

TPMS

Tire Pressure Monitoring Sensor

SP37

High integrated single-chip TPMS sensor with a low power embedded micro-controller and wireless FSK/ASK UHF transmitter

SP370 900kPa Version A5

ROM Library Function Guide

Revision 1.0, 2011-12-15

Edition 2011-12-15

**Published by
Infineon Technologies AG
81726 Munich, Germany**

**© 2011 Infineon Technologies AG
All Rights Reserved.**

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.

SP37 High integrated single-chip TPMS sensor with a low power embedded micro-controller and wireless FSK/ASK UHF transmitter

Revision History: 2011-12-15, Revision 1.0

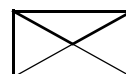
Previous Revision: Revision 0.1

Page	Subjects (major changes since last revision)
Page 17	WakeUp time updated
Page 22	Chapter Meas_Sensor updated
Page 27	Note for calling two additional bytes, valid for all functions
Page 31	Chapter Meas_Pressure updated
Page 36	Chapter Automatic Pressure Range Operation added
Page 42	Execution Time of the function Scale_Pressure updated
Page 45	Chapter Meas_Acceleration updated
Page 54	Execution Time of the function Raw_Temperature updated
Page 54	Register of Output Value of Raw_Temperature corrected
Page 71	Register CFG1 used in Powerdown Function
Page 72	Execution Time of the function Powerdown updated
Page 76	Execution Time of the function StartXtalOsc updated
Page 84	Execution Time of the function IntervalTimerCalibration updated
Page 86	Execution Time of the function LFBaudrateCalibration updated
Page 87	Function MullIntInt(): 2 additional bytes for stack added
Page 97	ReadID Input and Output Parameters updated
Page 101	FW_Revision_Nb Input and Output Parameters updated
Page 106	Execution Time of the function FlashUserConfigSectorLine updated
Page 107	New function WriteFlashCodeSectorLine
Page 110	Ressource Usage of the function FlashSetLock updated
Page 110	Execution Time of the function FlashSetLock updated
Page 121	Execution Time of the function Wait100usMultiples updated
Page 123	Execution Time of the function Send_RF_Telegram updated
Page 126	Flow Chart of Send_RF_Telegram Function

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

sensors@infineon.com



Trademarks of Infineon Technologies AG

AURIX™, C166™, CanPAK™, CIPOS™, CIPURSE™, EconoPACK™, CoolMOS™, CoolSET™, CORECONTROL™, CROSSAVE™, DAVE™, EasyPIM™, EconoBRIDGE™, EconoDUAL™, EconoPIM™, EiceDRIVER™, eupec™, FCOS™, HITFET™, HybridPACK™, I²RF™, ISOFACE™, IsoPACK™, MIPAQ™, ModSTACK™, my-d™, NovalithIC™, OptiMOS™, ORIGA™, PRIMARION™, PrimePACK™, PrimeSTACK™, PRO-SIL™, PROFET™, RASIC™, ReverSave™, SatRIC™, SIEGET™, SINDRION™, SIPMOS™, SmartLEWIS™, SOLID FLASH™, TEMPFET™, thinQ!™, TRENCHSTOP™, TriCore™.

Other Trademarks

Advance Design System™ (ADS) of Agilent Technologies, AMBA™, ARM™, MULTI-ICE™, KEIL™, PRIMECELL™, REALVIEW™, THUMB™, μ Vision™ of ARM Limited, UK. AUTOSAR™ is licensed by AUTOSAR development partnership. Bluetooth™ of Bluetooth SIG Inc. CAT-iq™ of DECT Forum. COLOSSUS™, FirstGPS™ of Trimble Navigation Ltd. EMV™ of EMVCo, LLC (Visa Holdings Inc.). EPCOS™ of Epcos AG. FLEXGO™ of Microsoft Corporation. FlexRay™ is licensed by FlexRay Consortium. HYPERTERMINAL™ of Hilgraeve Incorporated. IEC™ of Commission Electrotechnique Internationale. IrDA™ of Infrared Data Association Corporation. ISO™ of INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. MATLAB™ of MathWorks, Inc. MAXIM™ of Maxim Integrated Products, Inc. MICROTEC™, NUCLEUS™ of Mentor Graphics Corporation. Mifare™ of NXP. MIPI™ of MIPI Alliance, Inc. MIPS™ of MIPS Technologies, Inc., USA. muRata™ of MURATA MANUFACTURING CO., MICROWAVE OFFICE™ (MWO) of Applied Wave Research Inc., OmniVision™ of OmniVision Technologies, Inc. Openwave™ Openwave Systems Inc. RED HAT™ Red Hat, Inc. RFMD™ RF Micro Devices, Inc. SIRIUS™ of Sirius Satellite Radio Inc. SOLARIS™ of Sun Microsystems, Inc. SPANSION™ of Spansion LLC Ltd. Symbian™ of Symbian Software Limited. TAIYO YUDEN™ of Taiyo Yuden Co. TEAKLITE™ of CEVA, Inc. TEKTRONIX™ of Tektronix Inc. TOKO™ of TOKO KABUSHIKI KAISHA TA. UNIX™ of X/Open Company Limited. VERILOG™, PALLADIUM™ of Cadence Design Systems, Inc. VLYNQ™ of Texas Instruments Incorporated. VXWORKS™, WIND RIVER™ of WIND RIVER SYSTEMS, INC. ZETEX™ of Diodes Zetex Limited.

Last Trademarks Update 2011-02-24

Table of Contents

	Table of Contents	5
	List of Figures	12
	List of Tables	13
1	Introduction	17
1.1	General Considerations	17
1.2	Type definitions	17
1.3	Wakeup Handler	17
1.4	Restricted RAM and FLASH areas	18
1.4.1	Restricted RAM areas	18
1.4.2	Restricted FLASH areas	18
1.5	8051 Instruction Set Summary	18
2	ROM Library Functions	20
2.1	Meas_Sensor()	22
2.1.1	Description	22
2.1.2	Actions	22
2.1.3	Prototype	22
2.1.4	Inputs	23
2.1.5	Outputs	26
2.1.6	Resource Usage	27
2.1.7	Execution Information	28
2.2	Meas_Pressure()	31
2.2.1	Description	31
2.2.2	Actions	31
2.2.3	Prototype	31
2.2.4	Inputs	32
2.2.5	Outputs	35
2.2.6	Automatic Pressure Range Operation	36
2.2.7	Resource Usage	38
2.2.8	Execution Information	39
2.2.9	Code Example	41
2.3	Scale_Pressure()	42
2.3.1	Description	42
2.3.2	Actions	42
2.3.3	Prototype	42
2.3.4	Inputs	42
2.3.5	Outputs	42
2.3.6	Resource Usage	43
2.3.7	Execution Information	43
2.3.8	Code Example	44
2.4	Meas_Acceleration()	45
2.4.1	Description	45
2.4.2	Actions	45
2.4.3	Prototype	45
2.4.4	Inputs	46
2.4.5	Outputs	48
2.4.6	Resource Usage	48
2.4.7	Execution Information	49

Table of Contents

2.4.8	Code Example	50
2.5	Meas_Temperature()	51
2.5.1	Description	51
2.5.2	Actions	51
2.5.3	Prototype	51
2.5.4	Inputs	51
2.5.5	Outputs	51
2.5.6	Resource Usage	52
2.5.7	Execution Information	52
2.5.8	Code Example	53
2.6	Raw_Temperature()	54
2.6.1	Description	54
2.6.2	Actions	54
2.6.3	Prototype	54
2.6.4	Inputs	54
2.6.5	Outputs	54
2.6.6	Resource Usage	55
2.6.7	Execution Information	55
2.7	Comp_Temperature()	56
2.7.1	Description	56
2.7.2	Actions	56
2.7.3	Prototype	56
2.7.4	Inputs	56
2.7.5	Outputs	56
2.7.6	Resource Usage	57
2.7.7	Execution Information	57
2.7.8	Code Example	58
2.8	Meas_Supply_Voltage()	59
2.8.1	Description	59
2.8.2	Actions	59
2.8.3	Prototype	59
2.8.4	Inputs	59
2.8.5	Outputs	59
2.8.6	Resource Usage	60
2.8.7	Execution Information	60
2.8.8	Code Example	61
2.9	Start_Supply_Voltage()	62
2.9.1	Description	62
2.9.2	Actions	62
2.9.3	Prototype	62
2.9.4	Inputs	62
2.9.5	Outputs	62
2.9.6	Resource Usage	63
2.9.7	Execution Information	63
2.9.8	Code Example	63
2.10	Trig_Supply_Voltage()	64
2.10.1	Description	64
2.10.2	Actions	64
2.10.3	Prototype	64
2.10.4	Inputs	64
2.10.5	Outputs	64

Table of Contents

2.10.6	Resource Usage	64
2.10.7	Execution Information	65
2.10.8	Code Example	65
2.11	Get_Supply_Voltage()	66
2.11.1	Description	66
2.11.2	Actions	66
2.11.3	Prototype	66
2.11.4	Inputs	66
2.11.5	Outputs	66
2.11.6	Resource Usage	67
2.11.7	Execution Information	67
2.11.8	Code Example	68
2.12	ADC_Selftest()	69
2.12.1	Description	69
2.12.2	Actions	69
2.12.3	Prototype	69
2.12.4	Inputs	69
2.12.5	Outputs	69
2.12.6	Resource Usage	70
2.12.7	Execution Information	70
2.13	Powerdown()	71
2.13.1	Description	71
2.13.2	Actions	71
2.13.3	Prototype	71
2.13.4	Inputs	71
2.13.5	Outputs	71
2.13.6	Resource Usage	71
2.13.7	Execution Information	72
2.14	ThermalShutdown()	73
2.14.1	Description	73
2.14.2	Actions	73
2.14.3	Prototype	73
2.14.4	Inputs	73
2.14.5	Outputs	73
2.14.6	Resource Usage	73
2.14.7	Execution Information	74
2.15	StartXtalOsc()	75
2.15.1	Description	75
2.15.2	Actions	75
2.15.3	Prototype	75
2.15.4	Inputs	75
2.15.5	Outputs	75
2.15.6	Resource Usage	75
2.15.7	Execution Information	76
2.15.8	Code Example	76
2.16	StopXtalOsc()	77
2.16.1	Description	77
2.16.2	Actions	77
2.16.3	Prototype	77
2.16.4	Inputs	77
2.16.5	Outputs	77

Table of Contents

2.16.6	Resource Usage	77
2.16.7	Execution Information	78
2.16.8	Code Example	78
2.17	PLL_Ref_Signal_Check()	79
2.17.1	Description	79
2.17.2	Actions	79
2.17.3	Prototype	79
2.17.4	Inputs	79
2.17.5	Outputs	79
2.17.6	Resource Usage	79
2.17.7	Execution Information	80
2.18	VCO_Tuning()	81
2.18.1	Description	81
2.18.2	Actions	81
2.18.3	Prototype	81
2.18.4	Inputs	81
2.18.5	Outputs	81
2.18.6	Resource Usage	82
2.18.7	Execution Information	82
2.19	IntervalTimerCalibration()	83
2.19.1	Description	83
2.19.2	Actions	83
2.19.3	Prototype	83
2.19.4	Inputs	83
2.19.5	Outputs	84
2.19.6	Resource Usage	84
2.19.7	Execution Information	84
2.20	LFBaudrateCalibration()	85
2.20.1	Description	85
2.20.2	Actions	85
2.20.3	Prototype	85
2.20.4	Inputs	85
2.20.5	Outputs	85
2.20.6	Resource Usage	86
2.20.7	Execution Information	86
2.21	SMulIntInt() (16Bit * 16Bit)	87
2.21.1	Description	87
2.21.2	Actions	87
2.21.3	Prototype	87
2.21.4	Inputs	87
2.21.5	Outputs	87
2.21.6	Resource Usage	87
2.21.7	Execution Information	88
2.22	UDivLongLong() (32Bit : 32Bit)	89
2.22.1	Description	89
2.22.2	Actions	89
2.22.3	Prototype	89
2.22.4	Inputs	89
2.22.5	Outputs	89
2.22.6	Resource Usage	89
2.22.7	Execution Information	90

Table of Contents

2.23	UDivIntInt() (16Bit : 16Bit)	91
2.23.1	Description	91
2.23.2	Actions	91
2.23.3	Prototype	91
2.23.4	Inputs	91
2.23.5	Outputs	91
2.23.6	Resource Usage	91
2.23.7	Execution Information	92
2.24	CRC8_Calc()	93
2.24.1	Description	93
2.24.2	Actions	93
2.24.3	Prototype	93
2.24.4	Inputs	93
2.24.5	Outputs	93
2.24.6	Resource Usage	94
2.24.7	Execution Information	94
2.25	CRC_Baicheva_Calc()	95
2.25.1	Description	95
2.25.2	Actions	95
2.25.3	Prototype	95
2.25.4	Inputs	95
2.25.5	Outputs	95
2.25.6	Resource Usage	95
2.25.7	Execution Information	96
2.26	Read_ID()	97
2.26.1	Description	97
2.26.2	Actions	97
2.26.3	Prototype	97
2.26.4	Inputs	97
2.26.5	Outputs	97
2.26.6	Resource Usage	98
2.26.7	Execution Information	98
2.26.8	Code Example	98
2.27	ManuRevNb()	99
2.27.1	Description	99
2.27.2	Actions	99
2.27.3	Prototype	99
2.27.4	Inputs	99
2.27.5	Outputs	99
2.27.6	Resource Usage	99
2.27.7	Execution Information	100
2.28	FW_Revision_Nb()	101
2.28.1	Description	101
2.28.2	Actions	101
2.28.3	Prototype	101
2.28.4	Inputs	101
2.28.5	Outputs	101
2.28.6	Resource Usage	102
2.28.7	Execution Information	102
2.29	Erase_UserConfigSector()	103
2.29.1	Description	103

Table of Contents

2.29.2	Actions	103
2.29.3	Prototype	103
2.29.4	Inputs	103
2.29.5	Outputs	103
2.29.6	Resource Usage	103
2.29.7	Execution Information	104
2.30	WriteFlashUserConfigSectorLine()	105
2.30.1	Description	105
2.30.2	Actions	105
2.30.3	Prototype	105
2.30.4	Inputs	105
2.30.5	Outputs	106
2.30.6	Resource Usage	106
2.30.7	Execution Information	106
2.31	WriteFlashCodeSectorLine()	107
2.31.1	Description	107
2.31.2	Actions	107
2.31.3	Prototype	107
2.31.4	Inputs	107
2.31.5	Outputs	108
2.31.6	Resource Usage	108
2.31.7	Execution Information	108
2.32	FlashSetLock()	109
2.32.1	Description	109
2.32.2	Actions	109
2.32.3	Prototype	109
2.32.4	Inputs	109
2.32.5	Outputs	109
2.32.6	Resource Usage	110
2.32.7	Execution Information	110
2.33	ECC_Check()	111
2.33.1	Description	111
2.33.2	Actions	111
2.33.3	Prototype	111
2.33.4	Inputs	111
2.33.5	Outputs	111
2.33.6	Resource Usage	111
2.33.7	Execution Information	112
2.33.8	Code Example	112
2.34	CRC16_Check()	113
2.34.1	Description	113
2.34.2	Actions	113
2.34.3	Prototype	113
2.34.4	Inputs	113
2.34.5	Outputs	113
2.34.6	Resource Usage	114
2.34.7	Execution Information	114
2.34.8	Code Example	114
2.35	HIRC_Clock_Check()	115
2.35.1	Description	115
2.35.2	Actions	115

Table of Contents

2.35.3	Prototype	115
2.35.4	Inputs	115
2.35.5	Outputs	115
2.35.6	Resource Usage	116
2.35.7	Execution Information	116
2.36	GetCompValue()	117
2.36.1	Description	117
2.36.2	Actions	118
2.36.3	Prototype	118
2.36.4	Inputs	118
2.36.5	Outputs	120
2.36.6	Resource Usage	120
2.36.7	Execution Information	120
2.37	Wait100usMultiples()	121
2.37.1	Description	121
2.37.2	Actions	121
2.37.3	Prototype	121
2.37.4	Inputs	121
2.37.5	Outputs	121
2.37.6	Resource Usage	122
2.37.7	Execution Information	122
2.38	Send_RF_Telegram()	123
2.38.1	Description	123
2.38.1.1	Baud Rate parameter	123
2.38.1.2	Pattern Descriptor table	123
2.38.1.3	Start of Table indicator	124
2.38.1.4	Pattern Descriptor entries	124
2.38.1.5	Transmit Type Pattern Descriptor	124
2.38.1.6	Delay Pattern descriptor	125
2.38.1.7	End of Table pattern descriptor	125
2.38.2	Flow Chart of Send_RF_Telegram()	126
2.38.3	Actions	128
2.38.4	Prototype	128
2.38.5	Inputs	128
2.38.6	Outputs	129
2.38.7	Resource Usage	129
2.38.8	Execution Information	129
2.38.9	Code Example	131
2.39	Internal_SFR_Refresh()	132
2.39.1	Description	132
2.39.2	Actions	132
2.39.3	Prototype	132
2.39.4	Inputs	132
2.39.5	Outputs	132
2.39.6	Resource Usage	132
2.39.7	Execution Information	133
3	Reference Documents	134

List of Figures

Figure 1	SP37 OpCode Map.	19
Figure 2	Flow Chart for configuration of SensorConfig for Meas_Pressure() function.	37
Figure 3	Code example for usage of Meas_Pressure()	41
Figure 4	Code example for usage of Scale_Pressure()	44
Figure 5	Code example for usage of Meas_Acceleration().	50
Figure 6	Code example for usage of Meas_Temperature()	53
Figure 7	Code example for usage of Comp_Temperature().	58
Figure 8	Code example for usage of Meas_Supply_Voltage().	61
Figure 9	Code example for usage of the functions Start_Supply_Voltage, Trig_Supply_Voltage and Get_Supply_Voltage() 68	
Figure 10	Code example for usage of StartXtalOsc().	76
Figure 11	Code example for usage of StopXtalOsc().	78
Figure 12	Code example for usage of Read_ID().	98
Figure 13	Code example for usage of the functions ECC_Check() and CRC16_Check()	112
Figure 14	M by N matrix	117
Figure 15	Lookup table organization.	119
Figure 16	Datagram format.	123
Figure 17	Datagram format.	124
Figure 18	Transmit Type Pattern Descriptor.	124
Figure 19	Delay Type Pattern Descriptor	125
Figure 20	End of Table	126
Figure 21	Flow Chart of the Send_RF_Telegram() function.	127
Figure 22	Code example for usage of Send_RF_Telegram().	131

List of Tables

Table 1	Definition of types	17
Table 2	Wakeup Handler	17
Table 3	ROM Library functions	20
Table 4	Meas_Sensor: Input Parameters	23
Table 5	Meas_Sensor: Input Parameter: SensorConfig[15:8]	24
Table 6	Meas_Sensor: Input Parameter: SensorConfig[7:0]	24
Table 7	Meas_Sensor: Input Parameter: SampleRate	25
Table 8	Meas_Sensor: Output values	26
Table 9	Meas_Sensor: Resources	27
Table 10	Meas_Sensor: Pressure Measurement: Execution Time and Charge Consumption	28
Table 11	Meas_Sensor: Acceleration Measurement: Execution Time and Charge Consumption	30
Table 12	Meas_Pressure: Input Parameters	32
Table 13	Meas_Pressure: Input Parameter: SensorConfig[15:8]	33
Table 14	Meas_Pressure: Input Parameter: SensorConfig[7:0]	34
Table 15	Meas_Pressure: Input Parameter: SampleRate	34
Table 16	Meas_Pressure: Output values	35
Table 17	Meas_Pressure: Resources	38
Table 18	Meas_Pressure: Execution Time and Charge Consumption	39
Table 19	Scale_Pressure: Input Parameters	42
Table 20	Scale_Pressure: Output values	42
Table 21	Scale_Pressure: Resources	43
Table 22	Scale_Pressure: Execution Time and Charge Consumption	43
Table 23	Meas_Acceleration: Input Parameters	46
Table 24	Meas_Acceleration: Input Parameter: SensorConfig[15:8]	47
Table 25	Meas_Acceleration: Input Parameter: SensorConfig[7:0]	47
Table 26	Meas_Acceleration: Input Parameter: SampleRate	47
Table 27	Meas_Acceleration: Output values	48
Table 28	Meas_Acceleration: Resources	48
Table 29	Meas_Acceleration: Execution Time and Charge Consumption	49
Table 30	Meas_Temperature: Input Parameters	51
Table 31	Meas_Temperature: Output values	51
Table 32	Meas_Temperature: Resources	52
Table 33	Meas_Temperature: Execution Time and Charge Consumption	52
Table 34	Raw_Temperature: Input Parameters	54
Table 35	Raw_Temperature: Output values	54
Table 36	Raw_Temperature: Resources	55
Table 37	Raw_Temperature: Execution Time and Charge Consumption	55
Table 38	Comp_Temperature: Input Parameters	56
Table 39	Comp_Temperature: Output values	56
Table 40	Comp_Temperature: Resources	57
Table 41	Comp_Temperature: Execution Time and Charge Consumption	57
Table 42	Meas_Supply_Voltage: Input Parameters	59
Table 43	Meas_Supply_Voltage: Output values	59
Table 44	Meas_Supply_Voltage: Resources	60
Table 45	Meas_Supply_Voltage: Execution Time and Charge Consumption	60
Table 46	Start_Supply_Voltage: Input Parameters	62
Table 47	Start_Supply_Voltage: Output values	62
Table 48	Start_Supply_Voltage: Resources	63
Table 49	Start_Supply_Voltage: Execution Time and Charge Consumption	63

