820

08/02/11