Application Note 46

MICRF505/506 Basic: Handling the Control Interface

This app note is intended for users of the MICRF505/MICRF506 radio transceivers. Primary readers are micro controller programmers.

Summary

This application note emphasizes some of the basic micro-controller procedures used when working with Micrel's MICRF505/MICRF506 radio transceiver: How to handle the "control interface" and how to build the "control word". The actual values used in the control word are not described here.

Several simple procedures are included to show trade-offs, like use of program memory vs. execution speed.

Two complete sets of procedures are included.

Introduction

MICRF505/MICRF506 operation is controlled through a "control word" or "programming word".

The control word is read/written through the "control interface", which in the RF chip is a "Slave" (note that the RF chip is a "Master" in the separate "data interface").

Typically, a micro controller (MCU) is used to control the RF transceiver. Typical strategies are:

Not using any micro controller built-in hardware modules. This should be selected if the micro does not have an available hardware module. Writing and reading are done bit-by-bit, also referred to as "bit-banging".

Using a built-in hardware module. This can be used if the micro controller has an available hardware module. Typically, a synchronous serial module (like "SPI").

The Control Interface

The MICRF505/MICRF506 (MICRF50x) operation is controlled through a "control word" or "programming word". The MICRF50x stores the control word in a number of addressable registers. The RF chip has a total of 23 read-able 8-bit registers (address 0-22). Twenty-two of the registers are writeable (address 0-21).

Through the control word, the user can set the MICRF50x in transmit or receive mode, program a specific RF frequency and select a bit rate. Several other options are also available. Refer to the MICRF50x data sheet for a complete list of the fields in the control word.

When writing to the RF chip, the address of the (first) register to write to (7 bits) and a Read/Write (RW) bit, must be entered first. The RW bit is "0" for writing. These two fields make a complete 8-bit byte. After the "address and RW byte" is entered, then the value to write into the addressed register is entered. If more than 1 byte is entered after the "address and RW byte", then the entered values are filled into the next registers. Refer to Figure 1. Note that the address field is 7 bits wide. However, since the register address range is from 0 to 22, the 2 most significant bits are always "00".

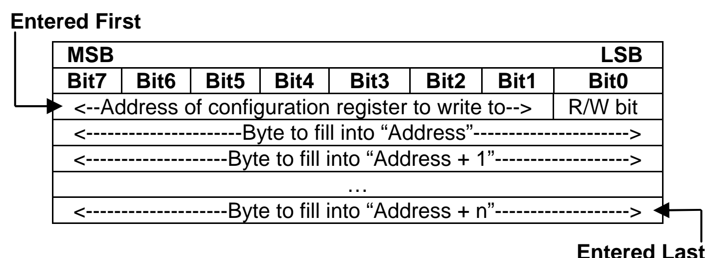

Figure 1. What to Write into the RF Chip

The control interface is a 3-wire interface. Refer to Figure 2 and Table 1. "MCU" means micro controller unit.

Pin Name	Function	Direction Micro RF chip	Comment
CS	Chip Select	→	Active high during programming
SCLK	Serial Clock	→	Output from MCU, i.e. MCU is "Master"
IO	Data In/Out	←→	Output from MCU when writing

Table 1. The 3-Wire Control Interface

Micrel Inc. • 2180 Fortune Drive • San Jose, CA 95131 • USA • tel +1 (408) 944-0800 • fax + 1 (408) 474-1000 • http://www.micrel.com

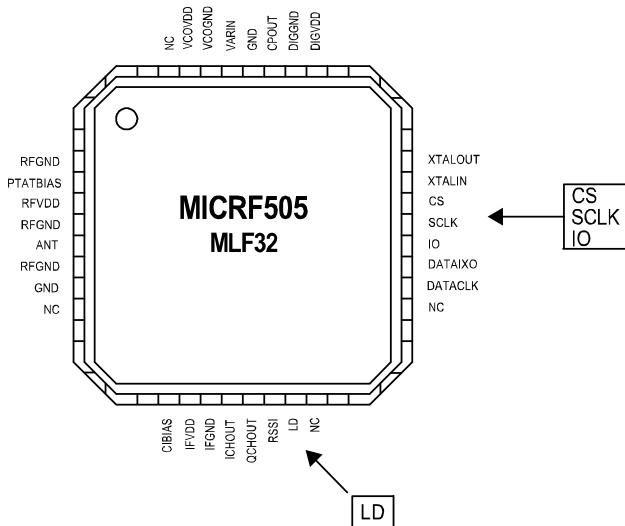

Figure 2. MICRF50x Pin Assignment

The programming is straightforward. This is the programming sequence:

- Bring CS high
- Clock in address and RW bit followed by 1 or more bytes, msb in every byte first (change state of IO on positive edges of SCLK)
- Bring CS low

Observe that address + RW is entered only 1 time in every programming sequence, even if more than 1 register is written to.

Note: The MICRF50x samples the IO line at the negative edges of SCLK. Then, if a SPI-like hw-module is used, make sure that the IO is changed at positive edges by the micro controller. If a bit-by-bit ("bit-banging") approach is used, then the IO line can be changed at any time, *as long as the IO is stable when a negative edge is made on SCLK.*

In Figure 3, 2 bytes are written into the MICRF50x. Observe that the RW-bit is "0".

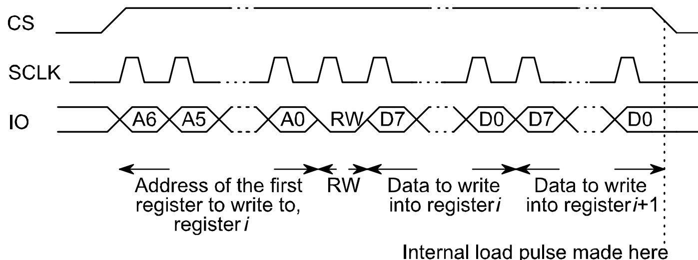

Figure 3. Writing 2 Bytes into the MICRF50x

Timing of signals: Refer to the appropriate MICRF50x data sheet for details. The minimum period of SCLK is in the 50nsec range, and the minimum high/low time of SCLK is in the 20nsec range.

Using Lock Detect Pin

The MICRF50x has an output pin called "Lock Detect" (LD). After a control word is loaded into the MICRF50x, the LD pin will (typically) go low, then high again when the frequency is "locked". In addition, the LD pin will go low when the PA (power amplifier) is starting to turn on, and go high when the PA is on.

The LD pin can be used as follows:

In receive mode:

Load control word, wait for LD, and then start to search for rxed data

(In receive mode, the use of LD can be skipped, simply start to search for rxed data right away.)

In transmit mode:

Load control word; set/keep PA off (to reduce spurious emissions)

Wait for LD

Load control word; set PA on (update register 0x00 only)

Wait for LD

Set DATAIXO as output from micro controller and

Start to transmit

Set DATAIXO back to input to MCU when the transmit is done

When CS is brought low (a load pulse is generated internally), LD will typically go low, but not always (if e.g. the entered control word equals the one already loaded). Therefore, a time delay can be included – if LD does not go low before timeout, then assume LD is OK.

In addition, to avoid a hang-up, a time delay for waiting for LD to go high should be included.

Finally, if LD is detected low, then high: Assume LD is immediately OK.

From the text above, a wait-for-lock procedure can be utilized – refer to Table 5 in the "Basic MCU Procedures" section.

If the number of I/O pins on the MCU is limited, the designer may then select to not route the LD pin to the MCU. In the program a fixed delay can be included instead of waiting for the LD pin to get active. The length of the "fixed delay" depends upon the modulation type and loop filter settings. Refer to the MICRF50x data sheet. It is suggested to experiment with the real application to get the maximum time for the LD pin to go high, then use this as a "fixed delay".

Special Considerations

When programming the MICRF50x, make sure to take the following points into consideration:

- After power-on, the MICRF50x is kept in power-on-reset for approx 3.5 msec. That is, before programming the first time after power-on, include a delay of 3.5 msec. This is important if the rf chip is powered up/down by the MCU; after power-on of the rf chip, include a delay. If the MCU controls a voltage regulator that supplies the rf chip, then add the start-up time of the regulator. If the MCU and rf chip is powered-on at the same time, it is not necessary to include this delay if the MCU is kept in a power-on-reset state for at least 3.5 msec.
- After power-on, the “mode” bits are “00”, i.e., the rf chip starts in power-down mode. The remaining bits are unknown – always enter a complete control word when programming the 1st time. In particular, make sure that the A0, N0, M0 and A1, N1, M1 frequency divider sets are filled with legal values (i.e. N <> 0 and M <> 0). If VCO modulation is selected, then only the “A0, N0, M0” set is used, but it is a good practice to always initiate the “A1, N1, M1” set as well.
- When entering rx or tx mode from power-down, the xtal oscillator starts first, then the other parts of the chip are started. Take this start-up delay into consideration
- For the frequency dividers, observe that “A” > “N” is illegal. That is: Always ensure that the “A” divider is less than (or equal to) the “N” divider.
- Always ensure that all dividers are > 0.

Basic MCU Procedures

The MCU program can be optimized for:

- Minimum use of program memory and/or
- Minimum use of data memory and/or
- Maximum speed and/or
- Available built-in hardware modules and/or
- Development time

Increased program and data memory, more instructions/second and additional built-in hardware modules are continuously added to the MCUs. However, reducing the required resources will always result in a lower-cost solution. Typically, a MCU program will have several tasks, not only controlling the RF interface. Based upon the resources that “the other tasks” need, the resources available for the RF-procedures may be limited. For example; If a MCU has a built-in synchronous serial port, the programmer may decide to use this resource for another task than RF. Then a “bit-banging” procedure must be used for the RF control interface.

For some of the procedures described in this document; both a “minimum use of program memory” and a “maximum speed” optimized version are given.

Typically, optimization can have different purposes. For example, “maximum speed” optimization (reducing the number of instruction cycles) can be done:

- if it is important to run the procedure as fast as possible or,
- if a low-frequency system clock (which results in lower power-consumption and, generally, lower cost) is used.

Compare the data charts in Figure 4, which illustrates the strategy for program-memory/data-memory/speed optimization.

After a power-on reset, the register values are random, except for the “mode” bits, which are “00” – i.e.: The RF chip starts in power-down mode.

Therefore, following a power-on, all registers should be initialized. Then, during operation, changing 1 register is sufficient to change the way the RF chip works.

For details on the register values, refer to the MICRF50x user guides. For assistance, in calculating the values, use e.g., the “RF TestBench” program, available from www.micrel.com.

