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

RF ASK/FSK Transmitter

Application Note

Revision 1.0, 2011-10-26

Sense & Control

Edition 2011-10-26

Published by
Infineon Technologies AG
81726 Munich, Germany

Confidential

Revision History	
Page or Item	**Subjects (major changes since previous revision)**
Revision 1.0, 2011-10-26	

Table of Contents

List of Figures

Introduction

The SP37 is a high integrated Tire Pressure Monitoring Sensor with a low power embedded micro-controller and wireless FSK/ASK RF transmitter. The RF transmitter can operate in either the 315 or 434 MHz frequency bands and can be configured for an output power of either 5 or 8dBm. This document describes how the application software must configure the RF transmitter for typical operation.

Typically, several steps must be performed for proper RF transmitter operation:

- RF transmitter band, modulation, and output power configuration
- RF symbol encoding configuration and baud rate generation
- Starting the Crystal Oscillator (Xtal Osc)
- Voltage Controlled Oscillator (VCO) Tuning
- Transmission RF data
- Stopping the Crystal Oscillator

1 RF Transmitter Configuration

The first step that must be performed is to configure the RF Transmitter Special Function Registers (SFR) RFTX to select the desired operating frequency band (315 or 434 MHz), RF Power Amplifier (PA) output power (5 or 8 dBm) and RF carrier modulation (ASK/FSK). The first two selections (frequency band, RF PA power) are straightforward. The RF carrier modulation, however, is a bit more complex.

To achieve FSK transmission, the PLL reference oscillator frequency is changed during transmission. The PLL reference frequency is that of the crystal oscillator, and is changed by switching between two different crystal load capacitances (one for the low and one for the high FSK frequency). These capacitance values are achieved with embedded switchable capacitors and/or with external capacitors, mounted in series to the crystal. For ASK transmission, only a single capacitance value is required. The SP37 offers the possibility to use either internal capacitors and/or external capacitors for pulling the crystal frequency. This is determined by the SFR bit RFTX.FSKSWITCH.

When the internal capacitors are used for FSK modulation, the desired capacitor values are determined by the values loaded into SFR XTAL1 and SFR XTAL0. The SFR bit FSKSWITCH must be cleared for this configuration. During FSK transmission the output of the Manchester/Biphase Encoder determines whether SFR XTAL1 or SFR XTAL0 is selected. In case of ASK transmission the value contained in SFR XTAL1 is always applied to the internal capacitors, and the value of SFR XTAL0 is irrelevant. Additional information is given in the datasheet (Chapter 3.8. Crystal Pulling).

When the external capacitors are used for FSK modulation, the SFR bit RFTX.FSKSWITCH must be set to enable FSK transmission. During FSK transmission the output of the Manchester/Biphase Encoder opens and closes the FSK switch, thereby applying either a single external capacitor, or both capacitors, as the crystal load capacitor. In case of ASK modulation the FSK switch is under control of the application software. Additional information is given in the datasheet (Chapter 3.8. Crystal Pulling).

1.1 Code Example

```
// RF Transmitter SFR Initialization
               // RFTX.7[FSKSWITCH]  FSK modulation switch
               // RFTX.6[ITXD]       Invert Transmit Data
               // RFTX.5[ASKFSK]     ASK/FSK Modulation Select (handled by Send RF Telegram)
               // RFTX.4[UNUSED]
               // RFTX.3:2[ISMB]     Frequency Band Select
               // RFTX.1:0[PAOP]     Power Amplifier Output Stage Select

       // Internal Capacitor Bank Setting
               // XTAL0.7:0[FSKLOW]  Internal capacitor value for FSK low frequency
               // XTAL1.7:0[FSKHASK] Internal capacitor value for FSK High or ASK center frequency

// RF Transmitter Configuration 1:
void RF_Configuration_2ICAP_FSK_434_8dbm (void)
{
       // FSK switch open, FSK with internal pulling capacitors, data inv., 433.92 MHz, 2PA stages
       RFTX = 0x45;
       // Internal capacitor bank values for 40kHz FSK deviation
       XTAL0 = 0x00;
       XTAL1 = 0x28;
}
// RF Transmitter Configuration 2:
void RF_Configuration_1ICAP_ASK_434_8dbm (void)
{
       // FSK switch open, ASK with internal pulling capacitors, data inv., 433.92 MHz, 2PA stages
       RFTX = 0x65;
       // Internal capacitor bank value for ASK center frequency of 433.92 MHz
       XTAL1 = 0x15;
}
// RF Transmitter Configuration 3:
void RF_Configuration_2ECAP_FSK_315_5dbm (void)
{
       // FSK switch closed, FSK/ASK (low frequency) with 2 external pulling capacitors
       // data not inverted, 315 MHz, 1PA stage
       RFTX = 0x80;
       // Internal capacitor bank must be set to zero for FSK with external pulling capacitors
       XTAL0 = 0x00;
       XTAL1 = 0x00;
}
// RF Transmitter Configuration 4:
void RF_Configuration_2ECAP_ASK_315_5dbm (void)
{
       // FSK switch open, ASK (high frequency) with 2 external pulling capacitors
       // data not inverted, 315 MHz, 1PA stage
       RFTX = 0x20;
       // Internal capacitor bank must be set to zero for FSK with external pulling capacitors
       XTAL0 = 0x00;
       XTAL1 = 0x00;
}
```

```
// RF Transmitter Configuration5:
void RF_Configuration_1ECAP_ASK_315_5dbm (void)
{
        // FSK switch open, ASK with 1 external pulling capacitor, data not inv., 315 MHz, 1PA stage
        RFTX = 0x20;
        // Internal Capacitor Bank can be used to tune the center frequency
        XTAL1 = 0x05; // slide tuning of center frequency
        //XTAL1 = 0x00; // no tuning
}
```

Figure 1 Configure SFR RFTX register for RF transmission

2 Starting the Crystal Oscillator

After the configuration of the RF Transmitter peripheral, the crystal oscillator (Xtal Osc) must be started to provide a stable reference frequency for the RF Phase Locked Loop (PLL) synthetizer. The Xtal Osc is enabled by calling the StartXtalOsc (delay) ROM Library Function (see [2]). This function enables the crystal oscillator circuit and then delays for a time duration proportional to the delay parameter. The delay time must be long enough that the crystal oscillator is stable, which is determined by the crystal start-up time (see [1]).

$$Delay\_Time[\mu s] = Delay \bullet 42{,}67[\mu s]$$

Figure 2 Equation for Delay

Note that the StartXtalOsc() function will return a result code upon completion, which may be examined before proceeding to subsequent steps before RF transmission. A result code of 0 indicates that the crystal oscillator has been started and the delay duration has elapsed. A result code of -1 indicates that the crystal oscillator was already running, so typically no further action is required in this case.

The crystal load capacitance must be established prior to enabling the crystal oscillator. Therefore, both XTAL0 and XTAL1 should be set to the appropriate values prior to calling the StartXtalOsc() function.

