SmartLEWIS™ MCU

Smart Low Energy Wireless Systems with a Microcontroller Unit

PMA51xx/PMA71xx

RF Transmitter ASK/FSK 315/434/868/915 MHz,
Embedded 8051 Microcontroller,
10-bit ADC,
125 kHz ASK LF Receiver

Function Library Guide

Revision 1.5, 2010-06-01

Wireless Control

Edition 2010-06-01

Published by
Infineon Technologies AG
81726 Munich, Germany

Legal Disclaimer

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation, warranties of non-infringement of intellectual property rights of any third party.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

PMA51xx/PMA71xx RF Transmitter ASK/FSK 315/434/868/915 MHz,
Embedded 8051 Microcontroller,
10-bit ADC,
125 kHz ASK LF Receiver

Revision History: 2010-06-01, Revision 1.5

Previous Revision: 1.4

Page	Subjects (major changes since last revision)
40	Library function LFSensitivityCalibration() added
81	Library function ReadManufacturerRevNum() added
18	Code example changed: Old function MeasureBatteryVoltage() replaced by MeasureSupplyVoltage()
	Document also valid for PMA71xx family

Table of contents

1 Introduction

1.1 General considerations

This document describes the Library functions that are available in the PMA5110 or PMA7110. For any other product out of the PMA51xx or PMA71xx family a reduced set of library functions is available related to the reduced feature set of the product.

Note: The PMA51xx/PMA71xx Function Library is designed to be used with Keil Cx51 compiler. If another compiler is used, special attention has to be taken to issues like parameter passing.

To be able to use the Library functions in the application code the following files have to be included in the software project**:**

- PMA71xx_PMA51xx_Library.h (the header file including the function prototypes)
- PMA71xx_PMA51xx_Library.lib (the precompiled Library functions)

Note: All execution times stated throughout this document are typical values at V_{Bat} = 3V and T_{amb} = 25°C.

1.2 Type definitions

1.2.1 Parameter types

The following table defines the parameter types used throughout this document.

Table 1 Definition of types

Type	Bits	Range	
		minimum	maximum
unsigned char	8	0	255
signed char	8	-128	127
unsigned int	16	0	65,535
signed int	16	-32,768	32,767
unsigned long	32	0	4,294,967,295
signed long	32	-2,147,483,648	2,147,483,647

Table 2 and **Table 3** show how the Keil Cx51 compiler passes the parameters of the Library functions in registers. The Library functions expect the parameters as shown in **Table 2**. If another compiler then Keil Cx51 is used, the parameter passing of this compiler must be equal to Keil Cx51.

Table 2 Parameters passed in registers

Parameter #	char or 1 byte pointer: unsigned char idata * unsigned int idata * unsigned long idata *	int (2 bytes)
1	R7	R6 & R7 (MSB in R6, LSB in R7)
2	R5	R4 & R5 (MSB in R4, LSB in R5)
3	R3	R2 & R3 (MSB in R2, LSB in R3)

Table 3 Examples for parameter passing

Declaration	Description
void **AES128Decrypt**(*const unsigned char idata* * InputData, *unsigned char idata* * OutputData, *const unsigned char idata* * Key);	This function uses three 1 byte pointers as parameters. The first parameter is passed in R7. The second parameter is passed in R5 and the third parameter is passed in R3.
signed char **CalibrateIntervalTimer**(*unsigned char* WU_Frequency, *unsigned long idata* * Xtal_Frequency);	This function uses one char and one 1 byte pointer as parameters. The first parameter is passed in R7 and the second parameter is passed in R5.
signed char **LFBaudrateCalibration**(*unsigned int* Baudrate, *unsigned long idata* * Xtal_Frequency);	This function uses one int (2 bytes) and one 1 byte pointer as parameters. The first parameter is passed in R6 (MSB) and R7 (LSB). The second parameter is passed in R5.
unsigned int **UIntDivision**(*unsigned int* Dividend, *unsigned int* Divisor);	This function uses two int (2 bytes) as parameters. The first parameter is passed in R6 (MSB) and R7 (LSB). The second parameter is passed in R4 (MSB) and R5 (LSB).

The enumeration types which are used in the struct RF_Config (see **Table 4 “RF Coder modes” on Page 30**) are mapped to 8 bit char by the Keil Cx51 compiler.

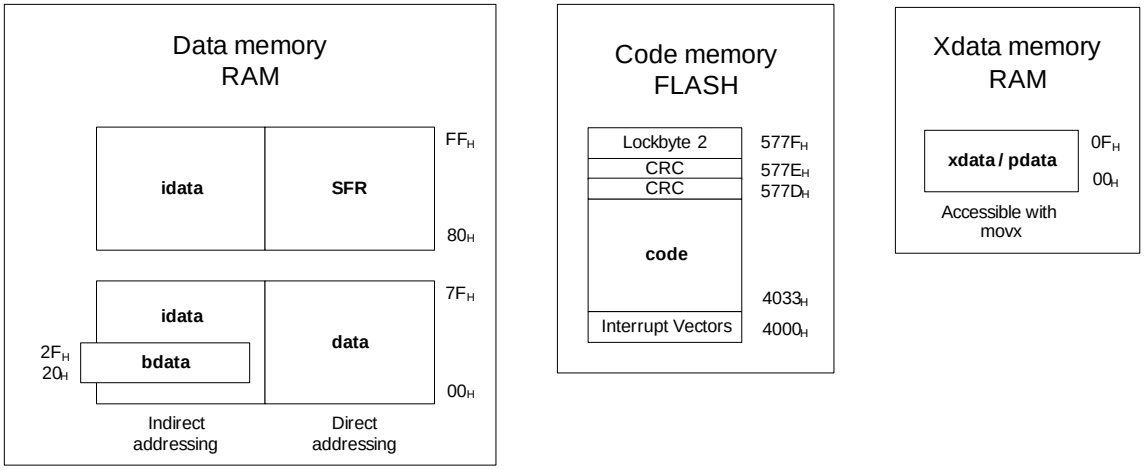
1.2.2 Memory types

The architecture of the PMA51xx/PMA71xx provides 6 different memory areas. Each variable may be explicitly assigned to a specific memory space using the memory types shown in **Table 4**. The location of the different memory types is shown in **Figure 1** .

Table 4 Description of memory types of PMA51xx/PMA71xx

Memory Type	Size	Description
code	6 kbytes	Code memory. Accessible with movc.
data	128 bytes	Directly accessible internal data memory, fastest access.
idata	256 bytes	Indirectly accessible internal data memory, accessed across the full address space. The RAM area $C0_H$ through FF_H is used for Library functions.
bdata	16 bytes	Bit-addressable internal data memory.
xdata	16 bytes	Internal xdata memory. Accessible with movx @DPTR.
pdata	16 bytes	Paged internal xdata memory. Accessible with movx @Rn. (n = 0..7)

Figure 1 Location of memory types of PMA51xx/PMA71xx



Some of the Library functions are using pointers in the parameter list. The memory type of these pointers has to be specified as **idata** to enable the compiler to pass the parameter in a single register. The Library function expects the pointer in a single register. If the memory type is not specified as idata, a generic pointer is used which needs three registers to pass the parameter, thereby the Library function does not get the correct pointer in the expected register. The result of the Library function will be undefined.

The internal data memory (IDATA) is a 256-byte on-chip RAM area that is indirectly accessed using 8-bit addresses. It may be used to declare variables only (no idata function declarations).

IDATA variables can be declared as follows:

unsigned char**idata** variable;

unsigned int**idata** *pointer;

1.3 Wakeup Handler

The Wakeup Handler is executed every time the device wakes up from POWER DOWN state. Possible wakeup sources are listed in **“[1]” on Page 82**.

Table 5 Wakeup Handler

Value	typ	max	Unit	Conditions
Wakeup from WU event until execution of first code line (independent of wakeup source)	1.0	2.1	ms	

1.4 Restricted RAM and FLASH areas

The Library functions use certain address areas in RAM and FLASH. They must not be changed by the application for proper operation.

1.4.1 Restricted RAM areas

The RAM area $C0_H$ through FF_H (upper 64 bytes) of the 256-bytes RAM is used for the Library functions so this RAM area is overwritten by the Library functions. Details on the use of RAM areas for each function can be found in the corresponding function descriptions.

1.4.2 Restricted FLASH areas

The FLASH address $57FF_H$ is reserved for Lockbyte 3. This value must not be changed by the application otherwise it might result in an unintentionally locked FLASH *User Data Sector I* and *II*. If the *User Data Sectors* have been locked, they can only be unlocked by setting the PMA51xx/PMA71xx into Test Mode and erasing the FLASH *Code Sector* and *User Data Sectors*. To lock the *User Data Sectors*, the FLASH *Code Sector* must also be locked (by setting Lockbyte 2) otherwise setting Lockbyte 3 does not have any effect.

*Note: If **“WriteUserDataSectorLine()” on Page 57WriteUserDataSectorLine()WriteUserDataSectorLine()** is used by the application it has to be ensured that the restricted Flash address is not overwritten unintentionally.*

1.5 FLASH organization

For the application the PMA51xx/PMA71xx offers a FLASH area of 256 bytes located from 5780_H to $587F_H$. This FLASH area is split into two sectors, *User Data Sector I* and *User Data Sector II*. **Figure 2** shows the organization of the 6 kbytes FLASH including the *User Data Sectors*. These sectors are organized in four *word lines*, where each of these *word lines* consists of 32 bytes. The *User Data Sectors* are shown in **Figure 3** .

Figure 2 PMA51xx/PMA71xx FLASH organization

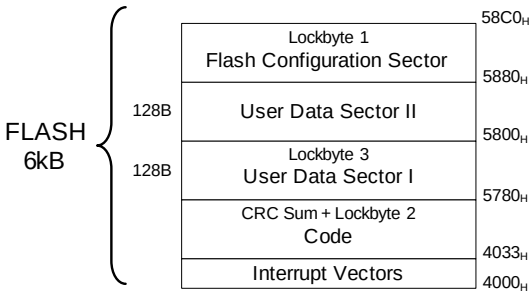


Figure 3 PMA51xx/PMA71xx Flash User Data Sectors

Flash User Data Sector II

	byte31	byte30	byte29	byte28	byte27	byte26	byte26	byte4	byte3	byte2	byte1	byte0	Address
Wordline 3	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	5860_H
Wordline 2	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	5840_H
Wordline 1	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	5820_H
Wordline 0	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	5800_H

Flash User Data Sector I

	byte31	byte30	byte29	byte28	byte27	byte26	byte26	byte4	byte3	byte2	byte1	byte0	Address
Wordline 3	**Lockbyte3**	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	$57E0_H$
Wordline 2	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	$57C0_H$
Wordline 1	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	$57A0_H$
Wordline 0	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	00_H	5780_H

2 Library Functions

The following Library functions are available for use in application code:

Table 6 Library functions

Library function	Description	Page
Temperature measurement		
MeasureTemperature(..)	Returns ambient temperature.	Page 13
Supply voltage measurement		
MeasureSupplyVoltage(..)	Returns the battery voltage.	Page 16
StartSupplyVoltage()	These three functions perform a Battery Voltage measurement during real-time critical applications like the RF Transmission.	Page 18
TriggerSupplyVoltage()		Page 20
GetSupplyVoltage(..)		Page 21
State change		
PowerDown()	Forces the device to POWER DOWN state	Page 23
ThermalShutdown()	Forces the device to THERMAL SHUTDOWN state.	Page 23
Clock switch		
Switch2XTAL()	Switches the system clock to the crystal and prepares the device for RF Transmission.	Page 26
Switch2RC()	Switches the system clock to the 12 MHz RC HF Oscillator.	Page 26
RF Transmission		
VCOTuning()	Adjusts the PLL frequency with respect to temperature and supply voltage (stops after lock).	Page 28
InitRF(..)	Sets up the device for RF Transmission.	Page 29
TransmitRF(..)	Performs an RF Transmission.	Page 34
Interval Timer		
CalibrateIntervalTimer(..)	Sets the Interval Timer precounter to the appropriate value.	Page 37
LF-Receiver		
LFBaudrateCalibration(..)	Sets the baudrate for the LF Receiver.	Page 40
LFSensitivityCalibration(..)	Sets the sensitivity level of the LF Receiver.	Page 40
Division		
ULongDivision(..)	Divides two unsigned long values (32 bit : 32 bit)	Page 44
UIntDivision(..)	Divides two unsigned integer values (16 bit : 16 bit)	Page 46
CRC Calculation		
CRC8Calculation(..)	Generation of an 8 bit checksum using the CRC polynomial ($x^8+x^2+x^1+x^0$)	Page 49
CRC8_410(..)	Generation of an 8 bit checksum using the CRC polynomial ($x^8+x^4+x^1+x^0$)	Page 50

2.1 Temperature measurement

The PMA51xx/PMA71xx provides an on chip temperature sensor which can be easily accessed by the user through the Library function **MeasureTemperature()**.