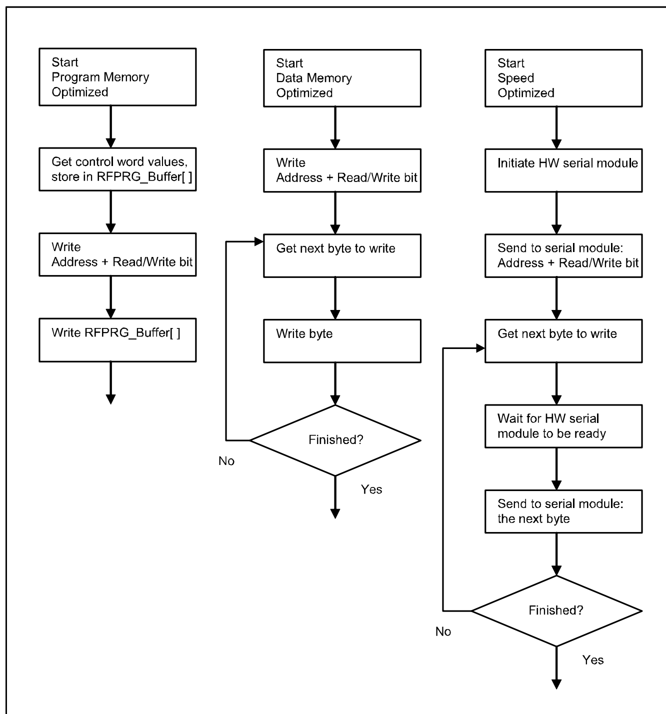

Figure 4. Using the Control Interface: Optimize for Program Memory / Data Memory / Speed

Procedure for Writing a Byte

The operation, "writing a byte" is a central part of "Programming the RF Chip". In this sub-section, only "writing a byte" is discussed.

Writing a Byte Using a Bit-by-Bit Method

This is the most straightforward way of entering a byte, without using any micro controller HW modules.

Program Memory Optimized

In this sub-section, a "program memory optimized" procedure for bit-by-bit writing is discussed.

Refer to Figure 5. The procedure in Figure 5 is "program memory optimized" (a "speed optimized" procedure is shown in Figure 6; it requires three times more program memory, but is twice as fast).

"Bit_Cnt" ("Bit-counter") is a local register used to count bits that are clocked in. Input to the procedure is an 8-bit byte.

Assumptions: SCLK and IO are configured as outputs from the micro controller.

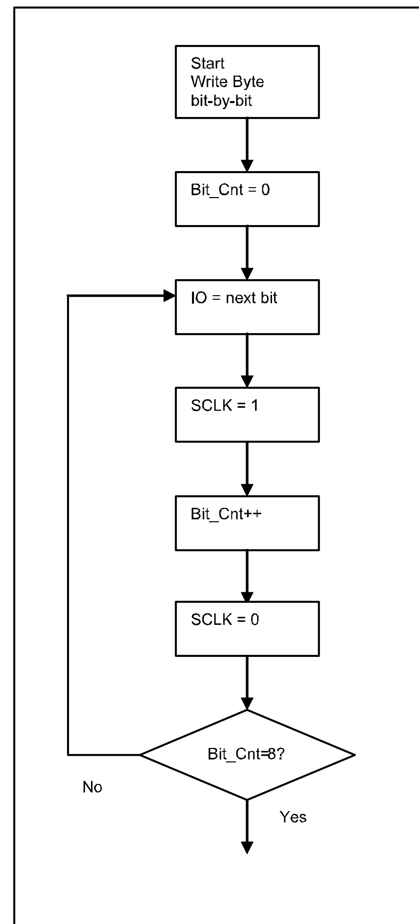

Figure 5. Writing a Byte, Bit-by-Bit (Program Memory Optimize)

The procedure in Figure 5 can be implemented in several ways. In Table 2, for example, using C-code, is given.

Notes to Table 2:

"uint8" is defined as "unsigned char"

bit_and is defined as "&"

The input ("byte_to_write") and the local variable ("temp") are made static to reduce the use of program memory and to increase the execution speed. The cost is that two data bytes are reserved for this procedure. If this cost is too high, then make the variables not static (this will increase the use of program memory and execution time).

"temp" is a local variable with 1 and only 1 bit equal to "1"; in the position of the next bit to give out. MSB in every byte is written first, making the initial value of "temp" equal to binary "10000000", or, in hexadecimal form, 0x80.

"temp" is right-shifted for every bit, and when "temp" is all-zeroes, all bits are written, and the procedure ends.

It may be necessary to include time delays if the minimum high and low times of SCLK are not met (minimum times are defined in the data sheet – they are in the 20nsec range). In addition, it may be necessary to include a time delay after changing I/O, to make sure the line is stable, before making a negative transition on SCLK. This depends upon the load of the I/O pin.

Also note: In many cases, using assembly code can generate a more efficient code than C.

```
//Based on PIC18F micro controller:
//   Program Memory: 15 instructions
//   Data Memory: 2 bytes
//   Execution time:
//      100 instruction cycles
void
RFPRG_Write_Byte( static uint8 byte_to_write)
{
   static uint8 temp;

   temp = 0x80;

   do
   {
      RF_IO_OUT = 0;
      if( byte_to_write bit_and temp)
      {
         RF_IO_OUT = 1;
      }
      RF_SCLK = 1;
      temp  >>= 1;
      RF_SCLK = 0;
   }
   while( temp);
}
```

Table 2. Write a Byte, Bit-by-Bit (Program Memory Optimized)

Speed Optimized

In this sub-section, a “speed optimized” procedure for bit-by-bit writing (not using a MCU built-in hw module) is discussed.

Refer to Figure 6. The procedure in Figure 6 is “speed optimized” (a “program memory optimized” procedure is shown in Figure 5; it requires 1/3 of the program memory, but is 2 times slower).

Assumptions: SCLK and IO are configured as outputs from the micro.

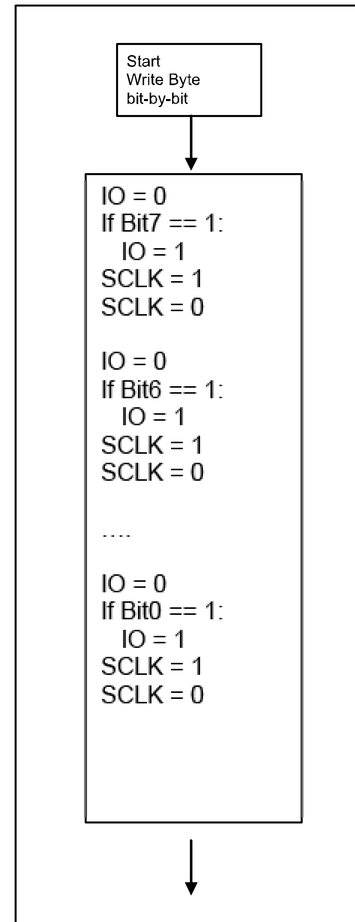

Figure 6. Writing a Byte, Bit-by-Bit (Speed Optimized)

In Table 3, an example using C-code is given.

Notes to Table 3:

“uint8” is def’ed as “unsigned char”

bit_and is def’ed as “&”

The input (“byte_to_write”) and the local variable “temp” are made static to reduce the use of program memory and to increase the execution speed. The cost is that 2 data bytes are reserved for this procedure. If this speed-optimized variant is chosen, then the cost should be acceptable.

The local variable “temp” is made as a union of structures; this is done to be able to do bit-wise operations.

It may be necessary to include time delays if the minimum high and low times of SCLK are not met (minimum times are defined in the data sheet – they are in the 20nsec range). In addition, it may be necessary to include a time delay after changing I/O, to make sure the line is stable, before making a negative transition on clock. This depends upon the load of the I/O pin.

Generally, this procedure should be used only if there is no need for inserting time delays.

Important note: For this speed-optimized procedure to be effective, the C-compiler must be told not to "optimize" the program memory, if so, the C-compiler may simply generate a sub-procedure and call it for every bit – the effect is that the speed is lowered. Refer to your C-compiler manual, and test the result of the compilation to confirm that no memory optimization has occurred.

The complete procedure is not described; it should be obvious how to fill out the remaining part marked as (……………).

```
//Based on PIC18F micro controller:
//   Program Memory: 44 instructions
//   Data Memory: 2 bytes
//   Execution time: 45 instruction cycles
void
RFPRG_Write_Byte( static uint8 byte_to_write)
{
   static union
   {
       struct
       {
           uint8 all;
       };
       struct
       {
          unsigned bit0:1;
          unsigned bit1:1;
          unsigned bit2:1;
          unsigned bit3:1;
          unsigned bit4:1;
          unsigned bit5:1;
          unsigned bit6:1;
          unsigned bit7:1;
       };
   } temp;

   temp.all = byte_to_write;

   RF_IO_OUT = 0;
   if( temp.bit7)
      RF_IO_OUT = 1;
   RF_SCLK = 1;
   RF_SCLK = 0;

(……………)

   RF_IO_OUT = 0;
   if( temp.bit0)
      RF_IO_OUT = 1;
   RF_SCLK = 1;
   RF_SCLK = 0;
}
```

Table 3. Write a Byte, Bit-by-Bit (Speed Optimized)

Writing a Byte Using a Built-in Hardware Module

If a serial interface (SI) module (like "SPI") is available in the micro controller, it can be used for writing bytes into Micrel's MICRF50x. Generally, this method requires less program memory than the bit-by-bit methods, and, additionally, it is faster. When making the firmware, the programmer must compare the available resources to these advantages. If the micro controller only has one serial interface module and a high-speed data interface is to be used (such as 200 kbps), it may be advantageous to use the serial interface module for the data interface, and use a bit-by-bit method for the control interface.

Typically, when a serial interface module is ready for a new byte, an interrupt is made. However, it may be advantageous not to enable the interrupt, but simply test if the module is ready ("polling"). In particular, the last method should be used if there is no priority for the interrupts. In fact, the polling method can be the fastest method; there are no overhead instructions associated with entering/leaving the interrupt service routine (ISR), and there is no "interrupt source testing" (if the micro controller must test the source of the interrupt).

Note that bytes can be entered with a random delay between them, there is no "maximum time" associated with the control interface.

Refer to the text that follows, where the procedure is described. Refer to the data sheet of your micro controller on how to set-up the serial interface module with the parameters described.

Finally, note that the active state of the CS (chip select) pin is high for the MICRF50x. If the built-in serial module includes automatic use of the CS pin, then make sure to get the active state correct (or operate the CS pin manually).

SI initiate:

- Disable SI module
- Clear interrupt-flag
- Make SI a Master
- Make SI clock out data on positive edges
- Make sure clock speed is OK (refer to MICRF data sheet)
- Enable SI module
- (Enable interrupts if used)

SI interrupt (if used):

Clear interrupt-flag

Set SI byte-out = next byte out

SI polling (if used):

Wait for interrupt flag to be set

Clear interrupt-flag

Set SI out-byte = next byte out

Procedure for Getting the Byte to Write

In the previous sub-section, *how to write a byte* was described. However, before writing, the value of the byte to write must be determined. There are twenty-two writeable registers in the control word. In particular, some of the registers can instruct the MICRF to enter receive or transmit (register 0x00), and which frequency to use (registers 0x0A-0x13).

"Which frequency to use" is determined through a number of frequency dividers, called A, N and M. Modulation can be done by switching between two sets of dividers – in this case, the dividers are referred to as A0, N0, M0 and A1, N1, M1 (for transmitting a "0" and a "1", respectively).

If "Transmit_0x00" and "Receive_0x00" constants are made (for register 0x00), and all register values except for frequency dividers are kept constant, then the challenge is:

How does the micro controller get the frequency dividers for a specific frequency?

That question is discussed in this sub-section (note: *how to calculate the values* are not described here, refer to e.g., "RF TestBench" and the MICRFx data sheet).

It is not obvious that this is an important point. However, consider this case:

- 50 RF channels are used
- Modulation type is "switching between two sets of dividers"
- The values of the frequency dividers can't be calc'ed on-the-fly (this is too complex)

For this case, there must be one set of A, N, M for every receive (rx) frequency, and two sets for every transmit (tx0 and tx1) frequency (a total of 3 sets). A is 1 byte long, N and M are 2 bytes long i.e., 5 bytes/set.