List of Tables

Table 50	Trig_Supply_Voltage: Input Parameters	64
Table 51	Trig_Supply_Voltage: Output values	64
Table 52	Trig_Supply_Voltage: Resources	64
Table 53	Trig_Supply_Voltage: Execution Time and Charge Consumption	65
Table 54	Get_Supply_Voltage: Input Parameters	66
Table 55	Get_Supply_Voltage: Output values	66
Table 56	Get_Supply_Voltage: Resources	67
Table 57	Get_Supply_Voltage: Execution Time and Charge Consumption	67
Table 58	ADC_Selftest: Input Parameters	69
Table 59	ADC_Selftest: Output values	69
Table 60	ADC_Selftest: Resources	70
Table 61	ADC_Selftest: Execution Time and Charge Consumption	70
Table 62	Powerdown: Input Parameters	71
Table 63	Powerdown: Output values	71
Table 64	Powerdown: Resources	71
Table 65	Powerdown: Execution Time and Charge Consumption	72
Table 66	ThermalShutdown: Input Parameters	73
Table 67	ThermalShutdown: Output values	73
Table 68	ThermalShutdown: Resources	73
Table 69	ThermalShutdown: Execution Time and Charge Consumption	74
Table 70	StartXtalOsc: Input Parameters	75
Table 71	StartXtalOsc: Output values	75
Table 72	StartXtalOsc: Resources	75
Table 73	StartXtalOsc: Execution Time and Charge Consumption	76
Table 74	StopXtalOsc: Input Parameters	77
Table 75	StopXtalOsc: Output values	77
Table 76	StopXtalOsc: Resources	77
Table 77	StopXtalOsc: Execution Time and Charge Consumption	78
Table 78	PLL_Ref_Signal_Check: Input Parameters	79
Table 79	PLL_Ref_Signal_Check: Output values	79
Table 80	PLL_Ref_Signal_Check: Resources	79
Table 81	PLL_Ref_Signal_Check: Execution Time and Charge Consumption	80
Table 82	VCO_Tuning: Input Parameters	81
Table 83	VCO_Tuning: Resources	82
Table 84	VCO_Tuning: Execution Time and Charge Consumption	82
Table 85	IntervalTimerCalibration: Input Parameters	83
Table 86	IntervalTimerCalibration: Output values	84
Table 87	IntervalTimerCalibration: Resources	84
Table 88	IntervalTimerCalibration: Execution Time and Charge Consumption	84
Table 89	LFBaudrateCalibration: Input Parameters	85
Table 90	LFBaudrateCalibration: Output values	85
Table 91	LFBaudrateCalibration: Resources	86
Table 92	LFBaudrateCalibration: Execution Time and Charge Consumption	86
Table 93	SMullIntInt: Input Parameters	87
Table 94	SMullIntInt: Output values	87
Table 95	SMullIntInt: Resources	87
Table 96	SMullIntInt: Execution Time and Charge Consumption	88
Table 97	UDivLongLong: Input Parameters	89
Table 98	UDivLongLong: Output values	89
Table 99	UDivLongLong: Resources	89
Table 100	UDivLongLong: Execution Time and Charge Consumption	90

List of Tables

Table 101	UDivIntInt: Input Parameters	91
Table 102	UDivIntInt: Output values	91
Table 103	UDivIntInt: Resources	91
Table 104	UDivIntInt: Execution Time and Charge Consumption	92
Table 105	CRC8_Calc: Input Parameters	93
Table 106	CRC8_Calc: Output values	93
Table 107	CRC8_Calc: Resources	94
Table 108	CRC8_Calc: Execution Time and Charge Consumption	94
Table 109	CRC_Baicheva_Calc: Input Parameters	95
Table 110	CRC_Baicheva_Calc: Output values	95
Table 111	CRC8_Baicheva_Calc: Resources	95
Table 112	CRC_Baicheva_Calc: Execution Time and Charge Consumption	96
Table 113	Read_ID: Input Parameters	97
Table 114	Read_ID: Input Parameters	97
Table 115	Read_ID: Resources	98
Table 116	Read_ID: Execution Time and Charge Consumption	98
Table 117	ManuRevNb: Input Parameters	99
Table 118	ManuRevNb: Output values	99
Table 119	ManuRevNb: Resources	99
Table 120	ManuRevNb: Execution Time and Charge Consumption	100
Table 121	FW_Revision_Nb: Input Parameters	101
Table 122	FW_Revision_Nb: Input Parameters	101
Table 123	FW_Revision_Nb: Resources	102
Table 124	FW_Revision_Nb: Execution Time and Charge Consumption	102
Table 125	Erase_UserConfigSector: Input Parameters	103
Table 126	Erase_UserConfigSector: Output values	103
Table 127	Erase_UserConfigSector: Resources	103
Table 128	Erase_UserConfigSector: Execution Time and Charge Consumption	104
Table 129	WriteFlashUserConfigurationSectorLine: Input Parameters	105
Table 130	WriteFlashUserConfigurationSectorLine: Output values	106
Table 131	WriteFlashUserConfigurationSectorLine: Resources	106
Table 132	WriteFlashUserConfigurationSectorLine: Execution Time and Charge Consumption	106
Table 133	WriteFlashCodeSectorLine: Input Parameters	107
Table 134	WriteFlashCodeSectorLine: Output values	108
Table 135	WriteFlashCodeSectorLine: Resources	108
Table 136	WriteFlashCodeSectorLine: Execution Time and Charge Consumption	108
Table 137	FlashSetLock: Input Parameters	109
Table 138	FlashSetLock: Output values	109
Table 139	FlashSetLock: Resources	110
Table 140	FlashSetLock: Execution Time and Charge Consumption	110
Table 141	ECC_Check: Input Parameters	111
Table 142	ECC_Check: Output values	111
Table 143	ECC_Check: Resources	111
Table 144	ECC_Check: Execution Time and Charge Consumption	112
Table 145	CRC16_Check: Input Parameters	113
Table 146	CRC16_Check: Output values	113
Table 147	CRC16_Check: Resources	114
Table 148	CRC16_Check: Execution Time and Charge Consumption	114
Table 149	HIRC_Clock_Check: Input Parameters	115
Table 150	HIRC_Clock_Check: Output values	115
Table 151	HIRC_Clock_Check: Resources	116

List of Tables

Table 152	HIRC_Clock_Check: Execution Time and Charge Consumption	116
Table 153	GetCompValue: Input Parameters	118
Table 154	GetCompValue: Output values	120
Table 155	GetCompValue: Resources	120
Table 156	GetCompValue: Execution Time and Charge Consumption	120
Table 157	Wait100usMultiples: Input Parameters	121
Table 158	Wait100usMultiples: Output values	121
Table 159	Wait100usMultiples: Resources	122
Table 160	Wait100usMultiples: Execution Time and Charge Consumption	122
Table 161	Send_RF_Telegram: Input Parameters	128
Table 162	Send_RF_Telegram: Output values	129
Table 163	Send_RF_Telegram: Resources	129
Table 164	Send_RF_Telegram: Execution Time.	129
Table 165	Send_RF_Telegram: Charge Consumption	130
Table 166	Internal_SFR_Refresh: Input Parameters	132
Table 167	Internal_SFR_Refresh: Output values	132
Table 168	Internal_SFR_Refresh: Resources	132
Table 169	Internal_SFR_Refresh: Execution Time and Charge Consumption	133

1 Introduction

1.1 General Considerations

This document describes the ROM Library functions that are available in the SP37 Version A5 step device. When the ROM Library Function **"FW_Revision_Nb()"** on Page 101 is called, these devices will return the value 0A32_H for the ROM revision.

In order for the application to use these ROM Library functions, the following files have to be included to the software project:

- SP37_ROMLibrary.h (the header file the function prototypes)
- SP37_ROMLibrary.lib (the precompiled ROM Library functions)

Notes

1. *The application must ensure that no IDLE-RESUME event source is active before any call of the ROM Library_functions.*
2. *All typical charge consumptions and typical execution times stated throughout this document were obtained from measurements of 10 devices at $V_{BAT} = 3.0\text{ V}$ and $T_{Ambient} = 25^{\circ}\text{C}$*
3. *Analysis of General Register, SFR, and Stack resource usage was performed on object code generated by Keil uVision3 (Assembler version: 8.00d, Compiler version: 8.08, Linker version: 6.05, Hex converter version: 2.6.0.1).*

1.2 Type definitions

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

Table 1 Definition of types

Type	Description	Range	
		Minimum	Maximum
Unsigned char	8 bit value without sign bit	0	255
Signed char	8 bit value with sign bit	-128	127
Unsigned int	16 bit value without sign bit	0	65,535
Signed int	16 bit value with sign bit	-32,768	32,767
Unsigned long	32 bit value without sign bit	0	4,294,967,295
Signed long	32 bit value with sign bit	-2,147,483,648	2,147,483,647

The Keil C51 Compiler stores data in big endian format (MSB first).

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 SP37 Datasheet [1].

Table 2 Wakeup Handler

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Wakeup from POWER DOWN state (independent of Wakeup source)	t_{wakeup}	—	0.85	1.8	ms	

1.4 Restricted RAM and FLASH areas

The ROM Library functions use certain address areas in RAM and FLASH. To ensure that the ROM Library functions operate properly, the application may not use these locations for storage, nor alter the contents of these memory locations

1.4.1 Restricted RAM areas

The RAM area C1_H through FF_H (upper 63 Bytes) of the 256 Bytes RAM is used and overwritten by the ROM Library functions.

1.4.2 Restricted FLASH areas

The FLASH area 57FA_H through 57FC_H is reserved for storage of the crystal frequency to be used in the application. The crystal frequency (divided by two) has to be written to that location during FLASH programming in the production in order to provide the ROM Library functions with the correct timebase for calibration purposes. For 315 MHz applications a 19.6875 MHz crystal is used and the crystal frequency divided by two has to be written to that location.

$$19,687,500 \text{ Hz} : 2 = 9,843,750 \text{ Hz} = 963426_{\text{H}} \text{ Hz}$$

The values in FLASH have to be written in the following way:

- Flash address 57FA_H: 96_H
- Flash address 57FB_H: 34_H
- Flash address 57FC_H: 26_H

For 433.92 MHz applications an 18.0800 MHz crystal is used and the crystal frequency divided by two has to be written to that location.

$$18,080,000 \text{ Hz} : 2 = 9,040,000 \text{ Hz} = 89F080_{\text{H}} \text{ Hz}$$

The values in FLASH have to be written in the following way:

- Flash address 57FA_H: 89_H
- Flash address 57FB_H: F0_H
- Flash address 57FC_H: 80_H

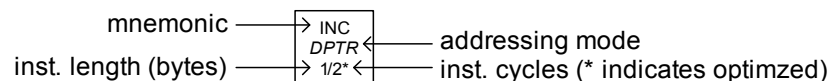
The FLASH address 57FF_H is reserved for Lockbyte 3. This value must not be changed by the application otherwise it might result in an unintentionally locked FLASH User Configuration Sector. Locking this sector is irreversible and shall only be done by either programming the Lockbyte 3 together with writing and locking the FLASH Code Sector or by a dedicated ROM Library function **FlashSetLock()**.

*Note: If **Erase_UserConfigSector()** or **WriteFlashUserConfigSectorLine()** are used by the application it has to be ensured that these restricted Flash locations are restored to the proper values.*

1.5 8051 Instruction Set Summary

As the SP37 incorporates an 8051 compatible microcontroller, **Figure 1** shows the SP37 OpCode Map.

		Least Significant Nybble															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		NOP 1/1	AJMP page0 2/2	LJMP addr16 3/2	RR A 1/1	INC A 1/1	INC dir 2/1	INC @R0 1/1	INC @R1 1/1	INC R0 1/1	INC R1 1/1	INC R2 1/1	INC R3 1/1	INC R4 1/1	INC R5 1/1	INC R6 1/1	INC R7 1/1
	1	JBC bit,rel 3/2	ACALL page0 2/2	LCALL addr16 3/2	RRC A 1/1	DEC A 1/1	DEC dir 2/1	DEC @R0 1/1	DEC @R1 1/1	DEC R0 1/1	DEC R1 1/1	DEC R2 1/1	DEC R3 1/1	DEC R4 1/1	DEC R5 1/1	DEC R6 1/1	DEC R7 1/1
2		JB bit,rel 3/2	AJMP page1 2/2	RET 1/2	RL A 1/1	ADD A,#imm 2/1	ADD A,dir 2/1	ADD A,@R0 1/1	ADD A,@R1 1/1	ADD A,R0 1/1	ADD A,R1 1/1	ADD A,R2 1/1	ADD A,R3 1/1	ADD A,R4 1/1	ADD A,R5 1/1	ADD A,R6 1/1	ADD A,R7 1/1
	3	JNB bit,rel 3/2	ACALL page1 2/2	RETI 1/2	RLC A 1/1	ADDC A,#imm 2/1	ADDC A,dir 2/1	ADDC A,@R0 1/1	ADDC A,@R1 1/1	ADDC A,R0 1/1	ADDC A,R1 1/1	ADDC A,R2 1/1	ADDC A,R3 1/1	ADDC A,R4 1/1	ADDC A,R5 1/1	ADDC A,R6 1/1	ADDC A,R7 1/1
4		JC rel 2/2	AJMP page2 2/2	ORL dir,A 2/1	ORL dir,#imm 3/2	ORL A,#imm 2/1	ORL A,dir 2/1	ORL A,@R0 1/1	ORL A,@R1 1/1	ORL A,R0 1/1	ORL A,R1 1/1	ORL A,R2 1/1	ORL A,R3 1/1	ORL A,R4 1/1	ORL A,R5 1/1	ORL A,R6 1/1	ORL A,R7 1/1
	5	JNC rel 2/2	ACALL page2 2/2	ANL dir,A 2/1	ANL dir,#imm 3/2	ANL A,#imm 2/1	ANL A,dir 2/1	ANL A,@R0 1/1	ANL A,@R1 1/1	ANL A,R0 1/1	ANL A,R1 1/1	ANL A,R2 1/1	ANL A,R3 1/1	ANL A,R4 1/1	ANL A,R5 1/1	ANL A,R6 1/1	ANL A,R7 1/1
6		JZ rel 2/2	AJMP page3 2/2	XRL dir,A 2/1	XRL dir,#imm 3/2	XRL A,#imm 2/1	XRL A,dir 2/1	XRL A,@R0 1/1	XRL A,@R1 1/1	XRL A,R0 1/1	XRL A,R1 1/1	XRL A,R2 1/1	XRL A,R3 1/1	XRL A,R4 1/1	XRL A,R5 1/1	XRL A,R6 1/1	XRL A,R7 1/1
	7	JNZ rel 2/2	ACALL page3 2/2	ORL C,bit 2/2*	JMP @A+DPTR 1/2	MOV A,#imm 2/1	MOV dir,#imm 3/2	MOV @R0,#imm 2/1	MOV @R1,#imm 2/1	MOV R0,#imm 2/1	MOV R1,#imm 2/1	MOV R2,#imm 2/1	MOV R3,#imm 2/1	MOV R4,#imm 2/1	MOV R5,#imm 2/1	MOV R6,#imm 2/1	MOV R7,#imm 2/1
8		SJMP rel 2/2	AJMP page4 2/2	ANL C,bit 2/2*	MOVC A,@A+PC 1/2	DIV AB 1/4	MOV dir,dir 3/2	MOV dir,@R0 2/2*	MOV dir,@R1 2/2*	MOV dir,R0 2/2*	MOV dir,R1 2/2*	MOV dir,R2 2/2*	MOV dir,R3 2/2*	MOV dir,R4 2/2*	MOV dir,R5 2/2*	MOV dir,R6 2/2*	MOV dir,R7 2/2*
	9	MOV DPTR,#imm 3/2	ACALL page4 2/2	MOV bit,C 2/2*	MOVC A,@A+DPTR 1/2	SUBB A,#imm 2/1	SUBB A,dir 2/1	SUBB A,@R0 1/1	SUBB A,@R1 1/1	SUBB A,R0 1/1	SUBB A,R1 1/1	SUBB A,R2 1/1	SUBB A,R3 1/1	SUBB A,R4 1/1	SUBB A,R5 1/1	SUBB A,R6 1/1	SUBB A,R7 1/1
A		ORL C,bit 2/2*	AJMP page5 2/2	MOV C,bit 2/1	INC DPTR 1/2*	MUL AB 1/4		MOV @R0,dir2 1/2*	MOV @R1,dir 2/2*	MOV R0,dir 2/2*	MOV R1,dir 2/2*	MOV R2,dir 2/2*	MOV R3,dir 2/2*	MOV R4,dir 2/2*	MOV R5,dir 2/2*	MOV R6,dir 2/2*	MOV R7,dir 2/2*
	B	ANL C,bit 2/2*	ACALL page5 2/2	CPL bit 2/1	CPL C 1/1	CJNE A,#imm 3/2	CJNE A,dir 3/2	CJNE @R0,#imm 3/2	CJNE @R1,#imm 3/2	CJNE R0,#imm 3/2	CJNE R1,#imm 3/2	CJNE R2,#imm 3/2	CJNE R3,#imm 3/2	CJNE R4,#imm 3/2	CJNE R5,#imm 3/2	CJNE R6,#imm 3/2	CJNE R7,#imm 3/2
C		PUSH dir 2/2*	AJMP page6 2/2	CLR bit 2/1	CLR C 1/1	SWAP A 1/1	XCH A,dir 2/1	XCH A,@R0 1/1	XCH A,@R1 1/1	XCH A,R0 1/1	XCH A,R1 1/1	XCH A,R2 1/1	XCH A,R3 1/1	XCH A,R4 1/1	XCH A,R5 1/1	XCH A,R6 1/1	XCH A,R7 1/1
	D	POP dir 2/2*	ACALL page6 2/2	SETB bit 2/1	SETB C 1/1	DA A 1/1	DJNZ dir,rel 3/2	XCHDA, @R0 1/1	XCHD A,@R1 1/1	DJNZ R0,rel 2/2	DJNZ R1,rel 2/2	DJNZ R2,rel 2/2	DJNZ R3,rel 2/2	DJNZ R4,rel 2/2	DJNZ R5,rel 2/2	DJNZ R6,rel 2/2	DJNZ R7,rel 2/2
E		MOVX A,@DPTR 1/2	AJMP page7 2/2	MOVX A,@R0 1/2	MOVX A,@R1 1/2	CLR A 1/1	MOV A,dir 2/1	MOV A,@R0 1/1	MOV A,@R1 1/1	MOV A,R0 1/1	MOV A,R1 1/1	MOV A,R2 1/1	MOV A,R3 1/1	MOV A,R4 1/1	MOV A,R5 1/1	MOV A,R6 1/1	MOV A,R7 1/1
	F	MOVX @DPTR,A 1/2	ACALL page7 2/2	MOVX @R0,A 1/2	MOVX @R1,A 1/2	CPL A 1/1	MOV dir,A 2/1	MOV @R0,A 1/1	MOV @R1,A 1/1	MOV R0,A 1/1	MOV R1,A 1/1	MOV R2,A 1/1	MOV R3,A 1/1	MOV R4,A 1/1	MOV R5,A 1/1	MOV R6,A 1/1	MOV R7,A 1/1



All mnemonics Copyright ©1980 Intel Corporation.

Figure 1 SP37 OpCode Map

2 ROM Library Functions

The following library functions are available for application usage:

Table 3 ROM Library functions

ROM Library function	Description	Page
Meas_Sensor()	Measures the ambient air pressure or acceleration	Page 22
Meas_Pressure()	Measures the ambient air pressure	Page 31
Scale_Pressure()	Scale pressure into a single byte for RF transmission	Page 42
Meas_Acceleration()	Measures the acceleration	Page 45
Meas_Temperature()	Measures the ambient temperature	Page 51
Raw_Temperature()	Measures the raw temperature	Page 54
Comp_Temperature()	Compensates raw temperature data	Page 56
Meas_Supply_Voltage()	Measures the battery voltage	Page 59
Start_Supply_Voltage()	These three functions perform a Battery Voltage measurement during an RF Transmission.	Page 62
Trig_Supply_Voltage()		Page 64
Get_Supply_Voltage()		Page 66
ADC_Selftest()	Returns the delta of ADC test measurements	Page 69
Powerdown()	Forces the device to POWER DOWN state	Page 71
ThermalShutdown()	Forces the device to THERMAL SHUTDOWN state	Page 73
StartXtalOsc()	Enables the Crystal Oscillator	Page 75
StopXtalOsc()	Stops the Crystal Oscillator	Page 77
PLL_Ref_Signal_Check()	Evaluates Crystal Resonator signal	Page 79
VCO_Tuning()	Tunes the VCO frequency	Page 81
IntervalTimerCalibration()	Calibrates the Interval Timer precounter	Page 83
LFBaudrateCalibration()	Calibrates the LF baudrate divider	Page 85
SMulIntInt()	Multiplies two signed values (16 bit * 16 bit)	Page 87
UDivLongLong()	Divides two unsigned values (32 bit : 32 bit)	Page 89
UDivIntInt()	Divides two unsigned values (16 bit : 16 bit)	Page 91
CRC8_Calc()	Calculates an 8 Bit CRC with polynom 83 _H	Page 93
CRC_Baicheva_Calc()	Calculates an 8 Bit CRC with polynom 97 _H	Page 95
Read_ID()	Returns the unique device ID	Page 97
ManuRevNb()	Returns the device revision number	Page 99
FW_Revision_Nb()	Returns the ROM- and Flash library revision number	Page 101
Erase_UserConfigSector()	Erases the FLASH user configuration sector	Page 103
WriteFlashUserConfigSectorLine()	Writes one FLASH line (32 Bytes) in the FLASH user configuration sector	Page 105
WriteFlashCodeSectorLine()	Writes one FLASH line (32 Bytes) in the FLASH Code Sector 0.	Page 107
FlashSetLock()	Sets the Lockbyte to protect the User Configuration sector	Page 109
ECC_Check()	Evaluates the ECC result bit	Page 111

Table 3 **ROM Library functions (cont'd)**

ROM Library function	Description	Page
CRC16_Check()	Evaluates the CRC16 result of a memory block	Page 113
HIRC_Clock_Check()	Evaluates the 12 MHz_RC_Oscillator frequency	Page 115
GetCompValue()	Returns a compensated value from look up table	Page 117
Wait100usMultiples()	Performs a delay of 100 µs multiples	Page 121
Send_RF_Telegram()	Configures and transmits RF frames	Page 123
Internal_SFR_Refresh()	Loads default values into the internal SFRs	Page 132

2.1 Meas_Sensor()

2.1.1 Description

This function performs a pressure or acceleration sensor measurement determined by **SensorConfig**.