2.1.1 MeasureTemperature()

2.1.1.1 Description

This function performs a temperature measurement.

2.1.1.2 Actions

- Measures the temperature with the temperature sensor
- Performs a device specific offset correction

2.1.1.3 Prototype

unsigned char **MeasureTemperature** (signed int **idata** * *Temp_Result*)

2.1.1.4 Parameters

Table 7 MeasureTemperature: Parameters

Register / Address	Type	Direction	Name	Description

R7	signed int idata*	Output	Temp_Result	iData pointer to the first element of a structure containing the measurement results.
				
				Temperature measurement results struct:
				struct{
				  signed int Temperature;
				  signed int Temperature_ADC_Output;
				} **idata** Temp_Result;
				
				Temp_Result.Temperature:
				8000_H = -256.0 °C
				0000_H = 0.0 °C
				$7FFF_H$ = 255.9921875 °C
				(= 256 °C - 1 LSB where 1 LSB = 1/128 °C)
				
				Temp_Result.Temperature_ADC_Output:
				10 bit ADC temperature output value (without device specific correction of the temperature sensor)
				0000.00xx.xxxx.xxxxb: x..valid bit

2.1.1.5 Return value

Table 8 MeasureTemperature: Return value

Register/ Address	Type	Description
R7	unsigned char	0000.0000b: Success xxxx.xxx1b: Underflow of ADC Result xxxx.xx1xb: Overflow of ADC Result

2.1.1.6 Resource Usage

Table 9 MeasureTemperature: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, ADCOFF, ADCC0, ADCC1, ADCDL, ADCDH, ADCM, ADCS, CFG0, CFG1, CFG2, DIVIC, DPL, DPH, IE
Stack	14 bytes (2 bytes for function call included)
RAM (IDATA)	$C2_H$ - $E7_H$

2.1.1.7 Execution Time

Table 10 MeasureTemperature: Execution Time

Value	typ	max	Unit	Conditions
Instruction cycles		1130		

2.1.1.8 Code example

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main()
{
        // Return value of temperature measurement is stored in StatusByte
        unsigned char StatusByte;

        // struct for temperature measurement results
        struct{
                signed int Temperature;
                signed int Temperature_ADC_Output;
        } idata Temp_Result;

        // Temperature measurement function call
        StatusByte=MeasureTemperature(&Temp_Result.Temperature);

        if(!StatusByte){
                // Temperature measurement was successful
        }
        else{
                // Temperature measurement was not successful, underflow or
                // overflow of ADC result occurred
        }
}
```

2.2 Supply voltage measurement

There are two different ways to measure the supply voltage (voltage on pin V_{Bat}) using Library Functions. Use MeasureSupplyVoltage() for all non-time critical applications. If temporary drops of the battery voltage are expected during time critical applications like the RF Transmission the supply voltage measurement can be split into three different functions: StartSupplyVoltage(), TriggerSupplyVoltage() and GetSupplyVoltage(..).

2.2.1 MeasureSupplyVoltage()

2.2.1.1 Description

This function performs a battery voltage measurement.

2.2.1.2 Actions

- Measures the supply voltage with the ADC
- Performs a device specific offset correction

2.2.1.3 Prototype

unsigned char **MeasureSupplyVoltage** (signed int **idata** * *Batt_Result*)

2.2.1.4 Parameters

Table 11 MeasureSupplyVoltage: Parameters

Register / Address	Type	Direction	Name	Description

R7	signed int idata*	Output	Batt_Result	iData pointer to the first element of a structure containing the measurement results.
				
				struct
				{
				  signed int BatteryVoltage;
				  signed int BatteryVoltage_ADC_Output;
				} **idata** Batt_Result;
				
				Batt_Result.BatteryVoltage:
				8000_H = -4096.0 mV (Only theoretical number)
				0000_H = 0.0 mV
				$7FFF_H$ = 4095.875 mV
				(= 4096 mV - 1 LSB where 1 LSB = 1/8 mV)
				
				Batt_Result.BatteryVoltage_ADC_Output:
				10 bit ADC battery voltage output value (without device specific correction of the voltage sensor)
				0000.00xx.xxxx.xxxxb: x..valid bit

2.2.1.5 Return value

Table 12 MeasureSupplyVoltage: Return value

Register/ Address	Type	Description
R7	unsigned char	0000.0000b: Success xxxx.xxx1b: Underflow of ADC Result xxxx.xx1xb: Overflow of ADC Result

2.2.1.6 Resource Usage

Table 13 MeasureSupplyVoltage: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, ADCOFF, ADCC0, ADCC1, ADCDL, ADCDH, ADCM, ADCS, CFG0, CFG1, CFG2, DPL, DPH, DIVIC, IE
Stack	8 bytes (2 bytes for function call included)
RAM (IDATA)	$C2_H$ - $C9_H$, DE_H - $E7_H$

2.2.1.7 Execution Time

Table 14 MeasureSupplyVoltage: Execution Time

Value	typ	max	Unit	Conditions
Instruction cycles		668		

2.2.1.8 Code Example

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main()
{
        // Return value of battery voltage measurement is stored in StatusByte
        unsigned char StatusByte;

        // struct for battery voltage measurement results
        struct{
                signed int BatteryVoltage;
                signed int BatteryVoltage_ADC_Output;
        }   idata Batt_Result;

        // Battery voltage measurement function call
        StatusByte=MeasureSupplyVoltage(&Batt_Result.BatteryVoltage);

        if(!StatusByte){
                // Battery voltage measurement was successful
        }
        else{
                // Battery voltage measurement was not successful, underflow or
               // overflow of ADC result occurred
        }
}
```

2.2.2 Supply voltage measurement in time critical applications

For measurements of the supply voltage during time critical applications the supply voltage measurement can be split into three different Library functions. **StartSupplyVoltage()** has to be called before the real time critical application is executed. To trigger a supply voltage measurement during the time critical application **TriggerSupplyVoltage()** is used. After the time critical function has finished **GetSupplyVoltage()** can be called to read out the supply voltage measurements results.

2.2.2.1 StartSupplyVoltage()

2.2.2.1.1 Description

This function sets up the ADC and the Supply Voltage Sensor for a battery voltage measurement.

2.2.2.1.2 Actions

- Prepares the ADC and the Supply Voltage Sensor for a measurement of the supply voltage

2.2.2.1.3 Prototype

void **StartSupplyVoltage** (void)

2.2.2.1.4 Resource Usage

Table 15 StartSupplyVoltage: Resources

Type	used or modified
Registers	R0, R7
SFR	A, PSW, DPH, DPL, ADCM, ADCC0, ADCC1, ADCOFF, CFG1, CFG2, IE

Type	used or modified
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.2.2.1.5 Execution Information

Table 16 StartSupplyVoltage: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		28		

2.2.2.1.6 Code Example

See **GetSupplyVoltage()**.

2.2.2.2 TriggerSupplyVoltage()

2.2.2.2.1 Description

This function triggers the ADC to measure the battery voltage with the Supply Voltage Sensor.

Example: When a measurement of the supply voltage during RF Transmission is done this function has to be called by the application after the last byte of the datagram is shifted into SFR RFD.

2.2.2.2.2 Actions

- Triggers the ADC for a Supply Voltage measurement

2.2.2.2.3 Prototype

void **TriggerSupplyVoltage** (void)

2.2.2.2.4 Resource Usage

2.2.2.2.5 TriggerSupplyVoltage: Resources

Type	used or modified
Registers	---
SFR	ADCM
Stack	2 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.2.2.2.6 Execution Information

Table 17 TriggerSupplyVoltage: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		4		

2.2.2.2.7 Code Example

See **GetSupplyVoltage()**.

2.2.2.3 GetSupplyVoltage()

2.2.2.3.1 Description

This function reads the Supply Voltage measurement result.

2.2.2.3.2 Actions

- Compensates the offset
- Reads out the Supply Voltage measurement result

2.2.2.3.3 Prototype

unsigned char **GetSupplyVoltage** (signed int **idata** * *Batt_Result*)

2.2.2.3.4 Parameters

Table 18 GetSupplyVoltage: Parameters

Register / Address	Type	Direction	Name	Description

R7	signed int idata*	Output	Batt_Result	iData pointer to the first element of a structure containing the measurement results.
				
				struct
				{
				  signed int BatteryVoltage;
				  signed int BatteryVoltage_ADC_Output;
				} **idata** Batt_Result;
				
				Batt_Result.BatteryVoltage:
				8000_H = -4096.0 mV (Only theoretical number)
				0000_H = 0.0 mV
				$7FFF_H$ = 4095.875 mV
				(= 4096 mV - 1 LSB where 1 LSB = 1/8 mV)
				
				Batt_Result.BatteryVoltage_ADC_Output:
				10 bit ADC battery voltage output value (without device specific correction of the voltage sensor)
				0000.00xx.xxxx.xxxxb: x..valid bit

2.2.2.3.5 Return value

Table 19 GetSupplyVoltage: Return value

Register/ Address	Type	Description
R7	unsigned char	0000.0000b: Success xxxx.xxx1b: Underflow of ADC Result xxxx.xx1xb: Overflow of ADC Result

2.2.2.3.6 Resource Usage

Table 20 GetSupplyVoltage: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPH, DHL, ADCS, ADCDL, ADCDH, CFG1, CFG2, IE
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	DD_H, DE_H, DF_H, $E0_H$ - $E5_H$

2.2.2.3.7 Execution Information

Table 21 GetSupplyVoltage: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		412		

2.2.2.3.8 Code Example

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main(){
        // Return value of battery voltage measurement is stored in StatusByte
        unsigned char StatusByte;

        // struct for battery voltage measurement results
        struct{
                signed int BatteryVoltage;
                signed int BatteryVoltage_ADC_Output;
        }   idata Batt_Result;

        // ADC setup for supply voltage measurement is done before real time critical
        // function is executed
        StartSupplyVoltage();

        // real time critical function starts here
        // ...
        TriggerSupplyVoltage();
        // ...
        // end of real time critical function

        // Get the measurement result after the real time critical function
        StatusByte = GetSupplyVoltage(&Batt_Result.BatteryVoltage);

        if(!StatusByte){
                // Battery voltage measurement was successful
        }
        else{
                // Battery voltage measurement was not successful, underflow or
                // overflow of ADC result occurred
        }
}
```

2.3 State Change

There are two Library functions available which support PMA51xx/PMA71xx state changes. To save battery life time **PowerDown()** can be called.**ThermalShutdown()** can be used to prevent uncontrolled operation of the devices outside the operating temperature range.

2.3.1 PowerDown()

2.3.1.1 Description

This function forces the device to POWER DOWN state.

2.3.1.2 Actions

- Waits until all peripherals (RF Transmission and Interval Timer value storing) have completed their ongoing operations
- Enters POWER DOWN state

2.3.1.3 Prototype

void **PowerDown**(void)

2.3.1.4 Resource Usage

Table 22 PowerDown: Resources

Type	used or modified
Registers	none
SFR	A, FCS, CFG0, IE
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.3.1.5 Execution Information

Table 23 PowerDown: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		19		ITinit = 0, RFSE = 1 before entering PDWN

2.3.2 ThermalShutdown()

2.3.2.1 Description

This function checks the ambient temperature and forces the device to THERMAL SHUTDOWN state in case the temperature exceeds the maximum allowed operating temperature.

The application should call this function whenever the ambient temperature is close to the maximum operating range (this can be detected by using **MeasureTemperature()**) to prevent uncontrolled behaviour of the device outside the operating temperature range.

If this function is called when the temperature is below the TMAX threshold, the function will return without any action and the application program will continue uninterrupted. If the temperature is above the TMAX threshold THERMAL SHUTDOWN state is entered.

The TMAX detector wakes up the device from THERMAL SHUTDOWN state when the ambient temperature is below TMAX threshold.

2.3.2.2 Actions

- Turns on the TMAX Detector
- Enters THERMAL SHUTDOWN state if the temperature exceeds the maximum specified operating range (see **“[1]” on Page 82**).

2.3.2.3 Prototype

void **ThermalShutdown**(void)

2.3.2.4 Resource Usage

Table 24 ThermalShutdown: Resources

Type	used or modified
Registers	R7
SFR	A, PSW, DPL, DPH, TMAX, CFG0, IE, DSR
Stack	3 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.3.2.5 Execution Information

Table 25 ThermalShutdown: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		136		

2.3.2.6 Code Example

```
// Register definition
#include "Reg_PMA71xx_PMA51xx.h"

// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

// Definition of some used bits
#define BIT_WUP  0x02
#define BIT_TMAX 0x40

void main (void){

        // Decision: Reset or Wakeup entry:
        // ================================
        if ((DSR & BIT_WUP)== 0) {