The total number of bytes in this case, then, is 50*3*5 = 750 bytes. This may be a significant portion of the memory in the micro controller.

There are (at least) four different approaches ("RFPRG_Frequency" is a register holding which frequency number, or channel, to enter):

1. Make a "switch" statement or an if-else structure, testing the value of "RFPRG_Frequency" and filling up the values to write
2. Make a look-up table in program memory, using "RFPRG_Frequency" as index
3. Make a look-up table in data memory, using "RFPRG_Frequency" as index
4. Calculate the frequency dividers on-the-fly based on "RFPRG_Frequency"

Method 1: Not described further, because it uses more program memory and is slower than method number 2).

Method 2: Described in the next sub-section (below).

Method 3: Not described further, because most micros can't spend "a lot of" data memory

Method 4: This can be used if the frequency dividers are easy to calculate on-the-fly; when using "VCO modulation" and the distance between the channels are a multiple of the comparison frequency. In this case, M can be kept constant, and A is incremented a number of times. If A overflows (i.e. > 63), then N is incremented and A is decremented with 31. To understand this algorithm, refer to the MICRF50x data sheet and view the equations for calculating the RF frequency.

Example, Method 4:

Use 25 frequencies, stepping 1 MHz/frequency. Start at 903MHz (f0), end at 927MHz (f24). The xtal used is 16MHz (fext), and the comparison frequency (fcomp) is 100kHz (fcomp = fext/M => M = 160). The MICRF505 chip is used.

A and N for f0:

fcomp = fext/M = fRF*2/(31*N+A) => (31*N+A) = fRF*2/fcomp = 903e6*2/100e3 = 18060

=>

N= 582, A=18

=>

A_f0 = 18

N_f0 = 582

M_f0 = 160

Now, from the equations, it can be seen that incrementing A with 1, will increment fRF with fcomp/2, i.e. 50kHz. Then, to change fRF 1 MHz, A must be increased with 20.

Use A_temp and N_temp as temporary help-registers when calc'ing A, N, M for fn:

A_temp = A_f0 + 20*n

N_temp = integer part of (A_temp / 31)

A_temp = A_temp mod 31

A_fn= A_temp

N_fn = N_f0 + N_temp

M_fn = M_f0

For example, f17 (920MHz) is then

A_temp=18+20*17=358

N_temp = int(358/31)=11

A_temp = 358 mod 31 = 17

A_f17 = 17

N_f17 = 582+11=593

M_f17 = 160

This method requires some calculations, but should be OK for most MCUs. It can be used for VCO modulation, where frequency spacing is a multiple of the comparison frequency.

Storing the Frequency Dividers in Program Memory

Probably the most efficient way to get the frequency dividers[1] is by making a program-memory look-up table. If using the C programming language, this can be done through the "const" type. In the example below, made for PIC18, the keyword "rom" is included.

The actual values to use for A, N and M, can be calculated using the "RF TestBench" program. In the example below, it is assumed that the dividers are included (in, for example, an .h file).

Then the "RFPRG_Frequency" variable is used as an index.

Example:

Using 4 frequencies (0-3) and modulation type "switching between 2 sets of dividers"

"uint8" and "uint16" are "unsigned char" (8 bit) and "unsigned int" (16 bit), respectively.

```
rom const uint8 RFPRG_A_Values_Rx[4]  = {A_0rx, A_1rx, A_2rx, A_3rx};
rom const uint8 RFPRG_A_Values_Tx0[4] = {A_0tx0, A_1tx0, A_2tx0, A_3tx0};
rom const uint8 RFPRG_A_Values_Tx1[4] = {A_0tx1, A_1tx1, A_2tx1, A_3tx1};

rom const uint16 RFPRG_N_Values_Rx[4]  = {N_0rx, N_1rx, N_2rx, N_3rx};
rom const uint16 RFPRG_N_Values_Tx0[4] = {N_0tx0, N_1tx0, N_2tx0, N_3tx0};
rom const uint16 RFPRG_N_Values_Tx1[4] = {N_0tx1, N_1tx1, N_2tx1, N_3tx1};

//assuming all M values <=255 (1 byte/value is enough):
rom const uint8 RFPRG_M_Values_Rx[4]  = {M_0rx, M_1rx, M_2rx, M_3rx};
rom const uint8 RFPRG_M_Values_Tx0[4] = {M_0tx0, M_1tx0, M_2tx0, M_3tx0};
rom const uint8 RFPRG_M_Values_Tx1[4] = {M_0tx1, M_1tx1, M_2tx1, M_3tx1};
```

Then, to get the value to enter, use a statement like this:

```
byte_to_write = RFPRG_A_Values_Tx0[
RFPRG_Frequency];
```

In Table 4, a procedure for filling the complete control word into the array RFPRG_Buffer[] is shown. The procedure is made for modulation "using 2 sets of dividers". It is assumed that all constants (like "cwrd_byte0x01") are included in an .h file.

Note: If "VCO modulation" is used, the procedure can be simplified – in this case, only one set of dividers can be filled in both rx and tx mode (the same set of dividers for rx and tx).

A flag called "RFPRG_ENTER_TX" is set if transmit-mode is to be entered (cleared otherwise). The variable "RFPRG_Frequency" holds the frequency number to enter.

[1] "Get the frequency dividers": This means how to get the already calc'ed A, N M values to use, it does not mean how to calculate the values. The A, N, M values are stored as constants in program memory in this case.

```
//Control word constants included in .h file
//It is assumed that all M-dividers are <= 255
//Pre:  RFPRG_ENTER TX_FLAG and RFPRG_Frequency updated
//Post: RFPPRG_Buffer[] filled with rx or tx word using the
//      specified frequency number
void
RFPRG_Fill_Complete_ControlWord( void)
{
   If( RFPRG_ENTER_TX)
   {
      RFPRG_Buffer[ 0x00] = cwrd_byte0x00_transmit;
   }
   else
   {
      RFPRG_Buffer[ 0x00] = cwrd_byte0x00_receive;
   }

   RFPRG_Buffer[ 0x01] = cwrd_byte0x01;
   RFPRG_Buffer[ 0x02] = cwrd_byte0x02;
   RFPRG_Buffer[ 0x03] = cwrd_byte0x03;
   RFPRG_Buffer[ 0x04] = cwrd_byte0x04;
   RFPRG_Buffer[ 0x05] = cwrd_byte0x05;
   RFPRG_Buffer[ 0x06] = cwrd_byte0x06;
   RFPRG_Buffer[ 0x07] = cwrd_byte0x07;
   RFPRG_Buffer[ 0x08] = cwrd_byte0x08;
   RFPRG_Buffer[ 0x09] = cwrd_byte0x09;

   If( RFPRG_ENTER_TX) // transmit mode, "use 2 sets of dividers"
   {
      RFPRG_Buffer[ 0x0A] = RFPRG_A_Values_Tx0[ RFPRG_Frequency];
      RFPRG_Buffer[ 0x0B] = RFPRG_N_Values_Tx0[ RFPRG_Frequency] >> 8;
      RFPRG_Buffer[ 0x0C] = (uint8)RFPRG_N_Values_Tx0[ RFPRG_Frequency];
      RFPRG_Buffer[ 0x0D] = 0;
      RFPRG_Buffer[ 0x0E] = (uint8)RFPRG_M_Values_Tx0[ RFPRG_Frequency];

      RFPRG_Buffer[ 0x0F] = RFPRG_A_Values_Tx1[ RFPRG_Frequency];
      RFPRG_Buffer[ 0x10] = RFPRG_N_Values_Tx1[ RFPRG_Frequency] >> 8;
      RFPRG_Buffer[ 0x11] = (uint8)RFPRG_N_Values_Tx1[ RFPRG_Frequency];
      RFPRG_Buffer[ 0x12] = 0;
      RFPRG_Buffer[ 0x13] = (uint8)RFPRG_M_Values_Tx1[ RFPRG_Frequency];
   }
   else // receive mode (A1, N1, M1 = A0, N0, M0)
   {
      RFPRG_Buffer[ 0x0A] = RFPRG_A_Values_Rx[ RFPRG_Frequency];
      RFPRG_Buffer[ 0x0B] = RFPRG_N_Values_Rx[ RFPRG_Frequency] >> 8;
      RFPRG_Buffer[ 0x0C] = (uint8)RFPRG_N_Values_Rx[ RFPRG_Frequency];
      RFPRG_Buffer[ 0x0D] = 0;
      RFPRG_Buffer[ 0x0E] = (uint8)RFPRG_M_Values_Rx[ RFPRG_Frequency];

      RFPRG_Buffer[ 0x0F] = RFPRG_Buffer[ 0x0A];
      RFPRG_Buffer[ 0x10] = RFPRG_Buffer[ 0x0B];
      RFPRG_Buffer[ 0x11] = RFPRG_Buffer[ 0x0C];
      RFPRG_Buffer[ 0x12] = 0;
      RFPRG_Buffer[ 0x13] = RFPRG_Buffer[ 0x0E];
   }

   RFPRG_Buffer[ 0x14] = cwrd_byte0x14;
   RFPRG_Buffer[ 0x15] = cwrd_byte0x15;
```

Table 4. Filling Complete Control Word

Using the LD Pin

The procedure below illustrates PIC18F. It uses a timer, called TMR0, to test for "timeouts". Refer to the data sheet for your micro controller to make a similar functionality. It also assumes a 11.0592MHz xtal connected to the micro controller (and an instruction time of ¼ of the xtal period).

In this procedure:

1. If LD pin goes from low to high, then return
2. If LD "stuck" high for x usec, then it is assumed that LD will not go low, and the procedure returns
3. If LD is "stuck" low for y msec, then the procedure returns anyway (to avoid a program hang-up)

```
// Returns when
//    1) LD detected going from low to high, or
//    2) LD "stuck" high for xusec or
//    3) LD "stuck" low for y msec

void
RFPRG_Wait_Lock( void)
{
   uint8
       in_lock,
       quit_lock;

   in_lock   = 0;
   quit_lock = 0;

   do
   {
      // Initiate tmr0 for 800 usec timeout ("Timeout1")
      T0CON = 0;
      TMR0L = 118; // 11.0592MHz xtal => timeout = 800 usec (pre 1:16 =>
                   // count 16*(256-118) = 16*138 times )

      INTCONbits.TMR0IE = 0;
      INTCONbits.TMR0IF = 0;
      T0CON = 0b11010011; // TMR0 on, 8-bit, prescaler 1:16

      while( ( not in_lock) and ( RF_LD))
      { // If LD "stuck" ON until Timeout1, or LD low: break
         if( INTCONbits.TMR0IF) // Test if Timeout1 reached
         {
            in_lock = 0xFF; // LD "stuck" ON until Timeout1: LD is valid
         }
      }

      if( not RF_LD) // LD detected low
      {
         //Initiate tmr0 for 8 msec (approx) timeout ("Timeout2")
         T0CON =   0;
         TMR0L = 170; // 11.0592MHz xtal => timeout = 8 msec (pre 1:256 =>
                      // count 256*(256-170) times
         INTCONbits.TMR0IF = 0;
         T0CON = 0b11010111; // TMR0 on, 8-bit, prescaler 1:256

         while( ( not quit_lock) and ( not RF_LD)) //If LD "stuck" OFF
                                                   //until Timeout2, or LD
                                                   //going OFF to ON: quit
         {
            if( INTCONbits.TMR0IF)
            {
               quit_lock = 0xFF; // Lock detect is not found! Quit
            }
         }

         if( RF_LD)         // note: returns asap if LD low, then high
         {                  //
            in_lock = 0xFF; //
         }                  //

      }
   } while( ( not in_lock) and ( not quit_lock));
}
```

Table 5. Waiting for LD

Programming Method 1: Always-Enter-All-Bytes

From the previous sub-sections, we can discern the building blocks to program the MICRF50x radio transceiver.