The pressure or acceleration result can be either:

- Compensated for sensitivity, offset and temperature
- Output as raw value without performing the compensation

The function can measure either pressure or acceleration with up to 64 samples at a specified sample rate. The function will return the maximum or average (arithmetic mean) pressure or acceleration value in order to compensate for noise. The function performs a new temperature measurement for compensating the pressure or acceleration measurements, unless a previously measured raw temperature value is supplied. Number of samples and raw temperature source are both determined by **SensorConfig**. SampleRate controls how frequently the measurement acquisitions occur and will not influence the measurement result under stable input pressure or acceleration conditions. The device is set to IDLE mode during the delay between measurement acquisitions.

In case of pressure measurement, **SensorConfig** can be configured to automatically determine the high/low pressure range where to perform the measurement. The choice of the pressure measurement range can also be performed manually. The pressure hysteresis thresholds can be selected: either predefined from Infineon factory-default setting, or specified when calling this function (see SensorResult).

2.1.2 Actions

Pressure Measurement

- Measure pressure sensor with channel and number of samples given by **SensorConfig**
- Check wire bonds between the ASIC and the sensor die
- Average the measurement result(s), or (optionally) use the maximum raw measurement, to obtain a raw pressure value
- Sensor gain is automatically configured
- (optionally) Switch between Low and High Pressure range automatically, using configured thresholds
- (optionally) Perform a raw temperature measurement
- (optionally) Compensate the raw pressure value

Acceleration Measurement

- Measure acceleration sensor with channel and number of samples given by **SensorConfig**
- Check wire bonds between the ASIC and the sensor die & sensor beam integrity
- Sensor gain is automatically configured
- Average the measurement(s), or (optionally) use the maximum raw measurement, to obtain a raw acceleration value
- (optionally) Perform a raw temperature measurement
- (optionally) Compensate the raw acceleration value

2.1.3 Prototype

unsigned char **Meas_Sensor** (unsigned int **SensorConfig**, unsigned char **SampRate**, signed int idata ***SensorResult**)

2.1.4 Inputs

Table 4 Meas_Sensor: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB) R7 (LSB)	unsigned int	SensorConfig	Defines Sensor Configuration (refer to Table 5 and Table 6)
R5	unsigned char	SampleRate	Defines the number of system clock cycles (12 MHz RC Oscillator) divided by 8 which is waited between consecutive measurements. <i>Note: Only applicable if SensorConfig.2-0 [2POWN2:0] greater than 1 Sample</i>
R3	signed int idata*	SensorResult	Pointer to an integer array in RAM to receive the measurement result
*(SensorResult+2)	signed int	RawTemperature	Previous Raw Temperature value can optionally be used as input parameter. Refer to bit RAWTemp in SensorConfig.
*(SensorResult+3)	signed int	LowPressureThreshold	(optional input parameter) Value for hysteresis (in 1/16kPa) used by the pressure auto-range selection to determine when to switch from High-to-Low. It must be specified only if the parameter SensorConfig.11 is set to 1 (Customer Configuration).
*(SensorResult+4)	signed int	HighPressureThreshold	(optional input parameter) Value for hysteresis (in 1/16kPa) used by the pressure auto-range selection to determine when to switch from Low-to-High. It must be specified only if the parameter SensorConfig.11 is set to 1 (Customer Configuration).

Table 5 Meas_Sensor: Input Parameter: SensorConfig[15:8]

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
RES	RES	RES	RES	HYST	ARANGE	PRANGE	RAWTemp
RES		RES bits have to be set to 0 _B					
HYST		Select the thresholds for pressure hysteresis: 0 _B : Internal Configuration: 550kPa is the threshold from low to high pressure range and 500kPa is the threshold from high to low pressure range. 1 _B : Customer Configuration - see (SensorResult+3) and (SensorResult+4)					
ARANGE		Automatic range selection for pressure measurement: 0 _B : Manual 1 _B : Automatic If compensation is disabled (SensorConfig.6[RAW]==.1) a manual pressure range measurement is performed, using the range specified by SensorConfig.9[PRANGE]. Automatic Pressure Range is not supported for raw (uncompensated) measurements.					
PRANGE		Pressure range: (ignored if SensorConfig.10[ARANGE]==1) 0 _B : Low range (500kPa) 1 _B : High range (900kPa)					
RAWTemp		Selects source of raw temperature data for compensation 0 _B : Perform new raw temperature measurement 1 _B : Use raw temperature data supplied in *(SensorResult+2)					

Table 6 Meas_Sensor: Input Parameter: SensorConfig[7:0]

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Sens_Type	RAW	RES	RES	MODE	2POWN.2	2POWN.1	2POWN.0
Sens_Type		Select Pressure or Acceleration Measurement 0 _B : Acceleration Measurement 1 _B : Pressure Measurement					
RAW		Defines if the raw ADC result is compensated 0 _B : Temperature compensation is performed. Returns compensated & raw value 1 _B : No compensation is performed. Returns only raw value. If SensorConfig bit RAWTemp is set to 1 _B , no RAW temperature measurement is performed. If no compensation is performed, the autorange selection is disabled and the range has to be set manually. <i>Note: See Table 8 for impact on ROM Library function output</i>					
RES		RES bits have to be set to 0 _B					

Table 6 Meas_Sensor: Input Parameter: SensorConfig[7:0] (cont'd)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Sens_Type	RAW	RES	RES	MODE	2POWN.2	2POWN.1	2POWN.0
Mode		Acceleration and Pressure Mode: 0 _B : Average value 1 _B : Maximum value					
2POWN[2:0]		Number of ADC measurements that are taken and averaged 111 _B : 64 Samples 110 _B : 64 Samples 101 _B : 32 Samples 100 _B : 16 Samples 011 _B : 8 Samples 010 _B : 4 Samples 001 _B : 2 Samples 000 _B : 1 Sample					

Table 7 Meas_Sensor: Input Parameter: SampleRate

Bit7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SR.7	SR.6	SR.5	SR.4	SR.3	SR.2	SR.1	SR0
SR[7:0]		Number of systemclock cycles divided by 8 between two consecutive samples (only applicable if more than one sample is taken): 00 _H : No delay (fastest possible sample rate) 01 _H ..4F _H : Not allowed 50 _H ..FF _H : 1/sample rate (sample rate in systemclocks divided by 8)					

2.1.5 Outputs

Table 8 Meas_Sensor: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result ¹⁾ xxxx.xx1x _B : Overflow of ADC Result ¹⁾ xxxx.x1xx _B : Sensor Fault Wire Bond Check xxxx.1xxx _B : Sensor Fault Diagnosis Resistor (only for acceleration measurement) xxx1.xxxx _B : VMIN warning
*(SensorResult+0)	signed int	Compensated Pressure	If Input Bit SensorConfig.7[Sens_Type] == 1; If Input Bit SensorConfig.6[RAW] == 0: 8000 _H = -2048.0 kPa (Only theoretical) 0000 _H = 0.0 kPa 7FFF _H = 2047.9375 kPa (= 2048 kPa - 1 LSB where 1 LSB = 1/16 kPa) If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no compensation is performed
		Compensated Acceleration	If Input Bit SensorConfig.7[Sens_Type] == 0; If Input Bit SensorConfig.6[RAW] == 0: 8000 _H = -2048.0 g 0000 _H = 0.0 g 7FFF _H = 2047.9375 g (= 2048 g - 1 LSB where 1 LSB = 1/16 g) If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no compensation is performed
*(SensorResult+1)	signed int	Raw Pressure	10 Bit ADC Result Value: 0000.00xx.xxxx.xxxx _B
		Raw Acceleration	10 Bit ADC Result Value: 0000.00xx.xxxx.xxxx _B
*(SensorResult+2)	signed int	Raw Temperature	If Input Bit SensorConfig.6[RAW] == 0: 16 Bit scaled signed ADC Result Value If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no temperature measurement is performed

1) If the sensor measurement result is within the input range for which its accuracy is specified, then the ADC underflow/overflow condition is due to a measurement failure and the measurement results are not valid. Otherwise, if the sensor measurement result is outside of this input range, then the ADC underflow/overflow bits may be ignored.

2.1.6 Resource Usage

Table 9 Meas_Sensor: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0 ¹⁾ , CFG1, CFG2, DIVIC, DPH, DPL, PSW, TCON ¹⁾ , TH0 ¹⁾ , TL0 ¹⁾ , TMOD ¹⁾
Stack	13 Bytes ²⁾

1) Only affected if more than 1 sample is taken

2) Two additional bytes, not included here, are needed to call the library function

2.1.7 Execution Information

Table 10 Meas_Sensor: Pressure Measurement: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time Compensated with new raw temperature measurement, 2 samples	$t_{comp,new}$	—	916	1016	μs	DIVIC = 00 _H , SensorConfig = 0081 _H , SampleRate = 00 _H
Execution Time Compensated when supplied with previously obtained raw temperature value, 2 samples	$t_{comp,prev}$	—	686	768	μs	DIVIC = 00 _H , SensorConfig = 0181 _H , SampleRate = 00 _H
Execution Time Raw (uncompensated) pressure result, 2 samples	t_{RAW}	—	304	332	μs	DIVIC = 00 _H , SensorConfig = 00C1 _H , SampleRate = 00 _H
Execution Time for each additional sample for averaging	t_{sample}	—	54	60	μs	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible
Execution Time in autorange selection Compensated with new raw temperature measurement, 2 samples ¹⁾	$t_{autocomp,new}$	—	944	1008	μs	DIVIC = 00 _H , SensorConfig = 0481 _H , SampleRate = 00 _H
Execution Time in autorange selection Compensated when supplied with previously obtained raw temperature value, 2 samples ¹⁾	$t_{autpcomp,prev}$	—	714	760	μs	DIVIC = 00 _H , SensorConfig = 0581 _H , SampleRate = 00 _H
Execution Time in autorange selection for each additional sample for averaging ¹⁾	$t_{autosample}$	—	54	59	μs	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible
Charge Consumption Compensated with new raw temperature measurement, 2 samples	$Q_{comp,new}$	—	1,62	2,535	μC	DIVIC = 00 _H , SensorConfig = 0081 _H , SampleRate = 00 _H
Charge Consumption Compensated when supplied with previously obtained raw temperature value, 2 samples	$Q_{comp,prev}$	—	1,175	1,889	μC	DIVIC = 00 _H , SensorConfig = 0181 _H , SampleRate = 00 _H
Charge Consumption Raw (uncompensated) pressure result, 2 samples	Q_{RAW}	—	0,549	0,821	μC	DIVIC = 00 _H , SensorConfig = 00C1 _H , SampleRate = 00 _H
Charge Consumption for each additional sample for averaging	Q_{sample}	—	0,118	0,151	μC	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible

Table 10 Meas_Sensor: Pressure Measurement: Execution Time and Charge Consumption (cont'd)

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Charge Consumption in autorange selection Compensated with new raw temperature measurement, 2 samples ¹⁾	$Q_{\text{autpcomp, new}}$	–	1,643	2,515	μC	DIVIC = 00 _H , SensorConfig = 0481 _H , SampleRate = 00 _H
Charge Consumption in autorange selection Compensated when supplied with previously obtained raw temperature value, 2 samples ¹⁾	$Q_{\text{autocomp, prev}}$	–	1,196	1,869	μC	DIVIC = 00 _H , SensorConfig = 0581 _H , SampleRate = 00 _H
Charge Consumption in autorange selection for each additional sample for averaging ¹⁾	$Q_{\text{autosample}}$	–	0,118	0,148	μC	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible

1) The execution time and charge consumption for automatic range selection have been verified when the proper pressure range is selected.

Table 11 Meas_Sensor: Acceleration Measurement: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time Compensated with new raw temperature measurement, 2 samples	$t_{\text{comp,new}}$	—	906	1013	μs	DIVIC = 00 _H , SensorConfig = 0001 _H , SampleRate = 00 _H
Execution Time Compensated when supplied with previously obtained raw temperature value, 2 samples	$t_{\text{comp,prev}}$	—	652	765	μs	DIVIC = 00 _H , SensorConfig = 0101 _H , SampleRate = 00 _H
Execution Time Raw (uncompensated) acceleration result, 2 samples	t_{RAW}	—	304	332	μs	DIVIC = 00 _H , SensorConfig = 0041 _H , SampleRate = 00 _H
Execution Time for each additional sample for averaging	t_{sample}	—	64	71	μs	SampleRate = 00 _H
Charge Consumption Compensated with new internal raw temperature measurement, 2 samples	$Q_{\text{comp,new}}$	—	1,603	2,512	μC	DIVIC = 00 _H , SensorConfig = 0001 _H , SampleRate = 00 _H
Charge Consumption Compensated when supplied with previously obtained raw temperature value, 2 samples	$Q_{\text{comp,prev}}$	—	1,122	1,866	μC	DIVIC = 00 _H , SensorConfig = 0101 _H , SampleRate = 00 _H
Charge Consumption Raw (uncompensated) acceleration result, 2 samples	Q_{RAW}	—	0,551	0,805	μC	DIVIC = 00 _H , SensorConfig = 0041 _H , SampleRate = 00 _H
Charge Consumption for each additional sample for averaging	Q_{sample}	—	0,135	0,17	μC	SampleRate = 00 _H

2.2 Meas_Pressure()

2.2.1 Description

This function performs a pressure sensor measurement.

The result can be either:

- Compensated for sensitivity, offset and temperature
- Output as raw value without performing the compensation

The function can measure pressure with up to 64 samples at a specified sample rate. The function will return the maximum or average (arithmetic mean) pressure value in order to compensate for noise. The function performs a new temperature measurement for compensating the pressure measurements, unless a previously measured raw temperature value is supplied. Number of samples and raw temperature source are both determined by **SensorConfig**. SampleRate controls how frequently the measurement acquisitions occur and will not influence the measurement result under stable input pressure conditions. The device is set to IDLE mode during the delay between measurement acquisitions. **SensorConfig** can be configured to automatically determine the high/low pressure range. The choice of the pressure measurement range can also be performed manually. The pressure hysteresis thresholds detection can be selected: either predefined from Infineon factory-default setting, or specified when calling this function - see PressResult.

2.2.2 Actions

- Measure pressure sensor with channel and number of samples given by **SensorConfig**
- Check wire bonds between the ASIC and the sensor die
- Average the measurement result(s), or (optionally) use the maximum raw measurement, to obtain a raw pressure value
- (optionally) Switch between Low and High pressure range automatically, using configured thresholds
- (optionally) Perform a raw temperature measurement
- (optionally) Compensate the raw pressure value

2.2.3 Prototype

unsigned char **Meas_Pressure** (unsigned int **SensorConfig**, unsigned char **SampRate**, signed int idata ***PressResult**)

2.2.4 Inputs

Table 12 Meas_Pressure: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB) R7 (LSB)	unsigned int	SensorConfig	Defines Sensor Configuration (refer to Table 13 and Table 14)
R5	unsigned char	SampleRate	Defines the number of system clock cycles (12 MHz RC Oscillator) divided by 8 which is waited between consecutive measurements. <i>Note: Only applicable if SensorConfig.2-0 [2POWN2:0] greater than 1 Sample</i>
R3	signed int idata*	PressResult	Pointer to an integer array in RAM to receive the measurement result
*(PressResult+2)	signed int	RawTemperature	Previous Raw Temperature value can optionally be used as input parameter. Refer to bit RAWTemp in SensorConfig.
*(PressResult+3)	signed int	LowPressureThreshold	(optional input parameter) Value for hysteresis (in 1/16kPa) used by the pressure auto-range selection to determine when to switch from High-to-Low. It must be specified only if the parameter SensorConfig.11 is set to 1 (Customer Configuration)
*(PressResult+4)	signed int	HighPressureThreshold	(optional input parameter) Value for hysteresis (in 1/16kPa) used by the pressure auto-range selection to determine when to switch from Low-to-High. It must be specified only if the parameter SensorConfig.11 is set to 1 (Customer Configuration).

Table 13 Meas_Pressure: Input Parameter: SensorConfig[15:8]

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
RES	RES	RES	RES	HYST	ARANGE	PRANGE	RAWTemp
RES		RES bits have to be set to 0 _B					
HYST		Select the thresholds for pressure hysteresis: 0 _B : Internal Configuration: 550kPa is the threshold from low to high pressure range and 500kPa is the threshold from high to low pressure range. 1 _B : Customer Configuration - See (PressResult+3) and (PressResult+4)					
ARANGE		Automatic range selection for pressure measurement: 0 _B : Manual 1 _B : Automatic If compensation is disabled (SensorConfig.6[RAW]==.1) a manual pressure range measurement is performed, using the range specified by SensorConfig.9[PRANGE]. Automatic Pressure Range is not supported for raw (uncompensated) measurements.					
PRANGE		Pressure range: (ignored if SensorConfig.10[ARANGE]==1) 0 _B : Low range (500kPa) 1 _B : High range (900kPa)					
RAWTemp		Selects source of raw temperature data for compensation 0 _B : Perform new raw temperature measurement 1 _B : Use raw temperature data supplied in *(PressResult+2)					

Table 14 Meas_Pressure: Input Parameter: SensorConfig[7:0]

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Sens_Type	RAW	RES	RES	MODE	2POWN.2	2POWN.1	2POWN.0
Sens_Type		Select Pressure Measurement 1 _B : Pressure Measurement					
RAW		Defines if the raw ADC result is compensated 0 _B : Temperature compensation is performed. Returns compensated & raw value 1 _B : No compensation is performed. Returns only raw value. If SensorConfig bit RAWTemp is set to 1 _B , no RAW temperature measurement is performed. If no compensation is performed, the autorange selection is disabled and the range has to be set manually. <i>Note: See Table 16 for impact on ROM Library function output</i>					
RES		RES bits have to be set to 0 _B					
Mode		Pressure Mode: 0 _B : Average value 1 _B : Maximum value					
2POWN[2:0]		Number of ADC measurements that are taken and averaged 111 _B : 64 Samples 110 _B : 64 Samples 101 _B : 32 Samples 100 _B : 16 Samples 011 _B : 8 Samples 010 _B : 4 Samples 001 _B : 2 Samples 000 _B : 1 Sample					

Table 15 Meas_Pressure: Input Parameter: SampleRate

Bit7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SR.7	SR.6	SR.5	SR.4	SR.3	SR.2	SR.1	SR0
SR[7:0]		Number of systemclock cycles divided by 8 between two consecutive samples (only applicable if more than one sample is taken): 00 _H : No delay (fastest possible sample rate) 01 _H ..4F _H : Not allowed 50 _H ..FF _H : 1/sample rate (sample rate in systemclocks divided by 8)					

2.2.5 Outputs

Table 16 Meas_Pressure: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result ¹⁾ xxxx.xx1x _B : Overflow of ADC Result ¹⁾ xxxx.x1xx _B : Sensor Fault Wire Bond Check xxx1.xxxx _B : VMIN warning
*(PressResult+0)	signed int	Compensated Pressure	If Input Bit SensorConfig.6[RAW] == 0: 8000 _H = -2048.0 kPa (Only theoretical) 0000 _H = 0.0 kPa 7FFF _H = 2047.9375 kPa (= 2048 kPa - 1 LSB where 1 LSB = 1/16 kPa) If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no compensation is performed
*(PressResult+1)	signed int	Raw Pressure	10 Bit ADC Result Value: 0000.00xx.xxxx.xxxx _B
*(PressResult+2)	signed int	Raw Temperature	If Input Bit SensorConfig.6[RAW] == 0: 16 Bit scaled signed ADC Result Value If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no temperature measurement is performed

- 1) If the pressure measurement result is within the input range for which its accuracy is specified (e.g. 100 kPa - 900 kPa), then the ADC underflow/overflow condition is due to a measurement failure and the measurement results are not valid. Otherwise, if the pressure measurement result is outside of this input range, then the ADC underflow/overflow bits and the pressure measurement results may be ignored.

2.2.6 Automatic Pressure Range Operation

The Automatic Pressure Range feature is implemented as two separate pressure range calibrations within the SP37 A5 device. One calibration is optimized for a “low range” of 100-500kPa, the other for a “high range” of 500-900kPa. When the Automatic Pressure Range feature is enabled, the SP37 applies the calibration from the most appropriate pressure range. For example, if the input pressure is within the low range, the low range calibration is used to obtain the best measurement accuracy. The Automatic Pressure Range feature is only available when compensated measurements are being performed.