                // Reset entry: initial entry of the Program,
                // ============ no wakeup is pending (Flag WUP)

                // Enable TMAX wakeup
                WUM &= ~(BIT_TMAX);

                // ... add your source code here

                // Check if ambient temperature is above TMAX
                ThermalShutdown();

                // ... this code is executed if temperature is below TMAX
        }
        else {
                // Wakeup entry:
                // =============

                // Evaluate the Wakeup Source:
                // ---------------------------

                if (WUF & BIT_TMAX)  {
                        // ... TMAX wakeup handling
                }
        }
}
```

2.4 Clock Switch

Functions **Switch2XTAL()** and **Switch2RC()** allow selection of the system clock source of the PMA51xx/PMA71xx.

2.4.1 Switch2XTAL()

2.4.1.1 Description

This function switches the system clock from the internal 12 MHz RC HF oscillator to the external crystal oscillator. Before this function is called an external crystal oscillator must be available.

2.4.1.2 Actions

- Switches system clock to crystal oscillator (Startup time for XTAL specified in SFR XTCFG)

2.4.1.3 Prototype

void **Switch2XTAL**(void)

2.4.1.4 Resource Usage

Table 26 Switch2XTAL: Resources

Type	used or modified
Registers	R0, R1, R5, R6, R7
SFR	A, PSW, DIVIC, CFG0, IE
Stack	9 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.4.1.5 Execution Time

Table 27 Switch2XTAL: Execution Time

Value	typ	max	Unit	Conditions
Instruction cycles		3000		Clock= 12 MHz RC HF Oscillator DIVIC = 00_H XTCFG = 03_H

2.4.2 Switch2RC()

2.4.2.1 Description

This function switches the system clock from the external crystal oscillator back to the internal 12 MHz RC HF oscillator.

Note: If RF-Transmission is performed before this function is called, it is recommended to disable the PLL by clearing SFR bit RFC.1[ENFSYN] to reduce the current consumption.

2.4.2.2 Actions

- Switches system clock to 12 MHz RC HF oscillator

2.4.2.3 Prototype

void **Switch2RC**(void)

2.4.2.4 Resource Usage

Table 28 Switch2RC: Resources

Type	used or modified
Registers	none
SFR	A, PSW, CFG0, TCON2, LFDIV1, DSR, IE
Stack	3 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.4.2.5 Execution Information

Table 29 Switch2RC: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		15		

2.5 RF-Transmission

Before an RF-Transmission can be started, the VCO has to be tuned and the RF-Transmitter has to be initialized. VCO tuning can be done by calling **VCOTuning()**. Recalibration of the tuning curve is typically necessary when the VDDA changes by more than 800mV or the temperature changes by more than 70 degrees. Use **VCOTuning: Execution Time** to set up the RF-Transmitter. For an RF-Transmission **TransmitRF()** can be called.

2.5.1 VCOTuning()

2.5.1.1 Description

This function selects an appropriate tuning curve for the VCO and enables the PLL.

Note: If RF-Transmission is not performed immediately after this function is called, it is recommended to disable the PLL by clearing SFR bit RFC.1[ENFSYN] to reduce the current consumption.

*Note: For this function the external crystal oscillator must be used as system clock. (see **Switch2XTAL()**).*

2.5.1.2 Actions

- Selects appropriate tuning curve
- Enables PLL for a RF-Transmission

2.5.1.3 Prototype

signed char **VCOTuning**(void)

2.5.1.4 Return value

Table 30 VCOTuning: Return value

Register/ Address	Type	Description
R7	signed char	0: Success of VCO Tuning -1: VCO Tuning not successful, mean tuning curve set to ’0’ -2: Wrong clock source selected (external crystal oscillator required)

2.5.1.5 Resource Usage

Table 31 VCOTuning: Resources

Type	used or modified
Registers	R0, R1, R5, R6, R7
SFR	A, PSW, RFFSLD, RFC, RFVCO, DIVIC, IE
Stack	7 bytes (2 bytes for function call included)
RAM (IDATA)	$E0_H$ - $E5_H$

2.5.1.6 Execution Time

Table 32 VCOTuning: Execution Time

Value	typ	max	Unit	Conditions
Instruction cycles	1887	4541		typ: retuning (reduced to neighboring tuning curves) max: performing all 16 tuning curves (first tuning after RESET) Clock = XTAL DIVIC = 00_H

2.5.2 InitRF()

2.5.2.1 Description

This function prepares an RF-Transmission by configuring the corresponding Special Function Registers (SFR).

2.5.2.2 Actions

- Sets all corresponding SFRs for an RF-Transmission according to user input structure.

2.5.2.3 Prototype

void **InitRF**(struct RF_Config **idata** * **idata** *RFConfig*)

2.5.2.4 Parameters

Table 33 InitRF: Parameters

Register / Address	Type	Direction	Name	Description

R7	struct RF_Config idata *	Input	RFConfig	Pointer to a struct with the following definition:
				struct RF_Config
				{
				ENCODING Encoding;
				unsigned char dataLength;
				QUIESCENT Quiescent;
				INTMASK IntMask;
				DUTYCONTROL DutyControl;
				DUTYCYCLE DutyCycle;
				MODULATION Modulation;
				INVERSION Invert;
				FREQUENCY Frequency;
				OUTSTAGES OutStages;
				XCAPSHORT XcapShort;
				unsigned char xtal0;
				unsigned char xtal1;
				unsigned int baudrate;
				}
				
				Remark: The struct and the enum variables are defined in file PMA71xx_PMA51xx_Library.h.

Figure 4 RF Coder modes

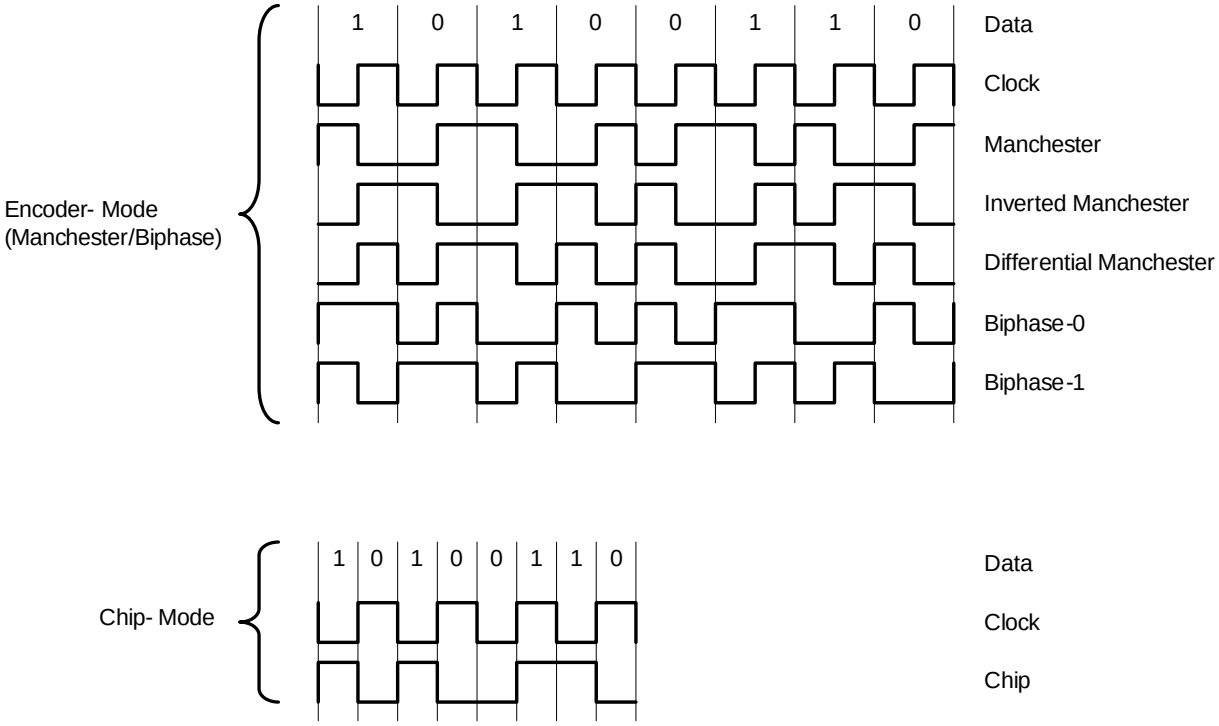


Table 34 Description of struct RF_Config

Type	Name	Description

enum {
MANCHESTER = 0,
INV_MANCHESTER = 1,
DIFF_MANCHESTER = 2,
BIPHASE_0 = 3,
BIPHASE_1 = 4,
CHIPS = 5
}
	Encoding	**Manchester / Biphase encoder settings.**
		Figure 4 shows the RF Encoder modes that can be choosen by the appropriate enum.
		
		MANCHESTER:
		Manchester encoding:
		’0’ is encoded as Low-to-High, ’1’ as High-to-Low transition.
		
		INV_MANCHESTER:
		Inverted Manchester encoding:
		’0’ is encoded as High-to-Low, ’1’ as Low-to-High transition.
		
		DIFF_MANCHESTER:
		Differential Manchester encoding:
		A '1' bit is indicated by making the first half of the signal equal to the last half of the previous bit's signal. A '0' bit is indicated by making the first half of the signal opposite to the last half of the previous bit's signal. In the middle of the bit-time there is always a transition, whether from high to low, or low to high.
		
		BIPHASE_0:
		Biphase 0 encoding:
		A transition occurs at the beginning of every bit interval. A ’0’ is represented by a second transition one half bit interval later. A ’1’ is represented by no second transition.
		
		BIPHASE_1:
		Biphase 1 encoding:
		A transition occurs at the beginning of every bit interval. A ’1’ is represented by a second transition one half bit interval later. A ’0’ is represented by no second transition.
		
		CHIPS:
		Chip data:
		It is also possible to send data with a user-defined encoding scheme, for example a preamble. For this the chip-mode can be used. The chip-mode sends each bit without encoding, but twice the data rate.

unsigned char	dataLength	Defines the number of bits of the Special Function Register RFD that are sent. **dataLength**: 0000.0xxx: Only the lower 3 bits are used. dataLength+1 bits are transmitted (MSB first).

Table 34 Description of struct RF_Config

Type	Name	Description

enum {
Q_ZERO = 0,
Q_ONE = 1
}
	Quiescent	Defines whether the RF-carrier is switched on or off, when PA is enabled and no data is available in Special Function Register RFD.
		
		Q_ZERO:
		RF-carrier is switched off, when PA is enabled and no data is available in RFD.
		
		Q_ONE:
		RF-carrier is switched on, when PA is enabled and no data is available in RFD.

enum {
MASK = 0,
NOMASK = 1
}
	IntMask	RF Encoder interrupt mask bit. This bit is used for both RF Encoder interrupts.
		RFS.0 [RFBF] RF Encoder Buffer Full
		RFS.1 [RFSE] RF Encoder Shift Register Empty
		MASK: Interrupt is not triggered
		NOMASK: Interrupt is triggered

enum {
DUTY_ON = 0,
DUTY_OFF = 1
}
	DutyControl	RF-Divider duty cycle control. The duty cycle control can be used to increase the power amplifier output power at low voltages.
		
		DUTY_ON:
		Duty cycle control used.
		
		DUTY_OFF:
		Duty cycle control not used.

enum {
DUTY_44 = 0,
DUTY_39 = 1,
DUTY_34 = 2,
DUTY_27 = 3
}
	DutyCycle	Duty cycle of the RF-Transmitter carrier. The duty cycle control can be used to increase the power amplifier output power at low voltages.

enum {
FSK = 0,
ASK = 1
}
	Modulation	Modulation type selection.
		
		ASK:
		ASK modulation used for RF-Transmission.
		
		FSK:
		FSK modulation used for RF-Transmission.

enum {
NO_INVERT = 0,
INVERT = 1
}
	Invert	RF-TX data inversion.
		
		NO_INVERT:
		TX data (written into RFD) not inverted.
		
		INVERT:
		TX data (written into RFD) inverted.

Table 34 Description of struct RF_Config

Type	Name	Description

enum {
RF_315MHZ = 0,
RF_434MHZ = 1,
RF_868MHZ = 2,
RF_915MHZ = 3
}
	Frequency	RF carrier frequency selection.
		
		RF_315MHZ:
		Carrier frequency is 315 MHz.
		
		RF_434MHZ:
		Carrier frequency is 434 MHz.
		
		RF_868MHZ:
		Carrier frequency is 868MHz.
		
		RF_915MHZ:
		Carrier frequency is 915 MHz.

enum {
STAGE_1 = 1,
STAGE_2 = 2,
STAGE_3 = 3
}
	OutStages	RF power amplifier output stage selection.
		
		STAGE_1:
		+5 dBm nominal transmit power into 50 ohm load at a supply voltage of 3.0 V.
		
		STAGE_2:
		+8 dBm nominal transmit power into 50 ohm load at a supply voltage of 3.0 V.
		
		STAGE_3:
		+10 dBm nominal transmit power into 50 ohm load at a supply voltage of 3.0 V.

enum {
DISABLE = 0,
ENABLE = 1
}
	XcapShort	Use of external capacitor and internal capacitor array.
		DISABLE: No external capacitor used. Internal capacitor array used.
		ENABLE: External capacitor used. Internal capacitor array not used.

unsigned char	xtal0	**FSK Low Frequency.** Pulling capacitor select for lower FSK modulation frequency if RFTX.5 [ASKFSK]=0. The capacitor array is binary weighted from FSKLOW7 = 1 ... 20pF (MSB) down to FSKLOW0 = 1 ... 156fF (LSB).

Table 34 Description of struct RF_Config

Type	Name	Description
unsigned char	xtal1	**FSK High Frequency / ASK Centre Frequency.** Pulling capacitor selection for upper FSK modulation frequency if RFTX.5 [ASKFSK]=0. ASK centre frequency fine tuning pulling capacitor selection if RFTX.5 [ASKFSK]=1. The capacitor array is binary weighted from FSKHASK7 = 1 ... 20pF (MSB) down to FSKHASK0 = 1 ... 156fF (LSB).
unsigned int	baudrate	**Baudrate for RF-Transmission.** *Note: If CHIPS is used for encoding, the datarate will be twice the selected one.*

2.5.2.5 Resource Usage

Table 35 InitRF: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, PSW, RFENC, RFTX, RFFSPLL, XTAL0, XTAL1, TMOD2, TCON2, TH3, TL3, TH2, TL2, IE, B
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	$C1_H$ - $C4_H$

2.5.2.6 Execution Information

Table 36 InitRF: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		1020		

2.5.2.7 Code Example

See **Chapter 2.5.4**.

2.5.3 TransmitRF()

2.5.3.1 Description

This function transmits a given data packet via RF according to the preset values of the corresponding SFRs.

Note: For this function the external crystal oscillator must be used as system clock.

Note: The RF-Transmitter remains enabled after this function returns. It is recommended to disable the RF-Transmitter to save energy consumption.

2.5.3.2 Actions

- Transmits a given data packet via RF.

2.5.3.3 Prototype

signed char **TransmitRF**(unsigned char **idata** **DataArray*,
unsigned char *BlockLength*)

2.5.3.4 Parameters

Table 37 TransmitRF: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char idata*	Input	DataArray	iData pointer to data block that will be transmitted.
R5	unsigned char	Input	BlockLength	Length of data block in bytes

2.5.3.5 Return value

Table 38 TransmitRF: Return value

Register/ Address	Type	Description
R7	signed char	0: Success -1: BlockLength = 0

2.5.3.6 Resource Usage

Table 39 TransmitRF: Resources

Type	used or modified
Registers	R0, R5, R7
SFR	A, PSW, TCON2, RFS, RFD, IE, CFG0
Stack	3 bytes
RAM (IDATA)	---

2.5.3.7 Execution Information

Table 40 TransmitRF: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		13 + 10*X + 1334*X (sending)		X...Blocklength Systemclock = XTAL Baudrate = 10kBaud

2.5.3.8 Code Example

See **Chapter 2.5.4**.

2.5.4 Code Example