The most straightforward way to program the chip is to always enter the complete control word. This is described in this sub-section (Programming Method 1). A complete control word is always entered, except: When in transmit-mode or, to set PA on only desktop register (reg0x00) is entered. A more sophisticated method is to only enter the necessary bytes. See next section (Programming Method 2). Method 2 requires more program memory, but has increased speed.

Refer to Figure 7.

Note that some of the building blocks procedures (like RFPRG_Write_Byte(…)) can be any of the procedures described previously.

Only "write-the-control word" procedures are given here. The user can make read-procedures as an exercise, or refer to the MICRF50x development firmware, available for free from www.micrel.com (many applications do not include the read-procedures).

The procedures are written in C, "bit-banging" is used.

A simple, complete program that uses these procedures is made. It is made for PIC18F4320. The project is called "Basic_Prog1". This program enters tx, makes a delay, enters rx, makes a delay, then this sequence is repeated on the next frequency.

For the constants and procedures below:

- MICRF505, 915MHz band
- 25 frequencies are used. Frequency numbers are stored randomly in a data memory table
- Modulation type: VCO modulation (A1, N1, M1 = A0, N0, M0)
- Fcomp is in the 100kHz range
- Bitrate: 38462 bps
- Including a 3.5 msec power-on reset delay
- Using LD pin when entering transmit-mode
- Initially, complete control word must be filled: Entering tx-mode with PA off
- Later, complete control word filled as well.
- A, N, M values for the selected frequency are calc'ed on-the-fly

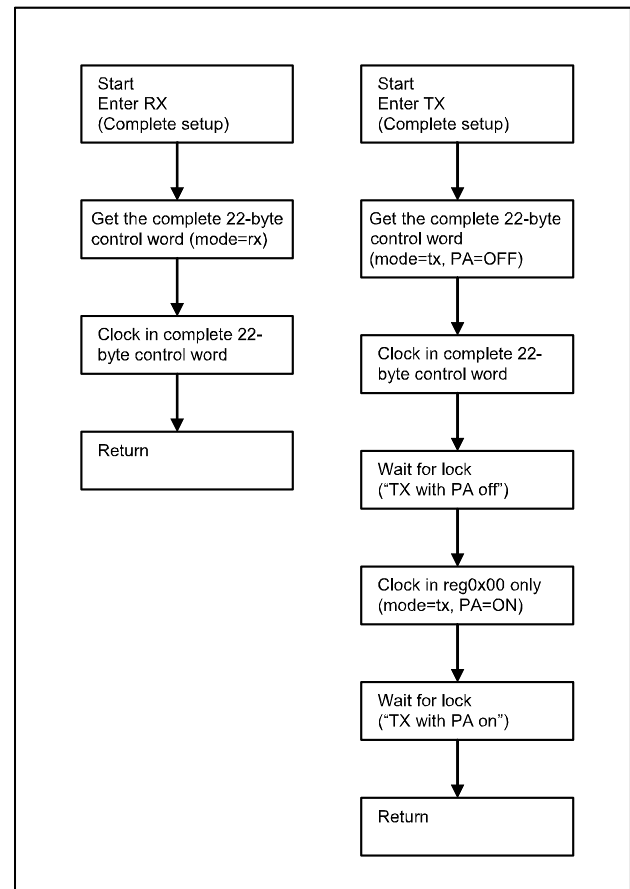

Figure 7. Enter RX/TX, Using Complete Control Word

Programming Method 2: Enter Some of the Registers Only

Complete control word is entered initially. Later, only desktop register (reg0) and/or "frequency divider registers" (reg10-reg19) are entered.

Refer to Figure 8.

Only "write-the-control word" procedures are given here. The user can make read-procedures as an exercise, or refer to the MICRF50x development firmware, available for free from www.micrel.com (many applications do not include the read-procedures).

The procedures are written in C. "bit-banging" is used.

A simple, complete program that uses these procedures is made. It is made for PIC18F4320. The project is called "Basic_Prog2". This program enters tx, makes a delay, enters rx, makes a delay, then this sequence is repeated on the next frequency.

For the constants and procedures below:

- MICRF505, 915MHz band
- 25 frequencies are used. Frequency numbers are stored randomly in a data memory table
- Modulation type: “switching between 2 sets of dividers” (A1, N1, M1 <> A0, N0, M0 in transmit-mode)
- Fcomp is in the 500kHz range
- All N and M values are < 256, that is: Sufficient to store 1 byte/divider
- Bitrate: 19231 bps
- Including a 3.5 msec power-on reset delay
- Using LD pin when entering transmit-mode
- Initially, complete control word must be filled: Entering tx-mode with PA off
- Later, only desktop reg and/or frequency dividers are filled
- A, N, M values for all frequencies are stored in program memory and the wanted values are fetched by memory table look-ups

Refer to Appendix A for a complete listing of the procedures used.

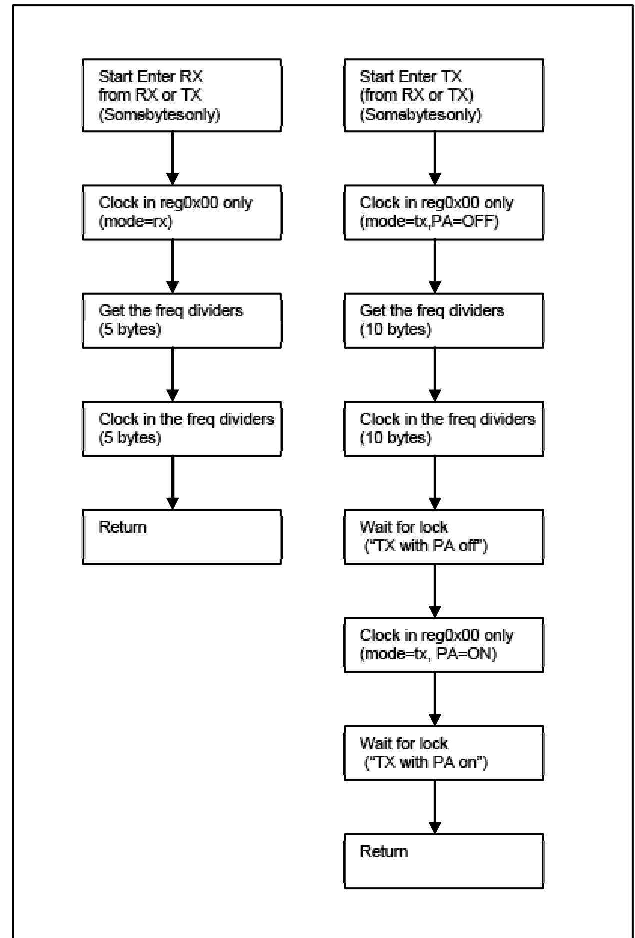

Figure 8. Enter RX/TX Write Some-of-the-Registers Only

Appendix A: Complete Set of Procedures, No. 1

```
//Fields in the Control word (constants)
// ****************************************************
//     915MHz band: f0=903MHz, f24=927MHz
//     Closed loop VCO mod
//     38462 bauds
//     Sallen key: 230KHz
//     Switched cap: 200kHz
//     Deviation: 101 kHz (903MHz) to 110kHz (927MHz)
//
// ****************************************************

// address 0x00
#define By_LNA_v  0
#define PA_v      0
#define Sync_en_v 1
#define Mode_v    0
#define Load_en_v 1
#define ChxxWrd0x00  ( (By_LNA_v<<7) + (PA_v<<4) + (Sync_en_v<<3) + (Mode_v<<1) + (Load_en_v))
#define ChxxWrd0x00r ( (By_LNA_v<<7) + (PA_v<<4) + (Sync_en_v<<3) + (      2<<1) + (Load_en_v))
#define ChxxWrd0x00t ( (By_LNA_v<<7) + (PA_v<<4) + (Sync_en_v<<3) + (      3<<1) + (Load_en_v))
#define RFPRG_TX_PA_ON ( (By_LNA_v<<7)+ ( 7 <<4) + (Sync_en_v<<3) + (      3<<1) + (Load_en_v))


// address 0x01
#define Modulation_v    0  // Closed loop VCO mod
#define RSSI_en_v       1
#define LD_en_v         1
#define PF_FC_v         2  // Sallen Key: 230kHz
#define ChxxWrd0x01 ( (Modulation_v<<6) + (RSSI_en_v<<3) + (LD_en_v<<2) + (PF_FC_v) )


// address 0x02
#define CP_HI_v  0
#define SC_by_v  0
#define VCO_by_v 0
#define PA_by_v  0
#define OUTS_v   0
#define ChxxWrd0x02 ( (CP_HI_v<<7) + (SC_by_v<<6) + (VCO_by_v<<5) + (PA_by_v<<4) + (OUTS_v) )


// address 0x03
#define VCO_IB_v    1 // VCO_IB: 915MHz setting
#define VCO_freq_v  2 // VCO_freq: 915MHz setting
#define ChxxWrd0x03 ( (0xC0) + (VCO_IB_v<<2) + (VCO_freq_v) )


// 101-110kHz deviation
// address 0x04
#define Mod_F_v    3
#define Mod_I_v    3
#define ChxxWrd0x04 ( ( Mod_F_v<<5) + (Mod_I_v) )


// address 0x05
#define Mod_A_v  1
#define ChxxWrd0x05 ( ( 0x10) + (Mod_A_v) )


// 38462 bps
// address 0x06 and 0x07
#define Mod_clkS_v      5
#define BitSync_clkS_v  6
#define BitRate_clkS_v  2
#define RefClk_K_v     13
#define ChxxWrd0x06 ( ( Mod_clkS_v<<4) + (BitSync_clkS_v<<1) + (BitRate_clkS_v>>2) )
#define ChxxWrd0x07 ( ( ( BitRate_clkS_v<<6) + (RefClk_K_v) ) & 0x00FF )
```

```
// address 0x08
#define ScClk_v  2
#define ChxxWrd0x08 ( ( 0xC0) + (ScClk_v) )


// address 0x09
#define XCOtune_v  16
#define ChxxWrd0x09 ( ( 0x60) + (XCOtune_v) )


//address 0x0A-0x13: Dividers
// ************************
// f0
// 903MHz, 100kHz comp freq
//*************************
#define A_CH00  18
#define N_CH00  582
#define M_CH00  160


// address 0x14
#define ChxxWrd0x14 (0xB5)


// address 0x15
#define FEEC_v 0
#define ChxxWrd0x15 ( FEEC_v)
```

```
//List of Procedures:

void RFPRG_Init_Rf( void);
void RFPRG_Enter_Rx( void);
void RFPRG_Enter_Tx( void);

void RFPRG_Clock_Complete( void);
void RFPRG_Clock_Desktop( uint8 reg0_value);
void RFPRG_Wait_Lock( void);

void RFPRG_Fill_Controlword( void);
void RFPRG_Rf_Write( uint8 write_address, uint8 nr_of_bytes);
void RFPRG_Write_Byte( static uint8 byte_to_write);
void RFPRG_Inc_FreqIndex( void);



//List of Global Variables used, all are "uint8" ("unsigned char"):
#pragma udata access main_regs_access
near uint8
    RFPRG_Frequency,        //= Frequency number to use (0…24)
    RFPRG_FreqIndex,        //= Index to use in RFPRG_FreqTable[]
    RFPRG_FreqTable[25],    //= Randomly ordered list of frequencies
    RFPRG_Enter_Tx_Flag,    //= 1 if enter tx, =0 if enter rx
    RFPRG_Keep_PA_Off_Flag, //= 1 if PA should be off, = 1 if PA should be set on
    RFPRG_Prog_Buffer[23];  // = Control word to fill into the MICRF50x



#pragma udata
#pragma code
```

```
//Purpose: Fill up FreqTable with a randomly ordered list of frequency numbers.
//Pre: Nothing
//Post:RFPRG_FreqTable filled
//Notes: Users may select a different pattern.
//            A more advanced procedure may generate a unique pattern based on some input
//            If the application cant' spend data bytes on this, then the table should be
//            implemented in program memory
//            Sequence: 7,16,8,15,9,4,1,0,2,10,3,12,17,24,20,13,6,5,14,19,22,18,23,21,11
void
RFPRG_Fill_FreqTable( void)
{
    RFPRG_FreqTable[0] = 7;
    RFPRG_FreqTable[1] = 16;
    RFPRG_FreqTable[2] = 8;
    RFPRG_FreqTable[3] = 15;

    RFPRG_FreqTable[4] = 9;
    RFPRG_FreqTable[5] = 4;
    RFPRG_FreqTable[6] = 1;
    RFPRG_FreqTable[7] = 0;

    RFPRG_FreqTable[8] = 2;
    RFPRG_FreqTable[9] = 10;
    RFPRG_FreqTable[10] = 3;
    RFPRG_FreqTable[11] = 12;

    RFPRG_FreqTable[12] = 17;
    RFPRG_FreqTable[13] = 24;
    RFPRG_FreqTable[14] = 20;
    RFPRG_FreqTable[15] = 13;

    RFPRG_FreqTable[16] = 6;
    RFPRG_FreqTable[17] = 5;
    RFPRG_FreqTable[18] = 14;
    RFPRG_FreqTable[19] = 19;

    RFPRG_FreqTable[20] = 22;
    RFPRG_FreqTable[21] = 18;
    RFPRG_FreqTable[22] = 23;
    RFPRG_FreqTable[23] = 21;

    RFPRG_FreqTable[24] = 11;
}



void
RFPRG_Init_Rf( void)
{
    // Initiate the global variables used
    RFPRG_Fill_FreqTable();
    RFPRG_FreqIndex=0;
    RFPRG_Frequency = RFPRG_FreqTable[ RFPRG_FreqIndex];


    // Include a power-on-reset delay (may be skipped)
    GEN_Delay3p5msec();  // a GENeral procedure


    // Program the complete control word: use "TX with PA off"
    // The procedure waits for LD - this method makes sure that the initial time
    // from power-down to active mode is included before continuing
    RFPRG_Keep_PA_Off_Flag = 1;
    RFPRG_Enter_Tx();
    RFPRG_Keep_PA_Off_Flag = 0;
}
```

```
//Purpose: Enter receive-mode
//Pre: RFPRG_FreqIndex correct
//Post: Receive mode entered
//Note: Returns without waiting for LD
void
RFPRG_Enter_Rx( void)
{
    RFPRG_Enter_Tx_Flag = 0;
    RFPRG_Fill_Controlword();
    RFPRG_Clock_Complete();
}



//Purpose: Enter transmit-mode
//Pre: RFPRG_FreqIndex correct, RFPRG_Keep_PA_Off_Flag =1 or 0
//Post: Transmit mode entered
//Note: Returns after waiting for LD

void
RFPRG_Enter_Tx( void)
{
    // First, enter tx with PA off
    RFPRG_Enter_Tx_Flag = 1;
    RFPRG_Fill_Controlword(); // Note: PA-bits are set to "off"
    RFPRG_Clock_Complete();
    RFPRG_Wait_Lock(); // Wait for lock "TX with PA off" (or timeout)

    // Then, enter tx with PA on (if PA should be set on)
    if( !RFPRG_Keep_PA_Off_Flag)
    {
        RFPRG_Clock_Desktop( RFPRG_TX_PA_ON); //Clock in 1 register only
        RFPRG_Wait_Lock(); // Wait for lock "TX with PA on" (or timeout)
    }
}



//Purpose: Bring CS high, Clock in 22 bytes (reg0-reg21), bring Clock low
//Pre: RFPRG_Prog_Buffer[ 0 - 21]  filled
//Post: MICRF50x programmed
void
RFPRG_Clock_Complete( void)
{
    RF_CS = 1;
    RFPRG_Rf_Write( 0,22);  // Start writing from address 0, write 22 bytes of the controlword
    RF_CS = 0;
}

//Purpose: Bring CS high, Clock in 1 byte (reg0), bring Clock low
//Input: Byte to clock into desktop
//Post: MICRF50x programmed, RFPRG_Prog_Buffer[ 0] changed
void
RFPRG_Clock_Desktop( uint8 reg0_value)
{
    RFPRG_Prog_Buffer[ 0] = reg0_value;
    RF_CS = 1;
    RFPRG_Rf_Write( 0,1);  // Start writing from address 0, write 1 byte of the controlword
    RF_CS = 0;
}
```

```
// Purpose: Wait for lock
// Pre: Control word entered, CS brought low
// Post: Return from procedure when
//    1) LD going from low to high, or
//    2) LD "stuck" high for xusec or
//    3) LD "stuck" low for y msec
//       in the last case, RFchip is not in lock
// Note: If going from power-down or standby into rx or tx,
//       then some additional timeout may be required
// Important: This procedure uses a built-in MCU module (TMR0)
//            Make changes for the selected MCU and the MCU
//            speed. The selected “timeouts” may be changed
//            for a particular application
void
RFPRG_Wait_Lock( void)
{
    uint8 in_lock,
          quit_lock;

    in_lock   = 0;
    quit_lock = 0;

    do
    {
        // Initiate tmr0 for 800 usec timeout ("Timeout1")
        // 11.0592MHz xtal =>
        // pre 1:16 => count 16*(256-118) = 16*138 times
        T0CON = 0;
        TMR0L = 118;

        INTCONbits.TMR0IE = 0; // Disable interrupt
        INTCONbits.TMR0IF = 0; // Clear interrupt-flag
        T0CON = 0b11010011;    // TMR0 on, prescaler 1:16

        // If LD "stuck" on until Timeout1,
        // or LD detected low: quit
        while( ( not in_lock) and ( RF_LD))
        {
            if( INTCONbits.TMR0IF) // Test if Timeout1 reached
            {
                // LD "stuck" ON until Timeout1, LD is valid
                in_lock = 0xFF;
            }
        }

        if( not RF_LD) // LD detected low
        {
            // Initiate tmr0 for 8 msec (approx) timeout
            // ("Timeout2") 11.0592MHz xtal =>
            // pre 1:256 => count 256*(256-170) times
            T0CON = 0; //Stop timer0
            TMR0L = 170; //
            INTCONbits.TMR0IE = 0; // Disable interrupt
            INTCONbits.TMR0IF = 0; // Clear interrupt-flag
            T0CON = 0b11010111;    // TMR0 on, prescaler 1:256

            // If LD "stuck" OFF until Timeout2,
            // or LD going from OFF to ON: quit
            while( ( not quit_lock) and ( not RF_LD))
            {
                if( INTCONbits.TMR0IF) // test Timeout2
                {
                   // Lock detect is not found! Quit
                   quit_lock = 0xFF;
                }
             }

             // Return asap if LD detected low, then high
            if( RF_LD)
            {
              in_lock = 0xFF;
```

```
            }
        }
    } while( ( not in_lock) and ( not quit_lock));
    // wait until LD deteteced high for timeout1,
    //or low for timeout2

    // Finally, stop timer0
    T0CON = 0;
    TMR0L = 0;
    INTCONbits.TMR0IF = 0;
}




// Purpose: Get the control word to enter into MICRF50x
// Pre: RFPRG_FreqIndex and RFPRG_Enter_Tx_Flag updated
// Post: RFPRG_Prog_Buffer is filled
void
RFPRG_Fill_Controlword( void)
{
    uint16
        A_help,
        N_help,
        M_help;


    RFPRG_Frequency = RFPRG_FreqTable[ RFPRG_FreqIndex];

    //Fill reg0, different for rx and tx mode (note: “tx with PA off”)
    if( RFPRG_Enter_Tx_Flag)
    {
        RFPRG_Prog_Buffer[ 0] = ChxxWrd0x00t;
    }
    else
    {
        RFPRG_Prog_Buffer[ 0] = ChxxWrd0x00r;
    }

    //Fill reg1-reg9, equal for rx and tx
    RFPRG_Prog_Buffer[ 1] = ChxxWrd0x01;
    RFPRG_Prog_Buffer[ 2] = ChxxWrd0x02;
    RFPRG_Prog_Buffer[ 3]=  ChxxWrd0x03;
    RFPRG_Prog_Buffer[ 4] = ChxxWrd0x04;

    RFPRG_Prog_Buffer[ 5] = ChxxWrd0x05;
    RFPRG_Prog_Buffer[ 6] = ChxxWrd0x06;
    RFPRG_Prog_Buffer[ 7] = ChxxWrd0x07;
    RFPRG_Prog_Buffer[ 8] = ChxxWrd0x08;
    RFPRG_Prog_Buffer[ 9] = ChxxWrd0x09;


    //Fill reg10-reg19, equal for rx and tx IF VCO MOD is selected
    // and: reg15-19 = reg10-14 IF VCO MOD is selected
    // The method below requires 100kHz fcomp, i.e. inc’ing A will increase fRF with 50kHz
    // 1MHz spacing => A must be inc’ed 20 times for every frequency
    A_help = A_CH00 + 20*(uint16)RFPRG_Frequency;
    N_help = N_CH00;
    M_help = M_CH00;
    while( A_help > (uint8)31)
    {
        A_help -= 31;
        N_help +=  1;
    }
    RFPRG_Prog_Buffer[ 10] = (uint8)A_help;
    RFPRG_Prog_Buffer[ 11] = (uint8)(N_help >> 8); // Most significant part
    RFPRG_Prog_Buffer[ 12] = (uint8)N_help;      // Least significant part
    RFPRG_Prog_Buffer[ 13] = (uint8)(M_help >> 8); // Most significant part
    RFPRG_Prog_Buffer[ 14] = (uint8)M_help;      // Least significant part
```

```
    RFPRG_Prog_Buffer[ 15] = RFPRG_Prog_Buffer[ 10];
    RFPRG_Prog_Buffer[ 16] = RFPRG_Prog_Buffer[ 11];
    RFPRG_Prog_Buffer[ 17] = RFPRG_Prog_Buffer[ 12];
    RFPRG_Prog_Buffer[ 18] = RFPRG_Prog_Buffer[ 13];
    RFPRG_Prog_Buffer[ 19] = RFPRG_Prog_Buffer[ 14];

    //Fill reg20-reg21, equal for rx and tx
    RFPRG_Prog_Buffer[ 20] = ChxxWrd0x14;
    RFPRG_Prog_Buffer[ 21] = ChxxWrd0x15;
}



// Pre: RFPRG_Prog_Buffer holds bytes to write
// Input: write_address = address of 1st byte to write, nr_of_bytes = number of bytes to write
void
RFPRG_Rf_Write( uint8 write_address, uint8 nr_of_bytes)
{
    //made static to increase speed and decrease program memory
    static uint8
        i,
        j;

    RFPRG_Write_Byte( (write_address<<1)+0); //address is 7 msb, r/w-bit(lsb) = 0 for write

    j = nr_of_bytes;
    i = 0;

    while( j)
    {
        RFPRG_Write_Byte( RFPRG_Prog_Buffer[ i]);
        i++;
        j--;
    }
}



//Based on PIC18F micro controller:
//   Program Memory: 15 instructions
//   Data Memory: 2 bytes
//   Execution time:
//      100 instruction cycles
void
RFPRG_Write_Byte( static uint8 byte_to_write)
{
    static uint8 temp;

    temp = 0x80;

    do
    {
        RF_IO_OUT = 0;
        if( byte_to_write bit_and temp)
        {
            RF_IO_OUT = 1;
        }
        RF_SCLK = 1;
        temp  >>= 1;
        RF_SCLK = 0;
    }
    while( temp);
}



void
RFPRG_Inc_FreqIndex( void)
{
    RFPRG_FreqIndex++;
```

```
    if( RFPRG_FreqIndex > (uint8)24)
    {
        RFPRG_FreqIndex = 0;
    }
}
```