The Automatic Pressure Range function includes a hysteresis feature to prevent “range toggling” at the boundary between high range and low range calibrations. The hysteresis is implemented as an upper threshold, and a lower threshold. During an Automatic Pressure Range measurement, if the input pressure is below the lower threshold value, the lower range calibration is used during measurement. If the input pressure is above the higher threshold value, the upper range calibration is used during measurement. If the input pressure is between the two threshold values, the calibration range from the previous pressure measurement is used. The values of the hysteresis thresholds may be either the Infineon default values, or values supplied as parameters to the pressure measurement functions.

To apply the default hysteresis values when using Automatic Pressure Range, the **[SensorConfig.10[ARANGE]]** bit within the **[SensorConfig]** parameter must be set to 1_B, and the **[SensorConfig.11[HYST]]** bit within the **[SensorConfig]** parameter must be cleared to 0_B. For the 900kPa SP37 A5 device, the default upper threshold value is 550kPa, and the default lower threshold value is 500kPa. When these default threshold values are used, the pressure measurement error as specified within the SP37 A5 900kPa datasheet applies.

If the default hysteresis thresholds are not suitable, new threshold values may be supplied as parameters via the **[PressResult]** array as shown in [Table 16](#). In this case, both the **[SensorConfig.10[ARANGE]]** and **[SensorConfig.11[HYST]]** bits within the **[SensorConfig]** parameter must be set to 1_B. See [Table 13](#) in [Chapter 2.2.4](#) for more details.

Note: If the lower hysteresis level is set to less than 500kPa, the measurement error is unspecified for input pressures between the lower hysteresis threshold and 500kPa.

Note: The upper hysteresis level must not be set above 550kPa.

If Automatic Pressure Range is not desired, the pressure measurement function may be manually configured to use only the high range or only the low range. In this case, the **[SensorConfig.10[ARANGE]]** bit within the **[SensorConfig]** parameter must be cleared to 0_B, and the **[SensorConfig.9[PRANGE]]** bit is used to select high range calibration or low range calibration by being set to 1_B or cleared to 0_B, respectively.

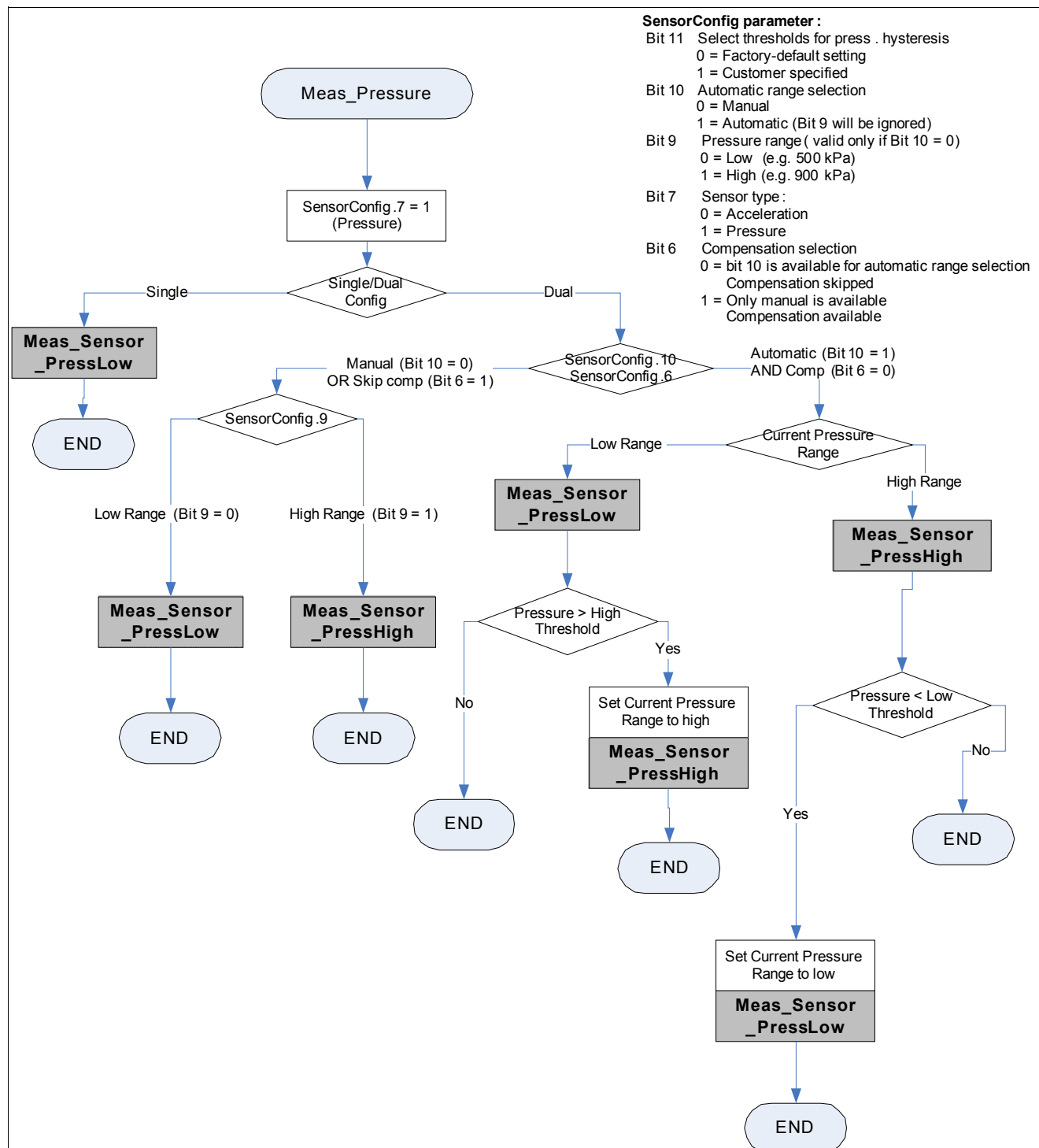


Figure 2 Flow Chart for configuration of SensorConfig for Meas_Pressure() function.

2.2.7 Resource Usage

Table 17 Meas_Pressure: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0 ¹⁾ , CFG1, CFG2, DIVIC, DPH, DPL, PSW, TCON ¹⁾ , TH0 ¹⁾ , TL0 ¹⁾ , TMOD ¹⁾
Stack	13 Bytes ²⁾

1) Only affected if more than 1 sample is taken

2) Two additional bytes, not included here, are needed to call the library function

2.2.8 Execution Information

Table 18 Meas_Pressure: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time Compensated with new raw temperature measurement, 2 samples	$t_{comp,new}$	—	918	1017	μs	DIVIC = 00 _H , SensorConfig = 0081 _H , SampleRate = 00 _H
Execution Time Compensated when supplied with previously obtained raw temperature value, 2 samples	$t_{comp,prev}$	—	689	770	μs	DIVIC = 00 _H , SensorConfig = 0181 _H , SampleRate = 00 _H
Execution Time Raw (uncompensated) pressure result, 2 samples	t_{RAW}	—	305	334	μs	DIVIC = 00 _H , SensorConfig = 00C1 _H , SampleRate = 00 _H
Execution Time for each additional sample for averaging	t_{sample}	—	54	60	μs	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible
Execution Time in automatic range selection compensated Pressure result with new raw temperature measurement, 2 samples ¹⁾	$t_{autocomp,new}$	—	948	1010	μs	DIVIC = 00 _H , SensorConfig = 0481 _H , SampleRate = 00 _H
Execution Time in automatic range compensated when supplied with previously obtained raw temperature value, 2 samples ¹⁾	$t_{autocomp,prev}$	—	716	762	μs	DIVIC = 00 _H , SensorConfig = 0581 _H , SampleRate = 00 _H
Execution Time in automatic pressure range for each additional sample for averaging ¹⁾	$t_{autosample}$	—	54	59	μs	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible
Charge Consumption Compensated with new raw temperature measurement, 2 samples	$Q_{comp,new}$	—	1,622	2,539	μC	DIVIC = 00 _H , SensorConfig = 0081 _H , SampleRate = 00 _H
Charge Consumption Compensated when supplied with previously obtained raw temperature value, 2 samples	$Q_{comp,prev}$	—	1,173	1,9	μC	DIVIC = 00 _H , SensorConfig = 0181 _H , SampleRate = 00 _H
Charge Consumption Raw (uncompensated) pressure result, 2 samples	Q_{RAW}	—	0,551	0,825	μC	DIVIC = 00 _H , SensorConfig = 00C1 _H , SampleRate = 00 _H
Charge Consumption for each additional sample for averaging	Q_{sample}	—	0,118	0,151	μC	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible

Table 18 Meas_Pressure: Execution Time and Charge Consumption (cont'd)

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Charge consumption in automatic range selection Compensated Pressure result with new raw temperature measurement, 2 samples ¹⁾	$Q_{\text{autocomp, new}}$	—	1,644	2,519	μC	DIVIC = 00 _H , SensorConfig = 0481 _H , SampleRate = 00 _H
Charge Consumption in automatic range selection compensated when supplied with previously obtained raw temperature value, 2 samples ¹⁾	$Q_{\text{autocomp, prev}}$	—	1,196	1,873	μC	DIVIC = 00 _H , SensorConfig = 0581 _H , SampleRate = 00 _H
Charge Consumption in automatic pressure range for each additional sample for averaging ¹⁾	$Q_{\text{autosample}}$	—	0,118	0,148	μC	SampleRate = 00 _H Only 1, 2, 4, 8, 16, 32, 64 samples possible

1) The execution time and charge consumption for automatic range selection have been verified when the proper pressure range is selected.

2.2.9 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

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

    // Input parameters for pressure measurement
    // Configure register to perform a 2-sample pressure measurement
    // Automatic Range Selection
    unsigned int SensorConfig = 0x0481;
    unsigned char SampRate = 0x00;

    // struct for pressure measurement results
    struct{
        signed int Pressure;
        signed int Raw_pressure;
        signed int Raw_temperature;
    } idata Press_Result;

    // Pressure measurement function call
    StatusByte = Meas_Pressure(SensorConfig, SampRate, &Press_Result.Pressure);

    if(!StatusByte)
    {
        // Pressure measurement was successful

    }
    else
    {
        // Pressure measurement was not successful, underflow or
        // overflow of ADC result, Sensor Fault Wire Bond Check,
        // or VMIN warning occurred
    }
}
```

Figure 3 Code example for usage of Meas_Pressure()

2.3 Scale_Pressure()

2.3.1 Description

A 450 kPa range TPMS sensor will typically transmit the pressure value as a single byte within an RF telegram: the pressure range 100...450 kPa is represented with an unsigned byte ranging from 0...255. A scale factor of 1.373 kPa/LSB is required to meet this typical requirement. The SP37 Meas_Pressure function, however, returns a signed integer value that represents pressure as 1/16 kPa/LSB. Conversion from 1/16 kPa/LSB to 1.373 kPa/LSB is therefore a commonly required task, but one that is not straightforward as care must be taken to avoid excessive rounding or loss of precision. The full scale error (FSE) of this function is less than 0.4%.

As a convenience, this Scale_Pressure() function performs this conversion. In addition, pressure bounds checking is performed so that 0 and 255 are returned for input pressure values below 100 kPa and above 450 kPa, respectively. Finally, to reduce the amount of data handling required by the application program, note that the input value to Scale_Pressure is passed as a pointer. This allows the same pointer to the [Meas_Pressure\(\)](#) output structure *(PressResult+0) to be re-used as the input pointer to Scale_Pressure().

Note: Scale function is intended to provide backward compatibility with SP30 100...450kPa numeric representation. The 100kPa and 450kPa bounds checking feature prevents this function from being useful outside of this range.

2.3.2 Actions

- Check 16 bit input value against 100 kPa and 450 kPa bounds, if value is outside of these bounds return 0 or 255, respectively
- Convert input value to unsigned 8 bit return value between 0 and 255

2.3.3 Prototype

unsigned char **Scale_Pressure**(signed int idata * **PressureValue**)

2.3.4 Inputs

Table 19 Scale_Pressure: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	PressureValue	Pointer to 16 bit input pressure value; Reuse of Meas_Pressure output structure *(PressResult+0)

2.3.5 Outputs

Table 20 Scale_Pressure: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	Result	8 bit output pressure value

2.3.6 Resource Usage

Table 21 Scale_Pressure: Resources

Type	Used or Modified
Registers	R0, R2, R3, R4, R5, R6, R7
SFR	A, B, PSW
Stack	4 bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.3.7 Execution Information

Table 22 Scale_Pressure: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	121	133	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,188	0,317	μC	DIVIC = 00 _H

2.3.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

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

    // 8-bit pressure value is stored in PressureByte
    unsigned char PressureByte;

    // Input parameters for pressure measurement
    unsigned int SensorConfig = 0x0081;
    unsigned char SampRate = 0x00;

    // struct for pressure measurement results
    struct{
        signed int Pressure;
        signed int Raw_pressure;
        signed int Raw_temperature;
    } idata Press_Result;

    // Pressure measurement function call
    StatusByte = Meas_Pressure(SensorConfig, SampRate, &Press_Result.Pressure);

    if(!StatusByte){

        // Pressure measurement was successful
        PressureByte = Scale_Pressure(&Press_Result.Pressure);

    }
    else{
        // Pressure measurement was not successful, underflow or
        // overflow of ADC result, Sensor Fault Wire Bond Check,
        // or VMIN warning occurred
        PressureByte = 0;

    }

    // 8-bit scaled pressure value is in PressureByte. If pressure was
    // 100kPa or less, or there was an error in measurement, PressureByte
    // is set to 0. If measured pressure was 450kPa or more, PressureByte
    // is set to 255. Otherwise, PressureByte contains a value proportional
    // to measured pressure, scaled by 1.373kPa/LSB for pressure above 100kPa.

}
}
```

Figure 4 Code example for usage of Scale_Pressure()

2.4 Meas_Acceleration()

2.4.1 Description

This function performs an acceleration sensor measurement.

The result can be either:

- Compensated for sensitivity, offset and temperature
- Output as raw value without performing the compensation

The function can measure acceleration with up to 64 samples at a specified sample rate. The function will return the average (arithmetic mean) acceleration value in order to compensate for noise. Optionally, this function can return the maximum value of the samples. The function performs a new temperature measurement for compensating the acceleration measurements, unless a previously measured raw temperature value is supplied. The number of samples and the raw temperature source are both determined by **SensorConfig**. **SampleRate** controls how frequently the measurement acquisitions occur and will not influence the measurement result under stable acceleration conditions. The device is set to IDLE mode during the delay between measurement acquisitions.

2.4.2 Actions

- Measure acceleration sensor with channel and number of samples given by **SensorConfig**
- Check wire bonds between the ASIC and the sensor die & sensor beam integrity
- Average the measurement(s), or (optionally) use the maximum raw measurement, to obtain a raw acceleration value
- (optionally) Perform a raw temperature measurement
- (optionally) Compensate the raw acceleration value

2.4.3 Prototype

unsigned char **Meas_Acceleration** (unsigned int **SensorConfig**, unsigned char **SampRate**, signed int idata * **AccelResult**)

2.4.4 Inputs

Table 23 Meas_Acceleration: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB) R7 (LSB)	unsigned int	SensorConfig	Defines Sensor Configuration (refer to Table 24 and Table 25)
R5	unsigned char	SampleRate	Defines the number of system clock cycles (12 MHz RC Oscillator) divided by 8 which is waited between consecutive measurements. <i>Note: Only applicable if SensorConfig.2-0 [2POWN2:0] greater than 1 Sample</i>
R3	signed int idata*	AccelResult	Pointer to an integer array in RAM to receive the measurement result
*(AccelResult+2)	signed int	Raw Temperature	Previous Raw Temperature value can optionally be used as input parameter. Refer to bit RAWTemp in SensorConfig.

Table 24 Meas_Acceleration: Input Parameter: SensorConfig[15:8]

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
RES	RES	RES	RES	RES	RES	RES	RAWTemp
RES		Must be set to 0.					
RAWTemp		Selects source of raw temperature data for compensation 0 _B : Perform new raw temperature measurement 1 _B : Use raw temperature data supplied in *(AccelResult+2)					

Table 25 Meas_Acceleration: Input Parameter: SensorConfig[7:0]

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Sens_Type	RAW	RES	RES	MODE	2POWN.2	2POWN.1	2POWN.0
Sens_Type		Select Pressure or Acceleration Measurement 0 _B : Acceleration Measurement					
RAW		Defines if the raw ADC result is compensated 0 _B : Temperature compensation is performed. Returns compensated & raw value 1 _B : No compensation is performed. Returns only raw value. If SensorConfig bit RAWTemp is set to 1 _B , no RAW temperature measurement is performed. <i>Note: See Table 29 for impact on ROM Library function output</i>					
RES		Must be set to 0 _B					
Mode		Acceleration Measure Mode: 0 _B : Average value 1 _B : Maximum value					
2POWN[2:0]		Number of ADC measurements that are taken and averaged 111 _B : 64 Samples 110 _B : 64 Samples 101 _B : 32 Samples 100 _B : 16 Samples 011 _B : 8 Samples 010 _B : 4 Samples 001 _B : 2 Samples 000 _B : 1 Sample					

Table 26 Meas_Acceleration: Input Parameter: SampleRate

Bit7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SR.7	SR.6	SR.5	SR.4	SR.3	SR.2	SR.1	SR0
SR[7:0]		Number of systemclock cycles divided by 8 between two consecutive samples (only applicable if more than one sample is taken and averaged): 00 _H : No delay (fastest possible sample rate) 01 _H ..5E _H : Not allowed 5F _H ..FF _H : 1/sample rate (sample rate in systemclocks divided by 8)					

2.4.5 Outputs

Table 27 Meas_Acceleration: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result ¹⁾ xxxx.xx1x _B : Overflow of ADC Result ¹⁾ xxxx.x1xx _B : Sensor Fault Wire Bond Check xxxx.1xxx _B : Sensor Fault Diagnosis Resistor xxx1.xxxx _B : VMIN warning
*AccelResult+0	signed int	Compensated Acceleration	If Input Bit SensorConfig.6[RAW] == 0: 8000 _H = -2048.0 g 0000 _H = 0.0 g 7FFF _H = 2047.9375 g (= 2048 g - 1 LSB where 1 LSB = 1/16 g) If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no compensation is performed
*AccelResult+1	signed int	Raw Acceleration Data	10 Bit ADC Result Value: 0000.00xx.xxxx.xxxx _B
*AccelResult+2	Singed int	Raw Temperature	If Input Bit SensorConfig.6[RAW] == 0: 16 Bit scaled signed ADC Result Value If Input Bit SensorConfig.6[RAW] == 1: 8000 _H since no temperature measurement is performed

- 1) If the acceleration measurement result is within the input range for which its accuracy is specified (e.g. -115 g - 115 g), then the ADC underflow/overflow condition is due to a measurement failure and the measurement results are not valid. Otherwise, if the acceleration measurement result is outside of this input range, then the ADC underflow/overflow bits and the acceleration measurement results may be ignored.