```
// Library function prototypes
// struct RF_Config, and enum variables are defined in <PMA71xx_PMA51xx_Library.h>
/ Register definition
#include "Reg_PMA71xx_PMA51xx.h"
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"
// Definition of TX data length
#define RF_TX_LEN 10

void main()
{
        // Return value of TransmitRF(..) is stored in StatusByte
        signed char StatusByte;
        // TX data array
        unsigned char idata DataArray[RF_TX_LEN] = {0,1,2,3,4,5,6,7,8,9};
        // variable for BlockLength
        unsigned char BlockLength = RF_TX_LEN;

        // delcare and initialize a variable of struct RF_Config
        struct RF_Config idata myRF_Config;
        myRF_Config.Quiescent = 0;
        myRF_Config.IntMask = 0;
        myRF_Config.DutyControl = DUTY_OFF;
        myRF_Config.XcapShort = DISABLE;
        myRF_Config.Encoding = MANCHESTER;
        myRF_Config.dataLength = 7;
        myRF_Config.DutyCycle = DUTY_27;
        myRF_Config.Modulation = ASK;
        myRF_Config.Invert = NO_INVERT;
        myRF_Config.Frequency = RF_315MHZ;
        myRF_Config.OutStages = STAGE_3;
        myRF_Config.xtal0 = 0x28;
        myRF_Config.xtal1 = 0x11;
        myRF_Config.baudrate = 9600;

        // call Library function InitRF(..)
        InitRF(&myRF_Config);

        // change clock to external crystal oscillator
        Switch2XTAL();

        // selects an appropriate tuning curve for the VCO and enables the PLL
        VCOTuning();

        // RF Transmission of TX data array
        StatusByte=TransmitRF(DataArray, BlockLength);
        if(!StatusByte){
                // .. RF Transmission successful
        }
        else{
                // .. RF Transmission failed
        }

        // disable PLL and PA after end of RF Transmission
        RFC = 0;
}
```

2.6 Interval Timer

2.6.1 CalibrateIntervalTimer()

2.6.1.1 Description

This function initiates a calibration of the interval timer precounter (ITPL and ITPH). The function works with both clock sources (12MHz RC Clock and Crystal clock).

Note: Due to accuracy reasons it is recommended to call this function with the crystal oscillator selected as system clock.

Note: The interval timer postcounter register ITPR has to be set separately (please refer to **"[1]" on Page 82***).*

In the case where the crystal oscillator is used, the crystal frequency divided by 2 has to be passed as a function parameter. If the value passed is not within the range of 9 to 10 MHz a default clock frequency of 9.5 MHz is assumed for the tuning.

This function also automatically calibrates the LF On/Off Timer precounter (SFR LFOOTP) to 50 ms as well as the interval timer.

2.6.1.2 Actions

- Calibrates the interval timer precounter (SFR ITPL, SFR ITPH)
- Calibrates the LF On/Off timer precounter (SFR LFOOTP)

2.6.1.3 Prototype

signed char **CalibrateIntervalTimer**(unsigned char *WU_Frequency*,
unsigned long **idata*** *Xtal_Frequency*)

2.6.1.4 Parameters

Table 41 CalibrateIntervalTimer: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	WU_Frequency	Base frequency of the interval timer precounter [Hz] 1dec: 1 Hz (precounter time ~1000 ms) 2dec: 2 Hz (precounter time ~500 ms) ... 19dec: 19 Hz (precounter time ~53 ms) 20dec: 20 Hz (precounter time ~50 ms).
R5	unsigned long idata*	Input	Xtal_Frequency	Pointer to a long value that equals half of the used crystal frequency (XTAL/2). Value range: 9 - 10 MHz Default: 9.5 MHz

2.6.1.5 Return value

Table 42 CalibrateIntervalTimer: Return value

Register/ Address	Type	Description
R7	signed char	0: Success -1: Xtal_Frequency parameter out of range (default used) -2: WU_Frequency out of range

2.6.1.6 Resource Usage

Table 43 CalibrateIntervalTimer: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, TCON2, TMOD2, TH2, TL2, CFG0, ITPL, ITPH, LFOOTP, IE
Stack	7 bytes (2 bytes for function call included)
RAM (IDATA)	$E9_H$, EA_H

2.6.1.7 Execution Time

Table 44 CalibrateIntervalTimer: Execution Time

Value	typ	max	Unit	Conditions
Instruction Cycles		3034		DIVIC = 00_H

2.6.1.8 Code Example

In this example the IntervalTimer is calibrated to wakeup the PMA51xx/PMA71xx from POWER DOWN mode every 100 ms. After every wakeup PP2 is toggled and the PMA51xx/PMA71xx is set into POWER DOWN mode again.

```
// Register definition
#include "Reg_PMA71xx_PMA51xx.h"

// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

// Definition of some used bits
#define BIT_WUP         0x02
#define BIT_WU_IT       0x01

void main()
{
        unsigned long idata Xtal_Frequency_2 = 9040000;
        unsigned char WU_Frequency = 10;
        signed char StatusByte;

        // Decision: Reset or Wakeup entry:
        // ================================
        if ((DSR & BIT_WUP)== 0) {

                // Reset entry: initial entry of the Program,
                // ============ no wakeup is pending (Flag WUP)
```

```
                P1Dir &= ~(0x04); // set PP2 to output

                // Wakeup Interval configuration:
                // ==============================
                // Dependent on accuracy criteria as well as time and voltage drift
                // recalibration is recommended.

                // Switch from 12MHz Hi RC oscillator to external crystal
                // to get higher accuracy.
                Switch2XTAL();

                // Interval timer is calibrated to wake up the PMA51xx/PMA71xx every
                // 100 ms
                StatusByte = CalibrateIntervalTimer(WU_Frequency, &Xtal_Frequency_2);
                if(!StatusByte){
                        // .. Interval timer calibration successful
                }
                else if(StatusByte==-1){
                        // .. Xtal_Frequency parameter out of range (default used)
                }
                else if(StatusByte==-2){
                        // .. WU_Frequency out of range
                }
                // Set interval timer post counter
                ITPR=1;
        }
        else {
                // Wakeup entry:
                // =============
                // interval timer (WUF clear on read)
                if((WUF & BIT_WU_IT) == BIT_WU_IT)
                {
                        // toggle PP2
                        P1Out ^= 0x04;
                }
        }
        // go into Powerdown mode
        PowerDown();
}
```

2.7 LF Receiver

The functions LFSensitivityCalibration() and LFBaudrateCalibration() are only relevant to product derivates that support the LF receiver feature in the PMA51xx and PMA71xx product family.

2.7.1 LFSensitivityCalibration()

2.7.1.1 Description

This function is used to set the LF sensitivity to the value specified in the datasheet.

2.7.1.2 Actions

- Measures voltage and temperature and calculates the Carrier Detector Threshold Level for the actual temperature and supply voltage. For this calculation measurement results from the production test, which are stored in the FLASH configuration sector, are used.
- Calculated value is written to SFR LFRX0.7-4 [CDETT3-0]

2.7.1.3 Prototype

signed char **LFSensitivityCalibration**(void)

2.7.1.4 Return value

Table 45 LFSensitivityCalibration: Return value

Register/ Address	Type	Description
R7	signed char	StatusByte: 0: Success -1: Failed, sensitivity set to device specific default value

2.7.1.5 Resource Usage

Table 46 LFSensitivityCalibration: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, ADCOFF, ADCC0, ADCC1, ADCDL, ADCDH, ADCM, ADCS, CFG0, CFG1, CFG2, DIVIC, DPL, DPH, IE, LFRX0
Stack	16 bytes (2 bytes for function call included)
RAM (IDATA)	$C0_H$ - $E7_H$

2.7.1.6 Execution Time

Table 47 LFSensitivityCalibration: Execution Time

Value	typ	max	Unit	Conditions
Instruction Cycles		1932		DIVIC = 00_H

2.7.1.7 Code Example

```
// Library include
#include "PMA71xx_PMA51xx_Library.h"

void main (void){
        signed char StatusByte;
        // .. add your source code here

        // .. do the settings for LF Receiver
        // Registers:          LFRXS, LFRXD, LFSYN0, LFSYN1, LFSYNCFG, LFCDFlt, LFDIV0,
        //                     LFDIV1, LFCDM, LFRX0, LFRX1, LFP0L, LFP0H, LFOOTP, LFOOT,
        //                     LFPCFG, LFP1L, LFP1H, LFRXC