Appendix A. Complete Set of Procedures, No. 2

```
//Fields in the Control word (constants)
// ****************************************************
//     915MHz band
//     2 sets of dividers mod
//     19231 bauds
//     Sallen key: 230KHz
//     Switched cap: 200kHz
// ****************************************************

// address 0x00
#define By_LNA_v  0
#define PA_v      0
#define Sync_en_v 1
#define Mode_v    0
#define Load_en_v 1
#define ChxxWrd0x00  ( (By_LNA_v<<7) + (PA_v<<4) + (Sync_en_v<<3) + (Mode_v<<1) + (Load_en_v))
#define ChxxWrd0x00r ( (By_LNA_v<<7) + (PA_v<<4) + (Sync_en_v<<3) + (     2<<1) + (Load_en_v))
#define ChxxWrd0x00t ( (By_LNA_v<<7) + (PA_v<<4) + (Sync_en_v<<3) + (     3<<1) + (Load_en_v))
#define RFPRG_TX_PA_ON ( (By_LNA_v<<7)+ ( 7 <<4) + (Sync_en_v<<3) + (     3<<1) + (Load_en_v))


// address 0x01
#define Modulation_v    2  // 2 sets of dividers mod
#define RSSI_en_v       1
#define LD_en_v         1
#define PF_FC_v         2  // Sallen Key: 230kHz
#define ChxxWrd0x01 ( (Modulation_v<<6) + (RSSI_en_v<<3) + (LD_en_v<<2) + (PF_FC_v) )


// address 0x02
#define CP_HI_v  0
#define SC_by_v  0
#define VCO_by_v 0
#define PA_by_v  0
#define OUTS_v   0
#define ChxxWrd0x02 ( (CP_HI_v<<7) + (SC_by_v<<6) + (VCO_by_v<<5) + (PA_by_v<<4) + (OUTS_v) )


// address 0x03
#define VCO_IB_v    1 // VCO_IB: 915MHz setting
#define VCO_freq_v  2 // VCO_freq: 915MHz setting
#define ChxxWrd0x03 ( (0xC0) + (VCO_IB_v<<2) + (VCO_freq_v) )


// address 0x04
#define Mod_F_v    0
#define Mod_I_v    0
#define ChxxWrd0x04 ( ( Mod_F_v<<5) + (Mod_I_v) )


// address 0x05
#define Mod_A_v  0
#define ChxxWrd0x05 ( ( 0x10) + (Mod_A_v) )


// 19231 bps
// address 0x06 and 0x07
#define Mod_clkS_v      0
#define BitSync_clkS_v  5
#define BitRate_clkS_v  1
#define RefClk_K_v     13
#define ChxxWrd0x06 ( ( Mod_clkS_v<<4) + (BitSync_clkS_v<<1) + (BitRate_clkS_v>>2) )
#define ChxxWrd0x07 ( ( ( BitRate_clkS_v<<6) + (RefClk_K_v) ) & 0x00FF )


// address 0x08
#define ScClk_v  2
```

```
#define ChxxWrd0x08 ( ( 0xC0) + (ScClk_v) )


// address 0x09
#define XCOtune_v  16
#define ChxxWrd0x09 ( ( 0x60) + (XCOtune_v) )



//address 0x0A-0x13: Dividers, for every freq, rx, tx0 and tx1-sets are made:
// The values are calc'ed with RF TestBench
//Frequency  0 (904.296MHz)
 #define A_0rx    14
 #define N_0rx    98
 #define M_0rx    27
 #define A_0tx0   26
 #define N_0tx0  145
 #define M_0tx0   40
 #define A_0tx1   27
 #define N_0tx1  145
 #define M_0tx1   40

//Frequency  1 (905.185MHz)
 #define A_1rx    17
 #define N_1rx    98
 #define M_1rx    27
 #define A_1tx0    1
 #define N_1tx0  135
 #define M_1tx0   37
 #define A_1tx1    5
 #define N_1tx1  113
 #define M_1tx1   31

//Frequency  2 (906.105MHz)
 #define A_2rx    26
 #define N_2rx   138
 #define M_2rx    38
 #define A_2tx0    9
 #define N_2tx0  102
 #define M_2tx0   28
 #define A_2tx1   30
 #define N_2tx1  105
 #define M_2tx1   29

//Frequency  3 (907.000MHz)
 #define A_3rx     1
 #define N_3rx   117
 #define M_3rx    32
 #define A_3tx0   21
 #define N_3tx0  120
 #define M_3tx0   33
 #define A_3tx1   12
 #define N_3tx1  113
 #define M_3tx1   31

//Frequency  4 (907.897MHz)
 #define A_4rx    24
 #define N_4rx   142
 #define M_4rx    39
 #define A_4tx0   13
 #define N_4tx0  146
 #define M_4tx0   40
 #define A_4tx1   16
 #define N_4tx1  102
 #define M_4tx1   28

//Frequency  5 (908.842MHz)
 #define A_5rx     8
 #define N_5rx   139
 #define M_5rx    38
 #define A_5tx0   29
```

```
 #define N_5tx0   98
 #define M_5tx0   27
 #define A_5tx1   19
 #define N_5tx1  124
 #define M_5tx1   34

//Frequency  6 (909.778MHz)
 #define A_6rx     2
 #define N_6rx   132
 #define M_6rx    36
 #define A_6tx0   22
 #define N_6tx0  113
 #define M_6tx0   31
 #define A_6tx1   23
 #define N_6tx1  124
 #define M_6tx1   34

//Frequency  7 (910.667MHz)
 #define A_7rx     5
 #define N_7rx   110
 #define M_7rx    30
 #define A_7tx0    6
 #define N_7tx0  143
 #define M_7tx0   39
 #define A_7tx1    7
 #define N_7tx1  143
 #define M_7tx1   39

//Frequency  8 (911.600MHz)
 #define A_8rx     1
 #define N_8rx   147
 #define M_8rx    40
 #define A_8tx0   19
 #define N_8tx0  117
 #define M_8tx0   32
 #define A_8tx1    8
 #define N_8tx1   99
 #define M_8tx1   27

//Frequency  9 (912.500MHz)
 #define A_9rx    23
 #define N_9rx   117
 #define M_9rx    32
 #define A_9tx0    5
 #define N_9tx0  147
 #define M_9tx0   40
 #define A_9tx1    6
 #define N_9tx1  147
 #define M_9tx1   40

//Frequency 10 (913.400MHz)
 #define A_10rx    10
 #define N_10rx   147
 #define M_10rx    40
 #define A_10tx0    8
 #define N_10tx0  136
 #define M_10tx0   37
 #define A_10tx1   27
 #define N_10tx1  117
 #define M_10tx1   32

//Frequency 11 (914.323MHz)
 #define A_11rx     9
 #define N_11rx   114
 #define M_11rx    31
 #define A_11tx0   22
 #define N_11tx0  132
 #define M_11tx0   36
 #define A_11tx1   21
 #define N_11tx1  121
 #define M_11tx1   33
```

```
//Frequency 12 (915.250MHz)
 #define A_12rx     3
 #define N_12rx   118
 #define M_12rx    32
 #define A_12tx0   24
 #define N_12tx0  121
 #define M_12tx0   33
 #define A_12tx1   13
 #define N_12tx1  114
 #define M_12tx1   31

//Frequency 13 (916.200MHz)
 #define A_13rx    24
 #define N_13rx   147
 #define M_13rx    40
 #define A_13tx0    2
 #define N_13tx0  144
 #define M_13tx0   39
 #define A_13tx1    3
 #define N_13tx1  144
 #define M_13tx1   39

//Frequency 14 (917.091MHz)
 #define A_14rx     1
 #define N_14rx   122
 #define M_14rx    33
 #define A_14tx0   10
 #define N_14tx0  118
 #define M_14tx0   32
 #define A_14tx1   26
 #define N_14tx1  136
 #define M_14tx1   37

//Frequency 15 (918.000MHz)
 #define A_15rx    14
 #define N_15rx   118
 #define M_15rx    32
 #define A_15tx0   20
 #define N_15tx0  140
 #define M_15tx0   38
 #define A_15tx1   21
 #define N_15tx1  140
 #define M_15tx1   38

//Frequency 16 (918.933MHz)
 #define A_16rx     5
 #define N_16rx   111
 #define M_16rx    30
 #define A_16tx0   30
 #define N_16tx0  125
 #define M_16tx0   34
 #define A_16tx1    9
 #define N_16tx1  122
 #define M_16tx1   33

//Frequency 17 (919.704MHz)
 #define A_17rx     4
 #define N_17rx   100
 #define M_17rx    27
 #define A_17tx0   10
 #define N_17tx0  148
 #define M_17tx0   40
 #define A_17tx1   11
 #define N_17tx1  148
 #define M_17tx1   40

//Frequency 18 (920.750MHz)
 #define A_18rx    25
 #define N_18rx   118
 #define M_18rx    32
```

```
 #define A_18tx0   11
 #define N_18tx0  137
 #define M_18tx0   37
 #define A_18tx1   30
 #define N_18tx1  103
 #define M_18tx1   28

//Frequency 19 (921.655MHz)
 #define A_19rx    24
 #define N_19rx   107
 #define M_19rx    29
 #define A_19tx0   24
 #define N_19tx0  133
 #define M_19tx0   36
 #define A_19tx1   29
 #define N_19tx1  118
 #define M_19tx1   32

//Frequency 20 (922.571MHz)
 #define A_20rx     5
 #define N_20rx   104
 #define M_20rx    28
 #define A_20tx0    2
 #define N_20tx0  145
 #define M_20tx0   39
 #define A_20tx1   14
 #define N_20tx1  100
 #define M_20tx1   27

//Frequency 21 (923.500MHz)
 #define A_21rx     5
 #define N_21rx   119
 #define M_21rx    32
 #define A_21tx0   29
 #define N_21tx0  148
 #define M_21tx0   40
 #define A_21tx1   30
 #define N_21tx1  148
 #define M_21tx1   40

//Frequency 22 (924.414MHz)
 #define A_22rx     3
 #define N_22rx   108
 #define M_22rx    29
 #define A_22tx0   11
 #define N_22tx0  145
 #define M_22tx0   39
 #define A_22tx1   12
 #define N_22tx1  145
 #define M_22tx1   39

//Frequency 23 (925.333MHz)
 #define A_23rx     4
 #define N_23rx   123
 #define M_23rx    33
 #define A_23tx0    6
 #define N_23tx0  108
 #define M_23tx0   29
 #define A_23tx1   15
 #define N_23tx1  104
 #define M_23tx1   28

//Frequency 24 (926.250MHz)
 #define A_24rx    16
 #define N_24rx   119
 #define M_24rx    32
 #define A_24tx0   20
 #define N_24tx0  145
 #define M_24tx0   39
 #define A_24tx1   31
 #define N_24tx1  126
```

```
 #define M_24tx1   34



// address 0x14
#define ChxxWrd0x14 (0xB5)


// address 0x15
#define FEEC_v 0
#define ChxxWrd0x15 ( FEEC_v)
```

```
//List of Procedures:

void RFPRG_Init_Rf( void);
void RFPRG_Enter_Rx( void);
void RFPRG_Enter_Tx( void);

void RFPRG_Clock_Complete( void);
void RFPRG_Clock_Desktop( uint8 reg0_value);
void RFPRG_Wait_Lock( void);

void RFPRG_Fill_Controlword( void);
void RFPRG_Fill_Dividers( void);

void RFPRG_Rf_Write( uint8 write_address, uint8 nr_of_bytes);
void RFPRG_Write_Byte( static uint8 byte_to_write);
void RFPRG_Inc_FreqIndex( void);



//List of Global Variables used, all are "uint8" ("unsigned char"):
#pragma udata access main_regs_access
near uint8
    RFPRG_Frequency,        //= Frequency number to use (0…24)
    RFPRG_FreqIndex,        //= Index to use in RFPRG_FreqTable[]
    RFPRG_FreqTable[25],    //= Randomly ordered list of frequencies
    RFPRG_Enter_Tx_Flag,    //= 1 if enter tx, =0 if enter rx
    RFPRG_Prog_Buffer[23];  // = Control word to fill into the MICRF50x


#pragma udata
#pragma code
```

```
//In this section, constants are stored in program memory
//Test that this implementation is OK for your MCU!
rom const uint8 RFPRG_A_Values_Rx[25]  =
   {A_0rx,  A_1rx,  A_2rx,  A_3rx,  A_4rx,  A_5rx,  A_6rx,  A_7rx,  A_8rx,  A_9rx,
    A_10rx, A_11rx, A_12rx, A_13rx, A_14rx, A_15rx, A_16rx, A_17rx, A_18rx, A_19rx,
    A_20rx, A_21rx, A_22rx, A_23rx, A_24rx};

rom const uint8 RFPRG_A_Values_Tx0[25]  =
   {A_0tx0,  A_1tx0,  A_2tx0,  A_3tx0,  A_4tx0,  A_5tx0,  A_6tx0,  A_7tx0,  A_8tx0,  A_9tx0,
    A_10tx0, A_11tx0, A_12tx0, A_13tx0, A_14tx0, A_15tx0, A_16tx0, A_17tx0, A_18tx0, A_19tx0,
    A_20tx0, A_21tx0, A_22tx0, A_23tx0, A_24tx0};

rom const uint8 RFPRG_A_Values_Tx1[25]  =
   {A_0tx1,  A_1tx1,  A_2tx1,  A_3tx1,  A_4tx1,  A_5tx1,  A_6tx1,  A_7tx1,  A_8tx1,  A_9tx1,
   A_10tx1, A_11tx1, A_12tx1, A_13tx1, A_14tx1, A_15tx1, A_16tx1, A_17tx1, A_18tx1, A_19tx1,
   A_20tx1, A_21tx1, A_22tx1, A_23tx1, A_24tx1};


//assuming all N values <=255 (1 byte/value is enough):
rom const uint8 RFPRG_N_Values_Rx[25] =
   {N_0rx,  N_1rx,  N_2rx,  N_3rx,  N_4rx,  N_5rx,  N_6rx,  N_7rx,  N_8rx,  N_9rx,
    N_10rx, N_11rx, N_12rx, N_13rx, N_14rx, N_15rx, N_16rx, N_17rx, N_18rx, N_19rx,
    N_20rx, N_21rx, N_22rx, N_23rx, N_24rx};

rom const uint16 RFPRG_N_Values_Tx0[25]  =
   {N_0tx0,  N_1tx0,  N_2tx0,  N_3tx0,  N_4tx0,  N_5tx0,  N_6tx0,  N_7tx0,  N_8tx0,  N_9tx0,
    N_10tx0, N_11tx0, N_12tx0, N_13tx0, N_14tx0, N_15tx0, N_16tx0, N_17tx0, N_18tx0, N_19tx0,
    N_20tx0, N_21tx0, N_22tx0, N_23tx0, N_24tx0};

rom const uint16 RFPRG_N_Values_Tx1[25]  =
   {N_0tx1,  N_1tx1,  N_2tx1,  N_3tx1,  N_4tx1,  N_5tx1,  N_6tx1,  N_7tx1,  N_8tx1,  N_9tx1,
   N_10tx1, N_11tx1, N_12tx1, N_13tx1, N_14tx1, N_15tx1, N_16tx1, N_17tx1, N_18tx1, N_19tx1,
   N_20tx1, N_21tx1, N_22tx1, N_23tx1, N_24tx1};



//assuming all M values <=255 (1 byte/value is enough):
rom const uint8 RFPRG_M_Values_Rx[25]  =
   {M_0rx,  M_1rx,  M_2rx,  M_3rx,  M_4rx,  M_5rx,  M_6rx,  M_7rx,  M_8rx,  M_9rx,
    M_10rx, M_11rx, M_12rx, M_13rx, M_14rx, M_15rx, M_16rx, M_17rx, M_18rx, M_19rx,
    M_20rx, M_21rx, M_22rx, M_23rx, M_24rx};

rom const uint8 RFPRG_M_Values_Tx0[25]  =
   {M_0tx0,  M_1tx0,  M_2tx0,  M_3tx0,  M_4tx0,  M_5tx0,  M_6tx0,  M_7tx0,  M_8tx0,  M_9tx0,
    M_10tx0, M_11tx0, M_12tx0, M_13tx0, M_14tx0, M_15tx0, M_16tx0, M_17tx0, M_18tx0, M_19tx0,
    M_20tx0, M_21tx0, M_22tx0, M_23tx0, M_24tx0};

rom const uint8 RFPRG_M_Values_Tx1[25]  =
   {M_0tx1,  M_1tx1,  M_2tx1,  M_3tx1,  M_4tx1,  M_5tx1,  M_6tx1,  M_7tx1,  M_8tx1,  M_9tx1,
    M_10tx1, M_11tx1, M_12tx1, M_13tx1, M_14tx1, M_15tx1, M_16tx1, M_17tx1, M_18tx1, M_19tx1,
    M_20tx1, M_21tx1, M_22tx1, M_23tx1, M_24tx1};
```

```
//Purpose: Fill up FreqTable with a randomly ordered list of frequency numbers.
//Pre: Nothing
//Post:RFPRG_FreqTable filled
//Notes: Users may select a different pattern.
//            A more advanced procedure may generate a unique pattern based on some input
//            If the application cant' spend data bytes on this, then the table should be
//            implemented in program memory
//            Sequence: 7,16,8,15,9,4,1,0,2,10,3,12,17,24,20,13,6,5,14,19,22,18,23,21,11
void
RFPRG_Fill_FreqTable( void)
{
    RFPRG_FreqTable[0] = 7;
    RFPRG_FreqTable[1] = 16;
    RFPRG_FreqTable[2] = 8;
    RFPRG_FreqTable[3] = 15;

    RFPRG_FreqTable[4] = 9;
    RFPRG_FreqTable[5] = 4;
    RFPRG_FreqTable[6] = 1;
    RFPRG_FreqTable[7] = 0;

    RFPRG_FreqTable[8] = 2;
    RFPRG_FreqTable[9] = 10;
    RFPRG_FreqTable[10] = 3;
    RFPRG_FreqTable[11] = 12;

    RFPRG_FreqTable[12] = 17;
    RFPRG_FreqTable[13] = 24;
    RFPRG_FreqTable[14] = 20;
    RFPRG_FreqTable[15] = 13;

    RFPRG_FreqTable[16] = 6;
    RFPRG_FreqTable[17] = 5;
    RFPRG_FreqTable[18] = 14;
    RFPRG_FreqTable[19] = 19;

    RFPRG_FreqTable[20] = 22;
    RFPRG_FreqTable[21] = 18;
    RFPRG_FreqTable[22] = 23;
    RFPRG_FreqTable[23] = 21;

    RFPRG_FreqTable[24] = 11;
}



void
RFPRG_Init_Rf( void)
{
    // Initiate the global variables used
    RFPRG_Fill_FreqTable();
    RFPRG_FreqIndex=0;
    RFPRG_Frequency = RFPRG_FreqTable[ RFPRG_FreqIndex];


    // Include a power-on-reset delay (may be skipped)
    GEN_Delay3p5msec();  // a GENeral procedure


    // Program the complete control word: use "TX with PA off"
    // The procedure waits for LD – this method makes sure that the initial time
    // from power-down to active mode is included before continuing
    RFPRG_Enter_Tx_Flag = 1;
    RFPRG_Fill_Controlword();
    RFPRG_Clock_Complete();
    RFPRG_Wait_Lock();      // Wait for lock "TX with PA off" (or timeout)
}
```

```
//Purpose: Enter receive-mode, not to be used for the 1st programming
//Pre: RFPRG_FreqIndex correct
//Post: Receive mode entered
//Note: Returns without waiting for LD
//Description: Enter the desktop-register, then the frequency dividers (A0,N0,M0 only)
void
RFPRG_Enter_Rx( void)
{
    RFPRG_Enter_Tx_Flag = 0;

    // First, make sure mode = rx
    RFPRG_Clock_Desktop( ChxxWrd0x00r);

    //Then, make sure the correct freq dividers are entered
    RFPRG_Fill_Dividers();  // Fill up the freq dividers
    RF_CS = 1;
    RFPRG_Rf_Write( 10,5);  // Start writing from address 0x0A, write 5 bytes
    RF_CS = 0;
}



//Purpose: Enter transmit-mode, not to be used for the 1st programming
//Pre: RFPRG_FreqIndex correct
//Post: Transmit mode entered
//Note: Returns after waiting for LD
//Description: Enter the desktop-register with PA= off,
//             then the frequency dividers (A0,N0,M0 and A1,N1,M1),
//             and finally desktop reg again, now with PA=on
void
RFPRG_Enter_Tx( void)
{
    RFPRG_Enter_Tx_Flag = 1;

    // First, make sure mode = tx with PA off
    RFPRG_Clock_Desktop( ChxxWrd0x00t);

    //Then, find and clock in the freq dividers
    RFPRG_Fill_Dividers();
    RF_CS = 1;
    RFPRG_Rf_Write( 10,10);  // Start writing from address 0x0A, write 10 bytes
    RF_CS = 0;

    //Finally, wait for lock on TX with PA off, enter tx with PA on and wait for lock
    RF_CS = 1;
    RFPRG_Prog_Buffer[ 0] = RFPRG_TX_PA_ON ; // PA on-setting
    RFPRG_Rf_Write( 0,1);   // Start writing from address 0, write 1 byte of the control word
    RFPRG_Wait_Lock();      // Wait for lock "TX with PA off" (or timeout)
    RF_CS = 0;              // Then generate a new loadpulse

    RFPRG_Wait_Lock();      // And wait for lock "TX with PA on" (or timeout)
}



//Purpose: Bring CS high, Clock in 22 bytes (reg0-reg21), bring Clock low
//Pre: RFPRG_Prog_Buffer[ 0 - 21]  filled
//Post: MICRF50x programmed
void
RFPRG_Clock_Complete( void)
{
    RF_CS = 1;
    RFPRG_Rf_Write( 0,22);  // Start writing from address 0, write 22 bytes of the controlword
    RF_CS = 0;
}
```

```
//Purpose: Bring CS high, Clock in 1 byte (reg0), bring Clock low
//Input: Byte to clock into desktop
//Post: MICRF50x programmed, RFPRG_Prog_Buffer[ 0] changed
void
RFPRG_Clock_Desktop( uint8 reg0_value)
{
    RFPRG_Prog_Buffer[ 0] = reg0_value;
    RF_CS = 1;
    RFPRG_Rf_Write( 0,1);  // Start writing from address 0, write 1 byte of the controlword
    RF_CS = 0;
}



//Pre: RFPRG_FreqIndex and RFPRG_Enter_Tx_Flag updated
//Post: The 5 (if rx) or 10 (if tx) first locations in RFPRG_Prog_Buffer
//      filled with freq dividers
void
RFPRG_Fill_Dividers( void)
{

    RFPRG_Frequency = RFPRG_FreqTable[RFPRG_FreqIndex];

    if( RFPRG_Enter_Tx_Flag)
    {
        RFPRG_Prog_Buffer[ 0] = RFPRG_A_Values_Tx0[ RFPRG_Frequency];
        RFPRG_Prog_Buffer[ 1] = 0;
        RFPRG_Prog_Buffer[ 2] = RFPRG_N_Values_Tx0[ RFPRG_Frequency];
        RFPRG_Prog_Buffer[ 3] = 0;
        RFPRG_Prog_Buffer[ 4] = RFPRG_M_Values_Tx0[ RFPRG_Frequency];

        RFPRG_Prog_Buffer[ 5] = RFPRG_A_Values_Tx1[ RFPRG_Frequency];
        RFPRG_Prog_Buffer[ 6] = 0;
        RFPRG_Prog_Buffer[ 7] = RFPRG_N_Values_Tx1[ RFPRG_Frequency];
        RFPRG_Prog_Buffer[ 8] = 0;
        RFPRG_Prog_Buffer[ 9] = RFPRG_M_Values_Tx1[ RFPRG_Frequency];
    }
    else
    {
        RFPRG_Prog_Buffer[ 0] = RFPRG_A_Values_Rx[ RFPRG_Frequency];
        RFPRG_Prog_Buffer[ 1] = 0;
        RFPRG_Prog_Buffer[ 2] = RFPRG_N_Values_Rx[ RFPRG_Frequency];
        RFPRG_Prog_Buffer[ 3] = 0;
        RFPRG_Prog_Buffer[ 4] = RFPRG_M_Values_Rx[ RFPRG_Frequency];
    }
}
```

```
// Purpose: Wait for lock
// Pre: Control word entered, CS brought low
// Post: Return from procedure when
//    1) LD going from low to high, or
//    2) LD "stuck" high for xusec or
//    3) LD "stuck" low for y msec
//       in the last case, RFchip is not in lock
// Note: If going from power-down or standby into rx or tx,
//       then some additional timeout may be required
// Important: This procedure uses a built-in MCU module (TMR0)
//            Make changes for the selected MCU and the MCU
//            speed. The selected “timeouts” may be changed
//            for a particular application
void
RFPRG_Wait_Lock( void)
{
    uint8 in_lock,
          quit_lock;

    in_lock   = 0;
    quit_lock = 0;

    do
    {
        // Initiate tmr0 for 800 usec timeout ("Timeout1")
        // 11.0592MHz xtal =>
        // pre 1:16 => count 16*(256-118) = 16*138 times
        T0CON = 0;
        TMR0L = 118;

        INTCONbits.TMR0IE = 0; // Disable interrupt
        INTCONbits.TMR0IF = 0; // Clear interrupt-flag
        T0CON = 0b11010011;    // TMR0 on, prescaler 1:16

        // If LD "stuck" on until Timeout1,
        // or LD detected low: quit
        while( ( not in_lock) and ( RF_LD))
        {
            if( INTCONbits.TMR0IF) // Test if Timeout1 reached
            {
                // LD "stuck" ON until Timeout1, LD is valid
                in_lock = 0xFF;
            }
        }

        if( not RF_LD) // LD detected low
        {
            // Initiate tmr0 for 8 msec (approx) timeout
            // ("Timeout2") 11.0592MHz xtal =>
            // pre 1:256 => count 256*(256-170) times
            T0CON = 0; //Stop timer0
            TMR0L = 170; //
            INTCONbits.TMR0IE = 0; // Disable interrupt
            INTCONbits.TMR0IF = 0; // Clear interrupt-flag
            T0CON = 0b11010111;    // TMR0 on, prescaler 1:256

            // If LD "stuck" OFF until Timeout2,
            // or LD going from OFF to ON: quit
            while( ( not quit_lock) and ( not RF_LD))
            {
                if( INTCONbits.TMR0IF) // test Timeout2
                {
                   // Lock detect is not found! Quit
                   quit_lock = 0xFF;
                }
             }

             // Return asap if LD detected low, then high
            if( RF_LD)
            {
              in_lock = 0xFF;
```

```
            }
        }
    } while( ( not in_lock) and ( not quit_lock));
    // wait until LD deteteced high for timeout1,
    //or low for timeout2

    // Finally, stop timer0
    T0CON = 0;
    TMR0L = 0;
    INTCONbits.TMR0IF = 0;
}



// Purpose: Get the control word to enter into MICRF50x
// Pre: RFPRG_FreqIndex and RFPRG_Enter_Tx_Flag updated
// Post: RFPRG_Prog_Buffer is filled
void
RFPRG_Fill_Controlword( void)
{

    //Fill reg10-reg19
    RFPRG_Fill_Dividers(); // This fills up the first 10 loc in Prog-buffer

    RFPRG_Prog_Buffer[ 10] = RFPRG_Prog_Buffer[ 0];
    RFPRG_Prog_Buffer[ 11] = RFPRG_Prog_Buffer[ 1];
    RFPRG_Prog_Buffer[ 12] = RFPRG_Prog_Buffer[ 2];
    RFPRG_Prog_Buffer[ 13] = RFPRG_Prog_Buffer[ 3];
    RFPRG_Prog_Buffer[ 14] = RFPRG_Prog_Buffer[ 4];

    RFPRG_Prog_Buffer[ 15] = RFPRG_Prog_Buffer[ 5];
    RFPRG_Prog_Buffer[ 16] = RFPRG_Prog_Buffer[ 6];
    RFPRG_Prog_Buffer[ 17] = RFPRG_Prog_Buffer[ 7];
    RFPRG_Prog_Buffer[ 18] = RFPRG_Prog_Buffer[ 8];
    RFPRG_Prog_Buffer[ 19] = RFPRG_Prog_Buffer[ 9];


    //Fill reg0, different for rx and tx mode (note: “tx with PA off”)
    if( RFPRG_Enter_Tx_Flag)
    {
        RFPRG_Prog_Buffer[ 0] = ChxxWrd0x00t;
    }
    else
    {
        RFPRG_Prog_Buffer[ 0] = ChxxWrd0x00r;
    }

    //Fill reg1-reg9, equal for rx and tx
    RFPRG_Prog_Buffer[ 1] = ChxxWrd0x01;
    RFPRG_Prog_Buffer[ 2] = ChxxWrd0x02;
    RFPRG_Prog_Buffer[ 3]=  ChxxWrd0x03;
    RFPRG_Prog_Buffer[ 4] = ChxxWrd0x04;

    RFPRG_Prog_Buffer[ 5] = ChxxWrd0x05;
    RFPRG_Prog_Buffer[ 6] = ChxxWrd0x06;
    RFPRG_Prog_Buffer[ 7] = ChxxWrd0x07;
    RFPRG_Prog_Buffer[ 8] = ChxxWrd0x08;
    RFPRG_Prog_Buffer[ 9] = ChxxWrd0x09;

    //Fill reg20-reg21, equal for rx and tx
    RFPRG_Prog_Buffer[ 20] = ChxxWrd0x14;
    RFPRG_Prog_Buffer[ 21] = ChxxWrd0x15;
}
```

```
// Pre: RFPRG_Prog_Buffer holds bytes to write
// Input: write_address = address of 1st byte to write, nr_of_bytes = number of bytes to write
void
RFPRG_Rf_Write( uint8 write_address, uint8 nr_of_bytes)
{
    //made static to increase speed and decrease program memory
    static uint8
        i,
        j;

    RFPRG_Write_Byte( (write_address<<1)+0); //address is 7 msb, r/w-bit(lsb) = 0 for write

    j = nr_of_bytes;
    i = 0;

    while( j)
    {
        RFPRG_Write_Byte( RFPRG_Prog_Buffer[ i]);
        i++;
        j--;
    }
}



//Based on PIC18F micro controller:
//   Program Memory: 15 instructions
//   Data Memory: 2 bytes
//   Execution time:
//      100 instruction cycles
void
RFPRG_Write_Byte( static uint8 byte_to_write)
{
    static uint8 temp;

    temp = 0x80;

    do
    {
        RF_IO_OUT = 0;
        if( byte_to_write bit_and temp)
        {
            RF_IO_OUT = 1;
        }
        RF_SCLK = 1;
        temp  >>= 1;
        RF_SCLK = 0;
    }
    while( temp);
}



void
RFPRG_Inc_FreqIndex( void)
{
    RFPRG_FreqIndex++;
    if( RFPRG_FreqIndex > (uint8)24)
    {
        RFPRG_FreqIndex = 0;
    }
}
```

MICREL, INC. 2180 FORTUNE DRIVE SAN JOSE, CA 95131 USA
TEL +1 (408) 944-0800 FAX +1 (408) 474-1000 WEB http:/www.micrel.com

The information furnished by Micrel in this data sheet is believed to be accurate and reliable. However, no responsibility is assumed by Micrel for its use. Micrel reserves the right to change circuitry and specifications at any time without notification to the customer.

Micrel Products are not designed or authorized for use as components in life support appliances, devices or systems where malfunction of a product can reasonably be expected to result in personal injury. Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body or (b) support or sustain life, and whose failure to perform can be reasonably expected to result in a significant injury to the user. A Purchaser's use or sale of Micrel Products for use in life support appliances, devices or systems is a Purchaser's own risk and Purchaser agrees to fully indemnify Micrel for any damages resulting from such use or sale.

© 2006 Micrel, Incorporated.