2.4.6 Resource Usage

Table 28 Meas_Acceleration: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0 ¹⁾ , CFG1, CFG2, DIVIC, DPH, DPL, PSW, TCON ¹⁾ , TH0 ¹⁾ , TL0 ¹⁾ , TMOD ¹⁾
Stack	13 Bytes ²⁾

- 1) Only affected if more than 1 sample is taken

- 2) Two additional bytes, not included here, are needed to call the library function

2.4.7 Execution Information

Table 29 Meas_Acceleration: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time Compensated with new raw temperature measurement, 2 samples	$t_{comp,new}$	—	909	1015	μs	DIVIC = 00 _H , SensorConfig = 0001 _H , SampleRate = 00 _H
Execution Time Compensated when supplied with previously obtained raw temperature value, 2 samples	$t_{comp,prevt}$	—	655	767	μs	DIVIC = 00 _H , SensorConfig = 0101 _H , SampleRate = 00 _H
Execution Time Raw (uncompensated) acceleration result, 2 samples	t_{RAW}	—	304	334	μs	DIVIC = 00 _H , SensorConfig = 0041 _H , SampleRate = 00 _H
Execution Time for each additional sample for averaging	t_{sample}	—	64	71	μs	SampleRate = 00 _H
Charge Consumption Compensated with new internal raw temperature measurement, 2 samples	$Q_{comp,new}$	—	1,611	2,516	μC	DIVIC = 00 _H , SensorConfig = 0001 _H , SampleRate = 00 _H
Charge Consumption Compensated when supplied with previously obtained raw temperature value, 2 samples	$Q_{comp,prev}$	—	1,125	1,87	μC	DIVIC = 00 _H , SensorConfig = 0101 _H , SampleRate = 00 _H
Charge Consumption Raw (uncompensated) acceleration result, 2 samples	Q_{RAW}	—	0,552	0,809	μC	DIVIC = 00 _H , SensorConfig = 0041 _H , SampleRate = 00 _H
Charge Consumption for each additional sample for averaging	Q_{sample}	—	0,135	0,17	μC	SampleRate = 00 _H

2.4.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

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

    // Input parameters for acceleration measurement
    unsigned int SensorConfig = 0x0001;
    unsigned char SampRate = 0x00;

    // struct for acceleration measurement results
    struct{
        signed int Acceleration;
        signed int Raw_acceleration;
        signed int Raw_temperature;
    } idata Accel_Result;

    // Acceleration measurement function call
    StatusByte = Meas_Acceleration(SensorConfig, SampRate, &Accel_Result.Acceleration);

    if(!StatusByte){
        // Acceleration measurement was successful
    }
    else{
        // Acceleration measurement was not successful, underflow or
        // overflow of ADC result, Sensor Fault Wire Bond Check,
        // Sensor Fault Diagnosis Resistor, or VMIN warning occurred
    }
}
```

Figure 5 Code example for usage of Meas_Acceleration()

2.5 Meas_Temperature()

2.5.1 Description

This function performs a temperature measurement and returns both raw and compensated temperature results. The Compensated Temperature result is compensated for sensitivity and offset errors. The Raw Temperature result is uncompensated and may be used as input for [Meas_Sensor\(\)](#), [Meas_Pressure\(\)](#), [Meas_Acceleration\(\)](#) and [Comp_Temperature\(\)](#).

2.5.2 Actions

- Measures the temperature sensor with 2 ADC samples for averaging (arithmetic mean).
- Compensate for offset using calibration data stored in FLASH (Compensated Temperature)

2.5.3 Prototype

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

2.5.4 Inputs

Table 30 Meas_Temperature: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	Temp_Result	Pointer to an integer array in RAM to receive the measurement result

2.5.5 Outputs

Table 31 Meas_Temperature: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result xxxx.xx1x _B : Overflow of ADC Result xxx1.xxxx _B : VMIN warning
*Temp_Result+0	signed int	Compensated 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+1	signed int	Raw Temperature	16 Bit scaled signed ADC Result Value

2.5.6 Resource Usage

Table 32 Meas_Temperature: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, CFG1, CFG2, DIVIC, DPH, DPL, PSW
Stack	9 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.5.7 Execution Information

Table 33 Meas_Temperature: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t_{comp}	–	668	726	μs	DIVIC = 00 _H , by default 2 ADC measurements are taken and averaged
Charge Consumption	Q_{comp}	–	1,2	1,82	μC	DIVIC = 00 _H , by default 2 ADC measurements are taken and averaged

2.5.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.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 Raw_temperature;
    } idata Temp_Result;

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

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

Figure 6 Code example for usage of Meas_Temperature()

2.6 Raw_Temperature()

2.6.1 Description

This function performs a raw temperature measurement. The Raw Temperature result is uncompensated and may be used as input for [Meas_Sensor\(\)](#), [Meas_Pressure\(\)](#), [Meas_Acceleration\(\)](#) and [Comp_Temperature\(\)](#).

2.6.2 Actions

- Measures the temperature sensor with 2 ADC samples for averaging (arithmetic mean)

2.6.3 Prototype

unsigned char **Raw_Temperature** (signed int idata * **TempResult**)

2.6.4 Inputs

Table 34 Raw_Temperature: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	TempResult	Pointer to an integer array in RAM to receive the measurement result

2.6.5 Outputs

Table 35 Raw_Temperature: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result xxxx.xx1x _B : Overflow of ADC Result xxx1.xxxx _B : VMIN warning
*Temp_Result+0	signed int	Raw Temperature	16 Bit scaled signed ADC Result Value

2.6.6 Resource Usage

Table 36 Raw_Temperature: Resources

Type	Used or Modified
Registers	R0, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, CFG1, CFG2, DPH, DPL, PSW
Stack	5 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.6.7 Execution Information

Table 37 Raw_Temperature: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t_{RAW}	–	274	299	μs	DIVIC = 00 _H , by default 2 ADC measurements are taken and averaged
Charge Consumption	Q_{RAW}	–	0,513	0,773	μC	DIVIC = 00 _H , by default 2 ADC measurements are taken and averaged

2.7 Comp_Temperature()

2.7.1 Description

This function will convert a previously obtained raw temperature value into a compensated temperature value. This function can be called after a pressure, acceleration or raw temperature measurement to compensate the raw temperature. Thus, the current ambient temperature is obtained without taking an extra temperature measurement.

2.7.2 Actions

- Compensate raw temperature data using calibration data stored in FLASH.

2.7.3 Prototype

unsigned char **Comp_Temperature** (signed int idata **TempRawIn**, signed int idata * **TempResult**)

2.7.4 Inputs

Table 38 Comp_Temperature: Input Parameters

Register / Address	Type	Name	Description
R6(MSB), R7(LSB)	signed int idata	TempRawIn	Raw measurement value (gathered from Meas_Pressure() or Meas_Acceleration() or Raw_Temperature)
R5	signed int idata*	Temp_Result	Pointer to an integer array in RAM to receive the measurement result

2.7.5 Outputs

Table 39 Comp_Temperature: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of Result xxxx.xx1x _B : Overflow of Result
*Temp_Result	signed int	Compensated 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+1	signed int	Raw Temperature	16 Bit scaled signed ADC Result Value

2.7.6 Resource Usage

Table 40 **Comp_Temperature: Resources**

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, DPH, DPL, PSW
Stack	4 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.7.7 Execution Information

Table 41 **Comp_Temperature: Execution Time and Charge Consumption**

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	420	454	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,63	1,089	μC	DIVIC = 00 _H

2.7.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

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

    // Input parameters for pressure measurement
    unsigned int SensorConfig = 0x0081;
    unsigned char SampRate = 0x00;

    // struct for pressure measurement results
    struct{
        signed int Pressure;
        signed int Raw_pressure;
        signed int Raw_temperature;
    } idata Press_Result;

    // struct for compensated temperature results
    struct{
        signed int Temperature;
        signed int Raw_temperature;
    } idata Temp_Result;

    // Pressure measurement function call
    StatusByte = Meas_Pressure(SensorConfig, SampRate, &Press_Result.Pressure);

    if(!StatusByte){
        // Pressure measurement was successful
    }
    else{
        // Pressure measurement was not successful, underflow or
        // overflow of ADC result, Sensor Fault Wire Bond Check,
        // or VMIN warning occurred
    }

    // Compensate Temperature function call
    StatusByte = Comp_Temperature(Press_Result.Raw_temperature, &Temp_Result.Temperature);

    if(!StatusByte){
        // Temperature compensation was successful
    }
    else{
        // Temperature compensation was not successful, underflow or
        // overflow during compensation occurred
    }
}
```

Figure 7 Code example for usage of `Comp_Temperature()`

2.8 Meas_Supply_Voltage()

2.8.1 Description

This function performs a battery voltage measurement and returns both raw and compensated voltage results. The compensated battery voltage result is compensated for offset error.

2.8.2 Actions

- Measure the supply voltage sensor with 2 ADC samples for averaging
- Compensate for offset using calibration data stored in FLASH

2.8.3 Prototype

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

2.8.4 Inputs

Table 42 Meas_Supply_Voltage: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	Batt_Result	Pointer to an integer array in RAM to receive the measurement result

2.8.5 Outputs

Table 43 Meas_Supply_Voltage: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result xxxx.xx1x _B : Overflow of ADC Result
*Batt_Result+0	signed int	Compensated battery voltage	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+1	signed int	Raw battery voltage	10 Bit ADC Result Value: 0000.00xx.xxxx.xxxx _B

2.8.6 Resource Usage

Table 44 Meas_Supply_Voltage: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, CFG1, CFG2, DIVIC, DPH, DPL, PSW
Stack	3 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.8.7 Execution Information

Table 45 Meas_Supply_Voltage: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	373	405	μs	DIVIC = 00 _H , by default 2 ADC measurements are taken and averaged
Charge Consumption	Q	–	0,634	1,01	μC	DIVIC = 00 _H , by default 2 ADC measurements are taken and averaged

2.8.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.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 Voltage;
        signed int Raw_voltage;
    } idata Volt_Result;

    // Battery voltage measurement function call
    StatusByte = Meas_Supply_Voltage(&Volt_Result.Voltage);

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

Figure 8 Code example for usage of Meas_Supply_Voltage()

2.9 Start_Supply_Voltage()

2.9.1 Description

The battery voltage typically shows a significant drop during RF transmission, when a considerable current is drawn. Calling the monolithic function **Meas_Supply_Voltage()** during RF transmission is often not feasible because of its execution time.

A set of three functions to allow battery voltage measurement during RF transmission has been implemented; **Start_Supply_Voltage()**, **Trig_Supply_Voltage()** and **Get_Supply_Voltage()**. These functions must be called in this particular sequence, and each function is separately described in detail.

Start_Supply_Voltage() enables and configures the ADC, allows settling of the analogue ADC part, and places the ADC into standby state. It must be called prior to calling the **Trig_Supply_Voltage()** function.

2.9.2 Actions

- Prepare the ADC and the Supply Voltage Sensor for a measurement

2.9.3 Prototype

unsigned char **Start_Supply_Voltage** (void)

2.9.4 Inputs

Table 46 Start_Supply_Voltage: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.9.5 Outputs

Table 47 Start_Supply_Voltage: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	Status Byte	Always returns 0

2.9.6 Resource Usage

Table 48 Start_Supply_Voltage: Resources

Type	Used or Modified
Registers	R7
SFR	ACC
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.9.7 Execution Information

Table 49 Start_Supply_Voltage: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	12	13	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,018	0,030	μC	DIVIC = 00 _H

2.9.8 Code Example

See [Figure 9 “Code example for usage of the functions Start_Supply_Voltage, Trig_Supply_Voltage and Get_Supply_Voltage\(\)” on Page 68](#)

2.10 Trig_Supply_Voltage()

2.10.1 Description

Trig_Supply_Voltage() is the second part of a set of three functions to measure the battery voltage during RF transmission. It is called after Start_Supply_Voltage(), and triggers an ADC battery voltage measurement. Upon the call of this function the ADC resumes from standby, performs a measurement and then goes back to standby, keeping the result. Typically this function is called by the application immediately after an RF datagram byte is shifted into SFR RFD to prevent disruption of RF data transmission.

2.10.2 Actions

- Bring ADC from standby to active state
- Measure the supply voltage sensor with ADC
- Place ADC into standby state

2.10.3 Prototype

void Trig_Supply_Voltage (void)

2.10.4 Inputs

Table 50 Trig_Supply_Voltage: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.10.5 Outputs

Table 51 Trig_Supply_Voltage: Output values

Register/ Address	Type	Name	Description
None	---	---	---

2.10.6 Resource Usage

Table 52 Trig_Supply_Voltage: Resources

Type	Used or Modified
Registers	---
SFR	---
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.10.7 Execution Information

Table 53 Trig_Supply_Voltage: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	$t_{\text{DIVIC}=0}$	—	3	4	μs	DIVIC = 00 _H
	$t_{\text{DIVIC}=3}$	—	192	208	μs	DIVIC = 03 _H
Charge Consumption	$Q_{\text{DIVIC}=0}$	—	0,005	0,008	μC	DIVIC = 00 _H
	$Q_{\text{DIVIC}=3}$	—	0,154	0,354	μC	DIVIC = 03 _H

2.10.8 Code Example

See [Figure 9 “Code example for usage of the functions Start_Supply_Voltage, Trig_Supply_Voltage and Get_Supply_Voltage\(\)”](#) on Page 68

2.11 Get_Supply_Voltage()

2.11.1 Description

Get_Supply_Voltage() is the third part of a set of three functions to measure the battery voltage during RF transmission. It reads the measured value obtained during Trig_Supply_Voltage(), turns off the ADC, and performs battery voltage compensation. Typically this function is called by the application after the RF transmission is finished.

2.11.2 Actions

- Read the ADC result register
- Turn off ADC
- Compensate the result for offset using calibration data stored in FLASH

2.11.3 Prototype

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

2.11.4 Inputs

Table 54 Get_Supply_Voltage: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	Batt_Result	Pointer to an integer array in RAM to receive the measurement result

2.11.5 Outputs

Table 55 Get_Supply_Voltage: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result xxxx.xx1x _B : Overflow of ADC Result
*Batt_Result+0	signed int	Compensated battery voltage	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+1	signed int	Raw battery voltage	10 Bit ADC Result Value: 0000.00xx.xxxx.xxxx _B

2.11.6 Resource Usage

Table 56 Get_Supply_Voltage: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG1, CFG2, DIVIC, DPH, DHL, PSW
Stack	3 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.11.7 Execution Information

Table 57 Get_Supply_Voltage: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	205	222	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,308	0,532	μC	DIVIC = 00 _H

2.11.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.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 Voltage;
        signed int Raw_voltage;
    } idata Volt_Result;

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

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

    // Get the measurement result after the real time critical function
    StatusByte = Get_Supply_Voltage(&Volt_Result.Voltage);

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

Figure 9 Code example for usage of the functions **Start_Supply_Voltage**, **Trig_Supply_Voltage** and **Get_Supply_Voltage()**

2.12 ADC_Selftest()

2.12.1 Description

The ADC self test is a combination of three measurements that use various channels as input and reference for the ADC. The output of this function is the delta deviation from the ideal value.

2.12.2 Actions

- Perform three ADC measurements
- Calculate delta from ideal value

2.12.3 Prototype

unsigned char **ADC_Selftest**(signed int idata * **Delta**)

2.12.4 Inputs

Table 58 ADC_Selftest: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	Delta	Pointer to an integer array in RAM to receive the ADC measurement result

2.12.5 Outputs

Table 59 ADC_Selftest: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	StatusByte	0000.0000 _B : Success xxxx.xxx1 _B : Underflow of ADC Result xxxx.xx1x _B : Overflow of ADC Result
Delta	signed int idata	Delta	The ADC can be considered as working when the value of Delta is between +/-6 LSBs after a call of this function during which the supply voltage was constant.

2.12.6 Resource Usage

Table 60 ADC_Selftest: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, CFG1, CFG2, DIVIC, DPH, DPL, PSW
Stack	7 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.12.7 Execution Information

Table 61 ADC_Selftest: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	645	701	μs	DIVIC = 00 _H , 3 Numbers of ADC measurements are taken
Charge Consumption	Q	–	1,2	1,84	μC	DIVIC = 00 _H 3 ADC measurements are taken

2.13 Powerdown()

2.13.1 Description

This function forces the device to POWER DOWN state.

2.13.2 Actions

- If an RF Transmission is in process, wait until it has completed
- If the SFR ITPR has been updated, wait for the Interval Timer to initialize
- Enter POWER DOWN state

2.13.3 Prototype

void **Powerdown**(void)

2.13.4 Inputs

Table 62 Powerdown: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.13.5 Outputs

Table 63 Powerdown: Output values

Register/ Address	Type	Name	Description
None	---	---	---

2.13.6 Resource Usage

Table 64 Powerdown: Resources

Type	Used or Modified
Registers	---
SFR	ACC, CFG0, CFG1, RFS
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.13.7 Execution Information

Table 65 Powerdown: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	11,36	21,6	μs	DIVIC = 00 _H , ITinit = 0, RFSE = 1 before entering PDWN
Charge Consumption	Q	–	0,018	0,053	μC	DIVIC = 00 _H , ITinit = 0, RFSE = 1 before entering PDWN

2.14 ThermalShutdown()

2.14.1 Description

This function forces the device to THERMAL SHUTDOWN state.

The application should call this function whenever the ambient temperature is close to the maximum operating range (this can be detected by using [Meas_Temperature\(\)](#)) to protect the device while the ambient temperature is above the specified operating conditions.

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, the TMAX threshold will be reduced (software hysteresis) and the THERMAL SHUTDOWN state is entered.

2.14.2 Actions

- Turn on the TMAX Detector
- Enter THERMAL SHUTDOWN state with hysteresis if TMAX Detector is set.

2.14.3 Prototype

void **ThermalShutdown**(void)

2.14.4 Inputs

Table 66 ThermalShutdown: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.14.5 Outputs

Table 67 ThermalShutdown: Output values

Register/ Address	Type	Name	Description
None	---	---	---

2.14.6 Resource Usage

Table 68 ThermalShutdown: Resources

Type	Used or Modified
Registers	R7
SFR	ACC, CFG0, DPH, DPL, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.14.7 Execution Information

Table 69 ThermalShutdown: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	$t_{T<TMAX}$	—	73	79	μs	Temp. < TMAX: function will return without any action, DIVIC = 00 _H
	$t_{T>TMAX}$	—	71	77	μs	Temp. > TMAX: function will enter THERMAL SHUTDOWN state, DIVIC = 00 _H
Charge Consumption	$Q_{T<TMAX}$	—	0,109	0,188	μC	Temp. < TMAX: function will return without any action, DIVIC = 00 _H
	$Q_{T>TMAX}$	—	0,106	0,183	μC	Temp. > TMAX: function will enter THERMAL SHUTDOWN state, DIVIC = 00 _H

2.15 StartXtalOsc()

2.15.1 Description

This function enables the crystal oscillator clock and delays for a defined amount of time. The delay time should be long enough that the crystal oscillator is stable, which is determined by the crystal startup time (see [1]). The minimum/maximum tolerance of the delay time may be derived by considering the tolerance of the 12 MHz RC Oscillator (see [1]).

2.15.2 Actions

- Enable the crystal oscillator clock
- Wait in IDLE for set Delay time

2.15.3 Prototype

signed char **StartXtalOsc**(unsigned char **Delay**)

2.15.4 Inputs

Table 70 StartXtalOsc: Input Parameters

Register / Address	Type	Name	Description
R7	unsigned char	Delay	Delay time to wait in IDLE after XTAL is enabled. Duration[μs] = Delay x 42.67[μs]

2.15.5 Outputs

Table 71 StartXtalOsc: Output values

Register/Address	Type	Name	Description
R7	signed char	StatusByte	StatusByte: 0: XTAL started and delay passed -1: XTAL already on (no action)

2.15.6 Resource Usage

Table 72 StartXtalOsc: Resources

Type	Used or Modified
Registers	R7
SFR	ACC, B, CFG0, DPTR, DIVIC, PSW, REF, TCON, TH0, TL0, TMOD
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.15.7 Execution Information

Table 73 StartXtalOsc: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time ¹⁾	t_{30}	–	1323	1440	μs	DIVIC is 0; delay is 30
	t_{40}	–	1750	1901	μs	DIVIC is 0; delay is 40
	t_{delta}	–	52,7	56,1	μs	DIVIC is 0; delta delay is 1
Charge Consumption ¹⁾	Q_{30}	–	1,22	2,15	μC	DIVIC is 0; delay is 30
	Q_{40}	–	1,6	2,83	μC	DIVIC is 0; delay is 40
	Q_{delta}	–	0,05	0,09	μC	DIVIC is 0; delta delay is 1

1) Values are obtained by extrapolation, not by characterization

2.15.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"
#include "Reg_SP37.h"

void main()
{
    // Return value of start xtal oscillator is stored in StatusByte
    unsigned char StatusByte;

    // Input parameters for start xtal oscillator
    unsigned char Delay = 33;

    // Start xtal oscillator function call
    StatusByte = StartXtalOsc(Delay);
}
```

Figure 10 Code example for usage of StartXtalOsc()

2.16 StopXtalOsc()

2.16.1 Description

This function disables the crystal oscillator clock if no other peripherals are using it. Therefore, peripherals using the crystal oscillator should be disabled prior to calling this function.

2.16.2 Actions

- Attempt to disable the crystal oscillator clock.

2.16.3 Prototype

signed char **StopXtalOsc**(void)

2.16.4 Inputs

Table 74 StopXtalOsc: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.16.5 Outputs

Table 75 StopXtalOsc: Output values

Register/Address	Type	Name	Description
R7	signed char	StatusByte	StatusByte: 0: XTAL stopped -1: XTAL already off (no action) -2: XTAL not stopped because it is still needed (e.g. due to an ongoing RF transmission)

2.16.6 Resource Usage

Table 76 StopXtalOsc: Resources

Type	Used or Modified
Registers	R7
SFR	ACC,B, CFG0, DPTR, PSW, RFS, TMOD
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.16.7 Execution Information

Table 77 StopXtalOsc: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	20	22	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,035	0,058	μC	DIVIC = 00 _H

2.16.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"
#include "Reg_SP37.h"

// Defines
#define bitmask_TMOD_CLK 0x08
#define bitmask_RFS_RFSE 0x02

void main()
{
    // Return value of stop xtal oscillator is stored in StatusByte
    signed char StatusByte;

    while (!(RFS & bitmask_RFS_RFSE));           // Wait for RF Transmission to end
    RFC &= ~0x03;                                // Disable PA & PLL
    TMOD &= ~bitmask_TMOD_CLK;                   // Clear TMOD_CLK

    StatusByte = StopXtalOsc();                   // Stop XTAL oscillator

    if(StatusByte != -2){
        // Xtal oscillator is stopped or was already off
    }
}
```

Figure 11 Code example for usage of StopXtalOsc()

2.17 PLL_Ref_Signal_Check()

2.17.1 Description

This function can be called prior to [VCO_Tuning\(\)](#) routine to test the lowest and highest VCO tuning curves that are disjoint in terms of operating frequencies. If the PLL Lock Detector indicates lock for both of the disjoint VCO tuning curves, a malfunction of the Crystal Resonator is most likely the source of the fault.