        // Register LFRX0.7-4[CDETT] is set by the following Library function
        // NOTE: Call this function every time the temperature changes by about 20°C
        //             or the voltage changes by about 200mV
        StatusByte=LFSensitivityCalibration();
        if(StatusByte==-1){
                // function failed (Temperature or Voltage out of range): sensitivity
                // set to device specific default value
        }
        else{
                // function executed correctly: sensitivity set to device specific
                // value according to actual temperature and voltage
        }
        // .. add your source code here
}
```

2.7.2 LFBaudrateCalibration()

2.7.2.1 Description

Calling this function calibrates the LF baudrate divider using the crystal oscillator, thus eliminating the impact of the drift and the offset of the 12 MHz RC HF oscillator on the LF baudrate accuracy.

$$\text{LFDIV1/0} = \frac{\text{Fsource}}{16 \times \text{baudrate}}$$

F_{SOURCE}: External crystal oscillator resonance frequency divided by 2.

The crystal frequency divided by 2 has to be passed as a function parameter. If the value passed is not within the range of 9 to 10 MHz the function uses 9.5 MHz as a default value for the crystal frequency:

Note: For this function the crystal oscillator should be used as system clock.

2.7.2.2 Actions

- Switches system clock to crystal oscillator if required (Delay time speficied in SFR XTCFG)
- Sets SFR LFDIV1 and SFR LFDIV0 according to the current frequency of the 12 MHz RC HF Oscillator

2.7.2.3 Prototype

signed char **LFBaudrateCalibration**(unsigned int *baudrate*,
unsigned long **idata*** *Xtal_Frequency*)

2.7.2.4 Parameters

Table 48 LFBaudrateCalibration: Parameters

Register / Address	Type	Direction	Name	Description
R6 (MSB) R7 (LSB)	unsigned int	Input	baudrate	Baudrate Value range: 2000 - 4000 baud e.g.: 3900dec..3900 baud
R5	unsigned long idata*	Input	Xtal_Frequency	Pointer to a long value that equals half of the used crystal frequency (XTAL/2). Value range: 9 - 10 MHz Default: 9.5 MHz

2.7.2.5 Return value

Table 49 LFBaudrateCalibration: Return value

Register/ Address	Type	Description
R7	signed char	StatusByte: 0: Success -1: Xtal_Frequency parameter out of range (default used) -2: Baudrate out of range

2.7.2.6 Resource Usage

Table 50 LFBaudrateCalibration: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, CFG0, LFDIV1, TH2, TL2, TMOD2, TCON2, IE
Stack	4 bytes (2 bytes for function call included)
RAM (IDATA)	EA_H

2.7.2.7 Execution Time

Table 51 LFBaudrateCalibration: Execution Time

Value	typ	max	Unit	Conditions
Instruction Cycles		1180		Clock= XTAL DIVIC = 00_H

2.7.2.8 Code Example

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main()
{
        unsigned long idata Xtal_Frequency_2 = 9040000;
        unsigned int Baudrate = 2000;
        signed char StatusByte;

        // .. add your source code here

        // Calibrate LF Baudrate for 2000 baud
        StatusByte=LFBaudrateCalibration(Baudrate, &Xtal_Frequency_2);
        if(!StatusByte){
                // .. LF Baudrate calibration successful
        }
        else if(StatusByte==-1){
                // .. Xtal_Frequency parameter out of range (default used)
        }
        else if(StatusByte==-2){
                // .. LF Baudrate out of range
        }

        // .. add your source code here
}
```

2.8 Division

Two Library functions are implemented for efficient division of long (32 bit) and integer (16 bit) values.

2.8.1 ULongDivision() (32 bit : 32 bit)

2.8.1.1 Description

This function divides the unsigned long value (32 bit) *Dividend* by the unsigned long value (32 bit) *Divisor*.

2.8.1.2 Actions

- Performs 32 bit division

2.8.1.3 Prototype

unsigned long **ULongDivision**(unsigned long **idata** * *Dividend*,
unsigned long **idata** * *Divisor*)

2.8.1.4 Parameters

Table 52 ULongDivision: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned long idata*	Input	Dividend	iData Pointer to 32 bit Dividend.
R5	unsigned long idata*	Input	Divisor	iData Pointer to 32 bit Divisor

2.8.1.5 Return value

Table 53 ULongDivision: Return value

Register/ Address	Type	Description
R4 (MSB), R5, R6, R7 (LSB)	signed long	32 bit Quotient of division (Dividend/Divisor)

2.8.1.6 Remainder

Table 54 ULongDivison: Remainder

Register/ Address	Type	Description
R0 (MSB), R1, R2, R3 (LSB)	signed long	32 bit remainder of division (Dividend/Divisor)

2.8.1.7 Resource Usage

Table 55 ULongDivision: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, IE
Stack	5 bytes (2 bytes for function call included)
RAM (IRATA)	---

2.8.1.8 Execution Information

Table 56 ULongDivision: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		760		

2.8.1.9 Code Example

This example shows how to call the Library function ULongDivision(..) and how to get the remainder of the division in C. It is important to switch the register bank directly after the function call to ensure that the remainder stored in registers R0 to R3 on bank 0 is not overwritten. Then the remainder can be read by dereferencing the idata pointer. Finally the register bank is switched back to bank 0.

```
// Register definition
#include "Reg_PMA71xx_PMA51xx.h"

// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main(){

        unsigned long idata Dividend = 0x5B05B057;
        unsigned long idata Divisor =  0x12345678;

        // idata pointer to R0
        unsigned long idata * pRemainder = 0;
        unsigned long Result;
        unsigned long Remainder;

        // .. add your source code here

        // Divide two unsigned long values
        Result = ULongDivision(&Dividend, &Divisor);

        // switch to register bank 1
        RS0=1;

        // Read Remainder from R0-R3
        Remainder = *pRemainder;

        // switch back to register bank 0
        RS0=0;

        // .. add your source code here
}
```

2.8.2 UIntDivision() (16 bit : 16 bit)

2.8.2.1 Description

This function divides the unsigned int value (16 bit) *Dividend* by the unsigned int value (16 bit) *Divisor*.

2.8.2.2 Actions

- Performs 16 bit division

2.8.2.3 Prototype

unsigned int **UIntDivision**(unsigned int *Dividend*,
unsigned int *Divisor*)

2.8.2.4 Parameters

Table 57 UIntDivision: Parameters

Register / Address	Type	Direction	Name	Description
R6(MSB) R7(LSB)	unsigned int	Input	Dividend	16 bit Dividend
R4(MSB) R5(LSB)	unsigned int	Input	Divisor	16 bit Divisor

2.8.2.5 Return value

Table 58 UIntDivision: Return value

Register/ Address	Type	Description
R6 (MSB), R7 (LSB)	signed int	16 bit Quotient of division (Dividend/Divisor)

2.8.2.6 Remainder

Table 59 UIntDivison: Remainder

Register/ Address	Type	Description
R4 (MSB), R5 (LSB)	signed int	16 bit Remainder of division (Dividend/Divisor)

2.8.2.7 Resource Usage

Table 60 UIntDivision: Resources

Type	used or modified
Registers	R0, R4, R5, R6, R7
SFR	A, B, PSW, IE
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.8.2.8 Execution Information

Table 61 UIntDivision: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		150		

2.8.2.9 Code Example

This example shows how to call the Library function UIntDivision(..) and how to get the remainder of the division in C. It is important to switch the register bank directly after the function call to ensure that the remainder stored in registers R4 to R5 on bank 0 is not overwritten. Then the remainder can be read by dereferencing the idata pointer. Finally the register bank is switched back to bank 0.

```
// Register definition
#include "Reg_PMA71xx_PMA51xx.h"

// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main(){

        unsigned int idata Dividend = 0x5B03;
        unsigned int idata Divisor =  0x1234;

        // idata pointer to R4
        unsigned int idata * pRemainder = 4;
        unsigned int Result;
        unsigned int Remainder;

        // .. add your source code here

        // Divide two unsigned int values
        Result = UIntDivision(Dividend, Divisor);

        // switch to register bank 1
        RS0=1;

        // read Remainder from R4-R5
        Remainder = *pRemainder;

        // switch back to register bank 0
        RS0=0;
        // .. add your source code here
}
```

2.9 CRC Calculation

The Library includes two functions for CRC calculation with two different polynomials.

2.9.1 CRC8Calculation()

2.9.1.1 Description

This function calculates the CRC-8 checksum for a memory area in RAM using a fixed polynomial (x^8+x^2+x+1).

2.9.1.2 Actions

- Calculates CRC-8 with polynomial (x^8+x^2+x+1)

2.9.1.3 Prototype

unsigned char **CRC8Calculation**(unsigned char *Preload*,
unsigned char **idata** * *BlockStart*,
unsigned char *BlockLength*)

2.9.1.4 Parameters

Table 62 CRC8Calculation: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Preload	Preload Value for the CRC Calculation. Usually set to FF_H.
R5	unsigned char idata*	Input	BlockStart	iData Pointer to first byte of the data to be used for calculating the checksum.
R3	unsinged char	Input	BlockLength	Length of block that is used for calculation of the checksum, starting with *BlockStart

2.9.1.5 Return value

Table 63 CRC8Calculation: Return value

Register/ Address	Type	Description
R7	unsigned chart	Calculated CRC8 checksum.

2.9.1.6 Resource Usage

Table 64 CRC8Calculation: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R7
SFR	A, PSW, IE
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.9.1.7 Execution Information

Table 65 CRC8Calculation: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		10+x*17		x = Number of bytes

2.9.1.8 Code Example

In this example the CRC8 with polynomial (x^8+x^2+x+1) is calculated over 8 bytes of data. A preload of FF_H is used. To check if the CRC value is correct, the 8 bit CRC value is appended to the data array and the CRC8 is calculated over the 8 bytes of data + 1 byte CRC. The result must be zero if the CRC value is correct.

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

#define CRC_BLOCK_LEN 8

void main()
{
        unsigned char CRC_Value;
        unsigned char Preload = 0xff;
        unsigned char idata DataBlock[CRC_BLOCK_LEN+1] = {1,2,3,4,5,6,7,8,0};
        unsigned char BlockLength = CRC_BLOCK_LEN;

        //.. add your source code here

        // CRC8 is calculated over DataBlock (8 bytes)
        CRC_Value = CRC8Calculation(Preload, DataBlock, BlockLength);

        // append CRC value at the end of the data array
        DataBlock[8] = CRC_Value;

        // check if CRC value is correct
        CRC_Value = CRC8Calculation(Preload, DataBlock, BlockLength+1);

        if(!CRC_Value){
                // .. CRC_Value is correct
        }
        else{
                // .. CRC_Value is not correct
        }
        //.. add your source code here
}
```

2.9.2 CRC8_410()

2.9.2.1 Description

This function calculates the CRC-8 checksum for a memory area in RAM using a fixed polynomial (x^8+x^4+x+1).

2.9.2.2 Actions

- Calculates CRC-8 with polynomial (x^8+x^4+x+1)

2.9.2.3 Prototype

unsigned char **CRC8_410**(unsigned char *Preload*,
unsigned char **idata** * *BlockStart*,
unsigned char *BlockLength*)

2.9.2.4 Parameters

Table 66 CRC8_410: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Preload	Preload value for the CRC Calculation. Usually set to FF_H.
R5	unsigned char idata*	Input	BlockStart	iData Pointer to first byte of the data to be used for calculating the checksum.
R3	unsinged char	Input	BlockLength	Length of block that is used for calculation of the checksum, starting with *BlockStart

2.9.2.5 Return value

Table 67 CRC8_410: Return value

Register/ Address	Type	Description
R7	unsigned char	Calculated CRC8 checksum.

2.9.2.6 Resource Usage

Table 68 CRC8_410: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R7
SFR	A, PSW, IE
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.9.2.7 Execution Information

Table 69 CRC8_410: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		10+x*20		x = Number of Bytes

2.9.2.8 Code Example

In this example the CRC8 with polynomial (x^8+x^4+x+1) is calculated over 8 bytes of data. A Preload of FF_H is used. To check if the CRC value is correct, the 8 bit CRC value is appended to the data array and the CRC8 is calculated over the 8 bytes of data + 1 byte CRC. The result must be zero, if the CRC value is correct.

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

#define CRC_BLOCK_LEN 8


void main()
{
        unsigned char CRC_Value;
        unsigned char Preload = 0xff;
        unsigned char idata DataBlock[CRC_BLOCK_LEN+1] = {1,2,3,4,5,6,7,8,0};
        unsigned char BlockLength = CRC_BLOCK_LEN;

        //.. add your source code here

        // CRC8 is calculated over DataBlock (8 bytes)
        CRC_Value = CRC8_410(Preload, DataBlock, BlockLength);

        // append CRC value at the end of the data array
        DataBlock[8] = CRC_Value;

        // check if CRC value is correct
        CRC_Value = CRC8_410(Preload, DataBlock, BlockLength+1);

        if(!CRC_Value){
                // .. CRC_Value is correct
        }
        else{
                // .. CRC_Value is not correct
        }

        //.. add your source code here
}
```

2.10 AES128 Encryption / Decryption

AES Encryption / Decryption is supported by the PMA51xx/PMA71xx Library for block sizes of 128 bit.

2.10.1 AES128Encrypt()

2.10.1.1 Description

This function performs an AES Encryption with 128-Bit data and a 128-Bit key.

2.10.1.2 Actions

- Encrypts 128-Bit data with a 128-Bit key using AES.

2.10.1.3 Prototype

void **AES128Encrypt**(const unsigned char **idata*** *InputData*,
unsigned char **idata*** *OutputData*,
const unsigned char **idata*** *Key*)

2.10.1.4 Parameters

Table 70 AES128Encrypt: Parameters

Register / Address	Type	Direction	Name	Description
R7	const unsigned char idata*	Input	InputData	Pointer to 128-Bit data for AES encryption.
R5	unsigned char idata*	Output	OutputData	Pointer to 128-Bit AES encryption result.
R3	const unsinged char idata*	Input	Key	Pointer to 128-Bit AES key.