2.1 Code Example

```
void Start_Xtal (void)
{
   // 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);
  If (StatusByte == -1)
  {
        // Xtal oscillator is started or was already running
       // ……
   }
  else
  {
      // xtal oscillator not started
     //…
  }
}
```

Figure 3 Start Crystal Oscillator

3 VCO Tuning

After configuration of the RF Transmitter peripheral and starting the crystal oscillator, the RF synthesizer's VCO must be tuned for temperature and voltage conditions. This is accomplished through a call to the ROM Library Function VCO_Tuning(). This ROM function selects an appropriate tuning curve for the VCO and enables the PLL. The VCO_Tuning() function will return a result code upon completion, which must be examined before proceeding with RF transmission. A result code of -1 indicates that the VCO tuning was not able to successfully find a VCO tuning configuration and a "default" tuning configuration is used. In general, in this situation the best course of action is to stop the PLL and not attempt any transmission. A return code of -2 occurs when the crystal oscillator has not been started prior to calling this function. Typically, this result code is only caused by improper application software organization, and the application program should be closely examined to determine why where StartXtalOsc() is not being called prior to VCO_Tuning().

Re-calibration of the tuning curve is necessary when VBAT changes more than 800mV or TAmbient changes more than 70°C. Due to the fast execution and low overhead of this ROM function it is recommended to call VCO_Tuning() when more than a few minutes have elapsed since this function was last called.

3.1 Code Example

```
void VCO_Tuning_Function (void)
{
	// Return value of start VCO Tuning is stored in StatusByte
	unsigned char StatusByte;
	// VCO Tuning function call
	StatusByte = VCO_Tuning();

	if (StatusByte == -1)
	{
		// VCO Tuning was not sucessful, default tuning curve selected
		// ...
	}
	if (StatusByte == -2)
	{
		// XTAL oscillator is not started
		// ...
	}
}
```

Figure 4 VCO Tuning

4 RF Encoder & Baudrate Generation

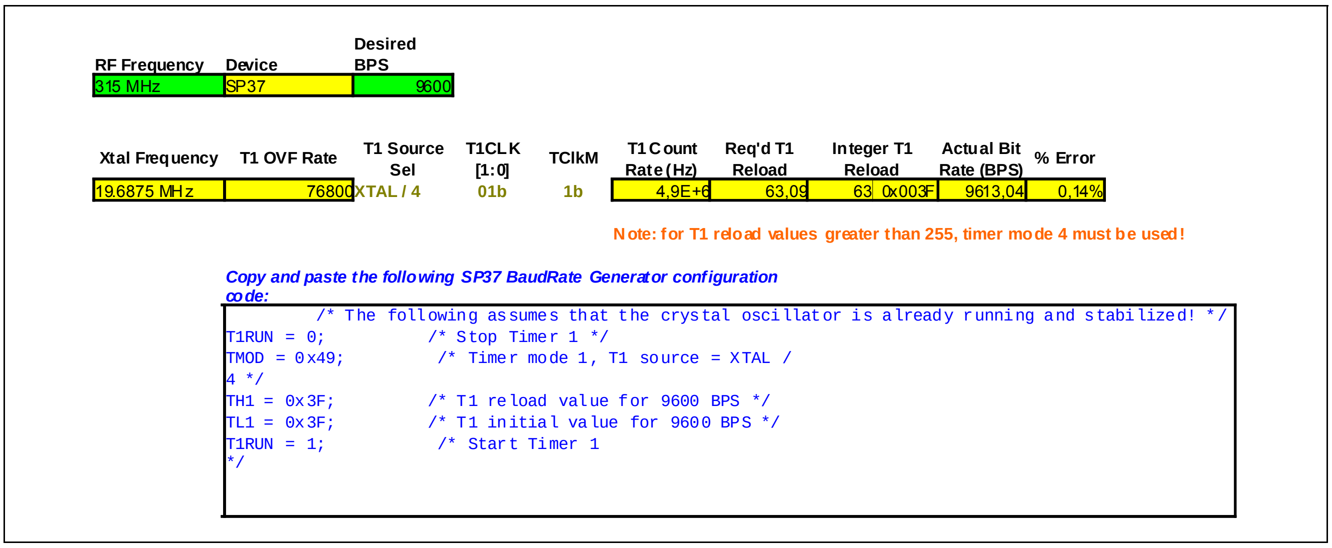
The RF Encoder is a very flexible and convenient peripheral block within the SP37. It frees the CPU from the necessity to directly encode ("bit bang") the baseband information used for RF carrier modulation. In principle, proper usage of the RF Encoder requires only that the RF Encoder itself is configured according to the desired baseband modulation, and that Timer1 is configured to generate the required baudrate strobe frequency. The encoder mode, transmit buffer length, and RF carrier idle state are all controlled by SFR RFENC. In some cases, the baseband encoding may need to be changed during the transmission of an RF telegram. For this reason, the RFENC settings for encoder mode and transmit buffer length take effect upon transfer of the data from SFR RFD to the internal shift register, not at the time of writing to SFR RFENC.

Timer 1 is used as baudrate generator for the RF Encoder. The recommended timer clock source is the crystal oscillator, which is selected by the SFR bit TMOD.TCLKM. The timer mode is determined by SFR TMOD and depends on the required baudrate. The required timer reload value can be calculated with the following formula:

$$timervalue = \left(\frac{f_{timerclock\ source}\,[Hz]}{8 \cdot Baudrate\left[\frac{1}{s}\right]} \right) - 1$$

Figure 5 Equation for Baudrate Generator

If a timer reload value of more than 255 is required, the Timer Mode 4 (16-bit counter w/reload) must be used and timer reload value is stored in SFRs TH1 and TL1. If a timer reload value of 255 or less is required, Timer Modes 1, 2 and 3 are possible in addition to Timer Mode 4. If these 8-bit counter w/reload modes are used, the reload value must be stored in SFR TH1. Additionally, a baudrate calculator [3] is available that calculates the proper timer values and generates example C-code.



RF Frequency	Device	Desired BPS
315 MHz	SP37	9600

Xtal Frequency	T1 OVF Rate	T1 Source Sel	T1CLK [1:0]	TClkM	T1 Count Rate (Hz)	Req'd T1 Reload	Integer T1 Reload		Actual Bit Rate (BPS)	% Error
19.6875 MHz	76800	XTAL / 4	01b	1b	4,9E+6	63,09	63	0x003F	9613,04	0,14%

Note: for T1 reload values greater than 255, timer mode 4 must be used!

Copy and paste the following SP37 BaudRate Generator configuration code:

```
         /* The following assumes that the crystal oscillator is already running and stabilized! */
T1RUN = 0;          /* Stop Timer 1 */
TMOD = 0x49;         /* Timer mode 1, T1 source = XTAL /
4 */
TH1 = 0x3F;         /* T1 reload value for 9600 BPS */
TL1 = 0x3F;         /* T1 initial value for 9600 BPS */
T1RUN = 1;           /* Start Timer 1
*/
```

Figure 6 RF baudrate calculator

4.1 RF Encoder Baseband Observation

For verification purposes the RF Encoder output can be configured to appear on pin PP2 during RF transmission. SFR bit CFG1.RFTXPEN controls this feature. When this bit is set, RF output is disabled and, instead, the RF Encoder baseband signal is output onto PP2. To observe the encoder output on the pin PP2 and the RF signal on a spectrum analyzer simultaneously, the RF power amplifier must be enabled manually via