2.17.2 Actions

- Select the lowest tuning curve and check the PLL Lock Detector result
- Select the highest tuning curve and check the PLL Lock Detector result

2.17.3 Prototype

signed char **PLL_Ref_Signal_Check**(void)

2.17.4 Inputs

Table 78 PLL_Ref_Signal_Check: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.17.5 Outputs

Table 79 PLL_Ref_Signal_Check: Output values

Register/Address	Type	Name	Description
R7	signed char	StatusByte	StatusByte: 0: PLL Reference Signal available -1: No PLL Reference Signal available

2.17.6 Resource Usage

Table 80 PLL_Ref_Signal_Check: Resources

Type	Used or Modified
Registers	R7
SFR	ACC, B, PSW, DIVIC, DPTR
Stack	2 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.17.7 Execution Information

Table 81 PLL_Ref_Signal_Check: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	273	295	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,41	0,71	μC	DIVIC = 00 _H

2.18 VCO_Tuning()

2.18.1 Description

This function selects an appropriate tuning curve for the VCO and enables the PLL. If no proper tuning curve could be selected the PLL will be disabled before the function returns with an error result.

Notes

1. *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[ENPLL] to reduce the current consumption.*
2. *Re-Calibration of the tuning curve is necessary when V_{BAT} changes more than 800mV or $T_{Ambient}$ changes more than 70 °C*

2.18.2 Actions

- Select appropriate tuning curve
- Enable PLL for a RF transmission
- Wait until PLL is locked

2.18.3 Prototype

signed char **VCO_Tuning**(void)

2.18.4 Inputs

Table 82 VCO_Tuning: Input Parameters

Register / Address	Type	Name	Description
None	void	---	---

2.18.5 Outputs

VCO_Tuning: Output values

Register/ Address	Type	Name	Description
R7	signed char	StatusByte	StatusByte: 0: Success of VCO Tuning -1: VCO Tuning not successful, PLL disabled -2: XTAL not enabled (crystal required)

2.18.6 Resource Usage

Table 83 VCO_Tuning: Resources

Type	Used or Modified
Registers	R0, R1, R5, R6, R7
SFR	ACC, DIVIC, PSW, RFC
Stack	4 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.18.7 Execution Information

Table 84 VCO_Tuning: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t_{firstTUN}	—	2360	2560	μs	First tuning (after RESET), DIVIC is 00 _H
	t_{reTUN}	—	915	989	μs	Re-tuning, DIVIC is 00 _H
Charge Consumption	Q_{firstTUN}	—	14,2	24,1	μC	First tuning (after RESET), DIVIC is 00 _H
	Q_{reTUN}	—	5,524	9,2	μC	Re-tuning, DIVIC is 00 _H

2.19 IntervalTimerCalibration()

2.19.1 Description

This function initiates a calibration of the Interval Timer precounter (ITPL and ITPH) to obtain a specific interval timer timebase between 1Hz and 20Hz. The function can work with both clock sources (12MHz RC Clock and Crystal clock), utilizing a special timer mode.

Note: To obtain the best possible Interval Timer accuracy, this function should be called after the crystal oscillator has already been enabled.

In case the crystal oscillator is used the crystal frequency in Hz divided by 2 has to be stored in the FLASH user configuration sector at address 57FA_H (MSByte) to 57FC_H (LSByte). If the value found at this FLASH location is not within the range of 9 MHz to 10 MHz a default clock frequency of 9.843750 MHz (XTAL/2 for 315 MHz carrier) is assumed for the tuning.

In addition, this function automatically calibrates the LF On/Off Timer precounter (SFR LFOOTP) to 50 ms.

2.19.2 Actions

- Calibrate the interval timer precounter (SFR ITPL, SFR ITPH) to the value passed in WU_Frequency
- Calibrate the LF On/Off Timer precounter (SFR LFOOTP) to 50 ms

2.19.3 Prototype

signed char **IntervalTimerCalibration**(unsigned char **WU_Frequency**)

2.19.4 Inputs

Table 85 IntervalTimerCalibration: Input Parameters

Register / Address	Type	Name	Description
R7	unsigned char	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)

2.19.5 Outputs

Table 86 IntervalTimerCalibration: Output values

Register/ Address	Type	Name	Description
R7	signed char	StatusByte	StatusByte: 0: Success -1: No valid crystal frequency found in FLASH -2: Input parameter out of range

2.19.6 Resource Usage

Table 87 IntervalTimerCalibration: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, DPTR, ITPH, ITPL, LFOOTP, PSW, TCON, TH0, TH1, TL0, TL1, TMOD
Stack	2 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.19.7 Execution Information

Table 88 IntervalTimerCalibration: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time ¹⁾	$t_{12\text{Mhz}}$	—	1190	1680	μs	DIVIC = 00 _H , Clock Source is 12 MHz RC HF Oscillator
Execution Time ¹⁾	t_{XTAL}	—	1130	1570	μs	DIVIC = 00 _H , Clock Source is XTAL
Charge Consumption ¹⁾	$Q_{12\text{MHz}}$	—	1,23	2,56	μC	DIVIC = 00 _H , Clock Source is 12 MHz RC HF Oscillator
Charge Consumption ¹⁾	Q_{XTAL}	—	1,60	3,04	μC	DIVIC = 00 _H , Clock Source is XTAL

1) Values are obtained by extrapolation, not by characterization

2.20 LFBaudrateCalibration()

2.20.1 Description

Calling this function calibrates the LF baudrate divider using the crystal oscillator as a frequency reference, thus reducing the impact of offset and the current drift of the LF RC Oscillator upon the LF baudrate accuracy.

It is mandatory to call this function prior to the first use of the LF Receiver.

The calibrated SFR LFDIV value may be stored in the FLASH and loaded into SFR LFDIV anytime the LF Receiver is operated. If this calibration is performed regularly by the application, the bitrate tolerance of the transmitted LF data may be increased beyond the value normally specified (see [1]).

Prior to calling this function the crystal oscillator must be enabled by calling StartXtalOsc() and the crystal frequency in Hz divided by 2 has to be stored in the FLASH user configuration sector at address 57FA_H (MSByte) to 57FC_H (LSByte). If the value found at this FLASH location is not within the range of 9 MHz to 10 MHz a default clock frequency of 9.843750 MHz (XTAL/2 for 315 MHz carrier) is used for the tuning and the function returns an error in the StatusByte and sets SFR LFDIV to a nominal value.

2.20.2 Actions

- Set SFR LFDIV according to the current frequency of the LF RC Oscillator

2.20.3 Prototype

signed char **LFBaudrateCalibration**(unsigned int **baudrate**)

2.20.4 Inputs

Table 89 LFBaudrateCalibration: Input Parameters

Register / Address	Type	Name	Description
R6, R7	unsigned int	baudrate	Baudrate 3900dec: 3900 baud

2.20.5 Outputs

Table 90 LFBaudrateCalibration: Output values

Register/ Address	Type	Name	Description
R7	signed char	StatusByte	StatusByte: 0: Success -1: XTAL frequency out of range -2: XTAL not enabled or input parameter out of range

2.20.6 Resource Usage

Table 91 LFBaudrateCalibration: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, DPTR, LFDIV, PSW, TCON, TMOD, TH0, TH1, TL0, TL1
Stack	4 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.20.7 Execution Information

Table 92 LFBaudrateCalibration: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time ¹⁾	t_{XTAL}	–	983	1070	μs	Baudrate = 3900 DIVIC = 00 _H
Charge Consumption ¹⁾	Q_{XTAL}	–	1,57	2,57	μC	Baudrate = 3900 DIVIC = 00 _H

1) Values are obtained by extrapolation, not by characterization

2.21 SMulIntInt() (16Bit * 16Bit)

2.21.1 Description

This function multiplies the first signed int value (16 bit) Multiplicand1 by the second signed int value (16bit) Multiplicand2 and produces a 32-bit signed result.

2.21.2 Actions

- Perform multiplication

2.21.3 Prototype

void **SMulIntInt**(signed int idata * **Multiplicand1**, signed int idata * **Multiplicand2**, signed long idata * **Product**)

2.21.4 Inputs

Table 93 SMulIntInt: Input Parameters

Register / Address	Type	Name	Description
R7	signed int idata*	Multiplicand1	Pointer to Multiplicand1
R5	signed int idata*	Multiplicand2	Pointer to Multiplicand2
R3	signed long idata*	Product	Pointer to an long array in RAM to the 32 Bit multiplication Product (Multiplicand1 * Multiplicand2)

2.21.5 Outputs

Table 94 SMulIntInt: Output values

Register/ Address	Type	Name	Description
None	---	---	---

2.21.6 Resource Usage

Table 95 SMulIntInt: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, PSW
Stack	2 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.21.7 Execution Information

Table 96 SMulIntInt: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	75	81	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,112	0,194	μC	DIVIC = 00 _H

2.22 UDivLongLong() (32Bit : 32Bit)

2.22.1 Description

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

2.22.2 Actions

- Perform division

2.22.3 Prototype

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

2.22.4 Inputs

Table 97 UDivLongLong: Input Parameters

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

2.22.5 Outputs

Table 98 UDivLongLong: Output values

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

Note: The output value for the remainder can be found in R0 (MSB), R1, R2, R3 (LSB).

2.22.6 Resource Usage

Table 99 UDivLongLong: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, DPH, DPL, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.22.7 Execution Information

Table 100 UDivLongLong: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	387	418	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,581	1,004	μC	DIVIC = 00 _H

2.23 UDivIntInt() (16Bit : 16Bit)

2.23.1 Description

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

2.23.2 Actions

- Perform division

2.23.3 Prototype

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

2.23.4 Inputs

Table 101 UDivIntInt: Input Parameters

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

2.23.5 Outputs

Table 102 UDivIntInt: Output values

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

Note: The output value for the remainder can be found in R4 (MSB) and R5 (LSB).

2.23.6 Resource Usage

Table 103 UDivIntInt: Resources

Type	Used or Modified
Registers	R0, R4, R5, R6, R7
SFR	ACC, B, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.23.7 Execution Information

Table 104 UDivIntInt: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	104	113	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,156	0,270	μC	DIVIC = 00 _H

2.24 CRC8_Calc()

2.24.1 Description

This function calculates the CRC-8 checksum for a memory area in RAM using a fixed polynomial (x^8+x^2+x+1). The CRC-8 calculation starts with a defined preload value.

2.24.2 Actions

- Calculate CRC-8

2.24.3 Prototype

unsigned char **CRC8_Calc**(unsigned char **Preload**, unsigned char idata * **BlockStart**, unsigned char **BlockLength**)

2.24.4 Inputs

Table 105 CRC8_Calc: Input Parameters

Register / Address	Type	Name	Description
R7	unsigned char	Preload	Preload Value for the CRC Calculation. According to CCITT a value FF _H is recommended.
R5	unsigned char idata*	BlockStart	Pointer to first Byte of the Data that is to be used for calculating checksum
R3	unsigned char	BlockLength	Length in Bytes of Block that is used for calculation of the checksum, starting with *BlockStart.

2.24.5 Outputs

Table 106 CRC8_Calc: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	CRC_Result	Calculated CRC8 checksum

2.24.6 Resource Usage

Table 107 CRC8_Calc: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R7
SFR	ACC, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.24.7 Execution Information

Table 108 CRC8_Calc: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	$8 + x \cdot 9$	$9 + x \cdot 10$	μs	DIVIC = 00 _H , x is the number of bytes
Charge Consumption	Q	–	$0,012 + x \cdot 0,013$	$0,021 + x \cdot 0,023$	μC	DIVIC = 00 _H , x is the number of bytes

2.25 CRC_Baicheva_Calc()

2.25.1 Description

This function calculates a 8-bit CRC checksum for a memory area in RAM using a fixed polynomial ($x^8+x^5+x^3+x^2+x+1$). It supports the standardized TPMS data protocol from the German Association of the Automotive Industry (VDA) and is optimal for data word length of 119 bits and below. The CRC Baicheva calculation starts with a defined preload value. The VDA protocol requires that a preload value of AA_H be used.

2.25.2 Actions

- Calculate CRC Baicheva

2.25.3 Prototype

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

2.25.4 Inputs

Table 109 CRC_Baicheva_Calc: Input Parameters

Register / Address	Type	Name	Description
R7	unsigned char	Preload	Preload Value for the CRC Calculation. According to VDA protocol value AA _H is used.
R5	unsigned char idata*	BlockStart	Pointer to first Byte of the Data that is to be used for calculating checksum.
R3	unsigned char	BlockLength	Length in Bytes of Block that is used for calculation of the checksum, starting with *BlockStart.

2.25.5 Outputs

Table 110 CRC_Baicheva_Calc: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	CRC_Result	Calculated CRC Baicheva checksum

2.25.6 Resource Usage

Table 111 CRC8_Baicheva_Calc: Resources

Type	Used or Modified
Registers	R0, R3, R5, R6, R7

Table 111 CRC8_Baicheva_Calc: Resources (cont'd)

Type	Used or Modified
SFR	ACC, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.25.7 Execution Information

Table 112 CRC_Baicheva_Calc: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	11 + x*85	12 + x*92	μs	DIVIC = 00 _H , x is the number of bytes
Charge Consumption	Q	–	0,016 + x*0,128	0,028 + x*0,221	μC	DIVIC = 00 _H , x is the number of bytes

2.26 Read_ID()

2.26.1 Description

This function returns the unique serial number of the device and a product description code.

2.26.2 Actions

- Read 4-byte ID
- Read 1 byte product code

2.26.3 Prototype

void **Read_ID** (struct ID_Struct idata * idata **ID_Result**)

Note: Structure ID_Struct is defined in SP37_ROMLibrary.h

2.26.4 Inputs

Table 113 Read_ID: Input Parameters

Register / Address	Type	Name	Description
R7	idata*	ID_Result	Pointer to a structure according to the following definition: idata struct ID_Struct {unsigned long ID; unsigned char ProdCode;}

2.26.5 Outputs

Table 114 Read_ID: Input Parameters

Register / Address	Type	Name	Description
R7	idata*	ID_Result	Pointer to a structure according to the following definition: idata struct ID_Struct {unsigned long ID; unsigned char ProdCode;} Product Code pressure range indication: xxxx.x000 _B : reserved xxxx.x100 _B : reserved xxxx.x001 _B : reserved xxxx.x101 _B : 500/900 kPa, Autorange, Green Package ¹⁾ xxxx.xx10 _B : reserved xxxx.xx11 _B : reserved

1) For definition "Green Package" please refer to [\[1\]](#)

2.26.6 Resource Usage

Table 115 Read_ID: Resources

Type	Used or Modified
Registers	R0, R3, R4, R5, R6, R7
SFR	ACC, DPH, DPL, PSW
Stack	2 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.26.7 Execution Information

Table 116 Read_ID: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	32	35	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,048	0,082	μC	DIVIC = 00 _H

2.26.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

// ID structure is defined in SP37_ROMLibrary.h
idata ID_Struct ID_Result;

void main()
{
    // Read Sensor ID function call
    Read_ID(&ID_Result);
}
```

Figure 12 Code example for usage of Read_ID()

2.27 ManuRevNb()

2.27.1 Description

This function returns the Infineon SP37 revision number

2.27.2 Actions

- Read 2-byte Revision Number from ROM and return it

2.27.3 Prototype

signed int **ManuRevNb** (void)

2.27.4 Inputs

Table 117 ManuRevNb: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.27.5 Outputs

Table 118 ManuRevNb: Output values

Register/ Address	Type	Name	Description
R6(MSB), R7(LSB)	signed int	Manufacturer Revision Number	37XY _H : X = Design Step (A...F) Y = Version (0...9)

2.27.6 Resource Usage

Table 119 ManuRevNb: Resources

Type	Used or Modified
Registers	R6, R7
SFR	ACC, PSW, DPTR
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.27.7 Execution Information

Table 120 ManuRevNb: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	9	10	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,013	0,023	μC	DIVIC = 00 _H

2.28 FW_Revision_Nb()

2.28.1 Description

This function returns the ROM Library revision number and the FLASH Library revision number.

2.28.2 Actions

- Read 2-byte Revision Number from ROM and return it
- Read 2-byte Revision Number from FLASH Library and return it

2.28.3 Prototype

void **FW_Revision_Nb** (unsigned int idata * **ROM_Rev**, unsigned int idata * **Lib_Rev**)

2.28.4 Inputs

Table 121 FW_Revision_Nb: Input Parameters

Register/ Address	Type	Name	Description
R4(MSB), R5(LSB)	unsigned int idata*	Lib_Rev	Pointer to the RAM location where the FLASH Library code revision will be returned
R6(MSB), R7(LSB)	unsigned int idata*	ROM_Rev	Pointer to the RAM location where the ROM code revision will be returned

2.28.5 Outputs

Table 122 FW_Revision_Nb: Input Parameters

Register/ Address	Type	Name	Description
R4(MSB), R5(LSB)	unsigned int idata*	Lib_Rev	Pointer to the RAM location where the FLASH Library code revision will be returned xxxx _H = SP37 FLASH Library version
R6(MSB), R7(LSB)	unsigned int idata*	ROM_Rev	Pointer to the RAM location where the ROM code revision will be returned 0A32 _H = SP37 ROM Library Version A5

2.28.6 Resource Usage

Table 123 FW_Revision_Nb: Resources

Type	Used or Modified
Registers	R0, R5, R7
SFR	ACC
Stack	2 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.28.7 Execution Information

Table 124 FW_Revision_Nb: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	12	13	μs	
Charge Consumption	Q	–	0,018	0,030	μC	

2.29 Erase_UserConfigSector()

2.29.1 Description

This function erases the FLASH user configuration sector located at FLASH address 5780_H -- 57FF_H if the Lockbyte 3 is not set. If Lockbyte 3 is set this function will return -1 without any action.

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

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} are within specified range (see [1]).

2.29.2 Actions

- Erase the FLASH user configuration sector

2.29.3 Prototype

signed char **Erase_UserConfigSector** (void)

2.29.4 Inputs

Table 125 Erase_UserConfigSector: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.29.5 Outputs

Table 126 Erase_UserConfigSector: Output values

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

2.29.6 Resource Usage

Table 127 Erase_UserConfigSector: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CRC0, CRC1, CRCC, CRCD, DIVIC, DPH, DPL, PSW, TCON, TH0, TH1, TL0, TL1
Stack	5 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.29.7 Execution Information

Table 128 Erase_UserConfigSector: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	103000	111300	μs	DIVIC = 00 _H
Charge Consumption	Q	–	222	358	μC	DIVIC = 00 _H

2.30 WriteFlashUserConfigSectorLine()

2.30.1 Description

This function writes one line in the FLASH user configuration sector located at FLASH address 5780_H -- 57FF_H if the Lockbyte 3 is not set. If Lockbyte 3 is set this function will return -1 without any action.

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

The written data is verified after the programming. In case the verification fails this 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 Endurance En_{FL} are within specified range (see [1]).

Note: Before written new data, the Flash User Config Sector needs to be erased by the user.

2.30.2 Actions

- Write one 32 Byte FLASH Line of the FLASH user configuration sector

2.30.3 Prototype

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

2.30.4 Inputs

Table 129 WriteFlashUserConfigurationSectorLine: Input Parameters

Register / Address	Type	Name	Description
R7 R6	unsigned int	Startaddress	Startaddress 5780 _H : FLASH Line 0 57A0 _H : FLASH Line 1 57C0 _H : FLASH Line 2 57E0 _H : FLASH Line 3
R5	unsigned char idata*	WrData	Pointer to first Byte of the 32 Byte Data array that is going to be written to the FLASH Line.

2.30.5 Outputs

Table 130 WriteFlashUserConfigurationSectorLine: Output values

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

2.30.6 Resource Usage

Table 131 WriteFlashUserConfigurationSectorLine: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, CFG1, CRC0, CRC1, CRCC, CRCD, DIVIC, DPH, DPL, PSW, TCON, TH0, TH1, TL0, TL1, TMOD
Stack	5 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.30.7 Execution Information

Table 132 WriteFlashUserConfigurationSectorLine: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time ¹⁾	t	–	3917	4260	μs	DIVIC = 00 _H , FLASH written with all bit 1
Charge Consumption ¹⁾	Q	–	8,97	12	μC	DIVIC = 00 _H , FLASH written with all bit 1

1) Values are obtained by extrapolation, not by characterization

2.31 WriteFlashCodeSectorLine()

2.31.1 Description

This function offers the possibility to write one Code Sector (sector 0) flash line. It enables the application program to modify the code sector, thus enabling code patches or configuration blocks to be applied to the Flash during application execution.

Some requirements to use this function are mandatory:

- The code sector must be unlocked
- The flash-line to be written must be already in erased-state
- No provision for erasing the Code Sector via ROM library call will be provided
- The function can not be called in debug mode

This function writes one line (32 bytes) to the Flash Code Sector located at Flash address 4000_H -- 575F_H. If the SP37 is in RUN mode and Lockbyte 2 is not set the write is performed, otherwise this function will return -1. The line contents are verified after the entire line has been written. In the event that the write verification fails, this function will return -2.

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_{bat} and Endurance En_{FL} are within specified range (see [1]).

Note: Each Flash Code Sector Line may be written only once. Once a line is written with data, it must not be written again. there is no corresponding ROM function to erase the Flash Code Sector under application program control.

2.31.2 Actions

Write one Code Sector flash line.

2.31.3 Prototype

signed char **WriteFlashCodeSectorLine** (unsigned int **StartAddress**, unsigned char idata ***WrData**)

2.31.4 Inputs

Table 133 WriteFlashCodeSectorLine: Input Parameters

Register / Address	Type	Name	Description
R6, R7	unsigned int	Startaddress	Must be between 4000 _H and 575F _H (the line containing Lockbyte2 is excluding) Must be multiple of 32
R5	unsigned char idata*	WrData	Pointer to 32 Byte buffer in RAM holding the contents to be written to the specified Flash line.

2.31.5 Outputs

Table 134 WriteFlashCodeSectorLine: Output values

Register/ Address	Type	Name	Description
R7	signed char	Statusbyte	0: success -1: Lockbyte 2 is set (Code Sector Lock), without writing anything; or the function was called while the device is in debug Mode. -2: After data verification, errors are detected -3: The StartAddress" is not multiple of 32 or the "StartAddress" is out-of-range (nothing was performed); or "

2.31.6 Resource Usage

Table 135 WriteFlashCodeSectorLine: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, PSW, DPTR, CFG1, TCON, TMOD, TH0, TL0, TH1, TL1, DSR, CRC0, CRC1, CRCD, CRCC, DIVIC
Stack	5 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.31.7 Execution Information

Table 136 WriteFlashCodeSectorLine: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	3850	4317	μs	DIVIC = 00 _H , FLASH written with all bit 1
Charge Consumption	Q	–	7,857	11,379	μC	DIVIC = 00 _H , FLASH written with all bit 1

2.32 FlashSetLock()

2.32.1 Description

This function sets the Lockbyte 3 which protects the User Configuration sector. This function returns -1 and has no effect if executed in DEBUG mode.

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 Endurance En_{FL} are within specified range (see [1]). This function returns -1 and has no effect if executed in DEBUG mode.

2.32.2 Actions

- Set the Lockbyte 3 protecting the User Configuration sector

2.32.3 Prototype

signed char **FlashSetLock**(void)

2.32.4 Inputs

Table 137 FlashSetLock: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.32.5 Outputs

Table 138 FlashSetLock: Output values

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

2.32.6 Resource Usage

Table 139 FlashSetLock: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG1, CRC0, CRC1, CRCC, CRCD, DIVIC, DPH, DPL, TCON, TH1, TL1
Stack	8 Bytes ¹⁾
RAM ²⁾	0x10 to 0x8F

1) Two additional bytes, not included here, are needed to call the library function

2) The FlashSetLock() overwrites RAM addresses 0x10 to 0x8F during its execution. For this reason it is recommended to perform a software reset (CFG2.0[RESET] == 1) after this function returns.

2.32.7 Execution Information

Table 140 FlashSetLock: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time ¹⁾	t	–	116030	126010	μs	DIVIC = 00 _H
Charge Consumption ¹⁾	Q	–	291	404	μC	DIVIC = 00 _H

1) Values are obtained by extrapolation, not by characterization

2.33 ECC_Check()

2.33.1 Description

This function evaluates the **Error Code Correction (ECC)** result bit. This bit shows if an error in the read/executed FLASH has been detected since the last call of this function.

Note: When a FLASH byte is programmed, the ECC unit generates and stores four ECC bits. With these additional bits the ECC is able to correct single bit errors and detects two bit errors. The ECC is calculated from read/executed FLASH and the result is stored in the ECC result bit. It cannot distinguish between a single bit or two bit error.

2.33.2 Actions

- Check the ECC result bit

2.33.3 Prototype

signed char **ECC_Check**(void)

2.33.4 Inputs

Table 141 ECC_Check: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.33.5 Outputs

Table 142 ECC_Check: Output values

Register/ Address	Type	Name	Description
R7	signed char	Statusbyte	0: success -1: ECC error detected

2.33.6 Resource Usage

Table 143 ECC_Check: Resources

Type	Used or Modified
Registers	R7
SFR	ACC, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.33.7 Execution Information

Table 144 ECC_Check: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	6	7	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,009	0,016	μC	DIVIC = 00 _H

2.33.8 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

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

    // ECC check function call to reset the ECC check function
    StatusByte = ECC_Check();

    // CRC16 check function call to test User Data Sector
    StatusByte = CRC16_Check(0x5780,0x80);

    // ECC check function call
    StatusByte += ECC_Check();

    if(!StatusByte){
        // Both ECC and CRC16 check were successful
    }
    else{
        // ECC and or CRC16 check was not successful
    }
}
```