2.10.1.5 Resource Usage

Table 71 AES128Encrypt: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPH, DPL, IE
Stack	13 bytes (2 bytes for function call included)
RAM (IDATA)	$C1_H$ - $E0_H$

2.10.1.6 Execution Information

Table 72 AES128Encrypt: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		4144		Clock= 12MHz RC RF Oscillator DIVIC = 00_H

2.10.1.7 Code Example

See **Chapter 2.10.3**.

2.10.2 AES128Decrypt()

2.10.2.1 Description

This function performs an AES Decryption with 128-Bit data and a 128-Bit key.

2.10.2.2 Actions

- Decrypts 128-Bit data with a 128-Bit key using AES.

2.10.2.3 Prototype

void **AES128Decrypt**(const unsigned char **idata*** *InputData*,
unsigned char **idata*** *OutputData*,
const unsigned char **idata*** *Key*)

2.10.2.4 Parameters

Table 73 AES128Decrypt: Parameters

Register / Address	Type	Direction	Name	Description
R7	const unsigned char idata*	Input	InputData	Pointer to 128-Bit data for AES decryption.
R5	unsigned char idata*	Output	OutputData	Pointer to 128-Bit AES decryption result.
R3	const unsinged char idata*	Input	Key	Pointer to 128-Bit AES key.

2.10.2.5 Resource Usage

Table 74 AES128Decrypt: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPH, DPL, IE

Type	used or modified
Stack	13 bytes (2 bytes for function call included)
RAM (IDATA)	$C1_H$ - $E0_H$

2.10.2.6 Execution Information

Table 75 AES128Decrypt: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		6110		Clock= High RC DIVIC = 00_H

2.10.2.7 Code Example

See **Chapter 2.10.3**.

2.10.3 Code Example

This example shows how to encrypt and decrypt a data block of 128 bits using the PMA51xx/PMA71xx Library functions **AES128Encrypt()** and **AES128Decrypt()**.

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main()
{
        const unsigned char idata InputData[16] = {1,2,3,4,5,6,7,8,9,
                                                   10,11,12,13,14,15,16};
        const unsigned char idata Key[16] = {10,20,30,40,50,60,70,80,
                                             90,100,110,120,130,140,150,160};
        unsigned char idata EncryptedData[16];
        unsigned char idata DecryptedData[16];

        // .. add your source code here

        // Encrypt InputData with AES128 using a predefined Key
        AES128Encrypt(InputData, EncryptedData, Key);

        // .. add your source code here

        // Decrypt InputData with AES128 using a predefined Key
        AES128Decrypt(EncryptedData, DecryptedData, Key);

        // .. add your source code here
}
```

2.11 Data FLASH access

The PMA51xx/PMA71xx Library provides two functions for directly accessing certain FLASH areas, *User Data Sector I* and *User Data Sector II* (**EraseUserDataSector()** and **WriteUserDataSectorLine()**). The FLASH organisation and the *User Data Sectors* are described in detail in **"FLASH organization" on Page 10**.

Note: The FLASH areas can only be erased sector-wise. It is not possible to erase only a single byte or word line.

Note: The write access to the FLASH is done by writing a single word line. It is not possible to write single bytes. Before a word line is written into one of the FLASH User Data Sectors the appropriate word line must have been erased. If a word line that already contains data is overwritten, the result is undefined.

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()** or **WriteUserDataSectorLine()**) when using the EEPROM-Emulation.*

2.11.1 EraseUserDataSector()

2.11.1.1 Description

This function erases the FLASH *User Data Sectors* located at FLASH address 5780_H -- $57FF_H$ and 5800_H -- $587F_H$ if the Lockbyte 3 is not set. If Lockbyte 3 is set this function will return -1 without any action.

Note: For this function the 12 MHz RC HF oscillator must be used as system clock

*Note: The application software has to ensure that the FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{BatFL} and Erase Cycle ER_{FLData} are within specified range (see **"[1]" on Page 82**).*
This function returns -1 and has no effect if executed in DEBUG mode.

2.11.1.2 Actions

- Erases the selected FLASH *User Data Sector(s)*

2.11.1.3 Prototype

signed char **EraseUserDataSector**(unsigned char *Sector*)

2.11.1.4 Parameter

Table 76 EraseUserDataSector: Parameter

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Sector	Sector Byte - 0000.0xx0 xx: 01 - User Data Sector 1 xx: 10 - User Data Sector 2 xx: 11 - User Data Sector 1 + 2

2.11.1.5 Return value

Table 77 EraseUserDataSector: Return value

Register/ Address	Type	Description
R7	signed char	0: success -1: failed

2.11.1.6 Resource Usage

Table 78 EraseUserDataSector: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, CRC0, CRC1, CRCC, CRCD, DIVIC, TCON2, TMOD2, TH2, TL2, TH3, TL3, FCS, FCSERM, FCPP0, FCTKAS, FCSP, IE
Stack	9 bytes (2 bytes for function call included)
RAM (IDATA)	$E0_H$ - $0E6_H$, $E9_H$, EA_H

2.11.1.7 Execution Information

Table 79 EraseUserDataSector: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		204000		

2.11.1.8 Code Example

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main()
{
        signed char StatusByte;
        unsigned char Sector = 0x06;

        // .. add your source code here

        // Erase FLASH User Data Sector I + II (0x06)
        StatusByte=EraseUserDataSector(Sector);

        if(!StatusByte){
                // .. Sector successfully erased
        }
        else{
                // .. Sector erase failed
        }

        // .. add your source code here
}
```

2.11.2 WriteUserDataSectorLine()

2.11.2.1 Description

This function writes one line in one of the two FLASH *User Data Sectors* located at FLASH address 5780_H -- $587F_H$ and 5800_H -- $587F_H$ if the Lockbyte 3 is not set. If Lockbyte 3 is set this function will return -1 without any action.

The written data is verified after the programming. In case the verification fails this function will return -1.

The write access to the FLASH is done by writing a word line. It is not possible to write single bytes. Due to the FLASH organisation there are 8 word lines (User Data Secor Lines) that can be written. Therefore the Library function WriteUserDataSectorLine(..) has only 8 possible *Startaddresses* (see **Table 80**). If an invalid *Startaddress* is used the function will return -1.

*Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{batFL} and Erase Cycle ER_{FLData} within specified range (see **"[1]" on Page 82**.)*
This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF oscillator must be used as system clock

Note: If data is written to a word line that already contains data, the result is undefined.

2.11.2.2 Actions

- Writes one 32 Byte FLASH Line of one of the two FLASH *User Data Sectors*

2.11.2.3 Prototype

signed char **WriteUserDataSectorLine** (unsigned int *Startaddress*,
unsigned char **idata** * *WrData*)

2.11.2.4 Parameters

Table 80 WriteUserDataSectorLine: Parameters

Register / Address	Type	Direction	Name	Description
R7, R6	unsigned int	Input	Startaddress	Possible Startaddresses: 5780_H: FLASH Line 0 $57A0_H$: FLASH Line 1 $57C0_H$: FLASH Line 2 $57E0_H$: FLASH Line 3 5800_H: FLASH Line 4 5820_H: FLASH Line 5 5840_H: FLASH Line 6 5860_H: FLASH Line 7
R5	unsigned char idata*	Input	WrData	iData Pointer to first Byte of the 32 Byte Data array that is going to be written to the FLASH Line.

2.11.2.5 Return value

Table 81 WriteUserDataSectorLine: Return value

Register/ Address	Type	Description
R7	signed char	0: success -1: failed (Wrong start address or data verification failed. Has an already written FLASH Line been rewritten ?)

2.11.2.6 Resource Usage

Table 82 WriteUserDataSectorLine: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, CFG1, DIVIC, CRCD, CRCC, CRC0, CRC1, DPL, DPH, FCSERM, FCTKAS, FCPP0, FCPP1, FCSP, FCS, TCON2, TMOD2, TH3, TL3, TH2, TL2, , IE

Type	used or modified
Stack	9 bytes (2 bytes for function call included)
RAM (IDATA)	$E7_H$ - EA_H

2.11.2.7 Execution Information

Table 83 WriteUserDataSectorLine: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		4400		

2.11.2.8 Code Example

```
// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main()
{
        unsigned char idata WrData[32];
        signed char StatusByte;
        unsigned char Sector = 0x02;
        unsigned char i = 0;

        // Init WrData with "dummy data"
        for(i=0;i<32;i++) WrData[i] = i;

        // .. add your source code here

        // Erase Sector I (0x02)
        StatusByte=EraseUserDataSector(Sector);

        if(!StatusByte){
                // .. Sector successfully erased
        }
        else{
                // .. Sector erase failed
        }

        // .. add your source code here

        StatusByte=WriteUserDataSectorLine(0x57A0, WrData);
        if(!StatusByte){
                // .. word line successfully written
        }
        else{
                // .. word line write access failed
        }
        // .. add your source code here
}
```

2.12 FLASH Lock

Two possibilities to lock PMA51xx/PMA71xx FLASH *User Data Sectors*

1) Write and lock the FLASH *Code Sector* (Lockbyte 2) and *User Data Sectors* (Lockbyte 3) during program download.

2) Write and lock the FLASH *Code Sector* (Lockbyte 2) during program download and lock the *User Data Sectors* (Lockbyte 3) by the dedicated Library function **SetLockbyte3()** or by writing $D1_H$ to $57FF_H$ using **WriteUserDataSectorLine()**.

Unlock FLASH User Data Sectors

If the *User Data Sectors* have been locked, they can only be unlocked by setting the PMA51xx/PMA71xx into Test Mode and erasing the the FLASH *Code Sector* and *User Data Sectors*.

2.12.1 SetLockbyte3()

2.12.1.1 Description

This function sets the Lockbyte 3 which protects the *User Data Sectors*.

A reset will be triggered in case of a succesfully set Lockbyte.

Attention: This function shows only effect if the Lockbyte 2 that protects the Code Sector is set.

Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{batFL} and Erase Cycle ER_{FLData} within specified range (see **"[1]" on Page 82***.)*
This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF oscillator should be used as system clock

2.12.1.2 Actions

- Sets the Lockbyte 3
- Verifies Lockbyte 3 write access
- Resets PMA51xx/PMA71xx to activate the Lock

2.12.1.3 Prototype

signed char **SetLockbyte3**(void)

2.12.1.4 Return value

Table 84 SetLockbyte3: Return value

Register/ Address	Type	Description
R7	signed char	0: success (only theoretical value because PMA51xx or PMA71xx is reset and return value can not be read.) -1: failed

2.12.1.5 Resource Usage

Table 85 SetLockbyte3: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, CFG1, DIVIC, CRCD, CRCC, CRC0, CRC1, DPL, DPH, DSR, FCSERM, FCTKAS, FCPP0, FCPP1, FCSP, FCS, TCON2, TMOD2, TH3, TL3, TH2, TL2, IE
Stack	14 bytes (2 bytes for function call included)
RAM (IDATA)	---

2.12.1.6 Execution Information

Table 86 SetLockbyte3: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		235500		

2.13 EEPROM Emulation

The PMA51xx/PMA71xx Library supports several functions to emulate an EEPROM with the FLASH *User Data Sector I* and *II*.

This emulation supports an EEPROM-like behaviour. 31 bytes of data can be read, written and erased byte-, integer- (2 bytes), long- (4 bytes) or stringwise (up to 31 bytes). Data written to the emulated EEPROM is non-volatile. The entire data management of the emulated EEPROM is done by the PMA51xx/PMA71xx Library functions.

For the best performance ensure that all bytes are written with a single Library function call. Use **Wr_EEByte()** for bytewise, **Wr_EEInt()** for integerwise, **Wr_EELong()** for longwise and **Wr_EEString()** for stringwise write accesses.

If EEPROM Emulation is used, ensure that data FLASH access functions ***EraseUserDataSector()*** *or* , ***WriteUserDataSectorLine()*** and FLASH *User Data Sector* Lock function ***SetLockbyte3()*** are not called.

2.13.1 Wakeup pin sampling

The write functions for the EEPROM Emulation support sampling of the wakeup pins. This functionality can be used to monitor the external wakeups during EEPROM Emulation write access. **Figure 5** shows how the wakeup flags are stored into one byte. Wakeup pins are sampled every 5 ms and appended to the sampling array. The sampling array has to be 21 bytes long. Ensure that the 12 MHz RC HF oscillator is used.

Note: If an extra erase is necessary during write functions, the values of the first erase of the sample array are overwritten. The user does not need to reserve extra bytes for the second erase.