SFR bit RFC.ENPA prior to the start of transmission. It is recommended to have the RFTXPEN bit cleared for normal operation; this feature is intended for RF Encoder testing.

```
void RF_Baudrate_5k_434 (void)
{
       // timer Settings for 5Kbaud, 434MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Timer 0 16bit w/o reload; Timebase 12MHz/DIVIC/8
                      // Timer 1 8bit reload; timebase XTAL/4
       TL1 = 0x70;    // Configure Timer 1 Counter for 5KBaud with reload
       TH1 = 0x70;
       T1RUN = 1;     // Start Baudrate Timer
}
void RF_Baudrate_9k6_434 (void)
{
       // Timer Settings for 9.6Kbaud, 434MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Reload Timer 1; timebase XTAL/4
       TL1 = 0x3A;    // Configure Timer 1 Counter for 9.6KBaud with reload
       TH1 = 0x3A;
       T1RUN = 1;     // Start Baudrate Timer
}
void RF_Baudrate_10k_434 (void)
{
       // Timer Settings for 10Kbaud, 434MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x59;   // Use Timer Mode 1
                      // Timer 0 16bit w/o reload; Timebase 12MHz/DIVIC/8
                      // Timer 1 8bit reload; timebase XTAL/4
       TL1 = 0x38;    // Configure Timer 1 Counter for 10KBaud with reload
       TH1 = 0x38;
       T1RUN = 1;     // Start Baudrate Timer
}
void RF_Baudrate_19k2_434 (void)
{
       // Timer Settings for 19.2Kbaud, 434MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Reload Timer 1; timebase XTAL/4
       TL1 = 0x1C;    // Configure Timer 1 Counter for 19.2KBaud with reload
       TH1 = 0x1C;
       T1RUN = 1;     // Start Baudrate Timer
}
void RF_Baudrate_5k_315 (void)
{
       // Timer Settings for 5Kbaud, 315MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Timer 0 16bit w/o reload; Timebase 12MHz/DIVIC/8
                      // Timer 1 8bit reload; timebase XTAL/4
       TL1 = 0x7A;    // Configure Timer 1 Counter for 5KBaud with reload
       TH1 = 0x7A;
       T1RUN = 1;     // Start Baudrate Timer
}
```

```
void RF_Baudrate_9k6_315 (void)
{
       // Timer Settings for 9.6Kbaud, 315MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Reload Timer 1; timebase XTAL/4
       TL1 = 0x3F;    // Configure Timer 1 Counter for 9.6KBaud with reload
       TH1 = 0x3F;
       T1RUN = 1;     // Start Baudrate Timer
}
void RF_Baudrate_10k_315 (void)
{
       // Timer Settings for 10Kbaud, 315MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Timer 0 16bit w/o reload; Timebase 12MHz/DIVIC/8
                      // Timer 1 8bit reload; timebase XTAL/4
       TL1 = 0x3D;    // Configure Timer 1 Counter for 10KBaud with reload
       TH1 = 0x3D;
       T1RUN = 1;     // Start Baudrate Timer
}
void RF_Baudrate_19k2_315 (void)
{
       // Timer Settings for 19.2Kbaud, 315MHz
       T1RUN = 0;     // Stop Baud rate Timer
       TMOD = 0x49;   // Use Timer Mode 1
                      // Reload Timer 1; timebase XTAL/4
       TL1 = 0x1F;    // Configure Timer 1 Counter for 19.2KBaud with reload
       TH1 = 0x1F;
       T1RUN = 1;     // Start Baudrate Timer
}
```

Figure 7 Configure RF Baudrate Timer

```
void RF_Encoder_Manchester_8Bit (void)
{
       RFENC = 0xE0;  // Manchester Encoding, RF data length: 8 bit
}
void RF_Encoder_Manchester_4Bit (void)
{
       RFENC = 0x60;  // Manchester Encoding, RF data length: 4 bit
}
void PP2_Encoder_Output (void)
{
       RFTXPEN = 1;   // Echos RF transmission baseband data on port PP2/ TxData
       RFC |= 0x01;   // Enable PA
}
```

Figure 8 Configure RF Encoder

5 RF Telegram Data Transmission

The typical application approach is to build the RF telegram contents in RAM prior to transmission, then simply move each RF telegram data byte into SFR RFD, in sequence, until the entire telegram has been sent. The RF Encoder allows baseband symbols of arbitrary nature (e.g. Manchester “code violations”) and bit lengths other than eight to be sent if required. The RFENC settings for encoder mode and transmit buffer length take effect upon transfer of the data from SFR RFD to the internal shift register, not at the time of writing to SFR RFENC. After the final data has been loaded into RFD, the application must ensure that the actual RF transmission is finished before taking further action. The bit RFSE gives the indication if the RF data transmission is complete. When this is “0”, the data transmission is still in progress. When it is “1”, the data transmission is complete.

5.1 Minimize Power Consumption

Using the RF Encoder peripheral allows that the CPU to operate at a reduced clock rate thereby reducing the peak current consumption during RF transmission. By setting SFR DIVIC to 0x03 the CPU clock rate is divided by a factor of 64. This reduced CPU clock rate also reduces the possibility of clock noise artifacts in the transmitted RF signal. Furthermore, the encoder generates a resume event after sending each byte so that the application can enter IDLE state after each byte is loaded into SFR RFD. It is recommended to use both reduced clock rate and IDLE mode for best performance during RF transmission.

For verification purposes the RF Encoder output can be configured to appear on pin PP2 during RF transmission. See Section 4.1 for more details.

5.2 Code Example

```
// Define Variables
#define SENSOR_ID1 0x0C
#define SENSOR_ID2 0x02
#define SENSOR_ID3 0x5A
#define SENSOR_ID4 0x58
#define BATT_STATUS 0xF0
#define PRESSURE 0x5D
#define TEMPERATURE 0x19
#define ACCELERATION 0x00
//Sends one Databyte via RF
char putchar_RF (char Result)
{
       RFD = Result;
       do
       {
              IDLE = 1;              // Wait for RF Buffer Empty RESUME Event in IDLE Mode
       } while ((REF & 0x04) == 0);  // While RF Transmit Buffer is NOT empty, go to IDLE Mode
       return Result;
}
void Tx_RF_Telegram_Low_consumption (void)
{
       DIVIC = 0x03;          // Turn on clock divider
       // Preamble with code violation
       RFENC = 0xC1;          // Number of bits to be transmitted: 7MSBs, inverted Manchester encoding
       putchar_RF(0xFE);      // Send 7 bits "1"
       // Code violation
       RFENC = 0x65;          // Number of bits to be transmitted: 4 MSBs, Chip Mode
       putchar_RF(0xC0);      // Send two chips "1" and two chips "0"
       // Number of bits to be transmitted: all bits, inverted Manchester encoding
       RFENC = 0xE1;
       // Transmit payload
       putchar_RF(SENSOR_ID1);      // Send first MsByte of sensor's ID
       putchar_RF(SENSOR_ID2);      // Send second MsByte of sensor's ID
       putchar_RF(SENSOR_ID3);      // Send third MsByte of sensor's ID
       putchar_RF(SENSOR_ID4);      // Send fourth MsByte of sensor's ID
       putchar_RF(BATT_STATUS);     // Send information about battery voltage
       putchar_RF(PRESSURE);        // Send measured pressure
       putchar_RF(TEMPERATURE);     // Send measured temperature
       putchar_RF(ACCELERATION);    // Send measured acceleration
       DIVIC = 0x00;
}
```

Figure 9 Transmit RF Telegram with minimized power consumption

```
void putchar_RFMain (void)
{
       unsigned char StatusByte;
       signed int Voltage;

       // Configures the RFTX Register according to the SP37_DevLib.h
       RF_Config();
       // Start the XTAL with a delay of about 1260 µs
       Start_Xtal();     // Additional details in chapter 2
       // VCO Tuning
       VCO_Tuning_Function();     // Additional details in chapter 3
       // Configure 8Bit Manchester Coding
       RFENC = 0xE0;
       // Timer settings for 9.6 kBaud - 434MHz
       RF_Baudrate_9k6_434();     // Additional details in chapter 4
       // FSK +/- 35kHz deviation
       XTAL0 = 0x28;
       XTAL1 = 0x11;
        // Output on PP2
       // CFG1 = 0x10;          // to be uncommented for observation of the data on PP2. If data are
       observed on PP2, the PA is disabled.
       // Send the telegram
       TX_RX_Telegram_Low_consumption();
       // Wait for RF Transmission to end
       while(!(REF & 8));
        // Disable PA & PLL
       RFC &= 0x00;
       // Configure Interval Timer
       // 500ms Interval Precounter
       IntervalTimerCalibration(2);
       // Configure Postcounter to 1
       ITPR = 0x01;
       // Stop Xtal
      StopXtalOsc();
       // Enter in Power Down
       Powerdown();
}
```

Figure 10 Transmit RF Telegram with minimized power consumption – Main function

6 Continuous Wave Transmission

During development and production, and for EMC Compliance testing purposes, a continuous wave (CW) transmission can be used. This can be achieved either by controlling the RF PA manually or by using the RF baseband encoder in “chip-mode”.

6.1 Manual control of the PA

Before taking manual control of the PA, the application should secure that any ongoing RF transmission is finished. The bit RFSE gives the indication if the RF data transmission is complete. When this bit is “0”, the data transmission is still in progress. When it is “1”, the data transmission is complete.

The SFR bit RFC.ENPA allows manual control of the RF PA. When the RF Transmit Shift Register is empty, the RF Carrier is controlled by the SFR bit RFENC.TXDD (see [1]). Depending on the selected modulation (ASK/FSK) the PA is turned on or off, and the carrier frequency is adjusted according to XTAL0, XTAL1, and FSKSWITCH SFR.

6.1.1 Code Example

```
void Carrier_ManualPA(void)
{
       int i=0;

       StartXtalOsc(33);       // Start XTAL with a delay of about 1260µs
       VCO_Tuning();           // VCO-Tuning
       DIVIC = 0x03;
       while(!(RFS |= 0x02));  // Wait for RF Data Transmission complete, RFSE = 1
       RFC |= 0x01;            // Enable PA
       DIVIC = 0x00;
       for(i=0; i<4000;  i++ ) // Generation of the continuous wave signal
       {
          T0RUN = 0;
          TMOD = 0x19;
          TH0 = 0xF0;
          TL0 = 0x89;
          T0RUN = 1;
          WDRES = 1;
          IDLE = 1;            // wait for RF Buffer Empty RESUME Event in IDLE Mode
       }
       RFC &= 0x00;            // Disable PA & PLL
      StopXtalOsc();           // turn off crystal oscillator
}
```

Figure 11 Transmit Continuous Wave signal by manual control of PA

6.2 Chip Mode of RF baseband encoder

In addition to Manchester/ and BiPhase encoding, it is also possible to send data with a user-defined encoding scheme. This is particularly helpful if the desired RF telegram format includes a “code violation” to separate preamble from payload. This is supported by chipmode (SFR bits RFENC.2-0[RFMODE2-0] = 101b). In chipmode the encoder sends each bit in SFR RFD without any encoding.

The frequency and the baud rate need also to be defined. By defining a low data rate, the duration of the carrier ON will be longer.

```
void RF_Encoder_chip_Mode (void)
{
       unsigned int i=0;

       RFTX = RF_Config();   // Configures the RFTX Register according to SP37_DevLib.h
       StartXtalOsc(33);     // Start XTAL with a delay of about 1260µs
       VCO_Tuning();         // VCO-Tuning
       DIVIC = 0x03;
       // Transmit all bits, chip mode
       RFENC = 0xE5;
       // Configuration of the baud rate, 16bps, 434MHz
       T1RUN = 0;
       TMOD = 0x4C;           // Mode 4
       TH1 = 0x89;
       TL1 = 0xF0;
       TH0 = 0x89;
       TL0 = 0xF0;
       T1RUN = 1;
       // Carrier transmission during 30s
       for(i=0; i<120; i++)
       {
              WDRES = 1;
              RFD = 0xFF;
              IDLE = 1;      // wait for RF Buffer Empty RESUME Event in IDLE Mode
       }
       DIVIC = 0x00;
       while (!(REF & 8));   // Wait for RF Transmission to end
       RFC &= 0x00;          // Disable PA & PLL
       StopXtalOsc();        // turn off crystal oscillator
}
```

Figure 12 Chip Mode of RF baseband Encoder

7 Stop Crystal Oscillator

After RF transmissions are concluded, the RF PLL and Crystal Oscillator should be stopped to conserve current. The crystal oscillator cannot be stopped if there are still peripherals using it, such as PLL and Timer Unit. Therefore, prior to stopping the crystal oscillator the RF PLL must be disabled by clearing the SFR bit RFC.ENPLL, and the Timer Unit usage of the crystal oscillator must be cancelled by clearing SFR bit TMOD.TCLKM. After these actions have been taken, the application software may call StopXtalOsc() to stop the crystal oscillator and check the result code to be sure that the crystal oscillator is stopped.

7.1 Code Example

```
// Define variables
#define bitmask_TMOD_CLK 0x08
#define bitmask_RFS_RFSE 0x02

// Start Xtal Oscillator Function
void Stop_XtalOscillator (void)
{
   // Return value of stop XTAL Oscillator stored in StatusByte
   unsigned char StatusByte;

   while(!(RFS & bitmask_RFS_RFSE));        // Wait for RF Transmission to end
   RFS &= ~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 13 Stop Crystal Oscillator

8 Measure Battery Voltage during RF Transmission

The motivation to measure battery voltage during the RF transmission is that the load of the battery is typically at maximum during RF transmission, so this is a good opportunity to determine the state of health of the battery. To measure the battery voltage during RF transmission three functions have been provided, which are to be called in sequential order: Start_Supply_Voltage(), Trig_Supply_Voltage() and finally Get_Supply_Voltage(). Before RF transmission begins the application software should call Start_Supply_Voltage() . It enables and configures the ADC, allows settling of the analogue ADC part, and places the ADC into idle state. Trig_Supply_Voltage() is called after Start_Supply_Voltage(), and triggers an ADC battery voltage measurement. This function causes the ADC to perform a measurement and then return to idle state, preserving the result. To prevent disruption of RF data transmission, this function should be called by the application immediately after an RF telegram byte is shifted into SFR RFD. Get_Supply_Voltage() reads the measured value obtained during Trig_Supply_Voltage(), turns off the ADC, and performs battery voltage compensation. This function should be called by the application after the RF transmission is finished.

8.1 Code Example

```
void Batt_Voltage_Measurement (void)
{
   // 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);

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

Figure 14 Battery Voltage measurement during RF Transmission

9 PLL Monitoring

In order to avoid unwanted out-of-band emissions in case the PLL unlocks, the SP37 includes a PLL Monitoring feature. This feature can be enabled via SFR bit RFC.7[ENPLLMON]. If PLL Monitoring is enabled and the PLL becomes unlocked, the RF Power Amplifier is automatically disabled and SFR bit RFS.7[PADIS] bit is set. The application program can verify that an RF transmission was successful by checking PADIS at the completion of the RF Telegram - if PADIS is clear, the PLL remained locked throughout the transmission. If the PLL Monitoring feature is enabled, the application software must ensure that the PADIS bit is clear prior to attempting transmission of each RF telegram.

Note: The Manchester/BiPhase Encoder state machine continues executing transmission even though PA is off.

9.1 Code Example

```
// Define Variables
#define SENSOR_ID1 0x0C
#define SENSOR_ID2 0x02
#define SENSOR_ID3 0x5A
#define SENSOR_ID4 0x58
#define BATT_STATUS 0xF0
#define PRESSURE 0x5D
#define TEMPERATURE 0x19
#define ACCELERATION 0x00

#define USE_PLL_MONITORING
#define bitmask_RFS_RFSE 0x02

/****************************************************************************************************
Sends one Databyte via RF
****************************************************************************************************/
char putchar_RF_PLL (char Result)
{
       RFD = Result;
       IDLE = 1;              // Wait for RF Buffer Empty RESUME Event in IDLE Mode
       #ifdef USE_PLL_MONITORING
              if ((RFS & 0x80) == 0x80)     // Check PLL Monitoring Status (PADIS) to see if PLL
                                               unlock occurred
                     Result = -1;
              else
                     Result = 0;
       #else
              Result = 0;           // no PLL Monitoring is performed, status is always '0'
       #endif
       return Result;
}
```

Figure 15 Putchar_RF function with PLL Monitoring

```
void PLL_Monitoring(void)
{
       // Return value of putchar_RF with PLL Monitoring
       unsigned char RF_PLL_Status;

       DIVIC = 0x03;          // Turn on clock divider
       RFC |= 0x80;           // Enable PLL  Monitoring feature
       // Preamble with code violation
       // Number of bits to be transmitted: 7MSBs, inverted Manchester encoding
       RF_PLL_Status = putchar_RF_PLL(0xFE);       // Send 7 bits "1"
       // Code violation
       // Number of bits to be transmitted: 4MSBs, chip Mode
       RFENC = 0x65;
       if (RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(0xC0); // Send two chips "1" and two chips "0"
       // Number of bits to be transmitted: 7 MSBs, inverted Manchester encoding
       RFENC= 0xC1;
       if (RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(0xFE);               // Send 7 bits "1"
       // End of preamble with code violation

       // Number of bits to be transmitted: all bits, inverted Manchester encoding
       RFENC = 0xE1;
       // Transmit payload
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(SENSOR_ID1);        // Send first MSByte of sensor's ID
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(SENSOR_ID2);        // Send second MSByte of sensor's ID
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(SENSOR_ID3);        // Send third MSByte of sensor's ID
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(SENSOR_ID4);        // Send fourth MSByte of sensor's ID
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(BATT_STATUS);       // Send information about battery
                                                                    voltage
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(PRESSURE);          // Send measured pressure
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(TEMPERATURE);       // Send measured temperature
       if(RF_PLL_Status == 0)
              RF_PLL_Status = putchar_RF_PLL(ACCELERATION);      // Send measured acceleration
       if(RF_PLL_Status == 0)
              while(!(RFS& bitmask_RFS_RFSE));                   // Wait for RF Transmission to end
       else
       {
              // PLL is out of lock
              // ...
       }
}
```

Figure 16 Transmit RF Telegram with PLL Monitoring

10 Send RF_Telegram() ROM Library Function

10.1 Description

Chapter 5 presents one solution for using the RF Encoder to send data via RF transmission. The SP37 ROM Library provides a single function that performs all of this work. By using the ROM Library function Send_RF_Telegram() the application program needs only to perform the Initialization steps; Send_RF_Telegram() addresses the Data Transmission. At the end of the Data Transmission the Crystal Oscillator and the RF Transmitter circuitry are disabled. A flexible Pattern Descriptor Table is used to define the format and the content of the RF Telegram. This function is described in more details in the SP37 ROM Library guide.

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

- The RF Transmitter SFR RFTX must be configured to select the appropriate Frequency Band, Output Power, etc. Further details about SFR RFTX are given in the Chapter 1 and Datasheet.
- The Crystal Oscillator must be enabled via the StartXtalOsc() ROM Library function. See Chapter 2.
- The RF PLL must be enabled and initialized via the VCO_Tuning() ROM Library function. See Chapter 3.
- An RF Telegram Pattern descriptor must be placed into RAM.

After successful Initialization, the application code may call the function Send_RF_Telegram() function. Only two parameters are required to setup the function Send_RF_telegram(): the desired baud rate and the starting RAM address of the Pattern Descriptor Table. If an error is encountered while processing the Pattern Descriptor Table, the function Send_RF_Telegram() will terminate with a return code of -1.

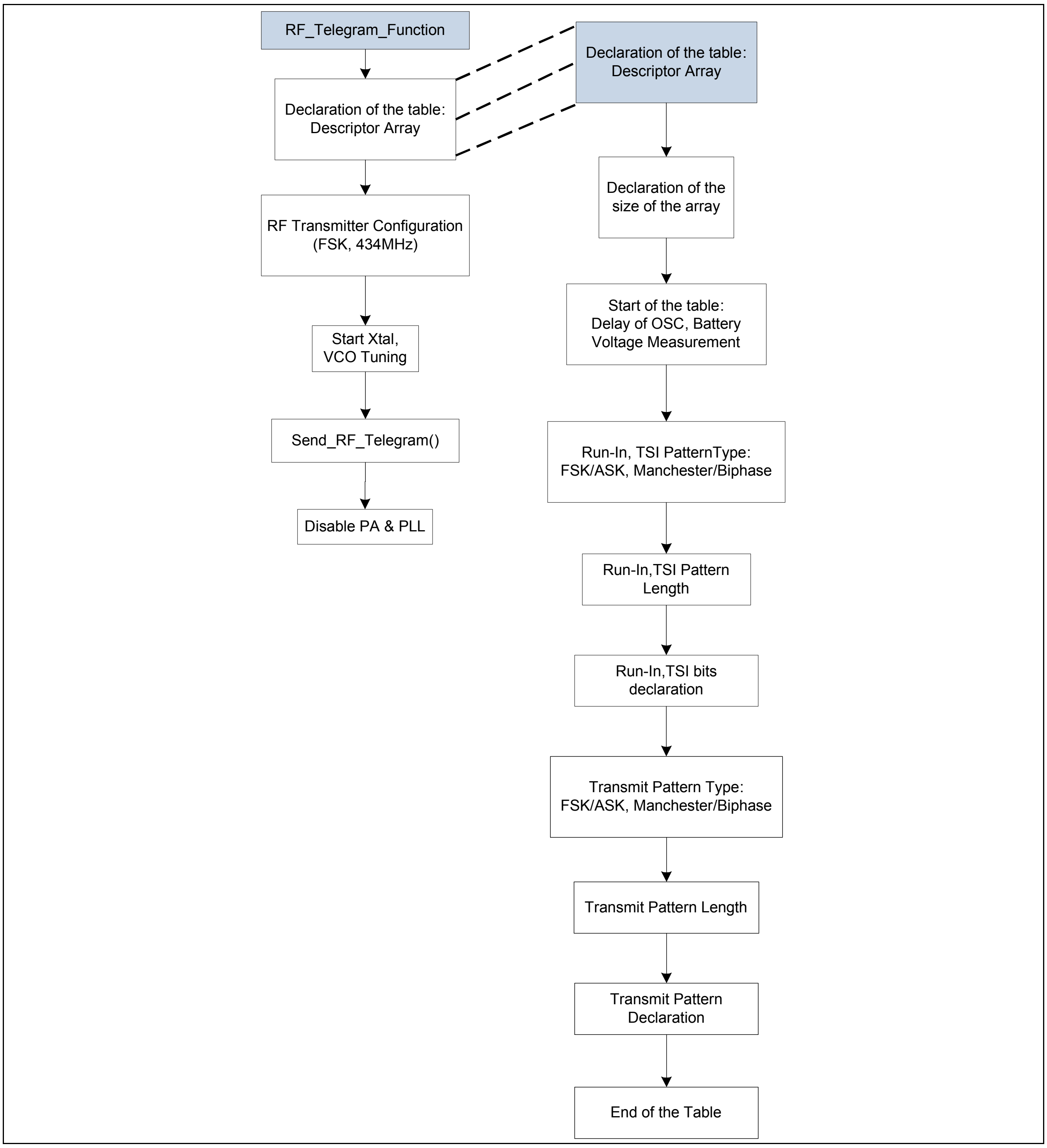

RF_Telegram_Function
Declaration of the table: Descriptor Array
RF Transmitter Configuration (FSK, 434MHz)
Start Xtal, VCO Tuning
Send_RF_Telegram()
Disable PA & PLL
Declaration of the table: Descriptor Array
Declaration of the size of the array
Start of the table: Delay of OSC, Battery Voltage Measurement
Run-In, TSI PatternType: FSK/ASK, Manchester/Biphase
Run-In,TSI Pattern Length
Run-In,TSI bits declaration
Transmit Pattern Type: FSK/ASK, Manchester/Biphase
Transmit Pattern Length
Transmit Pattern Declaration
End of the Table

Figure 17 Send_RF_Telegram Function Flow Chart.

10.2 Code Example

```
void RF_Telegram (void)
{
        // Return value of Send_RF_Telegram stored in sendRF_StatusByte
        signed char sendRF_StatusByte;
        // Array for pattern descriptor table
        unsigned char idata descriptorPtr[10];

        // Start of the 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 = 9600bit/s
        sendRF_StatusByte = Send_RF_Telegram(9600,descriptorPtr);

        // Get supply Voltage function call (if required
}
```

Figure 18 Send_RF_Telegram Function.

10.3 Battery Voltage Measurement with Send RF Telegram function

As described in Chapter 8, RF transmission provides a good opportunity to measure the battery voltage under load. The same three functions Start_Supply_Voltage(), Trig_Supply_Voltage and Get_Supply_Voltage() may be used. The Start_Supply_Voltage() function must be called before Send_RF_Telegram() is called. The Trig_Supply_Voltage() function is called from within Send_RF_telegram function if the MSB of the first byte of the PDT is set. In order to read the measured supply voltage after RF Telegram has completed, the Get_Supply_Voltage() function must be called.

```
//Frame definition
#define SENSOR_ID1 0x0C
#define SENSOR_ID2 0x02
#define SENSOR_ID3 0x5A
#define SENSOR_ID4 0x58
#define BATT_STATUS 0xF0
#define PRESSURE 0x5D
#define TEMPERATURE 0x19
#define ACCELERATION 0x00
#define PAYLOAD 0x44

// Telegram sends with the ROM library function Send_RF_Telegram and Battery voltage measurement
// performed.
// FSK modulation
void RF_Telegram_Function(void)
{
       unsigned char StatusByte;
       signed int Voltage;
       unsigned char data descriptorArray[19];

       // RF Transmitter Configuration (SFR RFTXX and SFR XTAL 0/ 1)
       RF_Configuration_2ICAP_FSK_434_8dbm();
       // Start XTAL with a delay of about 1260µs
       StartXtalOsc(33);
       // VCO-Tuning
       VCO_Tuning();
       // Transmit Type Pattern Descriptor for the Run-In and TSI
       descriptorArray[0] = 0x80;          // Start of Table, Voltage Measurement ON
       descriptorArray[1] = 0x00;          // Noninverted Manchester Code, FSK
       descriptorArray[2] = 0x14;          // Length is 20Bits
       descriptorArray[3] = 0xFF;          // Send 4Bits '1' Run-In 4Bits '1' for TSI
       descriptorArray[4] = 0xFF;          // Send 8Bits '1' for TSI
       descriptorArray[5] = 0xE0;          // Send 3Bits '1' and 1Bit '0' for TSI
       // Transmit Type Pattern Descriptor for the Payload
       descriptorArray[6] = 0x00;          // Noninverted Manchester Code, FSK
       descriptorArray[7] = 0x70;          // Length is 8Bytes = 64Bits
       // Transmission of the payload
       descriptorArray[8] = SENSOR_ID1;    // Send first MSByte of sensor's ID
       descriptorArray[9] = SENSOR_ID2;    // Send second MSByte of sensor's ID
       descriptorArray[10] = SENSOR_ID3;   // Send third MSByte of sensor's ID
       descriptorArray[11] = SENSOR_ID4;   // Send fourth MSByte of sensor's ID
       descriptorArray[12] = BATT_STATUS;  // Send information about battery voltage
       descriptorArray[13] = PRESSURE;     // Send measured pressure
       descriptorArray[14] = TEMPERATURE;  // Send measured temperature
       descriptorArray[15] = ACCELERATION; // Send measured acceleration
       descriptorArray[16] = PAYLOAD;      // Send payload
```

```
        descriptorArray[17] = PAYLOAD;       // Send payload
        // End Of Table
        descriptorArray[18] = 0xF1;
        // Prepare the ADC and the Supply Voltage Sensor for Supply Voltage Measurement
        Start_Supply_Voltage();
        // Transmission of the datagram
        Send_RF_Telegram(9600, descriptorArray);
        // Disable PA & PLL
        RFC &= 0x00;
        StatusByte = Get_Supply_Voltage(&Voltage);

        // Configure Interval Timer
        IntervalTimerCalibration(2);         // Configure for 50ms Interval Precounter
        ITPR = 0x01;                         // Configure for Postcounter to 1
        // Stop XTAL Oscillator
        StopXtalOsc();
}
```

Figure 19 Battery Voltage Measurement with the Send_RF_Telegram Function

10.4 PLL Monitoring with Send RF Telegram function

It is possible to avoid unwanted out-of-band emissions in case of PLL unlock while sending data with the function Send_RF_Telegram() as long as the Send_RF_Telegram “Delay” type pattern descriptors are not used. Otherwise, the PLL Monitoring function works exactly as described in chapter 9.

```
//Frame definition
#define SENSOR_ID1 0x0C
#define SENSOR_ID2 0x02
#define SENSOR_ID3 0x5A
#define SENSOR_ID4 0x58
#define BATT_STATUS 0xF0
#define PRESSURE 0x5D
#define TEMPERATURE 0x19
#define ACCELERATION 0x00
#define PAYLOAD 0x44

// Telegram sends with the ROM library function Send_RF_Telegram and PLL Monitoring performed.
// FSK modulation
void RF_TelegramPLL_Function(void)
{
        unsigned char StatusByte;
        signed int Voltage;
        unsigned char data descriptorArray[19];
       unsigned char RF_PLL_Status;

        // RF Transmitter Configuration (SFR RFTXX and SFR XTAL 0/ 1)
        RF_Configuration_2ICAP_FSK_434_8dbm();
       // Enable PLL Monitoring PA Status
       RF_PLL_Status = RFS & 0x80;
        // Start XTAL with a delay of about 1260µs
        StartXtalOsc(33);
       // VCO-Tuning
        VCO_Tuning();
        // Transmit Type Pattern Descriptor for the Run-In and TSI
        descriptorArray[0] = 0x80;    // Start of Table, Voltage Measurement ON
        descriptorArray[1] = 0x00;    // Noninverted Manchester Code, FSK
        descriptorArray[2] = 0x14;    // Length is 20Bits
        descriptorArray[3] = 0xFF;    // Send 4Bits '1' Run-In 4Bits '1' for TSI
        descriptorArray[4] = 0xFF;    // Send 8Bits '1' for TSI
        descriptorArray[5] = 0xE0;    // Send 3Bits '1' and 1Bit '0' for TSI
        // Transmit Type Pattern Descriptor for the Payload
        descriptorArray[6] = 0x00;    // Noninverted Manchester Code, FSK
        descriptorArray[7] = 0x70;    // Length is 8Bytes = 64Bits
        // Transmission of the payload
        descriptorArray[8] = SENSOR_ID1;            // Send first MSByte of sensor's ID
        descriptorArray[9] = SENSOR_ID2;            // Send second MSByte of sensor's ID
        descriptorArray[10] = SENSOR_ID3;           // Send third MSByte of sensor's ID
        descriptorArray[11] = SENSOR_ID4;           // Send fourth MSByte of sensor's ID
        descriptorArray[12] = BATT_STATUS;          // Send information about battery voltage
        descriptorArray[13] = PRESSURE;             // Send measured pressure
        descriptorArray[14] = TEMPERATURE;          // Send measured temperature
        descriptorArray[15] = ACCELERATION;         // Send measured acceleration
        descriptorArray[16] = PAYLOAD;              // Send payload
        descriptorArray[17] = RF_PLL_Status;        // Send RF_PLL_Status
        // End Of Table
        descriptorArray[18] = 0xF1;

        // Transmission of the datagram
                 If (!(RF_PLL_Status == 0x80))       // Check PLL Monitoring Status (PADIS) to see if no
PLL unlock occurred
                         Send_RF_Telegram(9600, descriptorArray);
        // Disable PA & PLL
        RFC &= 0x00;
        // Configure Interval Timer
        IntervalTimerCalibration(2);                // Configure for 500ms Interval Precounter
        ITPR = 0x01;                                // Configure for Postcounter to 1
        StopXtalOsc();
}
```

Figure 20 PLL Monitoring with Send RF Telegram

11 Redundant Telegram Transmission

In a typical TPMS application, redundant copies of the RF telegram containing the pressure and data are sent to ensure that the receiver gets the information. This redundancy is intended to overcome temporary disturbance in the RF link, such as RF interference, as well allow multiple TPMS transmitter to coexist in the same RF channel. There is typically a delay, on the order of 10's to 100's of milliseconds, in between these redundant RF Telegrams. The delay between redundant RF telegrams serves three purposes. First, the delay allows the lithium cell to recover after the deep discharge required by RF transmission. Second, if the delay is non-uniform (or even randomized) it provides some means to avoid RF collisions between all the sensors on the vehicle. Third, some government EMC regulations impose some limits on the duration and period of unlicensed device emissions. One approach to implementing these delay times is to use the Timer Unit to time the delay interval. An improvement over this method is to use the SP37 IDLE mode during the delay time interval, in order to reduce current consumption during this delay even further. To achieve the lowest possible current consumption during this delay time, it is necessary to place the SP37 in Power Down mode in between RF telegram copies. This section details some guidelines on how this may be accomplished.

The Interval Timer (IT) is the only means to bring the SP37 out of Power Down mode at specific time. The Interval Timer is normally used already to cause periodic wakeup, measurement and RF Telegram transmission. A typical periodic wakeup rate is once per minute. Therefore, the Interval Timer is already in use to provide this 60 second periodic wakeup. The 8-bit width of the IT post-scalar SFR ITPR demands that the fattest IT timebase is therefore 4 Hz (250 ms), with a ITPR value of 240 to achieve the required 60 second period. This 4 Hz timebase is too slow to realize the short delays in between redundant copies of the RF Telegram. Therefore, a different timebase, faster than 4 Hz, must be used throughout the transmission of the RF telegram copies.

For instance, when the application is sending three redundant copies of the RF Telegram, the timebase can be configured to 20 Hz (50ms). SFR ITPR is set to 2, so that each telegram frame is separated by an interval of 100ms. After each telegram is transmitted, the application software places the SP37 into Power Down mode to save current.

When all three telegrams copies have been sent, a calculation is performed to determine the remaining time that the device must stay in Power Down before a new sequence starts. Here, the Interval Timer is reconfigured to a timebase of 1 Hz (1s) and SFR ITPR of 60. At the end of the last frame transmitted, the application software calculates the remaining time that the SP37 must stay in Powerdown Mode (60s–(2*100ms)). This reconfiguration of the Interval Timer timebase allows the SP37 to stay in Power Down Mode for a longer time in between transmission events, and a short time in between redundant RF Telegrams. Indeed, if the timebase was only set for 20 Hz (50ms), the device could stay in Powerdown for only 12.8s (256*0.050ms =12.8s).

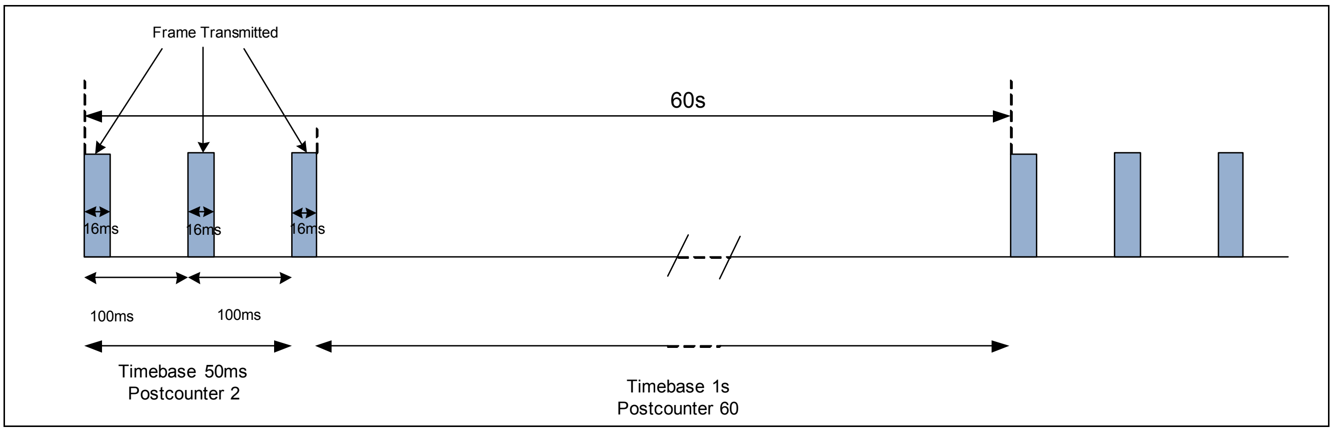


Figure 21 MultiFrame Diagram.

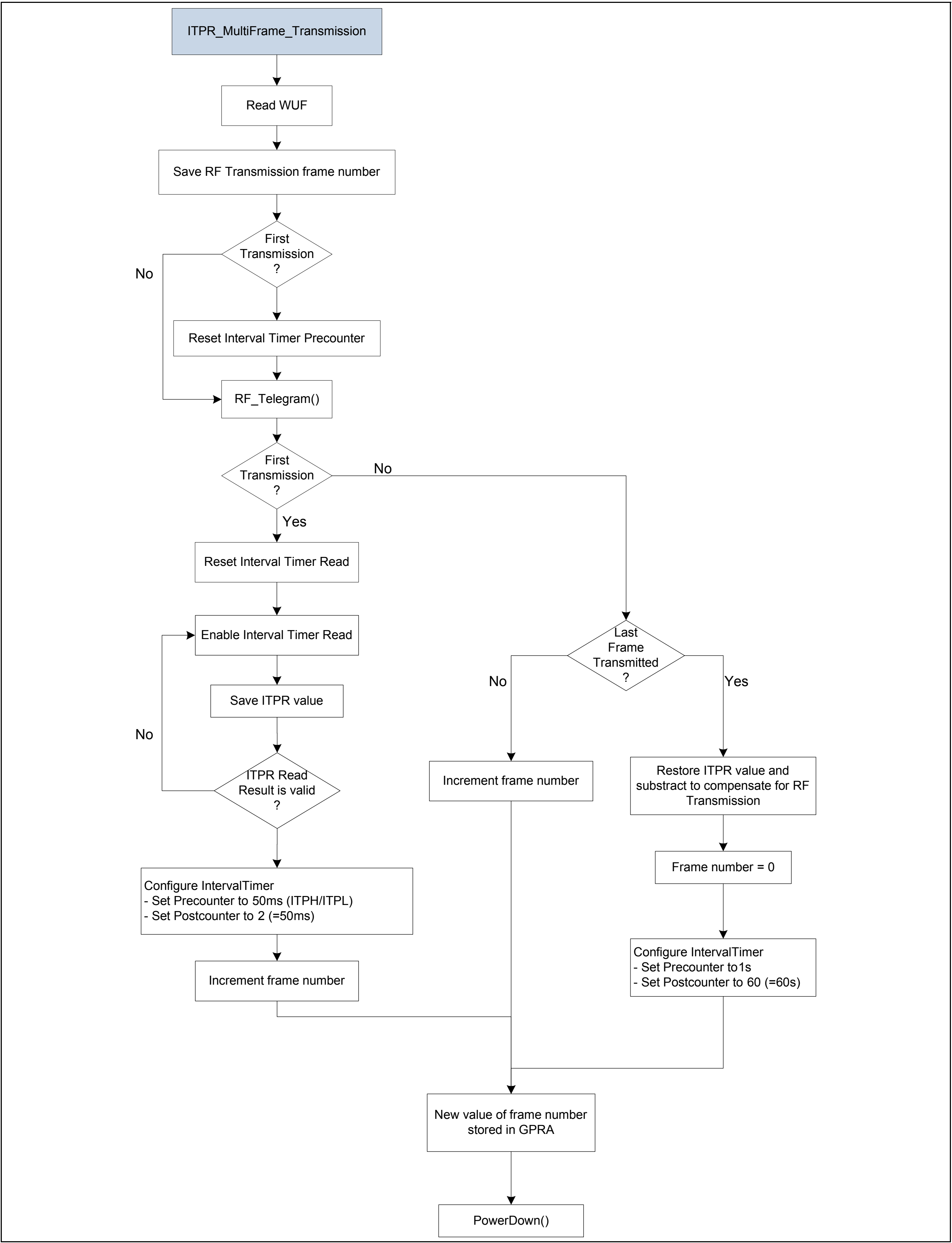

ITPR_MultiFrame_Transmission
Read WUF
Save RF Transmission frame number
First Transmission ?
No
Reset Interval Timer Precounter
RF_Telegram()
First Transmission ?
No
Yes
Reset Interval Timer Read
Enable Interval Timer Read
Save ITPR value
No
ITPR Read Result is valid ?
Configure IntervalTimer
- Set Precounter to 50ms (ITPH/ITPL)
- Set Postcounter to 2 (=50ms)
Increment frame number
Last Frame Transmitted ?
No
Yes
Increment frame number
Restore ITPR value and substract to compensate for RF Transmission
Frame number = 0
Configure IntervalTimer
- Set Precounter to1s
- Set Postcounter to 60 (=60s)
New value of frame number stored in GPRA
PowerDown()

Figure 22 ITPR Multi Frame Transmission Flow Chart

```
void ITPR_MultiFrame_Transmission(void)
{
       // Time for transmission in multiples of 100ms (3 RF CW bursts of 1s + 2 powerdown breaks of
       // 100ms = 3200ms = 32x100ms)
       unsigned char RFTx_time = 0x01;
       // Read Wakeup Flag Register
       char WUF_man = WUF;
       // Save RF transmission frame number
       char frame_number = GPRA;

       if(WUF_man == 0x00)
       {
              if (frame_number == 0)
              {
                     GPRA = 0;
                     frame_number = GPRA;
               }
       }
       // Send RF Telegram
       RF_Telegram_Function();
       CFG1 = 0x10;                        // Enable RF encoder output on PP2
       if(frame_number == 0)
              ITPR = 0xFF;                 // Reset Interval Timer Precounter after last transmission
       if(frame_number == 0)
       {
              ITRD = 0;                    // Reset Interval Timer Read
              do
              {
                      ITRD = 1;            // Enable Interval Timer Read
                      GPRB = ITPR;         // Save ITPR value
              } while( !ITRD);
              ITPH = 0x00;                 // Set Precounter to 50ms
              ITPL = 0x64;
              ITPR = 0x02;                 // Set Postcounter to 2 (100ms)
              frame_number ++;
       //frame_number = 1;
       }
       else if (frame_number == 2)
       {
              GPRB = ITPR;                 // Save ITPR value
              ITPR = GPRB - RFTx_time;     // Restore ITPR Value and subtract to compensate for RF
                                           // Transmission
              frame_number = 0;
              ITPH = 0x07;                 // Set Precounter to 1s
              ITPL = 0xD0;
              ITPR = 0x3C;                 // Set Postcounter to 60
       }
       else frame_number++;                //frame_number = 2;
       GPRA = frame_number;
}
```

Figure 23 Multi Frame Transmission

```
// Main function to use the MultiFrame ITPR function
void main (void)
{
       while(1)
       {
              ITPR_MultiFrame_Transmission();
              Powerdown();
       }
}
```

Figure 24 Main Function of the Multi Frame Transmission

12 References

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

[1] SP37 Datasheet

[2] SP37 ROM Library Function Guide

[3] SP37 RF Baudrate Calculator