Figure 13 Code example for usage of the functions ECC_Check() and CRC16_Check()

2.34 CRC16_Check()

2.34.1 Description

This function computes a 16-bit CRC of a code block from a start address with a defined length from FLASH, which includes a pre-computed 16-bit CRC, which is stored in the last 2 Bytes of the memory block ((StartAddr + Length - 2) is CRC High) and ((StartAddr + Length - 1) is CRC Low).

Note: The pre-computed CRC value has to be written by the application for code blocks in FLASH.

2.34.2 Actions

- Calculate 16-bit CRC
- Compare CRC with pre-computed CRC located in the last 2 Bytes of the code block

2.34.3 Prototype

signed char **CRC16_Check**(unsigned char code * **StartAddr**, unsigned int **Length**)

2.34.4 Inputs

Table 145 CRC16_Check: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB) R7 (LSB)	unsigned char code*	StartAddr	Pointer to first Byte of the Data that is used for calculating the checksum
R4 (MSB) R5 (LSB)	unsigned int	Length	Length in Bytes of the code block, including the pre-computed CRC value, that is used for calculating the checksum.

2.34.5 Outputs

Table 146 CRC16_Check: Output values

Register/ Address	Type	Name	Description
R7	signed char	Statusbyte	0: CRC matches -1: CRC does not match

2.34.6 Resource Usage

Table 147 CRC16_Check: Resources

Type	Used or Modified
Registers	R4, R5, R6, R7
SFR	ACC, B, CRC0, CRC1, CRCD, DPH, DPL, PSW
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.34.7 Execution Information

Table 148 CRC16_Check: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	$22 + x \cdot 8$	$24 + x \cdot 9$	μs	DIVIC = 00 _H , x is the number of bytes
Charge Consumption	Q	–	$0,033 + x \cdot 0,012$	$0,056 + x \cdot 0,020$	μC	DIVIC = 00 _H , x is the number of bytes

2.34.8 Code Example

See [Figure 13 “Code example for usage of the functions ECC_Check\(\) and CRC16_Check\(\)” on Page 112](#)

2.35 HIRC_Clock_Check()

2.35.1 Description

This function measures the frequency of the 12 MHz RC Oscillator, using the crystal oscillator as a measurement standard. It evaluates whether the 12 MHz RC Oscillator is working in the specified range (see [1]).

The crystal oscillator is automatically started, if not already enabled, and allowed a fixed startup time of approximately 2.5ms (five 2 kHz RC LP Oscillator periods). The crystal frequency divided by 2 has to be stored in the FLASH user configuration sector at address 57FA_H (MSByte) to 57FC_H (LSByte). If the value found at this FLASH location is not within the range of 9 MHz to 10 MHz a default clock frequency of 9.843750 MHz is assumed for the checking. Before the function returns the previous clock settings are restored.

2.35.2 Actions

- Check the frequency of the 12 MHz RC Oscillator using the crystal oscillator as measurement standard

2.35.3 Prototype

signed char **HIRC_Clock_Check**(void)

2.35.4 Inputs

Table 149 HIRC_Clock_Check: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.35.5 Outputs

Table 150 HIRC_Clock_Check: Output values

Register/ Address	Type	Name	Description
R7	signed char	Statusbyte	0: success -1: 12 MHz RC Oscillator out of specified range -2: No valid crystal frequency found in FLASH (12 MHz RC Oscillator in specified range for default crystal frequency) -3: No valid crystal frequency found in FLASH (12 MHz RC Oscillator out of specified range for default crystal frequency)

2.35.6 Resource Usage

Table 151 HIRC_Clock_Check: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, DIVIC, DPH, DPL, PSW, TCON, TH0, TH1, TL0, TL1, TMOD
Stack	5 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.35.7 Execution Information

Table 152 HIRC_Clock_Check: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t_{OFF}	–	3430	5210	μs	DIVIC = 00H, XtalOSC is OFF at the start of the function
Execution Time	t_{ON}	–	820	891	μs	DIVIC = 00H, XtalOSC is ON at the start of the function
Charge Consumption	Q_{OFF}	–	3,81	8,69	μC	DIVIC = 00H, XtalOSC is OFF at the start of the function
Charge Consumption	Q_{ON}	–	1,48	2,35	μC	DIVIC = 00H, XtalOSC is ON at the start of the function

2.36 GetCompValue()

2.36.1 Description

This function retrieves an 8 bit value from a 2 dimensional lookup table depending on input values Value1 and Value2. The lookup table is a M by N matrix and can be of any size from 2 x 2 up to 15 x 15 holding 225 different values in its maximum configuration. This function may be used to obtain a compensation value from a lookup table with 2 threshold types (e.g. temperature and voltage), according to the measured input values (Value1 and Value2). The lookup table has to be stored in the FLASH.

The input values are compared against threshold points defined in the lookup table, and the thresholds must be sorted in increasing order. The column is chosen by Value1 thresholds and the row is selected by Value2 thresholds respectively. If Value1 is lower than its lowest threshold then the left-most column is selected, and likewise if Value2 is lower than its lowest threshold the top-most row is selected.

The matrix size is always 1 larger than the number of threshold points. Thus, a 12 by 7 matrix will have 6 threshold points for Value1 (column) and 11 threshold points for Value2 (row).

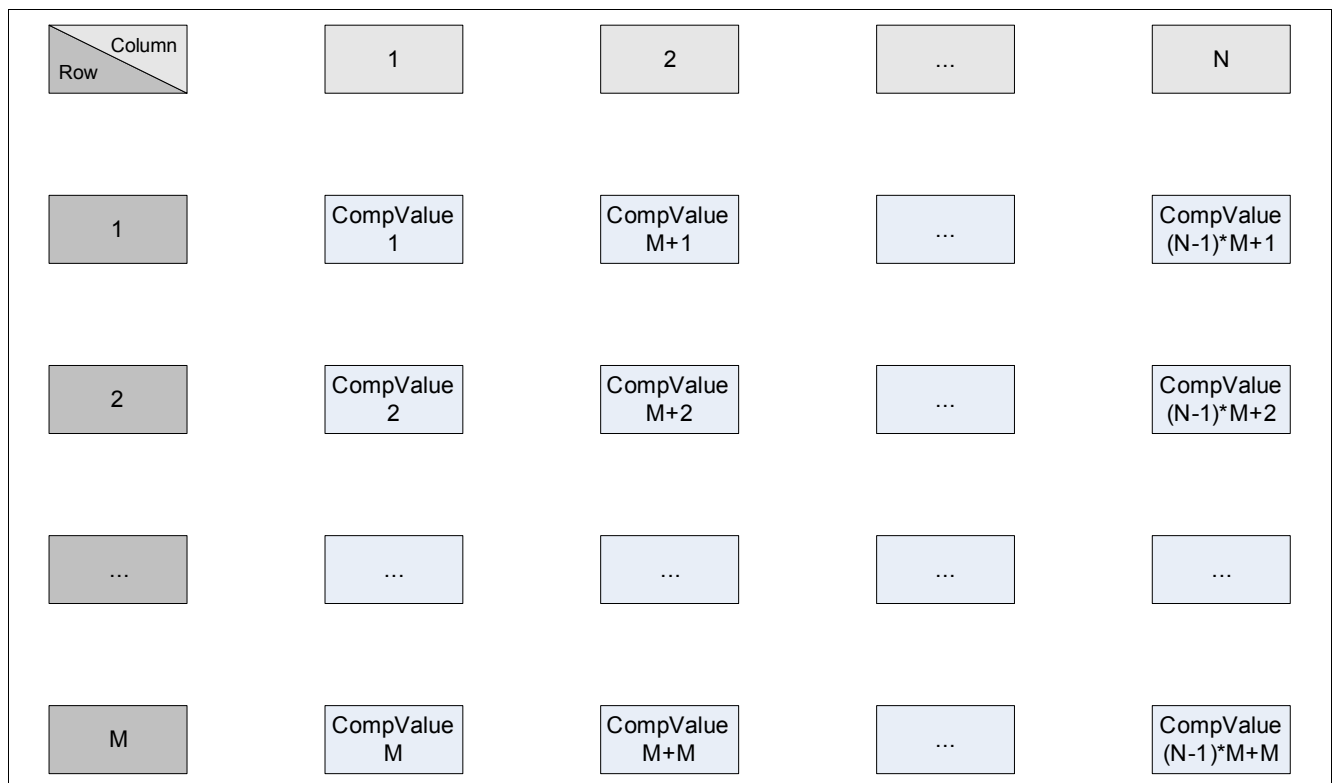


Figure 14 M by N matrix

2.36.2 Actions

- Compare Value1 with defined thresholds from threshold type 1
- Compare Value2 with defined thresholds from threshold type 2
- Return compensated value

2.36.3 Prototype

unsigned char **GetCompValue**(unsigned char code * **TablePointer**, unsigned char **Value1**, unsigned char **Value2**)

2.36.4 Inputs

Table 153 GetCompValue: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB) R7 (LSB)	unsigned char code*	TablePointer	Pointer to the first Byte of lookup table
R5	unsigned char	Value1	Measured Value1 for comparison with thresholds from threshold type 1 to select column
R3	unsigned char	Value2	Measured Value2 for comparison with thresholds from threshold type 2 to select row

Figure 15 shows how the compensated value table must appear in FLASH memory.

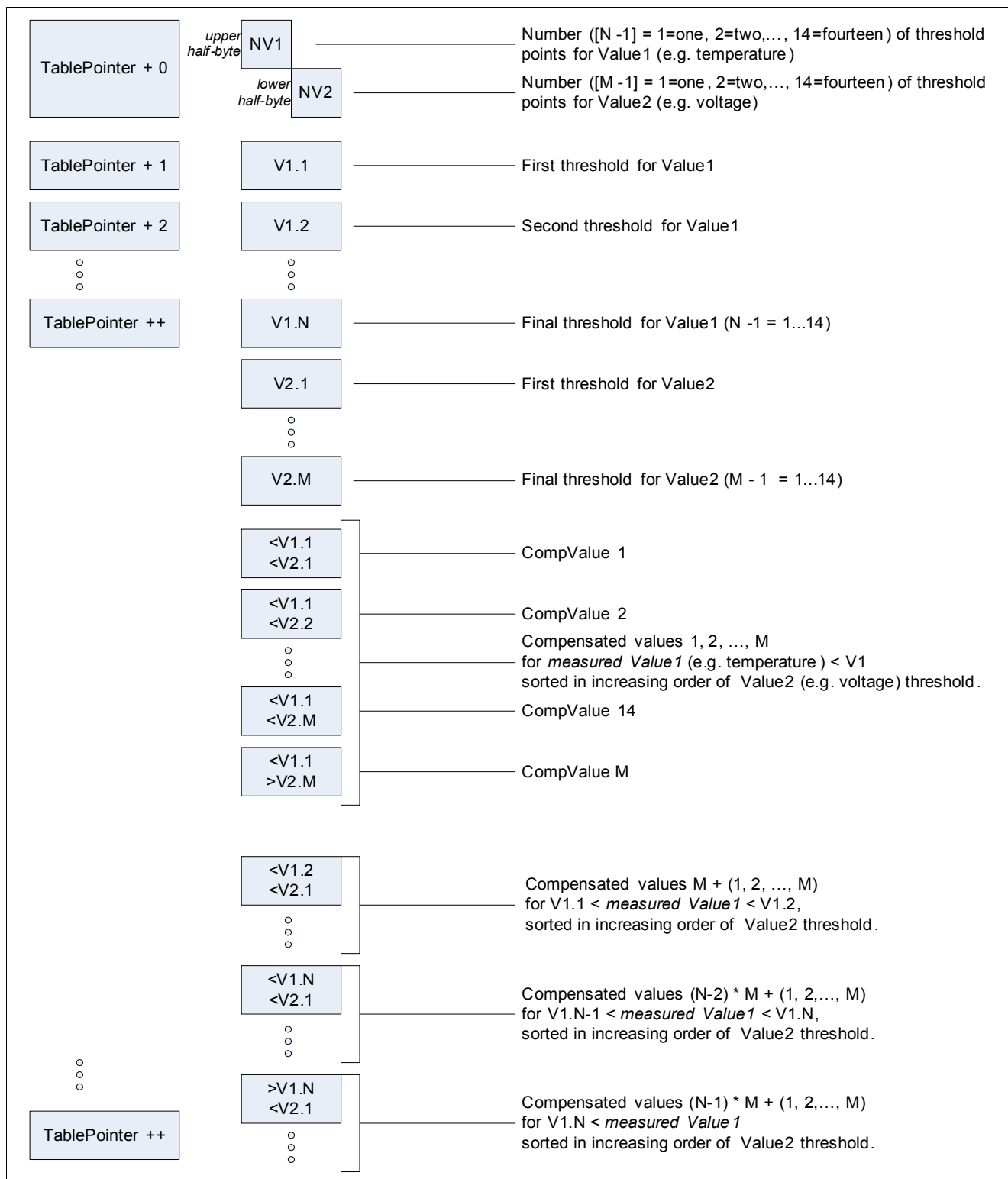


Figure 15 Lookup table organization

2.36.5 Outputs

Table 154 GetCompValue: Output values

Register/ Address	Type	Name	Description
R7	unsigned char	Compensated Value	Returns the compensated value from the lookup table defined by selected column and row

2.36.6 Resource Usage

Table 155 GetCompValue: Resources

Type	Used or Modified
Registers	R0, R1, R3, R5, R6, R7
SFR	ACC, DPH, DPL, PSW, SP
Stack	2 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.36.7 Execution Information

Table 156 GetCompValue: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	619	669	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,929	1,606	μC	DIVIC = 00 _H

2.37 Wait100usMultiples()

2.37.1 Description

This function performs a delay of 100 μ s multiples. It clears the CPU clock divider SFR DIVIC, initializes the timers according to the delay time, and uses IDLE state during the delay time. In case an unexpected resume event occurs, the IDLE state will be left and the function remain in RUN state until the delay is elapsed. The total duration of the delay can be determined by an offset time of 28 μ s plus the multiplication of the input parameter value with 100 μ s. The maximum delay time is limited in practice by the watchdog timer, which may be reset by the application prior to the function call.

2.37.2 Actions

- Set SFR DIVIC = 00_B
- Configure timers according to the delay time
- Enter IDLE state
- Delay until timer is elapsed

2.37.3 Prototype

void **Wait100usMultiples**(unsigned int **Counter**)

2.37.4 Inputs

Table 157 Wait100usMultiples: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB) R7 (LSB)	unsigned int	Counter	Number of 100 μ s multiples. Duration[μ s] = Counter x 100 μ s

2.37.5 Outputs

Table 158 Wait100usMultiples: Output values

Register/Address	Type	Name	Description

2.37.6 Resource Usage

Table 159 Wait100usMultiples: Resources

Type	Used or Modified
Registers	R6, R7
SFR	ACC, B, CFG0, CFG2, DIVIC, DPTR, PSW, TCON, TH0, TL0, TH1, TL1, TMOD
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.37.7 Execution Information

Table 160 Wait100usMultiples: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t_{100}	–	131	138	μs	DIVIC is 00 _H , counter = 1
	t_{600}	–	636	682	μs	DIVIC is 00 _H , counter = 6
	t_{+100}	–	100	109	μs	DIVIC is 00 _H , counter is increased
Charge Consumption	Q_{100}	–	0,113	0,19	μC	DIVIC is 00 _H , counter = 1
	Q_{600}	–	0,393	0,788	μC	DIVIC is 00 _H , counter = 6
	Q_{+100}	–	0,057	0,12	μC	DIVIC is 00 _H , counter is increased

2.38 Send_RF_Telegram()

This ROM Library Function performs many of the critical tasks required to transmit RF telegrams, resulting in simplified and smaller application code.

2.38.1 Description

In principle, transmission of an RF telegram using the SP37 can be divided into two distinct portions: RF Peripheral Initialization, and RF Data Transmission. In the case of the Send_RF_Telegram() function, the application program must perform the Initialization, and the Send_RF_Telegram() function addresses Data Transmission. To permit arbitrary RF telegram formats, an interpreted table-driven approach is employed. The Data Transmission portion relies upon a Pattern Descriptor Table (PDT) to define the format and content of the RF telegram. At the end of the Data Transmission portion, the Crystal Oscillator and RF Transmitter circuitry are disabled.

Prior to calling the Send_RF_Telegram() function, the following Initialization actions must be completed:

- The RF Transmitter SFR RFTX must be configured to select the appropriate Frequency Band, Output Power, etc. Refer to the SFR RFTX description of the datasheet.
- The Crystal Oscillator must be enabled via the StartXtalOsc() ROM Library function (Ref. to [Chapter 2.15](#)).
- The RF PLL must be enabled and initialized via the VCO_Tuning() ROM Library function (Ref. to [Chapter 2.18](#))
- An RF Telegram Pattern Descriptor Table must be placed into RAM (see below for more details).

Note: Additionally, if the randomized delay capability of Send_RF_Telegram() is used, it is strongly recommended to seed the Pseudo-Random Number generator with a value unique to each sensor (e.g. the LSB of the unique Sensor ID) in order to obtain better randomization. This is particularly important if the random delay is used as part of an “anti-collision” technique to prevent overlapping RF telegrams from two different sensor modules.

After successful Initialization, the application code may call the Send_RF_Telegram() function.

Only two parameters are required by this function to completely specify the RF telegram: the desired baud rate and the starting RAM address of the PDT. If an error is encountered while parsing the PDT, the Send_RF_Telegram() function will terminate with a return code of -1.

2.38.1.1 Baud Rate parameter

The baud rate parameter is a formal parameter passed to the Send_RF_Telegram() function. This is an integer value, 1LSB = 1BPS. The Send_RF_Telegram function configures the SP37 Timer peripheral with the appropriate mode, and calculates the required timer reload value using the crystal frequency stored in the flash User Configuration Sector (reference here). The baud rate parameter must be between 13 and 10000, the minimum and maximum supported baud rates.

2.38.1.2 Pattern Descriptor table

The Pattern Descriptor Table (PDT) provides a flexible means to transmit an RF telegram of arbitrary structure and content. A PDT begins with a Start of Table indicator, followed by one or more Pattern Descriptor entries, and concludes with an End of Table indicator. A conceptual diagram of a PDT is shown in [Figure 16](#).

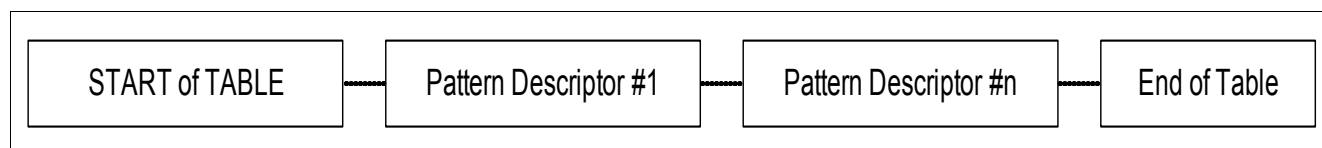


Figure 16 Datagram format

2.38.1.3 Start of Table indicator

The Start of Table is always the first byte of the PDT. The Start of Table is detailed in [Figure 17](#).

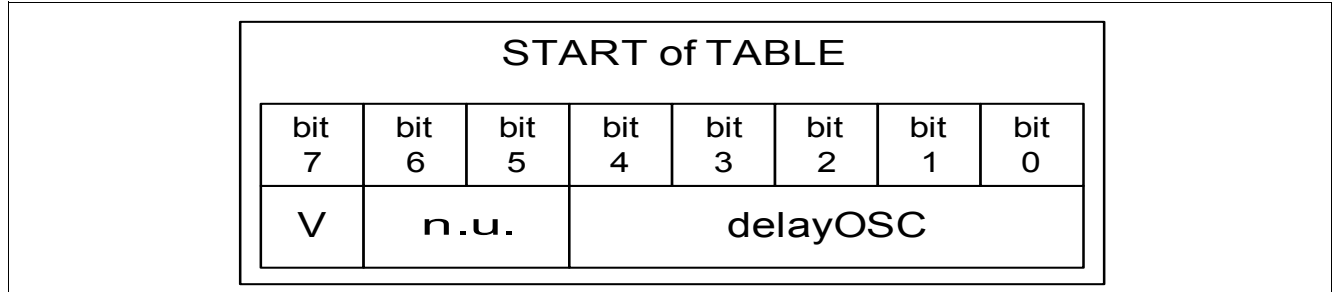


Figure 17 Datagram format

Bits 4..0 - delayOSC - define the delay for the StartXtalOsc() used in the RF_Transmission routine. The delay is calculated as $(30 + \text{delayOSC}) \times 42,67 \mu\text{s}$. This delay is used as that for crystal oscillator startup, which takes place before transmission of each repeated frame.

Bits 6..5 - not used - these bits should be set to 0.

Bit7 - V - enables the measurement of the battery voltage at the end of each transmitted frame, by means of the ADC. Only the last measurement is stored into RAM and can be read by the application.

Note: In order to operate the ADC for battery voltage measurements, Start_Supply_Voltage() must be called before Send_RF_Transmission is called. In order to read the measured supply voltage after the RF telegram has completed, the Get_Supply_Voltage function may be called. Note that the ADC is enabled and in standby mode during the duration of the Send_RF_Transmission function, therefore the total device current consumption during RF transmission will exceed the value typically specified.

2.38.1.4 Pattern Descriptor entries

In general, the concept is to divide the RF telegram into blocks of similar bit encoding and modulation which can be described by one Pattern Descriptor entry. A collection of Pattern Descriptors form the bulk of the PDT. Consider the following example: an RF telegram with some number of Manchester coded Run-In (or "preamble") bits, followed by a special Start of Message symbol (a "code violation") followed by the Manchester coded Message Payload bits. The Run-In and Message Payload are transmitted bitwise (e.g. Manchester coded) while the Start of Message symbol must be transmitted as chips because it is not Manchester coded data. The PDT for this example consists of three Pattern Descriptors, appearing in the table in order of transmission. In addition to Pattern Descriptors that define data and how it is transmitted, a "delay" type of Pattern Descriptor is also supported. A Pattern Descriptor is composed of byte data in RAM and can be one of two types; a Transmit type or a Delay type. Each of these Pattern Descriptor types is discussed in detail below.

2.38.1.5 Transmit Type Pattern Descriptor

A Transmit type Pattern Descriptor provides the RF Transmitter circuitry with serial data to encode and modulate. The Transmit type is detailed in the figure below.

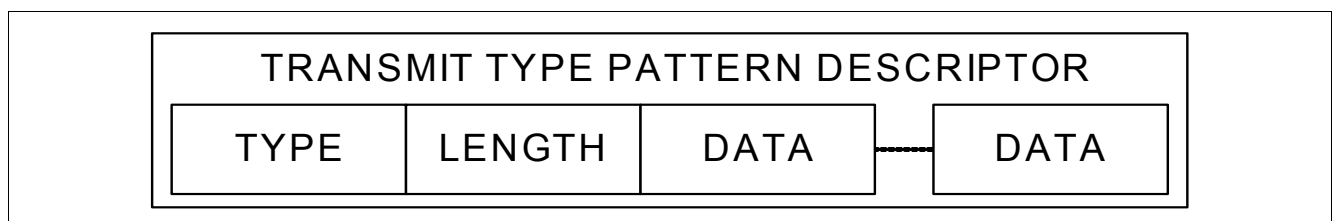


Figure 18 Transmit Type Pattern Descriptor

The first byte 'TYPE' of a Transmit type Pattern Descriptor specifies the encoding and modulation scheme for that Pattern:

- 00_H Manchester
- 01_H Inverted Manchester
- 02_H Differential Manchester
- 03_H Biphase 0
- 04_H Biphase 1
- 05_H Chip Mode (at chiprate, i.e. double datarate)
- +00_H for FSK Modulation
- +10_H for ASK Modulation

So for example:

- 03_H stands for Biphase 0 encoded data, FSK modulated
- 12_H stands for Differential Manchester encoded data, ASK modulated

The second byte 'LENGTH' specifies the length of the Pattern in bits (or chips in the case of TYPE = 05_H or 15_H). LENGTH can be any value from 01_H (1 bit/chip) to FF_H (255 bits/chips), and 00_H (256 bits/chips).

The remaining bytes in the Pattern Descriptor 'DATA' contain the data which has to be encoded and transmitted. Each byte in DATA is transmitted MSB first. In the case of a pattern with LENGTH that is not a multiple of eight, the final remaining bits are transmitted MSB first from the final DATA byte.

2.38.1.6 Delay Pattern descriptor

A Delay type Pattern Descriptor serves to disable the RF Transmitter circuitry for either a fixed or pseudo-random time delay. During Delay, the CPU is placed into IDLE mode to reduce overall current consumption. The Delay type is detailed in [Figure 19](#).

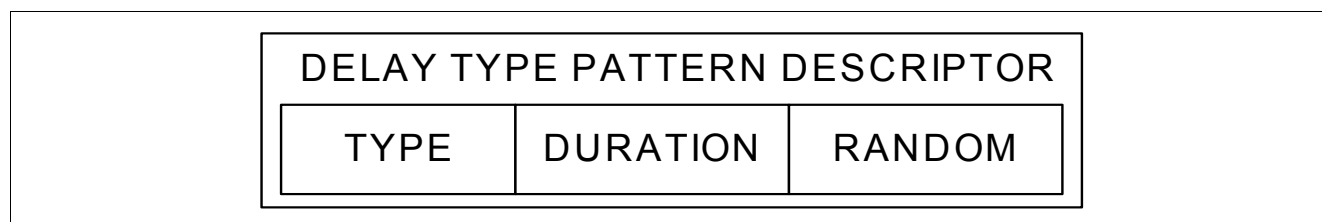


Figure 19 Delay Type Pattern Descriptor

The first byte 'TYPE' of a Delay type Pattern Descriptor is a fixed value of DD_H.

The second byte 'DURATION' specifies a fixed duration of delay time, in milliseconds [ms].

The third byte 'RANDOM' specifies the maximum duration of a randomly generated delay time, in milliseconds [ms]. The Random Delay will be uniformly distributed between 0 and the value of 'RANDOM', in milliseconds [ms].

The total delay achieved with a Delay type Pattern Descriptor is the sum of the fixed and randomly generated delay times. To realize a Pattern with a fixed delay time, simply set the DURATION byte accordingly and set the RANDOM byte to zero. To realize a Pattern with a random delay time, set the DURATION byte and the RANDOM byte accordingly: the result will be a fixed amount of time on the top of which a random delay is added.

The random delay is obtained using the SP37 Pseudo Random Number (PRN) generator peripheral. In order to achieve some "randomness" between individual devices, it is recommended to seed the PRN generator with a value unique to each device. e.g. the LSB of the Sensor ID.

2.38.1.7 End of Table pattern descriptor

A PDT must end with an End of Table (EOT) entry. In addition to marking the end of the table, the EOT allows for one or more repetitions of the entire PDT. The EOT is detailed in [Figure 20](#).

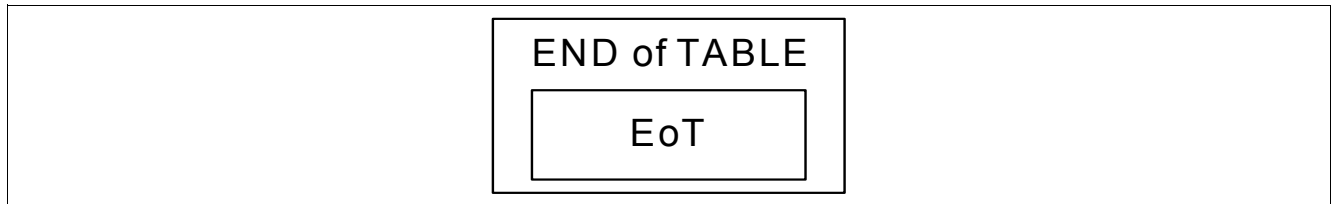


Figure 20 End of Table

The exact value of the EOT byte indicates how many times the PDT should be processed.

- F1_H process Pattern Descriptor once
- F2_H process Pattern Descriptor twice
- ...
- FF_H process Pattern Descriptor fifteen times
- F0_H process Pattern Descriptor sixteen times

For example, the PDT describes a single RF telegram, and if four duplicate telegrams are desired, then an EOT marker of F4H is required. Note that with each processing of the PDT, any RANDOM delays are taken with a new random value obtained from the Random Number

2.38.2 Flow Chart of Send_RF_Telegram()

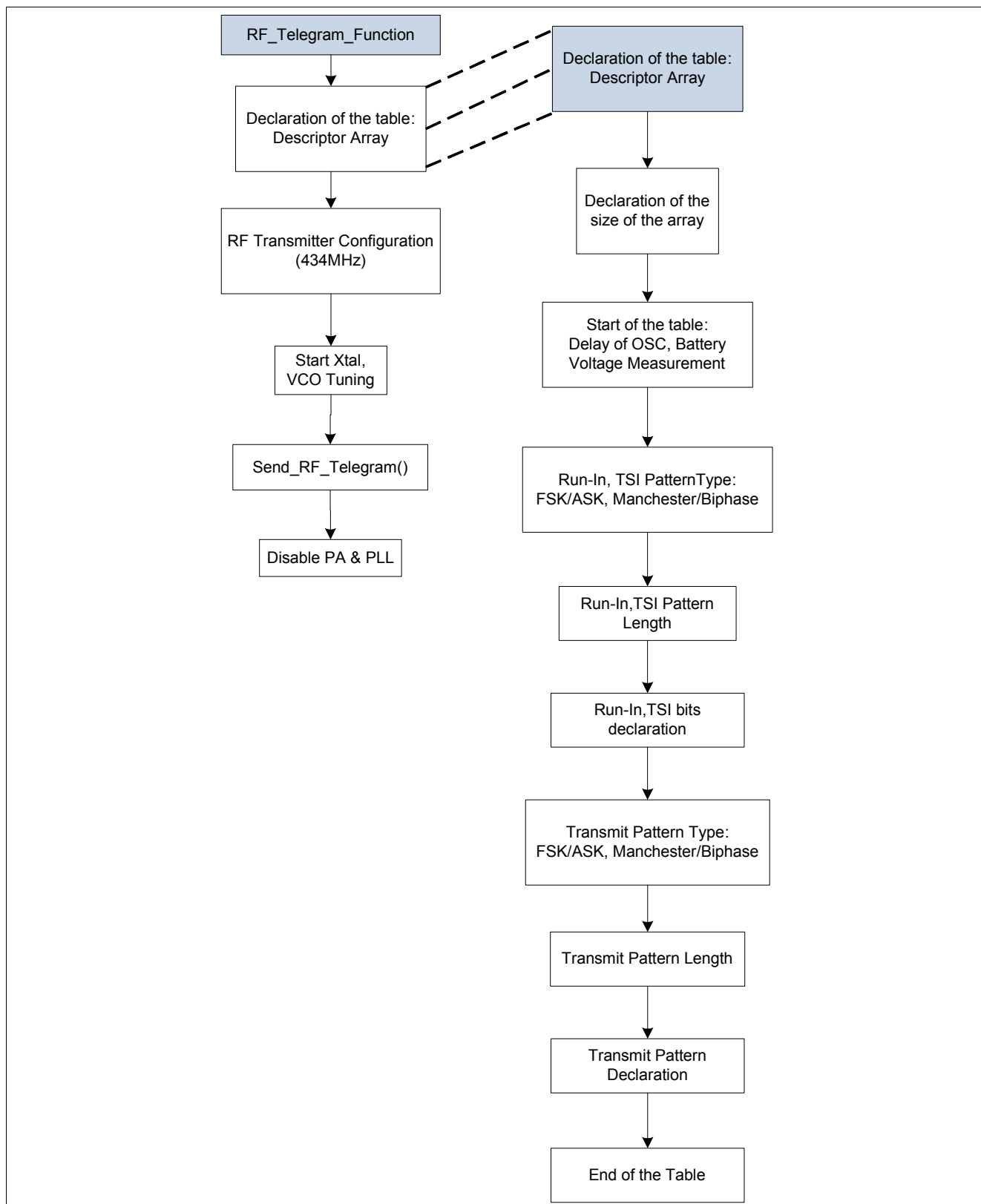


Figure 21 Flow Chart of the `Send_RF_Telegram()` function

2.38.3 Actions

- Set the Timer mode and reload values to achieve the appropriate baudrate
- Process the Pattern Descriptor Table (one or more repetitions, as determined by EOT byte) and transmit the data
- Disable the Crystal Oscillator and the RF transmitter PLL

2.38.4 Prototype

signed char Send_RF_Telegram(unsigned int baudrate, unsigned char idata * descriptorPtr)

2.38.5 Inputs

Table 161 Send_RF_Telegram: Input Parameters

Register / Address	Type	Name	Description
R6 (MSB), R7 (LSB)	unsigned int	baudrate	Baudrate [bps] of transmitted data
R5	unsigned char idata *	descriptorPtr	Register address for data vector descriptor

2.38.6 Outputs

Table 162 Send_RF_Telegram: Output values

Register/ Address	Type	Name	Description
R7	signed char		StatusByte: 0: Success -1: Error in data vector descriptor

2.38.7 Resource Usage

Table 163 Send_RF_Telegram: Resources

Type	Used or Modified
Registers	R0, R1, R2, R3, R4, R5, R6, R7
SFR	ACC, B, CFG0, CFG1, CFG2, DIVIC, DPH, DPL, PSW, RFC, RFD, RFENC, RFTX, RFS, RNGD, TCON, TH0, TH1, TL0, TL1, TMOD
Stack	10 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.38.8 Execution Information

Table 164 Send_RF_Telegram: Execution Time

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Function overhead for 1 transmission burst	<i>t</i>	–	1082	1176	μs	
1 chip transmitted @9600 bps	<i>t</i>	–	52,1	52,6	μs	
1 chip transmitted @10000 bps	<i>t</i>	–	50	50,6	μs	
1 chip transmitted @20000 bps ¹⁾	<i>t</i>	–	25	25,6	μs	
3 ms fixed delay	<i>t</i>	–	3160	3319	μs	
1 ms additional delay	<i>t</i>	–	1000	1087	μs	total delay >= 3ms

1) Values are obtained by extrapolation, not by characterization

Table 165 Send_RF_Telegram: Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Function overhead for 1 transmission burst	Q	–	3,23	5,498	μC	
1 chip transmitted @9600 bps	Q	–	0,278	0,512	μC	PA disabled (e.g. ASK off)
	Q	–	0,435	0,623	μC	SFR RFTX.PAOP = 00 _b P _{out} ~ 5 dBm Z _{load,434MHz} ~ 500 Ohm Z _{load,315MHz} ~ 650 Ohm
	Q	–	0,534	0,780	μC	SFR RFTX.PAOP = 01 _b P _{out} ~ 8 dBm Z _{load,434MHz} ~ 300 Ohm Z _{load,315MHz} ~ 450 Ohm
1 chip transmitted @10000 bps	Q	–	0,267	0,446	μC	PA disabled (e.g. ASK off)
	Q	–	0,418	0,598	μC	SFR RFTX.PAOP = 00 _b P _{out} ~ 5 dBm Z _{load,434MHz} ~ 500 Ohm Z _{load,315MHz} ~ 650 Ohm
	Q	–	0,513	0,749	μC	SFR RFTX.PAOP = 01 _b P _{out} ~ 8 dBm Z _{load,434MHz} ~ 300 Ohm Z _{load,315MHz} ~ 450 Ohm
1 chip transmitted @20000 bps ¹⁾	Q	–	0,134	0,249	μC	PA disabled (e.g. ASK off)
	Q	–	0,209	0,303	μC	SFR RFTX.PAOP = 00 _b P _{out} ~ 5 dBm Z _{load,434MHz} ~ 500 Ohm Z _{load,315MHz} ~ 650 Ohm
	Q	–	0,257	0,38	μC	SFR RFTX.PAOP = 01 _b P _{out} ~ 8 dBm Z _{load,434MHz} ~ 300 Ohm Z _{load,315MHz} ~ 450 Ohm
3 ms fixed delay	Q	–	4,223	8,17	μC	
1 ms additional delay	Q	–	0,673	1,196	μC	total delay >= 3ms

1) Values are obtained by extrapolation, not by characterization

2.38.9 Code Example

```
// Library function prototypes
#include "SP37_ROMLibrary.h"

void main()
{
    // Return value of send RF telegram is stored in sendRF_StatusByte
    signed char sendRF_StatusByte;

    // Array for pattern descriptor table
    unsigned char idata descriptorPtr[10];

    // Start of table pattern indicator
    // battery measurement enabled, no delay for delayOSC
    descriptorPtr[0] = 0x80;

    // Transmit type pattern descriptor
    // Type: ASK, Manchester
    descriptorPtr[1] = 0x10;

    // Length: 14 bits
    descriptorPtr[2] = 14;

    // Data: 8 bits (7->0) transmitted: 01010101
    descriptorPtr[3] = 0x55;

    // Data: 6 bits (7->2) transmitted: 101001
    descriptorPtr[4] = 0xA5;

    // Transmit type pattern descriptor
    // Type: FSK, Manchester
    descriptorPtr[5] = 0x00;

    // Length: 16 bits
    descriptorPtr[6] = 16;

    // Data: 8 bits (7->0) transmitted: 10101010
    descriptorPtr[7] = 0xAA;

    // Data: 8 bits (7->0) transmitted: 01010101
    descriptorPtr[8] = 0x55;

    // End of table pattern descriptor
    descriptorPtr[9] = 0xF1;

    // These tasks need to be done, details are application dependent
    // RF Transmitter SFR Initialization
    // Start xtal oscillator function call
    // VCO tuning function call
    // Start supply voltage function call (if required)

    // Send RF telegram function call, baudrate = 9600 bit/s
    sendRF_StatusByte = Send_RF_Telegram(9600, descriptorPtr);

    // Get supply voltage function call (if required)
}
```

Figure 22 Code example for usage of Send_RF_Telegram()

2.39 Internal_SFR_Refresh()

2.39.1 Description

This function refreshes on demand all internal registers which are refreshed by reset only.

2.39.2 Actions

- Loads default values into the internal SFRs which are refreshed at reset only

2.39.3 Prototype

void Internal_SFR_Refresh(void)

2.39.4 Inputs

Table 166 Internal_SFR_Refresh: Input Parameters

Register / Address	Type	Name	Description
None	---	---	---

2.39.5 Outputs

Table 167 Internal_SFR_Refresh: Output values

Register/ Address	Type	Name	Description
None	---	---	---

2.39.6 Resource Usage

Table 168 Internal_SFR_Refresh: Resources

Type	Used or Modified
Registers	---
SFR	ACC, DPTR
Stack	0 Bytes ¹⁾

1) Two additional bytes, not included here, are needed to call the library function

2.39.7 Execution Information

Table 169 Internal_SFR_Refresh: Execution Time and Charge Consumption

Parameter	Symbol	Values			Unit	Note / Test Condition
		Min.	Typ.	Max.		
Execution Time	t	–	27	29,2	μs	DIVIC = 00 _H
Charge Consumption	Q	–	0,041	0,071	μC	DIVIC = 00 _H

3 Reference Documents

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

[1] SP37 Datasheet

www.infineon.com