Figure 5 Bit order of a single wakeup pin sample

	7	6	5	4	3	2	1	0
Pin	PP9	PP8	PP7	PP6	PP4	PP3	PP2	PP1
External Wakeup	WU7	WU6	WU5	WU4	WU3	WU2	WU1	WU0

2.13.2 EEPROM_Init()

2.13.2.1 Description

This function prepares the emulated EEPROM data area. It has to be called before using the EEPROM Read- and Write functions, otherwise the results will be undefined.

This function can be used for port sampling:
If the parameter SamplearrayPtr is unequal to 0, the wakeup pins will be scanned every 5ms and the results will be saved into an array (Address = SamplearrayPtr)

Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{batFL} and Erase Cycle ER_{FLData} are within specified range (see **"[1]" on Page 82***).*

Note: This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF oscillator must be used as system clock

Note: Do not manipulate any byte in the User Data Sector I or II (by using ***EraseUserDataSector()****,* ***WriteUserDataSectorLine()*** *or* ***SetLockbyte3()****) when using the EEPROM emulation.*

2.13.2.2 Actions

- Erases both User Data Sectors
- Port strobing: reads the wake up Pins every 5ms and saves the result in an array (optional)

2.13.2.3 Prototype

signed char **EEPROM_Init**(unsigned char **idata** * *SamplearrayPtr*)

2.13.2.4 Parameter

Table 87 EEPROM_Init: Parameter

Register / Address	Type	Direction	Name	Description
R7	unsigned char idata*	Output	SamplearrayPtr	Pointer to array for Port Sampling 0:.no sampling not 0: address for sampling array. The array size must be 21 bytes.

2.13.2.5 Return value

Table 88 EEPROM_Init: Return value

Register/ Address	Type	Description
R7	signed char	0: Success -1: Initialisation failed -2: Lockbyte 3 set

2.13.2.6 Resource Usage

Table 89 EEPROM_Init: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, CRC0, CRC1, CRCC, CRCD, DIVIC, TCON2, TMOD2, TH2, TL2, TH3, TL3, FCS, FCSERM, FCPP0, FCTKAS, FCSP, CFG1, FCPP1, IE
Stack	9 bytes (2 bytes for function call included)
RAM (IDATA)	$C7_H$ - CF_H, $D0_H$ - EA_H

2.13.2.7 Execution Information

Table 90 EEPROM_Init: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		208550		

2.13.2.8 Code Example

See **Chapter 2.13.11**.

2.13.3 Wr_EEByte()

2.13.3.1 Description

This function writes a single byte into the emulated EEPROM data area.
This function can be used for port sampling:

If the parameter SamplearrayPtr is unequal to 0, the wakeup pins will be scanned every 5ms and the results will be saved into an array (Address = SamplearrayPtr)

Attention: This function shows only effect if the Lockbyte 3 that protects the User Data Sectors is NOT set.

*Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{batFL} and Erase Cycle ER_{FLData} within specified range (see **"[1]" on Page 82**).*
This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF Oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.3.2 Actions

- Gets current EEPROM content
- Adds byte to content
- Verifies new EEPROM content
- Port sampling: reads the Wakeup Pins every 5ms and saves the result in an array (optional)

2.13.3.3 Prototype

signed char **Wr_EEByte**(unsigned char *Address*,
unsigned char *EEByte*,
unsigned char **idata** * *SamplearrayPtr*)

2.13.3.4 Parameters

Table 91 Wr_EEByte: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1E_H$).
R5	unsigned char	Input	EEByte	Byte written into the emulated EEPROM.
R3	unsigned char idata*	Output	SamplearrayPtr	Pointer to array for Port Sampling 0:.no sampling not 0: address for sampling array. The array size must be 21 bytes.

2.13.3.5 Return value

Table 92 Wr_EEByte: Return value

Register/ Address	Type	Description
R7	signed char	0: Data written and verified -1: Lockbyte 3 set / address out of range -2: The current data was written properly but data from previous writes (max. 3) have been lost. -3: Fatal error 1 (erase mismatch on sector change) -4: Fatal error 2 (write mismatch in current line) -5: Fatal error 3 (no reliable data found in history log)

2.13.3.6 Resource Usage

Table 93 Wr_EEByte: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, CFG0, CRC0, CRC1, CRCC, CRCD, DIVIC, TCON2, TMOD2, TH2, TL2, TH3, TL3, FCS, FCSERM, FCPP0, FCPP1, FCTKAS, FCSP, IE
Stack	19 bytes (2 bytes for function call included)
RAM (IDATA)	$C2_H$ - $E6_H$

2.13.3.7 Execution Information

Table 94 Wr_EEByte: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles	4760	212750 + 204000 (extra erase)		typ.: Writing of a line without erasing max.: Writing a line with erasing (startup and every fourth time) extra erase: In worst case (previous interruption) an extra erase is necessary.

2.13.3.8 Code Example

See **Chapter 2.13.11**.

2.13.4 Wr_EEInt()

2.13.4.1 Description

This function writes an integer value into the emulated EEPROM data area.
This function can be used for port sampling:
If the parameter SamplearrayPtr is unequal to 0, the wakeup pins will be scanned every 5ms and the results will be saved into an array (Address = SamplearrayPtr)

Attention: This function shows only effect if the Lockbyte 3 that protects the User Data Sectors is NOT set.

Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply

*voltage V_{batFL} and Erase Cycle ER_{FLData} within specified range (see **"[1]" on Page 82**).*
This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.4.2 Actions

- Gets current EEPROM content
- Adds int value to content
- Verifies new EEPROM content
- Port sampling: reads the Wakeup Pins every 5ms and saves the result in an array (optional)

2.13.4.3 Prototype

signed char **Wr_EEInt**(unsigned char *Address*,
unsigned int *EEInt*,
unsigend char **idata** * *SamplearrayPtr*)

2.13.4.4 Parameters

Table 95 Wr_EEInt: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1D_H$).
R5	unsigned int	Input	EEInt	Int value written into the emulated EEPROM.
R3	unsigned char idata*	Output	SamplearrayPtr	Pointer to array for Port Sampling 0:.no sampling not 0: address for sampling array. The array size must be 21 bytes.

2.13.4.5 Return value

Table 96 Wr_EEInt: Return value

Register/ Address	Type	Description
R7	signed char	0: Data written and verified -1: Lockbyte 3 set / address out of range -2: The current data was written properly but data from previous writes (max. 3) have been lost. -3: Fatal error 1 (erase mismatch on sector change) -4: Fatal error 2 (write mismatch in current line) -5: Fatal error 3 (no reliable data found in history log)

2.13.4.6 Resource Usage

Table 97 Wr_EEInt: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, CFG0, CRC0, CRC1, CRCC, CRCD, DIVIC, TCON2, TMOD2, TH2, TL2, TH3, TL3, FCS, FCSERM, FCPP0, FCPP1, FCTKAS, FCSP, IE
Stack	13 bytes (2 bytes for function call included)
RAM (IDATA)	$C2_H$ - $E6_H$

2.13.4.7 Execution Information

Table 98 Wr_EEInt: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles	4760	212750 + 204000 (extra erase)		typ.: Writing of a line without erasing max.: Writing a line with erasing (startup and every fourth time) extra erase: In worst case (previous interruption) an extra erase is necessary.

2.13.4.8 Code Example

See **Chapter 2.13.11**.

2.13.5 Wr_EELong()

2.13.5.1 Description

This function writes a long value into emulated EEPROM data area.
This function can be used for port sampling:
If the parameter SamplearrayPtr is unequal to 0, the wakeup pins will be scant every 5ms and the results will be saved in to an array (Addresse = SamplearrayPtr)

Attention: This function shows only effect if the Lockbyte 3 that protects the User Data Sectors is NOT set.

*Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{batFL} and Endurance En_{FL} within specified range (see **"[1]" on Page 82**).*
This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.5.2 Actions

- Gets current EEPROM content
- Adds long value to content
- Verifies new EEPROM content
- Port sampling: reads the Wakeup Pins every 5ms and saves the result in an array (optional)

2.13.5.3 Prototype

signed char **Wr_EELong**(unsigned char *Address*,
unsigned long **idata** * *EELong*,
unsigend char **idata** * *SamplearrayPtr*)

2.13.5.4 Parameters

Table 99 Wr_EELong: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1B_H$).
R5	unsigned long idata*	Input	EELong	Long value written into the emulated EEPROM.
R3	unsigned char idata*	Output	SamplearrayPtr	Pointer to array for Port Sampling 0:.no sampling not 0: address for sampling array. The array size must be 21 bytes.

2.13.5.5 Return value

Table 100 Wr_EELong: Return value

Register/ Address	Type	Description
R7	signed char	0: Data written and verified -1: Lockbyte 3 set / address out of range -2: The current data was written properly but data from previous writes (max. 3) have been lost. -3: Fatal error 1 (erase mismatch on sector change) -4: Fatal error 2 (write mismatch in current line) -5: Fatal error 3 (no reliable data found in history log)

2.13.5.6 Resource Usage

Table 101 Wr_EELong: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, CFG0, CRC0, CRC1, CRCC, CRCD, DIVIC, TCON2, TMOD2, TH2, TL2, TH3, TL3, FCS, FCSERM, FCPP0, FCPP1, FCTKAS, FCSP, IE
Stack	13 bytes (2 bytes for function call included)
RAM (IDATA)	$C2_H$ - $E6_H$

2.13.5.7 Execution Information

Table 102 Wr_EELong: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles	4760	212750 + 204000 (extra erase)		typ.: Writing of a line without erasing max.: Writing a line with erasing (startup and every fourth time) extra erase: In worst case (previous interruption) an extra erase is necessary.

2.13.5.8 Code Example

See **Chapter 2.13.11**.

2.13.6 Wr_EEString()

2.13.6.1 Description

This function writes a string into emulated EEPROM data area.
This function can be used for port sampling:
If the parameter SamplearrayPtr is unequal to 0, the wakeup pins will be scant every 5ms and the results will be saved in to an array (Addresse = SamplearrayPtr)

Attention: This function shows only effect if the Lockbyte 3 that protects the User Data Sectors is NOT set.

*Note: The application software has to ensure that FLASH is only programmed or erased when all required environmental conditions are fulfilled. Special care has to be taken that ambient temperature T_{FL}, supply voltage V_{batFL} and Endurance En_{FL} within specified range (see **"[1]" on Page 82**).*
This function returns -1 and has no effect if executed in DEBUG mode.

Note: For this function the 12 MHz RC HF Oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.6.2 Actions

- Gets current EEPROM content
- Adds string to content
- Verifies new EEPROM content
- Port sampling: reads the Wakeup Pins every 5ms and saves the result in an array (optional)

2.13.6.3 Prototype

signed char **Wr_EEString**(unsigned char *Address*,
unsigned char *Length*,
unsigned char **idata** * *Buffer*,
unsigned char **idata** * *SamplearrayPtr*)

2.13.6.4 Parameters

Table 103 Wr_EEString: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1E_H$).
R5	unsigned char	Input	Length	String length (0-31) (addr + length < 32)
R3	unsigned char idata*	Input	Buffer	Pointer to buffer with string content that is written into the emulated EEPROM.
IDATA Memory	unsigned char idata*	Output	SamplearrayPtr	Pointer to array for Port Sampling 0:.no sampling not 0: address for sampling array. The array size must be 21 bytes.

2.13.6.5 Return value

Table 104 Wr_EEString: Return value

Register/ Address	Type	Description
R7	signed char	0: Data written and verified -1: Lockbyte 3 set / address out of range -2: The current data was written properly but data from previous writes (max. 3) have been lost. -3: Fatal error 1 (erase mismatch on sector change) -4: Fatal error 2 (write mismatch in current line) -5: Fatal error 3 (no reliable data found in history log)

2.13.6.6 Resource Usage

Table 105 Wr_EEString: Resources

Type	used or modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, CFG0, CRC0, CRC1, CRCC, CRCD, DIVIC, TCON2, TMOD2, TH2, TL2, TH3, TL3, FCS, FCSERM, FCPP0, FCPP1, FCTKAS, FCSP, IE
Stack	13 bytes (2 bytes for function call included)
RAM (IDATA)	$C2_H$ - $E6_H$

2.13.6.7 Execution Information

Table 106 Wr_EEString: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles	4760	212750 + 204000 (extra erase)		typ.: Writing of a line without erasing max.: Writing a line with erasing (startup and every fourth time) extra erase: In worst case (previous interruption) an extra erase is necessary.

2.13.6.8 Code Example

See **Chapter 2.13.11**.

2.13.7 Get_EEByte()

2.13.7.1 Description

This function reads a single byte from the emulated EEPROM data area.

Note: For this function the 12 MHz RC HF Oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.7.2 Actions

- Reads a single byte from current EEPROM content

2.13.7.3 Prototype

signed char **Get_EEByte**(unsigned char *Address*,
unsigned char **idata** * *Data*)

2.13.7.4 Parameters

Table 107 Get_EEByte: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1E_H$).
R5	unsigned char idata*	Output	Data	Pointer to EEPROM byte read

2.13.7.5 Return value

Table 108 Get_EEByte: Return value

Register/ Address	Type	Description
R7	signed char	0: Data sucessfully read -1: Address out of range -2: Reliable data found in history log but data from previous write accesses (max. 3) have been lost. -3: Fatal error (no reliable data found in history log)

2.13.7.6 Resource Usage

Table 109 Get_EEByte: Resources

Type	used or modified
Registers	R0, R1, R2, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, IE, FCSERM, FCSP
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	$C7_H$ - $E6_H$

2.13.7.7 Execution Information

Table 110 Get_EEByte: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		60		

2.13.7.8 Code Example

See **Chapter 2.13.11**.

2.13.8 Get_EEInt()

2.13.8.1 Description

This function reads an integer from the emulated EEPROM data area.

Note: For this function the 12 MHz RC HF Oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.8.2 Actions

- Reads an int from current EEPROM content

2.13.8.3 Prototype

signed char **Get_EEInt**(unsigned char *Address*,
unsigned int **idata** * *Data*)

2.13.8.4 Parameters

Table 111 Get_EEInt: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1D_H$).
R5	unsigned int idata*	Output	Data	Pointer to EEPROM byte read

2.13.8.5 Return value

Table 112 Get_EEInt: Return value

Register/ Address	Type	Description
R7	signed char	0: Data sucessfully read -1: Address out of range -2: Reliable data found in history log but data from previous write accesses (max. 3) have been lost. -3: Fatal error (no reliable data found in history log)

2.13.8.6 Resource Usage

Table 113 Get_EEInt: Resources

Type	used or modified
Registers	R0, R1, R2, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, IE, FCSERM, FCSP
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	$C7_H$ - $E6_H$

2.13.8.7 Execution Information

Table 114 Get_EEInt: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		64		

2.13.8.8 Code Example

See **Chapter 2.13.11**.

2.13.9 Get_EELong()

2.13.9.1 Description

This function reads a long from the emulated EEPROM data area.

Note: For this function the 12 MHz RC HF Oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.9.2 Actions

- Reads a long from current EEPROM content

2.13.9.3 Prototype

signed char **Get_EELong**(unsigned char *Address*,
unsigned long **idata** * *Data*)

2.13.9.4 Parameters

Table 115 Get_EELong: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1B_H$).
R5	unsigned long idata*	Output	Data	Pointer to EEPROM long read

2.13.9.5 Return value

Table 116 Get_EELong: Return value

Register/ Address	Type	Description
R7	signed char	0: Data sucessfully read -1: Address out of range -2: Reliable data found in history log but data from previous write accesses (max. 3) have been lost. -3: Fatal error (no reliable data found in history log)

2.13.9.6 Resource Usage

Table 117 Get_EELong: Resources

Type	used or modified
Registers	R0, R1, R2, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, IE, FCSERM, FCSP
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	$C7_H$ - $E6_H$

2.13.9.7 Execution Information

Table 118 Get_EELong: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		72		

2.13.9.8 Code Example

See **Chapter 2.13.11**.

2.13.10 Get_EEString()

2.13.10.1 Description

This function reads a string from the emulated EEPROM data area.

Note: For this function the 12 MHz RC HF Oscillator must be used as system clock

*Note: Do not manipulate any byte in the User Data Sector I or II (by using **EraseUserDataSector()**, **WriteUserDataSectorLine()** or **SetLockbyte3()**) when using the EEPROM emulation.*

2.13.10.2 Actions

- Reads a string from current EEPROM content

2.13.10.3 Prototype

signed char **Get_EEString**(unsigned char *Address*,
unsigned char *Length*,
unsigend char **idata** * *Buffer*)

2.13.10.4 Parameters

Table 119 Get_EEString: Parameters

Register / Address	Type	Direction	Name	Description
R7	unsigned char	Input	Address	Address of byte location (00_H-$1E_H$).
R5	unsigned char	Input	Length	String length (0-31) (addr + length < 32)
R3	unsigned char idata*	Output	Buffer	Pointer to location to place string content

2.13.10.5 Return value

Table 120 Get_EEString: Return value

Register/ Address	Type	Description
R7	signed char	0: Data sucessfully read -1: Address out of range -2: Reliable data found in history log but data from previous write accesses (max. 3) have been lost. -3: Fatal error (no reliable data found in history log)

2.13.10.6 Resource Usage

Table 121 Get_EEString: Resources

Type	used or modified
Registers	R0, R1, R2, R4, R5, R6, R7
SFR	A, B, PSW, DPL, DPH, IE, FCSERM, FCSP

Type	used or modified
Stack	5 bytes (2 bytes for function call included)
RAM (IDATA)	$C7_H$ - $E6_H$

2.13.10.7 Execution Information

Table 122 Get_EEString: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		63 + 7*Length		

2.13.11 Code Example

```
// Register definition
#include "Reg_PMA71xx_PMA51xx.h"

// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

// Length of unsigned char idata Buffer
#define BUFFER_LEN      31

// Define of wake-up bit in register DSR
#define BIT_WUP         0x02

void main()
{
        signed char StatusByte;
        unsigned char idata Samplearray[21];
        unsigned char EEByte = 0xAB;
        unsigned int EEInt = 0x1234;
        unsigned long idata EELong = 0x56789ABC;
        unsigned char idata Buffer[BUFFER_LEN];

        unsigned char idata EEReadByte = 0xee;
        unsigned int idata EEReadInt;
        unsigned long idata EEReadLong;
        unsigned char i;

        // Decision: Reset or wake-up entry:
        // ================================
        if ((DSR & BIT_WUP)== 0) {

                // Reset entry:
                // ============

                // .. Enable wake-up sources here

                // Initialize Buffer with Dummy data
                for(i=0; i<BUFFER_LEN; i++) Buffer[i] = i;

                // Prepare the FLASH User Data Sectors for EEPROM emulation
                StatusByte = EEPROM_Init(Samplearray);
                if(StatusByte==0){
                        // .. Initialization of EEPROM Emulation successful
                }
                else if(StatusByte==-1){
                        // .. Initialization of EEPROM Emulation failed
```

```
        }
        else if(StatusByte==-2){
                // .. Initialization of EEPROM Emulation failed (Lockbyte3 set!)
        }

        // .. The Samplearray can be checked here for any wakeups during
        // EEPROM_Init()
}
else {
   // wake-up entry:
   // =============

   // .. Check the wake-up source here

        // Write one byte (0xAB) to address 0x01
        StatusByte = Wr_EEByte(0x01, EEByte, Samplearray);
        if(StatusByte==0){
                // .. Write successful, data has been verified
        }
        else if (StatusByte==-1){
                // .. Write failed, Lockbyte3 set
        }
        else if (StatusByte==-2){
                // .. current data written proberly but previous write failed
        }
        else if (StatusByte==-3){
                // .. Fatal error 1 (erase mismatch on sector change)
        }
        else if (StatusByte==-4){
                // .. Fatal error 2 (write mismatch in current line)
        }
        else if (StatusByte==-5){
                // .. Fatal error 3 (no reliable data found in history log)
        }

        // .. The Samplearray can be checked here for any wakeups during
        // Wr_EEByte()

        // Write two bytes (0x1234)to address 0x02
        StatusByte = Wr_EEInt(0x02, EEInt, Samplearray);
        if(StatusByte==0){
                // .. Write successful, data has been verified
        }
        else if (StatusByte==-1){
                // .. Write failed, Lockbyte3 set
        }
        else if (StatusByte==-2){
                // .. current data written proberly but previous write failed
        }
        else if (StatusByte==-3){
                // .. Fatal error 1 (erase mismatch on sector change)
        }
        else if (StatusByte==-4){
                // .. Fatal error 2 (write mismatch in current line)
        }
        else if (StatusByte==-5){
                // .. Fatal error 3 (no reliable data found in history log)
        }

        // .. The Samplearray can be checked here for any wakeups during
```

```
// Wr_EEInt()

// Write four bytes (0x56789ABC) to address 0x04
StatusByte =  Wr_EELong(0x04, &EELong, Samplearray);
if(StatusByte==0){
        // .. Write successful, data has been verified
}
else if (StatusByte==-1){
        // .. Write failed, Lockbyte3 set
}
else if (StatusByte==-2){
        // .. current data written proberly but previous write failed
}
else if (StatusByte==-3){
        // .. Fatal error 1 (erase mismatch on sector change)
}
else if (StatusByte==-4){
        // .. Fatal error 2 (write mismatch in current line)
}
else if (StatusByte==-5){
        // .. Fatal error 3 (no reliable data found in history log)
}

// .. The Samplearray can be checked here for any wakeups during
// Wr_EELong()

// write 31 bytes to address 0x00
StatusByte =  Wr_EEString(0x00, BUFFER_LEN, Buffer, Samplearray);
if(StatusByte==0){
        // .. Write successful, data has been verified
}
else if (StatusByte==-1){
        // .. Write failed, Lockbyte3 set
}
else if (StatusByte==-2){
        // .. current data written proberly but previous write failed
}
else if (StatusByte==-3){
        // .. Fatal error 1 (erase mismatch on sector change)
}
else if (StatusByte==-4){
        // .. Fatal error 2 (write mismatch in current line)
}
else if (StatusByte==-5){
        // .. Fatal error 3 (no reliable data found in history log)
}

// .. The Samplearray can be checked here for any wakeups during
// Wr_EEString()

// Read one byte from the emulated EEPROM from address 0x12
StatusByte = Get_EEByte(0x12, &EEReadByte);
if(StatusByte==0){
        // .. Read successful
}else if (StatusByte==-1){
        // .. Address out of range
}
else if (StatusByte==-2){
        // .. Reliable data found in history log but data from
        //    previous write accesses (max. 3) has been lost
```

```
                }
                else if (StatusByte==-3){
                        // .. Fatal error (no reliable data found in history log)
                }

                // Read two bytes from the emulated EEPROM from address 0x08
                StatusByte = Get_EEInt(0x08, &EEReadInt);
                if(StatusByte==0){
                        // .. Read successful
                }else if (StatusByte==-1){
                        // .. Address out of range
                }
                else if (StatusByte==-2){
                        // .. Reliable data found in history log but data from
                        //    previous write accesses (max. 3) has been lost
                }
                else if (StatusByte==-3){
                        // .. Fatal error (no reliable data found in history log)
                }

                // Read four bytes from the emulated EEPROM from address 0x03
                StatusByte = Get_EELong(0x03, &EEReadLong);
                if(StatusByte==0){
                        // .. Read successful
                }else if (StatusByte==-1){
                        // .. Address out of range
                }
                else if (StatusByte==-2){
                        // .. Reliable data found in history log but data from
                        //    previous write accesses (max. 3) has been lost
                }
                else if (StatusByte==-3){
                        // .. Fatal error (no reliable data found in history log)
                }

                StatusByte = Get_EEString(0x00, BUFFER_LEN, Buffer);
                if(StatusByte==0){
                        // .. Read successful
                }else if (StatusByte==-1){
                        // .. Address out of range
                }
                else if (StatusByte==-2){
                        // .. Reliable data found in history log but data from
                        //    previous write accesses (max. 3) has been lost
                }
                else if (StatusByte==-3){
                        // .. Fatal error (no reliable data found in history log)
                }
        }
        // .. Set PMA51xx/PMA71xx into Power Down Mode and wait for wake up
        PowerDown();
}
```

2.14 Manufacturer Revision Number

2.14.1 ReadManufacturerRevNum()

2.14.1.1 Description

This function returns the manufacturer revision number.

2.14.1.2 Actions

- Read manufacturer revision number

2.14.1.3 Prototype

signed int **ReadManufacturerRevNum**(void)

2.14.1.4 Return value

Table 123 ReadManufacturerRevNum: Return value

Register/ Address	Type	Description
R6 (MSB), R7 (LSB)	signed int	Manufacturer revision number

2.14.1.5 Resource Usage

Table 124 ReadManufacturerRevNum: Resources

Type	used or modified
Registers	R6, R7
SFR	A, IE
Stack	2 bytes (2 bytes for function call included)
RAM (IDATA)	--

2.14.1.6 Execution Information

Table 125 ReadManufacturerRevNum: Execution Information

Value	typ	max	Unit	Conditions
Instruction Cycles		42		

2.14.1.7 Code Example

```
// Register definition
#include "Reg_PMA71xx_PMA51xx.h"

// Library function prototypes
#include "PMA71xx_PMA51xx_Library.h"

void main (void){

        signed int ManufacturerRevNum;

        ManufacturerRevNum = ReadManufacturerRevNum();

        // .. add your source code here

}
```

3 Reference Documents

This section contains documents used for cross- reference throughout this document.

Table 126 Reference Documents

Reference Number	Document description
[1]	PMA51xx Datasheet or PMA71xx Datasheet

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG