

EWD-HDTC User's Manual Rev 1.0



207 Industrial Blvd
Moore, OK 73170
405-794-7730

© 2002-2003 Radiotronics Inc, all rights reserved

Document Control

Created By	Steve Montgomery	11/2/02
Engineering Review		
Marketing Review		
Approved - Engineering		
Approved - Marketing		

Revision History

Revision	Author	Date	Description
1.0	SJM	11/2/02	Document Created
		11/22/02	Prerelease
1.0.1	TRM	12/27/2002	Update for initial release of APIs
1.0.2	TRM	2/5/2003	Correct document errors, function prototypes

Table of Contents

1.	Features and specifications	1
2.	Micro-controller interface	1
2.1.	Programming interface	1
2.2.	Data interface	3
2.3.	PROM interface	4
2.4.	Micro-controller interface circuit	4
2.5.	Reduced pin-count micro-controller interface circuit	5
3.	Application Programming Interface	5
3.1.	RFAPI	6
3.2.	uCAPI	23
4.	Frequency Programming	30
5.	Output Power Programming	31
6.	Calibration	32
7.	Timing recovery and data detection	33
8.	Sending and receiving data	35
9.	Low power operation	37
10.	RSSI output	37
11.	Printed circuit board layout	38
12.	Optional LC filter for improved harmonic performance	39
	Appendix A: Configuration Registers	40
	Appendix B: Channel Tables	50

Table of Figures and Tables

Figure 1: Write timing for programming interface	2
Figure 2: Programming interface read operation.....	2
Figure 3: Programming interface timing specifications.....	3
Figure 4: I2C Bus access waveforms	4
Figure 5: I2C random read waveforms	4
Figure 6: Micro-controller interface.....	5
Figure 7: Reduced pin-count micro-controller interface	5
Figure 8: API programming model.....	6
Table 1: RFAPI function synopsis	7
Table 2: rfDataMode parameter constants.....	9
Table 3: uCAPI function synopsis.....	23
Table 4: Hardware UART baud rate symbolic constants	24
Figure 9: Output power programming information.....	32
Figure 10: Automatic locking of averaging filter	34
Figure 11: Manual locking of averaging filter.....	35
Figure 12: Free-running averaging filter	35
Figure 13: Simple packet for Manchester mode.....	36
Figure 14: Simple packet for UART mode.....	36
Figure 15: Manchester data timing	36
Figure 16: UART data timing	36
Figure 17: Manchester encoding detail	37
Figure 18: RSSI response	38
Figure 19: RA.D. kit PCB - bottom layer.....	38
Figure 20: RA.D. kit PCB - top layer.....	39
Figure 21: Optional filter for improved harmonic performance.....	39
Table 6: 915MHz Legacy Channel Table	50
Table 7: Enhanced 915MHz 50-Channel Table	51
Table 8: Expanded 915MHz 197Channel Table	53
Table 9: Original 868MHz Channel Table	54

Function Reference

3.1.1.	rfAverageAutoLock.....	7
3.1.2.	rfAverageFreeRun.....	7
3.1.3.	rfAverageManualLock	8
3.1.4.	rfCalibrate.....	8
3.1.5.	rfCalibrateEx.....	8
3.1.6.	rfConfig.....	8
3.1.7.	rfDataMode.....	9
3.1.8.	rfGetChar.....	10
3.1.9.	rfGetFreqCal25C.....	10
3.1.10.	rfGetMAC	11
3.1.11.	rfGetMatch.....	11
3.1.12.	rfGetPacket	11
3.1.13.	rfGetString	12
3.1.14.	rfOverrideCurrentCalibration.....	12
3.1.15.	rfPutChar	12
3.1.16.	rfPutPacket.....	13
3.1.17.	rfPutString	13
3.1.18.	rfReadCurrentCalibration	13
3.1.19.	rfReadFromRegister.....	14
3.1.20.	rfReadRom	14
3.1.21.	rfReadRSSI	14
3.1.22.	rfReceivePacketsAvail	14
3.1.23.	rfReceivingPacket	15
3.1.24.	rfReset.....	15
3.1.25.	rfRxMode.....	15
3.1.26.	rfRxModeEx.....	15
3.1.27.	rfSetChannel	16
3.1.28.	rfSetDeviation.....	16
3.1.29.	rfSetFreqA.....	16
3.1.30.	rfSetFreqB	17
3.1.31.	rfSetMinimumFsep	17
3.1.32.	rfSetRefDiv	17
3.1.33.	rfSetTxPO.....	17
3.1.34.	rfSetupInterruptLockFlags.....	18
3.1.35.	rfSleep	18
3.1.36.	rfStopOverridingCalibration.....	18

3.1.37.	rfSwitchToRx	18
3.1.38.	rfSwitchToTx	18
3.1.39.	rfTransmitPacketsWaiting	19
3.1.40.	rfTransmittingPacket	19
3.1.41.	rfTxMode	19
3.1.42.	rfTxModeEx	19
3.1.43.	rfUpdateLockFilter	20
3.1.44.	rfWaitForChar	20
3.1.45.	rfWaitForString	21
3.1.46.	rfWakeToRx	21
3.1.47.	rfWakeToRxEx	21
3.1.48.	rfWakeToTx	21
3.1.49.	rfWakeToTxEx	22
3.1.50.	rfWriteToRegister	22
3.1.51.	rfWriteToRegisterWord	22
3.2.1.	delay_cycles	23
3.2.2.	ucComBaud	23
3.2.3.	ucComInit	24
3.2.4.	ucConfig	24
3.2.5.	ucDelayTicks	24
3.2.6.	ucGetAnalog	24
3.2.7.	ucGetChar	25
3.2.8.	ucGetPacketRF	25
3.2.9.	ucGetString	25
3.2.10.	ucPutChar	25
3.2.11.	ucPutPacketRF	26
3.2.12.	ucPutString	26
3.2.13.	ucKbhit	26
3.2.14.	ucReadRFDataBit	26
3.2.15.	ucReadRFDataByte	26
3.2.16.	ucReadRFProgByte	27
3.2.17.	ucReadSMBus	27
3.2.18.	ucReceivingRF	27
3.2.19.	ucReceivePacketsAvailRF	27
3.2.20.	ucSetOscFreq	27
3.2.21.	ucSetupForRxTx	28
3.2.22.	ucSetupInterruptLockFlagsRF	28
3.2.23.	ucStartAnalog	28

3.2.24.	ucTransmitPacketsWaitingRF	28
3.2.25.	ucTransmittingRF	28
3.2.26.	ucTriState	29
3.2.27.	ucWriteRFDataBit.....	29
3.2.28.	ucWriteRFDataByte.....	29
3.2.29.	ucWriteRFProgAddr	29
3.2.30.	ucWriteRFProgByte	29

1. Features and specifications

- 315,418/433, and 868/915 MHz versions available
- Half-duplex transmission and reception with on-board antenna switch
- Fast settling phased-locked-loop suitable for frequency hopping spread spectrum
- Programmable output power from –20 to +5dBm (+10dBm for 433 MHz version)
- Programmable data rates from 0.6 to 76.8 kbit/second
- On-board data encoding and decoding for Manchester or transparent UART operation
- Simple 3-wire serial programming interface
- Simple 2-wire data IO interface
- On-board PROM non-volatile memory stores calibration information and a unique, factory programmed 48-bit ID

2. Micro-controller interface

The EWD-HDTC (Embedded Wireless Data module – Half-Duplex Transceiver) provides a very simple interface for any micro-controller.

2.1. Programming interface

Programming is accomplished via a 3-wire serial interface consisting of:

- PCLK (pin 11-input)– programming clock
- PDAT (pin 12-input/output) – programming data
- PALE (pin 13-input) – programming latch enable

Figure 1 shows the typical waveforms for writing data to a register and Figure 2 shows the typical waveforms for reading data from a register.

There are 36 individual configuration registers that control the operations of the module, each one addressed by a 7-bit address. A write or read cycle begins by bringing PALE low. The address is then shifted into the module via the PDAT pin on the falling edge of PCLK. The bits are shifted MSB first. The LSB of the address byte determines whether the access is a read or a write operation. If the LSB is low, the access is a read. If the LSB is high, the access is a write.

If the operation is a read operation, PDAT pin switches to an output and the value of the addressed configuration register is shifted out MSB first on the falling edge of PCLK.

If the operation is a write operation, PDAT remains an input and the value of the data to be written to the addressed register is shifted in MSB first on the falling edge of PCLK.

Figure 3 shows the timing specifications for the programming interface.

Normally, the user should not have to directly program individual configuration registers. The RF-API routines provide a programmatic interface that isolates the user from the details of the configuration registers. However, to support situations where the user needs to access individual configuration registers, the RF-API exports functions to read and write configuration registers directly. For more information on the API, refer to Section 3.

The purpose of the API is to isolate the user from the details of the configuration registers and hardware. However, if the API needs to be ported to a processor that is not currently supported by Radiotronic, the information in Figures 1 thru 3 will be required.

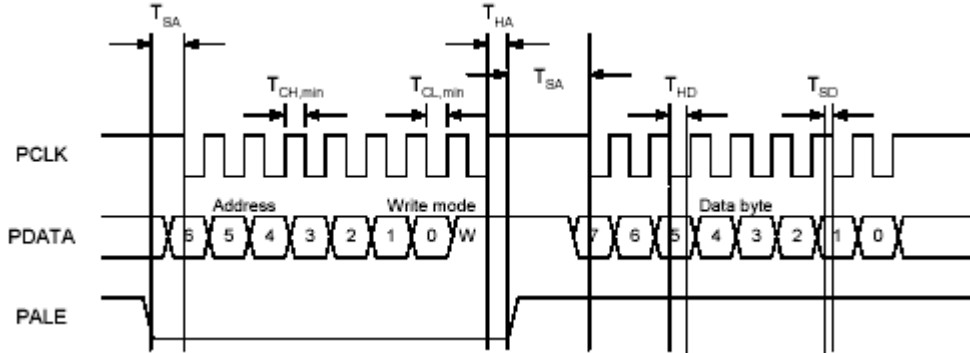


Figure 1: Write timing for programming interface

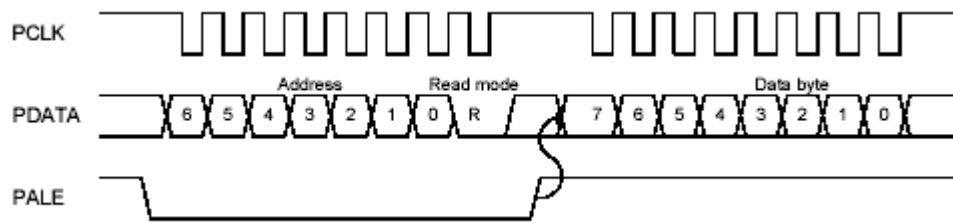


Figure 2: Programming interface read operation

Initial Release

Parameter	Symbol	Min	Max	Units	Conditions
PCLK, clock frequency	F_{CLOCK}	-	10	MHz	
PCLK low pulse duration	$T_{\text{CL, min}}$	50		ns	The minimum time PCLK must be low.
PCLK high pulse duration	$T_{\text{CH, min}}$	50		ns	The minimum time PCLK must be high.
PALE setup time	T_{SA}	10	-	ns	The minimum time PALE must be low before negative edge of PCLK.
PALE hold time	T_{HA}	10	-	ns	The minimum time PALE must be held low after the positive edge of PCLK.
PDATA setup time	T_{SD}	10	-	ns	The minimum time data on PDATA must be ready before the negative edge of PCLK.
PDATA hold time	T_{HD}	10	-	ns	The minimum time data must be held at PDATA, after the negative edge of PCLK.
Rise time	T_{rise}		100	ns	The maximum rise time for PCLK and PALE
Fall time	T_{fall}		100	ns	The maximum fall time for PCLK and PALE

Note: The set-up- and hold-times refer to 50% of VDD.

Figure 3: Programming interface timing specifications

2.2. Data interface

Data transmission and reception is accomplished via a 2-wire serial interface consisting of:

- DCLK (pin 10-i/o)
- DIO (pin 9-i/o)

The functions of the DCLK and DIO pins depend on the data mode.

The EWD-HDTC module supports three data modes: UART, Synchronous NRZ, and Manchester.

In UART mode, the transceiver is transparent and requires no special data formatting. In this mode, data is transmitted on DIO and data is received on DCLK. Data can be transmitted at rates from 0.6 to 76.8 kbits/second.

In Manchester mode, the transceiver encodes and decodes the data stream using Manchester encoding rules. Data is sent to or received from the transceiver on DIO. The data is sampled on the edge of DCLK, which is automatically generated by the module. Effective data rates of 0.3 to 38.4 kbits/second are supported in this mode.

The transceiver neither encodes nor decodes data in synchronous NRZ mode, however clock recovery is performed. In this mode, a preamble MUST be sent, as with the other modes. Additionally, the programmer must lock the average filter manually upon the detection of the preamble. Data is sent to or received from the transceiver on DIO. The data is sampled on the edge of DCLK, which is automatically generated by the module. Data rates of 0.6 to 78.6 kbits/second are supported in this mode.

For more information on sending and receiving data with the transceiver, please refer to section 9 "Sending and receiving data".

2.3. PROM interface

A third serial interface connects to the on-board PROM. This is a standard I2C bus interface and requires external pull-up resistors to ensure proper performance.

- ECLK (pin 3-input)
- EDAT (pin 2-open collector)

The PROM stores 48-bit serial number and calibration information for the module. The RF-API provides functions to programmatically access the serial number and calibration information, isolating the user from I2C bus timing concerns.

Designer's Note

The PROM contents should only be read. Altering these contents with a write operation voids the warranty of the module and may cause improper performance.

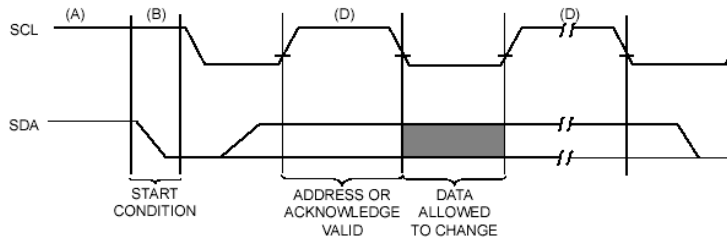


Figure 4: I2C Bus access waveforms

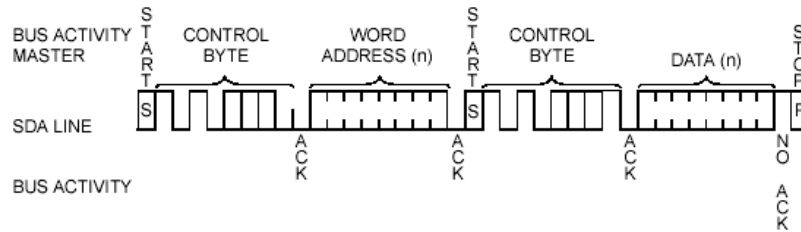


Figure 5: I2C random read waveforms

2.4. Micro-controller interface circuit

Initial Release

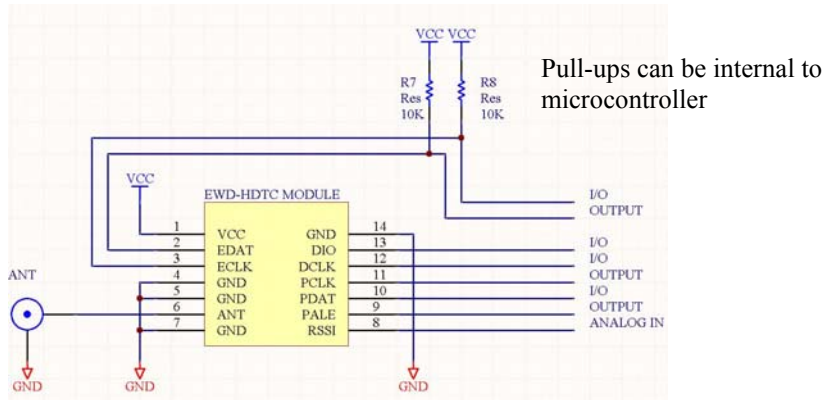


Figure 6: Micro-controller interface

2.5. Reduced pin-count micro-controller interface circuit

Figure 7 shows a reduced pin-count micro-controller interface. Using this circuit, only one interface can be active at a time. In other words, the circuit can access either the PROM or the transceiver, but not both at the same time.

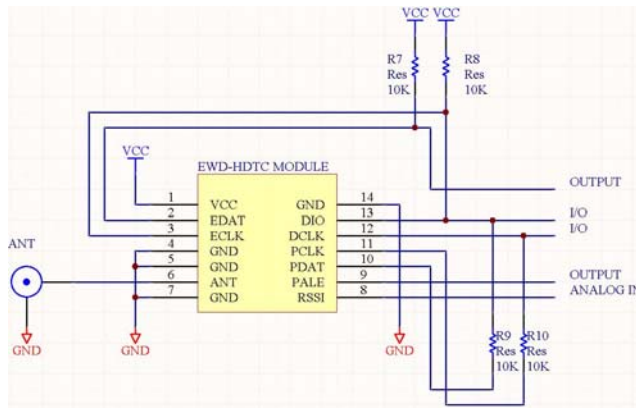


Figure 7: Reduced pin-count micro-controller interface

3. Application Programming Interface

Radiotronix has created an applications programming interface (API) to greatly simplify the effort required by the design engineer. This API contains routines, written in C, that configure and control all aspects of the transceiver module. The routines are written in ANSI C to ensure that they can be used with any micro-controller that supports ANSI C development tools.

The API has been written to be processor independent. No hardware specific operations occur in the API. Instead, the API functions make calls to external routines that are processor specific, which are

Attention

Radiotronix copyrights the RF-API. It can be used free of charge on Radiotronix modules. We do not allow its use for any other purpose.

contained in a separate API file. This creates a layered approach to software development that eases the programming burden.

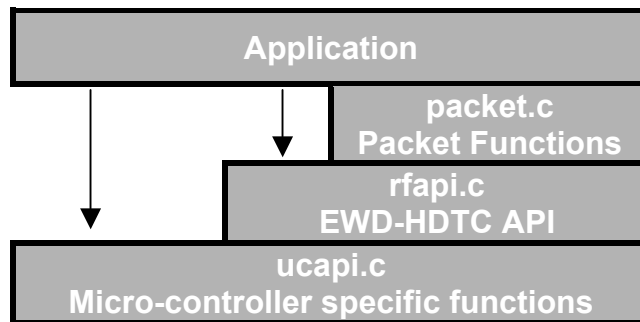


Figure 8: API programming model

3.1. RFAP I

The RFAP I exports functions that isolate the user from the underlying configuration details of the module. This layer of the programming model makes calls into the UCAP I, allowing the hardware-specific implementation to be easily ported to other micro-controllers. The RFAP I is designed to follow one of two models: polled and interrupt-driven. The interrupt-driven model is selected by default. To enable the polled model, comment out the “#define INTERRUPT_DRIVEN_RF” statement in the UCAP I.h header file. The following table details the functions available with each model.

Polled	Both		Interrupt Driven
rfGetChar	rfAverageAutoLock	rfSetDeviation	rfGetPacket
rfGetString	rfAverageFreeRun	rfSetFreqA	rfPutPacket
rfPutChar	rfAverageManualLock	rfSetFreqB	rfReceivePacketsAvail
rfPutString	rfCalibrate	rfSetMinimumFsep	rfReceivingPacket
rfWaitForChar	rfCalibrateEx	rfSetRefDiv	rfTransmitPacketsWaiting
rfWaitForString	rfConfig	rfSetTxPO	rfTransmittingPacket
	rfDataMode	rfSleep	rfUpdateLockFilter
	rfGetFreqCal25C	rfStopOverridingCalibration	rfSetupInterruptLockFlags
	rfGetMAC	rfSwitchToRx	
	rfGetMatch	rfSwitchToTx	
	rfOverrideCurrentCalibration		
	rfReadCurrentCalibration	rfTxModeEx	
	rfReadRom	rfWakeToRx	
	rfReadFromRegister	rfWakeToRxEx	
	rfReadRSSI	rfWakeToTx	
	rfReset	rfWakeToTxEx	
	rfRxMode	rfWriteToRegister	
	rfRxModeEx	rfWriteToRegisterWord	
	rfSetChannel		

Table 1: RF-API function synopsis

3.1.1. rfAverageAutoLock

```
void rfAverageAutoLock (void) ;
```

This function will set the averaging filter to automatically lock on detection of a valid preamble. For more information about the averaging filter, see section 8.

3.1.2. rfAverageFreeRun

```
void rfAverageFreeRun (void) ;
```

This function will set the averaging filter to free run. For more information about the averaging filter, see section 8.

Initial Release

3.1.3. rfAverageManualLock

```
void rfAverageManualLock(void);
```

This function manually locks the averaging filter. For more information about the averaging filter, see section 8.

3.1.4. rfCalibrate

```
unsigned INT8 rfCalibrate(void);
```

Calling this function will calibrate the module for the current temperature, and the A/B frequency settings. The function returns the contents of the LOCK register, indicating the current state of PLL lock.

3.1.5. rfCalibrateEx

```
INT8 rfCalibrateEx(unsigned INT8 rx_current, unsigned INT8 tx_current);
```

Calling this function will calibrate the module for the current temperature, the supplied current values, and the A/B frequency settings. The function returns the contents of the LOCK register, indicating the current state of PLL lock. You should only use this function if you are using custom current values for Tx and Rx.

3.1.6. rfConfig

```
void rfConfig(unsigned INT16 *configuration, INT8 count);
```

*Configuration contains an array of 16-bit words that define the configuration of the transceiver. Each word contains the address for a register and the data that should be programmed into that register. The address is stored in the most significant byte and the data is stored in the least significant byte.

Example:

```
// ** Configuration for EWD-HDTC **
//
// This data is stored in words in the following format-data:addr. A
// configuration tool will be available soon to automatically calculate
// the register values and create an array declaration that can be
// pasted into your source file.
//
//      addr   description           value      result
//      ----   -
//      0x00   Main                   0x51      receive mode
//      0x01   Freq 2a                 0x5C      RX Freq=915MHz
//      0x02   Freq 1a                 0xF0
//      0x03   Freq 0a                 0x38
//      0x04   Freq_2b                 0x5C      TX Freq=915MHz
//      0x05   Freq_1b                 0xF3
//      0x06   freq 0b                 0xCD
//      0x07   fsep1                   0x00      Deviation = 25 khz
//      0x08   fsep0                   0xA7
//      0x09   current                 0x88      Low-current receive
//      0x0A   front_end               0x3A      receive mode, max
//                                     sensitivity
```

```

//      0x0B  pa_pow          0x80          output power=0dBm
//      0x0C  pll             0x30          Ref Div of 6
//      0x0D  lock           0x10
//      0x0E  cal            0x26          disable calibration
//      0x0F  modem2        0x8E          peak detector set for
//                                     25kHz Fdev
//      0x10  modem1        0x6F          avg filter free-running
//      0x11  modem0        0x27          2.4kbit/sec Manchester
//      0x12  match         0xE2          common for EWD @ 915MHz
//      0x13  fsctrl        0x01          no spectral shaping
//      0x1C  prescale      0x00
short  RFInit[]={0x0051, 0x015C, 0x02F0, 0x0338, 0x045C, 0x05F3, 0x06CD,
                 0x0700, 0x08A7, 0x0988, 0x0A3A, 0x0B80, 0x0C30, 0x0D10,
                 0x0E26, 0x0F8E, 0x106F, 0x1127, 0x12E2, 0x1301, 0x1C00};

void func(void)
{
    ...
    rfConfigure(21,RFInit);
    ...
}

```

3.1.7. rfDataMode

```
void rfDataMode (INT8 datamode);
```

This function is used to set the data mode and speed of the transceiver module. The following table displays the symbolic constants for this function along with their meaning. For more information about the modes of data transfer, refer to Section 9.

Manchester Mode Constant	RF Baud Rate	NRZ Mode Constant	RF Baud Rate	UART Mode Constant	RF Baud Rate
dmMANCHESTER_300	300	dmNRZ_600	600	dmUART_600	600
dmMANCHESTER_600	600	dmNRZ_1200	1200	dmUART_1200	1200
dmMANCHESTER_1200	1200	dmNRZ_2400	2400	dmUART_2400	2400
dmMANCHESTER_2400	2400	dmNRZ_4800	4800	dmUART_4800	4800
dmMANCHESTER_4800	4800	dmNRZ_9600	9600	dmUART_9600	9600
dmMANCHESTER_9600	9600	dmNRZ_19200	19200	dmUART_19200	19200
dmMANCHESTER_19200	19200	dmNRZ_38400	38400	dmUART_38400	38400
dmMANCHESTER_38400	38400	dmNRZ_76800	76800	dmUART_76800	76800

Table 2: rfDataMode parameter constants

Example:

```

// Set the RF datalink to synchronous NRZ, 9600 symbols/sec
rfDataMode (dmNRZ_9600);

```

Initial Release

3.1.8. rfGetChar

```
char rfGetChar(void);
```

Polled Only. Once the preamble has been detected using `rfWaitForString` or `rfWaitForChar`, a call to `rfGetChar` will bring in the next 8 bits from the RF interface and assemble them into a byte. This function assumes that the data is sent least significant bit first. The RX section must be selected in order for this function to operate properly. This function is not compatible with asynchronous UART mode.

Example:

```

LONGBOOL preamble_found;
INT8 node num;
INT8 status;
INT8 event count;
INT8 sensor node value;

// Wait until a preamble is found
do
{
    preamble_found = rfWaitForString(RF_DETECT_PREAMBLE, RF_DETECT_PREAMBLE_SIZE);
}
while (!preamble_found);

// Preamble found. Grab next four characters as payload
node num = rfGetChar();
status = rfGetChar();
event count = rfGetChar();
sensor_node_value = rfGetChar();

```

3.1.9. rfGetFreqCal25C

```
INT16 rfGetFreqCal25C(INT8 band)
```

This function returns the 16-bit frequency calibration offset for 25 degrees C that is stored in PROM for each frequency band (fb915MHz, fb868MHz). This offset should ONLY be applied to transmitter tuning parameters. See Section 6 for more information about calibration.

Example:

```

INT16 fcal;
INT32 freq a = 0x00D90000;
INT32 freq b = 0x00D9072B;

fcal=rfGetFreqCal25C(fb915MHz);

// Manually set optimized channel 25, calibration factor applied to Tx ONLY
#ifdef HARD_WAY
rfSetFreqA(freq a);
rfSetFreqB(freq b - fcal);
#endif

// Automatically set channel to 25, applying offset to Tx.
#ifndef HARD_WAY
rfSetChannel(fb915MHz, 25, fcal);
#endif

rfCalibrate();

```

Initial Release

3.1.10. rfGetMAC

```
LONGBOOL rfGetMAC(unsigned INT32 *OUI, unsigned INT32 *sn);
```

This function returns the 48-bit unique identifier in its two parts: the OUI (Organizational Unique Identifier) and MAC (Media Access Control) address. The upper 8 bits of each part is not used. The function returns 1 if the numbers were successfully retrieved from the PROM-0 if an error was encountered during the I2C bus transfer.

Example:

```
if (rfGetMAC(&OUI, &MAC))
{
    sprintf(msg, "MAC Retrieved → %x:%6x", OUI, MAC);
    ucPutString(msg, length(msg));
}/*if*/
```

3.1.11. rfGetMatch

```
unsigned INT8 rfGetMatch(INT8 band);
```

Retrieves an unsigned, 8 bit match value for either the 868MHz or 915MHz band from the module's on-board ROM. This value should be programmed into the RF_MATCH register for optimum performance.

3.1.12. rfGetPacket

```
LONGBOOL rfGetPacket(char *buf, INT8 *num_chars);
```

Interrupt Driven Only. Gets a packet from the receive packet buffer and places it in the character buffer, *buf. If there are no packets available, the function will wait until there is at least one. The size of the buffer, *buf, MUST be provided in *num_chars at the time of the function call. The function returns 1 indicating that a packet was returned in *buf. If the function fails, the function returns 0. If the function is successful, the actual size of the packet is returned in *num_chars. This function is actually implemented as a macro in the RFAPI and makes a call directly to the UCAPI function, `ucGetPacketRF`. The RX section must be selected in order for this function to operate properly.

Example:

```
char buf[MAX_RF_PACKET_PAYLOAD_SIZE];
INT8 buf_size;

if (rfReceivePacketsAvail())
{
    buf_size = sizeof(buf);
    if (rfGetPacket(buf, buf_size))
    {
        // Put packet contents onto UART
        ucPutString(buf, buf_size);
    }/*if*/
}/*if*/
```

3.1.13. rfGetString

```
void rfGetString(char *buf, INT8 size);
```

Polled Only. Once the preamble has been detected using `rfWaitForString` or `rfWaitForChar`, a call to `rfGetString` will bring in the next `size` bytes from the RF interface and place them in the provided buffer, `buf`. This function assumes that the data is sent least significant bit to most significant bit of least significant byte to most significant byte. The RX section must be selected in order for this function to operate properly. This function is not compatible with asynchronous UART mode.

Example:

```
LONGBOOL preamble found;
enum {strNodeNum, strStatus, strEventCount, strSensorNodeValue, strEnd};
char message_payload[strEnd];

// Wait until a preamble is found
do
{
    preamble found = rfWaitForString(RF_DETECT_PREAMBLE, RF_DETECT_PREAMBLE_SIZE);
}
while (!preamble_found);

// Preamble found. Grab next four characters as payload
rfGetString(message_payload, sizeof(message_payload));
```

3.1.14. rfOverrideCurrentCalibration

```
void rfOverrideCurrentCalibration(INT8 val1, INT8 val2);
```

This function forces the module to use `val1` and `val2` for calibration. This is very useful in multiple-frequency applications where `val1` and `val2` represent the stored calibration data for a particular frequency and temperature.

3.1.15. rfPutChar

```
void rfPutChar(char out_byte);
```

Polled Only. Sends a single byte over the RF interface. The byte is sent least significant bit first. The TX section must be selected in order for this function to operate properly. This function is not compatible with asynchronous UART mode.

Example:

```
// Send preamble
rfPutString(RF_PREAMBLE, sizeof(RF_PREAMBLE));

// Preamble sent. Send next four characters as payload
rfPutChar(node_num);
rfPutChar(status);
rfPutChar(event_count);
rfPutChar(sensor_node_value);
```

Initial Release

3.1.16. rfPutPacket

```
LONGBOOL rfPutPacket(char *buf, INT8 num_chars);
```

Interrupt Driven Only. Puts a packet's payload into the transmit payload buffer, provided by the character buffer, *buf. The function returns 1 indicating that a packet was returned in *buf. If the function fails, the function returns 0. If the function is successful, the actual size of the packet is returned in *num_chars. This function is actually implemented as a macro in the RFAPL and makes a call directly to the UCAPL function, ucPutPacketRF. The TX section must be selected in order for this function to operate properly.

Example:

```
char buf[MAX_RF_PACKET_PAYLOAD_SIZE];

if ((!rfReceivePacketsAvail()) && (!rfReceivingPacket()))
{
    // Turn on transmitter
    rfTxMode(0xFF);

    sprintf(buf, "Hello\r\n");

    // If function fails, buffer is full. Keep trying until buffer is
    // empty
    while (!rfPutPacket(buf, strlen(buf)));

    // Put packet contents onto UART
    ucPutString(buf, strlen(buf));
}/*if*/
```

3.1.17. rfPutString

```
void rfPutString(char *out_buffer, unsigned INT8 len);
```

Polled Only. Sends a string of bytes over the RF interface. The string is sent least significant bit first, from index 0 to index len - 1. The TX section must be selected in order for this function to operate properly. This function is not compatible with asynchronous UART mode.

See rfPutChar for an example of usage.

3.1.18. rfReadCurrentCalibration

```
void rfReadCurrentCalibration(INT8 *val1, INT8 *val2);
```

This function returns the calibration values generated in the rfCalibrate function. This function is useful in multiple-frequency applications where all frequencies are calibrated and the calibration data is stored for each one.

Example:

```
INT8 cal_values[2][2];
main()
{
    rfReset();
    rfCalibrate();
    rfReadCurrentCalibration(&cal_values[0][0], &cal_values[0][1]);
    sleep(5000); // Allow device to warm up
    rfCalibrate();
}
```

Initial Release

```

rfReadCurrentCalibration(&cal_values[1][0], &cal_values[1][1]);
...
}

```

3.1.19. rfReadFromRegister

```
INT8 rfReadFromRegister(INT8 addr_byte);
```

This function returns the current value from a module register. See appendix A for a list of module configuration registers.

Example:

```

INT8 val;

val=rfReadFromRegister(RF_CAL);

if (val & 0x08)
{
    // calibration complete
}/*if*/

```

3.1.20. rfReadRom

```
LONGBOOL rfReadRom(unsigned INT8 addr_byte, INT8 *data_byte);
```

This function provides a way to read a single byte from the module's on-board ROM and returns it in *data_byte. Valid addresses for addr_byte range from 0 to 15. The function returns 1 if successful or 0 if a failure occurred. There are custom functions available to return the individual elements stored in the ROM. See `rfGetSN`, `rfGetFreqCa125C`, and `rfGetMatch`

3.1.21. rfReadRSSI

```
unsigned INT16 rfReadRSSI(void);
```

Returns a 16-bit A/D reading (resolution determined by CPU) of the receive signal strength indicator (RSSI).

3.1.22. rfReceivePacketsAvail

```
INT8 rfReceivePacketsAvail (void);
```

Interrupt Driven Only. This function returns the number of packets waiting in the packet receive buffer. If the receiver is not active, the function will return 0. This function is actually implemented as a macro in the RF API and makes a call directly to the UC API function, `ucReceivePacketsAvailRF`.

See `rfReceivingPacket` for example

Initial Release

3.1.23. rfReceivingPacket

```
LONGBOOL rfReceivingPacket(void);
```

Interrupt Driven Only. Checks to see if a packet is in the process of being received. The function returns 1 indicating that a packet receive is in progress: 0 if not. A packet is considered to be actively receiving when a preamble and packet header has been found, and the packet engine is actively retrieving packet payload. This function is actually implemented as a macro in the RF API and makes a call directly to the UC API function, **ucReceivingRF**.

Example:

```
char buf[MAX_RF_PACKET_PAYLOAD_SIZE];

if ((!rfReceivePacketsAvail()) && (!rfReceivingPacket()))
{
    // Turn on transmitter
    rfTxMode(0xFF);

    sprintf(buf, "Hello\r\n");

    // If function fails, buffer is full. Keep trying until buffer is
    // empty
    while (!rfPutPacket(buf, strlen(buf)));

    // Put packet contents onto UART
    ucPutString(buf, strlen(buf));
}/*if*/
```

3.1.24. rfReset

```
void rfReset(void);
```

Calling this function will reset the module. The module will be in sleep mode with frequency registers set to optimized channel 25 (see **rfSetChannel** for explanation of channels) and a deviation of 32kHz (Fsep = 64kHz). Manchester mode at 1.2kbps will be selected.

3.1.25. rfRxMode

```
void rfRxMode(void);
```

Activates the receive section and switches to RX state using predefined values for module. This function assumes that the Rx frequency value is stored in freq register A. It also assumes that the oscillator core, frequency synthesizer, and bias generator are already operating. This function is separated from **rfWakeToRx** because waking requires additional delays. To wake the module to the RX mode from sleep, use **rfWakeToRx**.

3.1.26. rfRxModeEx

```
void rfRxModeEx(INT8 current, INT8 pll, INT8 freq_register);
```

This function accomplishes the same task as **rfRxMode**, but allows the programmer to specify custom values for the CURRENT and PLL registers. It also allows the specification of either the A or B frequency registers for receiver tuning. This function is separated from

rfWakeToRxEx because waking requires additional delays. To wake the module to the Rx mode from sleep, use **rfWakeToRxEx** instead.

Example:

```
// Selects Rx Mode, but uses "low power" current settings. It is assumed that
// the current FREQA registers are populated with optimal channel 25 values.
rfRxModeEx(0x88, rf_915_optimal_channel_array[24][optRefDiv] << 3, fsFREQA);
```

3.1.27. rfSetChannel

```
LONGBOOL rfSetChannel(INT8 band, INT8 channel, INT16 offset);
```

This function sets the module up for communications using the provided channel for a given frequency band (fb915MHz, fb868MHz). Offset provides a way of "fine tuning" the center frequency. When selecting a channel, the frequency registers, reldiv setting, and frequency separation values are all updated. There are 49 optimized channels for the 902-928MHz band and 34 optimized channels for the 868-870MHz band. It should be noted that the 868MHz channel table contains channels that overlap, as well as channels that provide negated output from the chipset. These channel tables were taken from the chipset manufacturer. Appendix B details the channel tables.

Example:

```
INT16 fcal;

fcal=rfGetFreqCal25C(fb915MHz);

// Automatically set channel to 25, applying offset to Tx.
rfSetChannel(fb915MHz, 25, fcal);
rfCalibrate();
```

3.1.28. rfSetDeviation

```
void rfSetDeviation(INT16 deviation)
```

This function sets the peak-peak frequency deviation during transmit mode. Normally, the frequency deviation is set in the **rfSetChannel** and **rfSetChannelEx** functions. The user can override this setting by using this function. *deviation* is an 11 bit number that determines the actual frequency separation used to transmit 1's and 0's. See Section 4 for more information on frequency and deviation programming.

3.1.29. rfSetFreqA

```
void rfSetFreqA(INT32 freq);
```

This function is used to set the frequency A register. The lower 24-bits of *freq* determine the actual frequency. The calibration factor stored in the PROM should be applied to *freq* to ensure the accuracy of the frequency setting. This calibration factor should ONLY be applied to the *freq* word if *FREQ_A* is to be used as a transmitter tuning register. The calibration factor will actually degrade sensitivity if it is applied to receiver tuning parameters. See Section 4 for more information on frequency and deviation programming.

Example:

Initial Release

```

INT8 fcal;
INT32 freq_a = 0x00D90000;
INT32 freq_b = 0x00D9072B;

fcal=rfGetFreqCal25C();

// Manually set optimized channel 25, calibration factor applied to Tx ONLY
rfSetFreqA(freq_a);
rfSetFreqB(freq_b - fcal);
rfCalibrate();

```

3.1.30. rfSetFreqB

```
void rfSetFreqB(INT32 freq);
```

This function is used to set the frequency B register. The lower 24-bits of `freq` determine the actual frequency. The calibration factor stored in the PROM should be applied to `freq` to ensure the accuracy of the frequency setting. This calibration factor should ONLY be applied to the `freq` word if `FREQ_B` is to be used as a transmitter tuning register. The calibration factor will actually degrade sensitivity if it is applied to receiver tuning parameters. See Section 4 for more information on frequency and deviation programming.

See `rfSetFreqA` for an example of usage.

3.1.31. rfSetMinimumFsep

```
LONGBOOL rfSetMinimumFsep(INT8 datamode, INT8 band, INT8 channel);
```

This function calculates the minimum Fsep for a modulation index of 0.7 and programs the RF chipset for a given channel. It should be called following a call to `rfSetChannel`. If this function is not called following a call to `rfSetChannel`, the Fsep will be set to a default 64kHz.

3.1.32. rfSetRefDiv

```
void rfSetRefDiv(INT8 refdiv);
```

Use this function to alter the reference divider independently of channel. Be aware that altering the reference divider causes the meaning of the frequency words to change. It does this because the reference divider determines the size of frequency jump each count in the frequency registers represent.

3.1.33. rfSetTxPO

```
void rfSetTxPO(INT8 pa_level)
```

This function is used to set the output power when in transmit mode. See section 5.0 for more information on programming the output power.

Initial Release

3.1.34. rfSetupInterruptLockFlags

```
void rfSetupInterruptLockFlags (LONGBOOL *unlock_flag, LONGBOOL *lock_flag);
```

Interrupt Driven Only. This function Sets up the pointers to variables that allow the main routine to know when to lock and unlock the average filter. The packet engine will set `*lock_flag` when a preamble has been detected. Similarly, it will set `*unlock_flag` once the packet has been received. This function can be used in conjunction with `rfUpdateLockFilter` to control the state of the average filter. A call to `rfSetupInterruptLockFlags (0, 0)` tells the packet engine to stop using the flags.

See `rfUpdateLockFilter` for an example.

3.1.35. rfSleep

```
void rfSleep(void);
```

This function will place the module into power-down(or sleep) mode. To enable the transmitter or receiver from this state, use `rfWakeToTx` or `rfWakeToTxEx` and `rfWakeToRx` or `rfWakeToRxEx`, respectively. For power specifications, refer to the appropriate EWD-HDTC data sheet.

3.1.36. rfStopOverridingCalibration

```
void rfStopOverridingCalibration(void);
```

This function forces the module to use the calibration parameters from its last calibration operation.

3.1.37. rfSwitchToRx

```
void rfSwitchToRx(void);
```

Sets the internal antenna switch to Rx segment. Does not energize the receiver or de-energize the transmitter. See `rfWakeToRx` or `rfWakeToRxEx` to wake receiver. Similarly, see `rfRxMode` and `rfRxModeEx` to activate the receiver / deactivate the transmitter.

3.1.38. rfSwitchToTx

```
void rfSwitchToTx(void);
```

Sets the internal antenna switch to Tx segment. Does not energize the transmitter or de-energize the receiver. See `rfWakeToTx` or `rfWakeToTxEx` to wake transmitter. Similarly, see `rfTxMode` and `rfTxModeEx` to activate the transmitter / deactivate the receiver.

3.1.39. rfTransmitPacketsWaiting

```
INT8 rfTransmitPacketsWaiting (void);
```

Interrupt Driven Only. This function returns the number of packets waiting to be transmitted in the packet transmit buffer. If the transmitter is not active, the function will return 0. This function is actually implemented as a macro in the RFAPI and makes a call directly to the UCAPI function, **ucTransmitPacketsWaitingRF**.

Example :

```
if (!rfTransmitPacketsWaiting())
{
    // Turn on transmitter
    rfRxMode(0xFF);

    // Put packet contents onto UART
    rfPutString("Done Transmitting!!!\r\n", 0);
}/*if*/
```

3.1.40. rfTransmittingPacket

```
LONGBOOL rfTransmittingPacket (void);
```

Interrupt Driven Only. Checks to see if the packet engine is in the process of transmitting packets. The function returns 1 indicating that the packet engine is in the process of transmitting packets: 0 if not. This function is actually implemented as a macro in the RFAPI and makes a call directly to the UCAPI function, **ucTransmittingRF**.

3.1.41. rfTxMode

```
void rfTxMode(INT8 pa_level);
```

Activates the transmitter section and switches to Tx state using the value specified in **pa_level** for the power amp and other predefined values for the module. This function assumes that the Tx frequency value is stored in freq register B. It also assumes that the oscillator core, frequency synthesizer, and bias generator are already operating. This function is separated from **rfWakeToTx** because waking requires additional delays. To wake the module to the Tx mode from sleep, use **rfWakeToTx** instead.

3.1.42. rfTxModeEx

```
void rfTxModeEx(INT8 current, INT8 pll, INT8 pa_level, INT8 freq_register);
```

This function accomplishes the same task as **rfTxMode**, but allows the programmer to specify custom values for the CURRENT and PLL registers. It also allows the specification of either the A or B frequency registers for transmitter tuning. This function is separated from **rfWakeToTxEx** because waking requires additional delays. To wake the module to the Rx mode from sleep, use **rfWakeToTxEx** instead.

Example:

```
// Selects Tx Mode, but uses "low power" current settings. It is assumed that
// the current FREQB registers are populated with optimal channel 25 values.
```

```
rfTxModeEx(0xF2, rf_915_optimal_channel_array[24][optRefDiv] << 3, 0xF0, fsFREQA);
```

3.1.43. rfUpdateLockFilter

```
void rfUpdateLockFilter (LONGBOOL *unlock_flag, LONGBOOL *lock_flag);
```

Interrupt Driven Only. This function either locks or unlocks the average filter based on the two status flags. If `*unlock_flag` is 1, the average filter will be locked and `*unlock_flag` set to 0. If the `*lock_flag` is 1, the average filter will be locked and `*lock_flag` will be set to 0. `*lock_flag` has precedence.

Whether you use these functions or not, when using NRZ or UART modes of operation, you MUST lock the average filter after the preamble has been detected to ensure reliable data delivery.

The packet engine will set flags during its operation. You can setup these flags with a call to `rfSetupInterruptLockFlags`, which will tell the packet engine to start manipulating them.

Example:

```
INT8 buf size;

LONGBOOL __LOCAL_SEG__ lock_flag;
LONGBOOL __LOCAL_SEG__ unlock_flag;

ucConfig();
rfReset();

// Set pointers for packet engine to use.
rfSetupInterruptLockFlags(&unlock_flag, &lock_flag);

rfWakeToRx();

while (1)
{
    if (rfReceivePacketsAvail())
    {
        buf size = sizeof(buf);
        if (rfGetPacket(buf, buf size))
        {
            // Put packet contents onto UART
            ucPutString(buf, buf size);
        }/*if*/
    }/*if*/

    // Locks the average filter when a preamble is detected and
    // unlocks the average filter when packet is complete.
    rfUpdateLockFilter(&unlock_flag, &lock_flag);
}/*while*/
```

3.1.44. rfWaitForChar

```
LONGBOOL rfWaitForChar(char start_byte, unsigned INT16  
bit_times_to_wait);
```

Polled Only. Monitors the RF interface for the byte specified in `start_byte` until it is found. The RX section must be selected in order for this function to operate properly. This function is not compatible with asynchronous UART mode.

Initial Release

3.1.45. rfWaitForString

```
LONGBOOL rfWaitForString(char *find_array, unsigned INT8 len);
```

Polled Only. Monitors the RF interface for the byte string specified in `find_array` until it is found. Note that you must not specify a search array so large that the function cannot execute the comparisons in a single bit-time. The function assumes that the data is sent least to most significant byte, least significant bit to most. The RX section must be selected in order for this function to operate properly. This function is not compatible with asynchronous UART mode.

3.1.46. rfWakeToRx

```
void rfWakeToRx(void);
```

Wakes up transceiver and places it into receive mode using predefined module constants. This routine assumes that the receiver, transmitter, oscillator core, bias generator, and frequency synthesizer are all powered down. Frequency register A is used to set the Rx VCO. If the oscillator core and bias generator are running (i.e. not recovering from `rfSleep`), `rfRxMode` is a faster, better choice.

3.1.47. rfWakeToRxEx

```
void rfWakeToRxEx(INT8 current, INT8 pll, INT8 freq_register);
```

This function accomplishes the same task as `rfWakeToRx`, but allows the programmer to specify custom values for the CURRENT and PLL registers. It also allows the specification of either the A or B frequency registers for receiver tuning. The function `rfRxMode` is a better, faster alternative if the module is not sleeping.

Example:

```
// Wakes to Rx Mode, but uses "low power" current settings. It is assumed that
// the current FREQA registers are populated with optimal channel 25 values.
rfWakeToRxEx(0x88, rf_915_optimal_channel_array[24][optRefDiv] << 3, fsFREQA);
```

3.1.48. rfWakeToTx

```
void rfWakeToTx(INT8 pa_level);
```

Wakes up the transceiver and places it into transmit mode using predefined module constants. The power level is determined by `pa_level`. This routine assumes that the receiver, transmitter, oscillator core, bias generator, and frequency synthesizer are all powered down. Frequency register B is used to set the Tx VCO. If the oscillator core and bias generator are running (i.e. not recovering from `rfSleep`), `rfTxMode` is a faster, better choice.

3.1.49. rfWakeToTxEx

```
void rfWakeToTxEx(INT8 current, INT8 pll, INT9 pa_level, INT8
freq_register);
```

This function accomplishes the same task as `rfWakeToTx`, but allows the programmer to specify custom values for the CURRENT and PLL registers. It also allows the specification of either the A or B frequency registers for transmitter tuning. The function `rfTxMode` is a better, faster alternative if the module is not sleeping.

Example:

```
// Wakes to Tx Mode, but uses "low power" current settings. It is assumed that
// the current FREQB registers are populated with optimal channel 25 values.
rfWakeToTxEx(0xF2, rf_915_optimal_channel_array[24][optRefDiv] << 3, 0xF0, fsFREQA);
```

3.1.50. rfWriteToRegister

```
void rfWriteToRegister(INT8 addr_byte, INT8 data_byte);
```

This routine writes `data_byte` to register at address `addr_byte`. The registers are defined in `rfapi.h`.

Example:

```
rfWriteToRegister(RF_FREQ_2A, 0xD9); // setup 915 MHz operation
rfWriteToRegister(RF_FREQ_1A, 0x00);
rfWriteToRegister(RF_FREQ_0A, 0x00);
```

3.1.51. rfWriteToRegisterWord

```
void rfWriteToRegisterWord(unsigned INT16 addr_and_data);
```

This routine is the same as `rfWriteToRegister` except that it takes one 16-bit word parameter that contains the address in the upper 8 bits and the data in the lower 8 bits.

Example:

```
rfWriteToRegisterWord(0x013F); //writes a 0x3F to register 0x01 (sleep)
```

Initial Release

3.2. uCAPI

The uCAPI exports functions that provide access to the micro-controller hardware. The RFAPI makes calls into this module. The version 1 uCAPI is targeted towards 8051 architecture. Specifically, the Cygnal C8051F310/311 is targeted. If you wish to port the RFAPI to another processor, functions labeled "**Required for RFAPI**" will have to be re-implemented on the target processor. If you know that you will implement only one of the two models (**Polled Only, Interrupt Driven Only**), you can omit functions specific to the other model.

In addition to the RFAPI interface functions, there are a number of utility functions that provide convenient access to some of the 8051 peripherals. These functions are provided for your convenience and are not required for porting.

The uCAPI is designed to provide RF communications access following one of two models: polled and interrupt-driven. The interrupt-driven model is selected by default. To enable the polled model, comment out the "#define INTERRUPT_DRIVEN_RF" statement in the UCAPI.h header file. The following table details the functions available with each model.

Polled	Both		Interrupt Driven
ucReadRFDataByte	<i>ucComBaud</i>	ucReadRFDataBit	ucGetPacketRF
ucWriteRFDataByte	<i>ucComInit</i>	ucReadRFProgByte	
	<i>ucConfig</i>	ucReadSMBus	ucReceivePacketsAvailRF
	ucDelayTicks	<i>ucSetOscFreq</i>	ucReceivingRF
	ucGetAnalog	ucStartAnalog	ucSetupForRxTx
	<i>ucGetChar</i>	ucTriState	ucSetupInterruptLockFlagsRF
	<i>ucGetString</i>	ucWriteRFDataBit	ucTransmitPacketsWaitingRF
	<i>ucKbhit</i>	ucWriteRFProgAddr	
	<i>ucPutChar</i>	ucWriteRFProgByte	
	<i>ucPutString</i>	delay_cycles	

* Function names in italics are not required for RFAPI or uCAPI implementation

Table 3: uCAPI function synopsis

3.2.1. delay_cycles

```
void delay_cycles(unsigned INT8 cycles)
```

Required for RFAPI. This is a macro expansion of `_nop_` instructions (1-cycle) used by the uCAPI to implement hard delays.

3.2.2. ucComBaud

```
LONGBOOL ucComBaud (unsigned INT8 baud);
```

This routine sets up the timer hardware that generates the baud rate for the hardware UART. The following table lists the symbolic constants that may be passed as the parameter to this function. This function is not required by RFAPI.

Initial Release

Constant	Baud Rate
ubr1200	1,200 bps
ubr2400	2,400 bps
ubr9600	9,600 bps
ubr14400	14,400 bps
ubr19200	19,200 bps
ubr28800	28,800 bps
ubr38400	38,400 bps
ubr57600	57,600 bps
ubr76800	76,800 bps
ubr115200	115,200 bps

Table 4: Hardware UART baud rate symbolic constants

3.2.3. ucComInit

```
void ucComInit (void) ;
```

Initializes the communications port to 1,200 baud and sets up associated timers and interrupts. This function is not required by RFAPI.

3.2.4. ucConfig

```
void ucConfig (void) ;
```

Initializes the micro-controller for operation with the RFAPI and uCAPI. It should be called once at the beginning of `main()`. This function is not required by RFAPI, but it is **HIGHLY** recommended that it be implemented.

3.2.5. ucDelayTicks

```
char ucDelayTicks(unsigned INT16 delay) ;
```

Required for RFAPI. This function uses NOPs to generate a delay of delay microseconds, approximately. This function is written for the C8051F310/311 running at 24.5MHz and will have to be modified for other processors and speeds. Each tick should be no less than one microsecond; if there is error in timing, ensure that each tick is more than one microsecond.

3.2.6. ucGetAnalog

```
unsigned INT16 ucGetAnalog(void) ;
```

Required for RFAPI. Finishes the analog conversion process and returns the A/D result. This **MUST** be preceded by a call to `ucStartAnalog` otherwise the result will be indeterminate. Returns a 16-bit (10-bit resolution) A/D result. This function is used to read the receive signal strength indication (RSSI).

Initial Release

3.2.7. ucGetChar

```
char ucGetChar(void);
```

Gets a character from the UART receive ring buffer. If none are available, the routine waits until there are. This function is not required by RFAPL.

Example:

```
#define ACK 0x06
char buf[4];

if (ucKbhit())
{
    if ((unsigned char)ucGetChar() == 0xFF)
    {
        // Get fixed-length message
        ucGetString(buf, 4);

        // Send ACK to client
        ucPutChar(ACK);
        // Send human-readable response
        ucPutString("Message Retrieved\r\n", 0);
    }
}
/*if*/
/*if*/
```

3.2.8. ucGetPacketRF

```
LONGBOOL ucGetPacketRF(char *buf, INT8 *num_chars);
```

Required for RFAPL. Interrupt Driven Only. Gets a packet from the receive packet buffer. If none are available, the function waits until there are. This function only works when INTERRUPT_DRIVEN_RF is selected.

3.2.9. ucGetString

```
void ucGetString(char *buf, INT8 num_chars);
```

Gets a string of characters of size `num_chars` from the receive buffer and places them in the supplied buffer, `buf`. If the ring buffer does not contain enough characters to satisfy the request, the routine waits until there are. This function is not required by RFAPL.

See `ucGetChar` for an example.

3.2.10. ucPutChar

```
void ucPutChar (char c);
```

Sends a character, `c`, to the transmit ring buffer and jumpstarts the UART if necessary.

See `ucGetChar` for an example. This function is not required by RFAPL.

3.2.11. ucPutPacketRF

```
LONGBOOL ucPutPacketRF(char *buf, INT8 num_chars);
```

Required for RFAPI. Interrupt Driven Only. Gets a packet from the receive packet buffer. If none are available, the function waits until there are. This function is only available when `INTERRUPT_DRIVEN_RF` is #defined.

3.2.12. ucPutString

```
void ucPutString(char *c, INT8 len);
```

Sends a string of characters, `c`, of size `len` to the transmit buffer and jumpstarts the UART if necessary. This function is not required by RFAPI.

See `ucGetChar` for an example.

3.2.13. ucKbhit

```
unsigned INT8 ucKbhit(void);
```

Returns UART receive buffer length (in characters). This function is not required by RFAPI.

See `ucGetChar` for an example.

3.2.14. ucReadRFDataBit

```
void ucReadRFDataBit(unsigned INT8 *working_byte);
```

Required for RFAPI. Captures an RF data bit and shifts into the least-significant bit of the given `*working_byte`. There are two implementations in the uCAPI: an inline macro expansion and a function. You can select which implementation is compiled into the uCAPI and RFAPI by either commenting out or leaving the `#define` for `INLINE_RF_BIT_FUNCS`, found in the `uCAPI.h` header file. By default, `#define INLINE_RF_BIT_FUNCS` is uncommented (active), which results in a macro expansion rather than a function call. This function uses I/O pin defines, `DIO` and `DCLK`, both found in the `uCAPI.h` header file.

3.2.15. ucReadRFDataByte

```
void ucReadRFDataByte(void);
```

Required for RFAPI. Polled Only. Captures an RF data byte one bit at a time. This function assumes that the data is sent least significant bit to most significant bit. This function uses I/O pin defines, `DIO` and `DCLK`, both found in the `uCAPI.h` header file.

Initial Release

3.2.16. ucReadRFProgByte

```
unsigned INT8 ucReadRFProgByte(void);
```

Required for RFAPI. Captures an RF chipset register byte one bit at a time. This function uses I/O pin defines, `PDAT` and `PCLK`, both found in the `uCAPI.h` header file.

3.2.17. ucReadSMBus

```
BOOL ucReadSMBus(unsigned INT8 cmd_byte, unsigned INT8 addr_byte,  
INT8 *data_byte);
```

Required for RFAPI. Reads a single byte from an SMBus (I2C) peripheral. Returns a 1 if the read was successful, returns a 0 if the read failed.

3.2.18. ucReceivingRF

```
LONGBOOL ucReceivingRF(void);
```

Required for RFAPI. Interrupt Driven Only. Returns a Boolean value indicating whether the RF interrupt routine is tracking a packet (found a preamble) or not (still searching). Returns a 1 if the RF interface is receiving a packet, returns a 0 if otherwise. This function only works when `INTERRUPT_DRIVEN_RF` is selected. This is the content for the shell function `rfReceivingPacket`.

3.2.19. ucReceivePacketsAvailRF

```
INT8 ucReceivePacketsAvailRF(void);
```

Required for RFAPI. Interrupt Driven Only. This function returns the number of packets waiting in the packet receive buffer. If the receiver is not active, the function will return 0. This function is the content of the RFAPI shell function, `rfReceivePacketsAvail`.

See `rfReceivePacketsAvail` for example

3.2.20. ucSetOscFreq

```
void ucSetOscFreq(unsigned INT8 osc_freq_select);
```

Sets up the internal oscillator frequency to the integer constant defined by `osc_freq_select`. Currently, there are only two constants defined - `ofs22_1184MHz` for 22.1184MHz and `ofs24_5MHz` for 24.5MHz. By default, `ucConfig` selects the maximum internal oscillator speed, 24.5MHz. This function is not required by RFAPI.

3.2.21. ucSetupForRxTx

```
void ucSetupForRxTx(unsigned INT8 mode);
```

Required for RFAPI. Interrupt Driven Only. Sets up the micro-controller for either transmission or reception based on the mode parameter. This function is used in conjunction with INTERRUPT_DRIVEN_RF.

3.2.22. ucSetupInterruptLockFlagsRF

```
void ucSetupInterruptLockFlagsRF (LONGBOOL *unlock_flag, LONGBOOL *lock_flag);
```

Required for RFAPI. Interrupt Driven Only. This function Sets up the pointers to variables that allow the main routine to know when to lock and unlock the average filter. The packet engine will set *lock_flag when a preamble has been detected. Similarly, it will set *unlock_flag once the packet has been received. This function can be used in conjunction with rfUpdateLockFilter to control the state of the average filter. A call to ucSetupInterruptLockFlagsRF(0, 0) tells the packet engine to stop using the flags. This function is the content for the RFAPI shell function rfSetInterruptLockFlags.

See rfUpdateLockFilter for an example.

3.2.23. ucStartAnalog

```
LONGBOOL ucStartAnalog(unsigned INT8 ucPort, unsigned INT8 ucPin);
```

Required for RFAPI. Begins the analog reading process for a pin given the Port and Pin. The pin must already have been defined as an analog input. Returns a 1 if successful, returns a 0 if failed. This function is used to read the receive signal strength indication (RSSI).

3.2.24. ucTransmitPacketsWaitingRF

```
INT8 ucTransmitPacketsWaitingRF (void);
```

Required for RFAPI. Interrupt Driven Only. This function returns the number of packets waiting to be transmitted in the packet transmit buffer. If the transmitter is not active, the function will return 0. This function is the content of the RFAPI shell function, rfTransmitPacketsWaiting.

See rfTransmitPacketsWaiting for example

3.2.25. ucTransmittingRF

```
LONGBOOL ucTransmittingRF(void);
```

Required for RFAPI. Interrupt Driven Only. Returns a boolean value indicating whether the RF interrupt routine is transmitting a packet (buffer not empty) or not (buffer empty). Returns a 1 if transmitting a packet over RF interface, returns a 0 if otherwise. This function only works when INTERRUPT_DRIVEN_RF is selected. This is the content for the RFAPI shell function rfTransmittingPacket.

Initial Release

3.2.26. ucTriState

```
LONGBOOL ucTriState(unsigned INT8 ucPort, unsigned INT8 ucPin,  
unsigned char direction);
```

Required for RFAPI. Change the "direction" of a general purpose I/O pin. If direction is `tsINPUT`, the GPIO pin will be configured as a HiZ input. If direction is `tsOUTPUT`, the GPIO pin will be configured as a push-pull output. 8051 GPIO pins, as well as many other micro-controller types, are arranged in 8-pin banks called ports. This port number is what the `ucPort` parameter refers to. Similarly, the `ucPin` parameter is 0-based index into this port. This function can use I/O port and pin `#defines` found in the `uCAPI.h` header file (e.g. `PORT_PALE`, `PIN_PALE`).

3.2.27. ucWriteRFDataBit

```
void ucWriteRFDataBit(unsigned INT8 *working_byte);
```

Required for RFAPI. Sends an RF data bit and shifts the sent bit out of the source byte. There are two implementations in the `uCAPI`: an inline macro expansion and a function. You can select which implementation is compiled into the `uCAPI` and `RFAPI` by either commenting out or leaving the `#define` for `INLINE_RF_BIT_FUNCS`, found in the `uCAPI.h` header file. By default, `#define INLINE_RF_BIT_FUNCS` is uncommented (active), which results in a macro expansion rather than a function call. This function uses I/O pin defines, `DIO` and `DCLK`, both found in the `uCAPI.h` header file.

3.2.28. ucWriteRFDataByte

```
void ucWriteRFDataByte(unsigned char byte_to_send);
```

Required for RFAPI. Polled Only. Sends a data byte out the RF interface least significant bit first. This function uses I/O pin defines, `DIO` and `DCLK`, both found in the `uCAPI.h` header file.

3.2.29. ucWriteRFProgAddr

```
void ucWriteRFProgAddr(unsigned INT8 addr_byte, BOOL r_w);
```

Required for RFAPI. Sends a RF chipset register address one bit at a time. The `r_w` Boolean flag is used to tell the chipset whether you will be reading (1) or writing (0) a register value. This function uses I/O pin defines, `PDAT`, `PCLK`, and `PALE`, all found in the `uCAPI.h` header file.

3.2.30. ucWriteRFProgByte

```
void ucWriteRFProgByte(unsigned INT8 data_byte);
```

Required for RFAPI. Writes a programming byte to the RF chipset. Must be preceded by a call to `ucWriteRFProgAddr`. This function uses I/O pin defines, `PDAT` and `PCLK`, both found in the `uCAPI.h` header file.

4. Frequency Programming

The EWD-HDTC transceiver module contains two frequency-programming registers (FREQA and FREQB). Generally, the frequency programmed into FREQA is used for receive and the frequency programmed into FREQB is used for transmit.

The FREQA and FREQB registers are programmed using the `rfSetFreqA` and `rfSetFreqB` functions respectively. Each of these functions is passed a 32-bit value, the lower 24-bits of which determine the actual operating frequency. The following equation is used to determine the 24-bit value required to obtain a particular frequency:

$$Fc(\text{MHz}) = Fref \times \frac{FREQA + 8192}{16384}$$

$$FREQA = \left\lfloor \left(\frac{Fc(\text{MHz}) \times 16384}{Fref} \right) - 8192 \right\rfloor$$

Designer's Note

Fxtal = 14.7456MHz

Equation 1: FREQx register calculation

This equation works for both the FREQA and FREQB registers.

$$Fref(\text{MHz}) = \frac{Fxtal(\text{MHz})}{REFDIV}$$

Equation 2: Fref calculation

The reference frequency, *Fref*, is derived from the crystal frequency, *Fxtal*, through division by the 4-bit *REFDIV*, which is in the PLL register. *Fxtal* is always 14.7456MHz. For more information about the PLL register, see Appendix A. Values for *REFDIV* are selected to provide optimum frequency matching. However, only *REFDIV* values of 6 – 14 are valid for this module.

Example:

Question: What FREQA value will yield an operating frequency of 915 MHz?

Solution: Step 1: Calculate *Fref*

$$Fref(\text{MHz}) = \frac{14.7456}{8}$$

$$\Rightarrow Fref(\text{MHz}) = 1.8432\text{MHz}$$

Step 2: Calculate *FREQA*

$$FREQA = \left\lfloor \left(\frac{915 \times 16384}{1.8432} \right) - 8192 \right\rfloor$$

$$\Rightarrow FREQA = 8125141_{dec}(0x7BFAD5_{hex})$$

Frequency deviation is programmed using the RFAPI function `rfSetSeparation`. Separation is calculated using the following equation.

$$Fsep = \frac{separation}{16384}$$

Equation 3: *Fsep* calculation

5. Output Power Programming

In transmit mode, the output power is programmed using the RFAPI function `rfSetTXPO`. Figure 9 shows the relationship between the variable *power* and the actual transmit power.

Initial Release

Output power [dBm]	RF frequency 433 MHz		RF frequency 868 MHz	
	PA_POW [hex]	Current consumption, typ. [mA]	PA_POW [hex]	Current consumption, typ. [mA]
-20	01	5.3	02	8.6
-19	01	6.9	02	8.8
-18	02	7.1	03	9.0
-17	02	7.1	03	9.0
-16	02	7.1	04	9.1
-15	03	7.4	05	9.3
-14	03	7.4	05	9.3
-13	03	7.4	06	9.5
-12	04	7.6	07	9.7
-11	04	7.6	08	9.9
-10	05	7.9	09	10.1
-9	05	7.9	0B	10.4
-8	06	8.2	0C	10.6
-7	07	8.4	0D	10.8
-6	08	8.7	0F	11.1
-5	09	8.9	40	13.8
-4	0A	9.6	50	14.5
-3	0B	9.4	50	14.5
-2	0C	9.7	60	15.1
-1	0E	10.2	70	15.8
0	0F	10.4	80	16.8
1	40	11.8	90	17.2
2	50	12.8	B0	18.5
3	50	12.8	C0	19.2
4	60	13.8	F0	21.3
5	70	14.8	FF	25.4
6	80	15.8		
7	90	16.8		
8	C0	20.0		
9	E0	22.1		
10	FF	26.7		

Figure 9: Output power programming information

6. Calibration

The EWD-HDTC module can be automatically calibrated to compensate for voltage, temperature, and process variations. `rfCalibrate` should be called each time the operating frequency changes by more than 1 MHz. If the power supply voltage changes by more than .5V or the operating temperature changes by more than 40 degrees C, the calibration should be performed again.

The automatic calibration does not compensate for temperature and process variations that affect the operating frequency of the reference crystal. The crystal is calibrated in the test stand at the factory and a frequency offset at 25 degrees C is stored in PROM on the module. This allows the module to be calibrated to +/- 10ppm accuracy. The following code example shows how to use the crystal calibration and auto calibration functions.

```
FCAL=rfGetFreqCal25C(fb915MHz);
rfSetFreqB(FREQB, FCAL);
rfRXMode();
if(!rfCalibrate())
    error_handler();
```

In this example, the crystal calibration value (FCAL) is retrieved from the module and added to the frequency programming value (FREQB) to determine a calibrated center frequency at 25 degrees C.

The calibration constant should ONLY BE APPLIED TO THE TX REGISTER. Then, the automatic calibration routine is executed. If a non-zero value is returned, there is a problem with the module, which prevents it from calibrating using the current frequency settings at the current operating voltage and temperature.

Frequency (MHz)	Calibration Time(mSec)
315	34
418	29
433.92	29
868	20
915	20

Table 5: Approximate calibration time based on operating frequency

7. Timing recovery and data detection

The EWD-HDTC module has built-in circuitry for data clock recovery and data detection. Data detection is performed by over-sampling and digital filtration of the incoming demodulated signal.

In Manchester mode, the timing recovery circuit controls the sample clock. This greatly simplifies the data reception software and hardware requirements of the user's circuit. In UART mode, the sample clock is free running. Thus, data reception performance will always be better at low signal levels when using Manchester mode.

The data decision circuit evaluates the demodulated signal level to determine if the data output should be a one or a zero. To accurately discriminate between a one and a zero, the data decision circuit uses a decision threshold that is in-between the zero level and the one level. If the data pattern is 1010101010, then the decision threshold is simply the average voltage of the demodulated data.

In the EWD-HDTC, the averaging filter establishes the decision threshold. At the beginning of each transmission, the transmitting module must send a pre-amble pattern of 1010.... During this period, the receiving module's averaging filter calculates the decision threshold. Once a valid pre-amble is detected and the averaging filter has established the decision threshold, the averaging filter is locked and actual data can be sent from the transmitting module.

The averaging filter can be set to lock automatically using the RF-API function `rfAverageAutoLock`. When the UART data mode is used, the automatic locking feature can be used to simplify the software burden of the host micro-controller. However, in auto-lock mode, the averaging filter does require a longer pre-amble than the manual-lock mode. The recommended pre-amble is 32-bits for the auto-lock mode.

Initial Release

If the requirement is for a shorter pre-amble, the user should manually lock the averaging filter using the RFAPI function `rfAverageManualLock` once a valid pre-amble is detected. The minimum pre-amble is 16-bits when using manual-lock mode.

```
while(1) // infinite data reception loop
{
    rfAverageFreeRun(); // averaging filter is free-running
    while(ucGetChar()!=PREAMBLE); // wait for pre-amble byte
    while(ucGetChar()!=PREAMBLE); // wait for pre-amble byte
    rfAverageManualLock(); // we have a valid pre-amble, so lock the averaging filter
    while(ucGetChar()!=STARTBYTE); // wait for start byte of packet
    . . . .
    put code here to receive and validate packet
    . . . .
}
```

The above example illustrates a polled method of receiving a packet. Initially, the averaging filter is free running because we are not confident that we are receiving a valid signal. Once we get two pre-amble bytes in a row, we can assume that we have locked onto a valid carrier from another module and that our averaging filter has settled, so we manually lock the filter and begin receiving the packet.

In Manchester mode, we recommend that the averaging filter be kept free-running. The module can be placed in this mode using the RFAPI function `rfAverageFreeRun`.

Regardless of the data mode selected, the data to be transmitted should be packetized to include, at a minimum, the pre-amble and data payload. For more information about data packetization, please refer to Section 9.

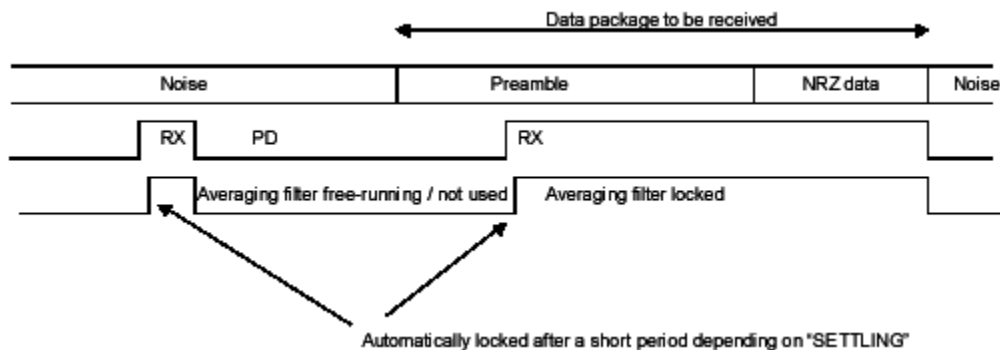


Figure 10: Automatic locking of averaging filter

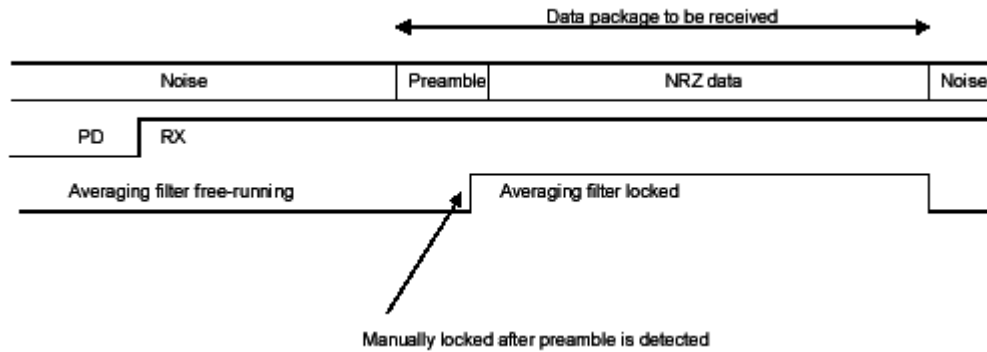


Figure 11: Manual locking of averaging filter

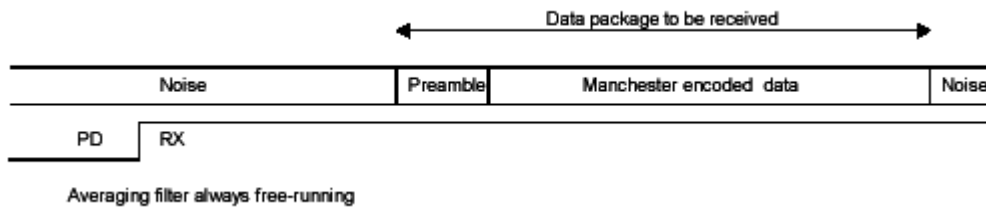


Figure 12: Free-running averaging filter

8. Sending and receiving data

DIO (pin 13) and DCLK (pin 12) are used to send and receive data to and from the module.

In Manchester mode, DCLK is driven by the module at the programmed data rate. If the module is in transmit mode, DIO is an input and the host micro-controller will shift data bits in on the falling edge of DCLK. In this mode, DIO is sampled by the module on the rising edge of DCLK. If the module is in receive mode, DIO is an output and the module will shift data bits out on the rising edge of DCLK. In this mode, DIO should be sampled by the host micro-controller on the falling edge of DCLK. All encoding and decoding is performed in the module. In this mode, the pre-amble consists of a repeated 10101010 pattern.

In UART mode, timing recovery is disabled. DCLK is the transmit data input and DIO is the received data output. In this mode, the pre-amble consists of a repeated 10101010 pattern.

In either mode, data should be packetized with a minimum of the pre-amble, a start byte sequence, and the data payload. An extra byte is required in UART mode to ensure that the receiving UART is in sync with the transmitting UART.



Figure 13: Simple packet for Manchester mode



Figure 14: Simple packet for UART mode

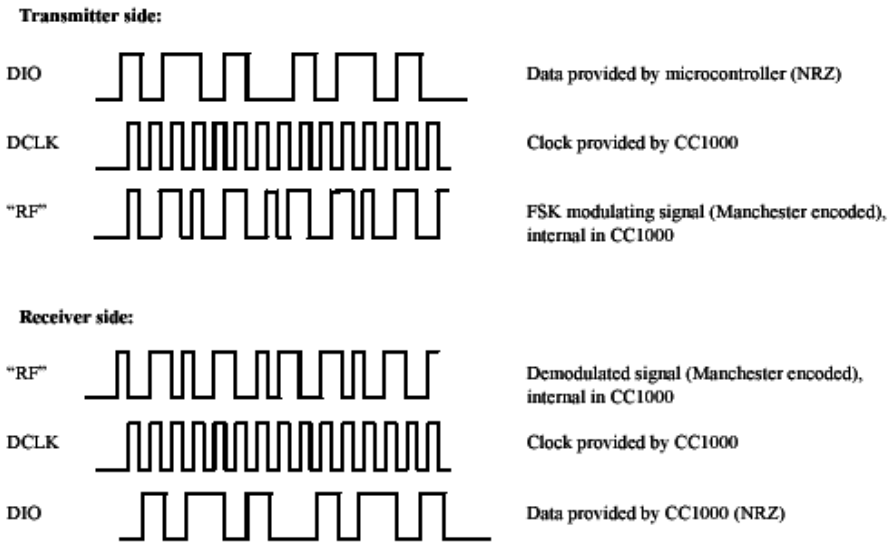


Figure 15: Manchester data timing

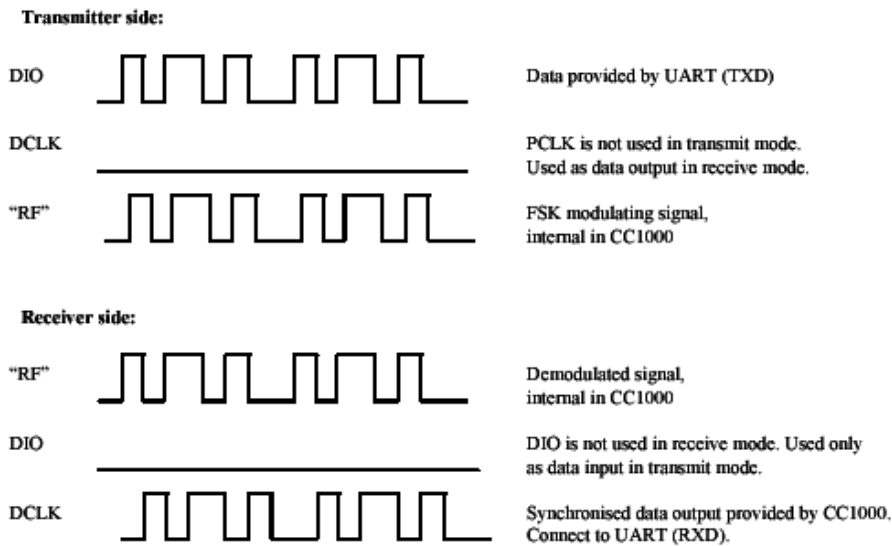


Figure 16: UART data timing

Initial Release

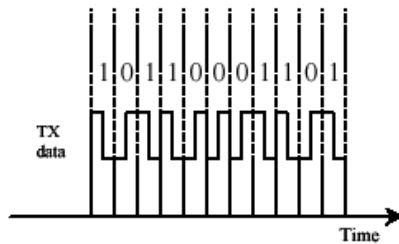


Figure 17: Manchester encoding detail

9. Low power operation

The module can be placed in a low-power sleep state by calling the RFAPI function `rfSleep`. Refer to the appropriate EWD-HDTC data sheet for power consumption specifications in this mode. In sleep mode, the micro-controller should tri-state PALE to eliminate leakage currents into the module.

When in the sleep state, the module can be awakened into either receive or transmit mode by using the RFAPI functions `rfWakeToRx` or `rfWakeToTx`, respectively.

10. RSSI output

The EWD-HDTC provides an analog indication of received signal strength in receive mode. The general response of the RSSI output is shown in figure 16. The RSSI output does require a filter capacitor to ground. The value of this capacitor will affect how quickly the RSSI can respond to changes in signal strength. A higher value will give a more accurate (less noisy) value with a slower response. A lower value will give a less accurate (more noisy) value with a faster response. If a fast response time is required, we recommend filtering the RSSI digitally by sampling with an A/D converter and calculating the mean (not average) value of the collected samples. For more information about this technique, please consult our technical support department.

Designer's Note

The RSSI pin requires a filter capacitor to ground. We recommend a value of 1nF for most cases.

Initial Release

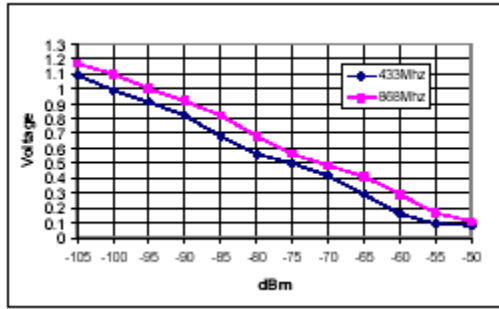


Figure 18: RSSI response

Initial Release

11. Printed circuit board layout

We recommend a 2-layer board with a large ground-plane on the bottom layer. The only critical layout considerations when designing a PCB using the EWD-HDTC module are sufficient grounding and the connections between the antenna and the module. Our RA.D. kit PCB is a good reference.

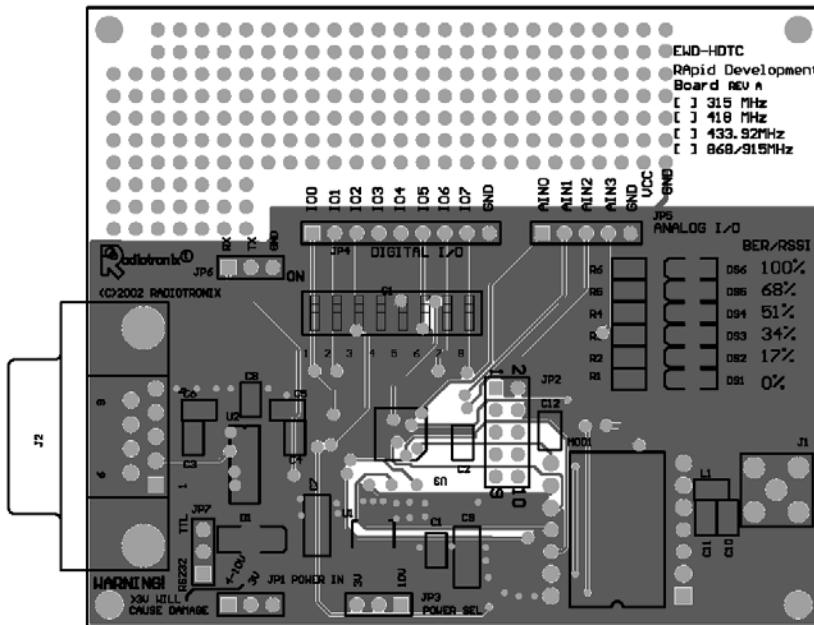


Figure 19: RA.D. kit PCB - bottom layer

Initial Release

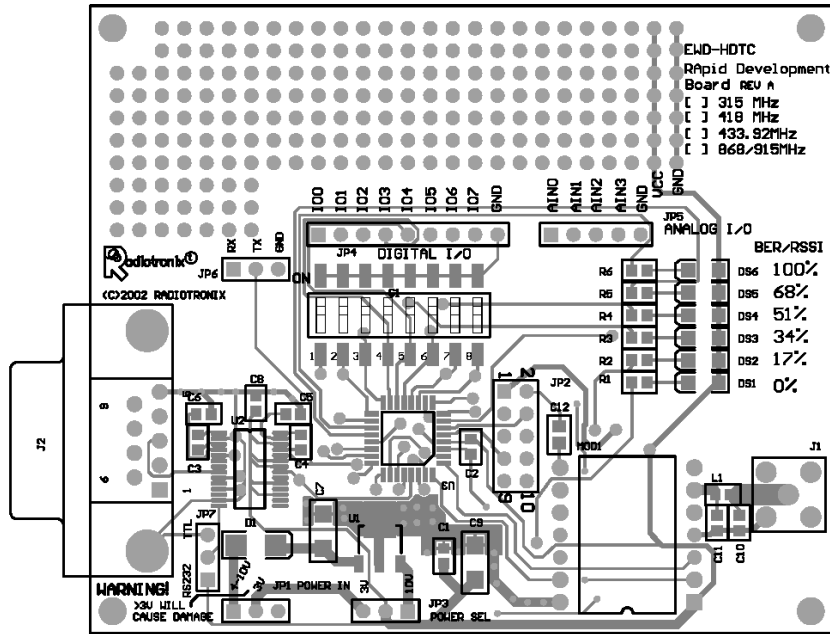
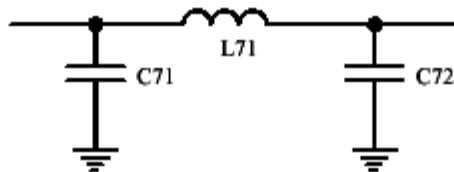


Figure 20: RA.D. kit PCB - top layer

12. Optional LC filter for improved harmonic performance

Although not required for proper operation, we do recommend that an external LC filter be added between the antenna and the module to simplify regulatory approvals.



Item	315 MHz	433 MHz	868 MHz	915 MHz
C71	30 pF	20 pF	10 pF	10 pF
C72	30 pF	20 pF	10 pF	10 pF
L71	15 nH	12 nH	5.6 nH	4.7 nH

Figure 21: Optional filter for improved harmonic performance

Appendix A: Configuration Registers

The EWD-HDTC is configured by programming twenty-nine 8-bit registers. An additional 7 registers are used for test purposes. Normally, we do not expect that the user will need to individually access these registers when the RF-API is used. However, to support situations where the RF-API is inadequate, we are publishing a detailed description of the registers.

REGISTER OVERVIEW

ADDRESS	Byte Name	Description
00h	MAIN	MAIN Register
01h	FREQ_2A	Frequency Register 2A
02h	FREQ_1A	Frequency Register 1A
03h	FREQ_0A	Frequency Register 0A
04h	FREQ_2B	Frequency Register 2B
05h	FREQ_1B	Frequency Register 1B
06h	FREQ_0B	Frequency Register 0B
07h	FSEP1	Frequency Separation Register 1
08h	FSEP0	Frequency Separation Register 0
09h	CURRENT	Current Consumption Control Register
0Ah	FRONT_END	Front End Control Register
0Bh	PA_POW	PA Output Power Control Register
0Ch	PLL	PLL Control Register
0Dh	LOCK	LOCK Status Register and signal select to CHP_OUT (LOCK) pin
0Eh	CAL	VCO Calibration Control and Status Register
0Fh	MODEM2	Modem Control Register 2
10h	MODEM1	Modem Control Register 1
11h	MODEM0	Modem Control Register 0
12h	MATCH	Match Capacitor Array Control Register for RX and TX impedance matching
13h	FSCTRL	Frequency Synthesiser Control Register
14h	FSHAPE7	Frequency Shaping Register 7
15h	FSHAPE6	Frequency Shaping Register 6
16h	FSHAPE5	Frequency Shaping Register 5
17h	FSHAPE4	Frequency Shaping Register 4
18h	FSHAPE3	Frequency Shaping Register 3
19h	FSHAPE2	Frequency Shaping Register 2
1Ah	FSHAPE1	Frequency Shaping Register 1
1Bh	FSDelay	Frequency Shaping Delay Register
1Ch	PRESCALER	Prescaler and IF-strip test control register
40h	TEST6	Test register for PLL LOOP
41h	TEST5	Test register for PLL LOOP
42h	TEST4	Test register for PLL LOOP
43h	TEST3	Test register for VCO
44h	TEST2	Test register for Calibration
45h	TEST1	Test register for Calibration
46h	TEST0	Test register for Calibration

Initial Release

MAIN Register (00h)

REGISTER	NAME	Default value	Active	Description
MAIN[7]	RXTX	-	-	RX/TX switch, 0 : RX , 1 : TX
MAIN[6]	F_REG	-	-	Selection of Frequency Register, 0 : Register A, 1 : Register B
MAIN[5]	RX_PD	-	H	Power Down of LNA, Mixer, IF, Demodulator, RX part of Signal Interface
MAIN[4]	TX_PD	-	H	Power Down of TX part of Signal Interface, PA
MAIN[3]	FS_PD	-	H	Power Down of Frequency Synthesiser
MAIN[2]	CORE_PD	-	H	Power Down of Crystal Oscillator Core
MAIN[1]	BIAS_PD	-	H	Power Down of BIAS (Global_Current_Generator) and Crystal Oscillator Buffer
MAIN[0]	RESET_N	-	L	Reset, active low. Writing RESET_N low will write default values to all other registers than MAIN. Bits in MAIN do not have a default value, and will be written directly through the configurations interface. Must be set high to complete reset.

FREQ_2A Register (01h)

REGISTER	NAME	Default value	Active	Description
FREQ_2A[7:0]	FREQ_A[23:16]	01110101	-	8 MSB of frequency control word A

FREQ_1A Register (02h)

REGISTER	NAME	Default value	Active	Description
FREQ_1A[7:0]	FREQ_A[15:8]	10100000	-	Bit 15 to 8 of frequency control word A

FREQ_0A Register (03h)

REGISTER	NAME	Default value	Active	Description
FREQ_0A[7:0]	FREQ_A[7:0]	11001011	-	8 LSB of frequency control word A

FREQ_2B Register (04h)

REGISTER	NAME	Default value	Active	Description
FREQ_2B[7:0]	FREQ_B[23:16]	01110101	-	8 MSB of frequency control word B

FREQ_1B Register (05h)

REGISTER	NAME	Default value	Active	Description
FREQ_1B[7:0]	FREQ_B[15:8]	10100101	-	Bit 15 to 8 of frequency control word B

FREQ_0B Register (06h)

REGISTER	NAME	Default value	Active	Description
FREQ_0B[7:0]	FREQ_B[7:0]	01001110	-	8 LSB of frequency control word B

FSEP1 Register (07h)

REGISTER	NAME	Default value	Active	Description
FSEP1[7:3]	-	-	-	Not used
FSEP1[2:0]	FSEP_MSB[2:0]	000	-	3 MSB of frequency separation control

FSEP0 Register (08h)

REGISTER	NAME	Default value	Active	Description
FSEP0[7:0]	FSEP_LSB[7:0]	01011001	-	8 LSB of frequency separation control

Initial Release

CURRENT Register (09h)

REGISTER	NAME	Default value	Active	Description
CURRENT[7:4]	VCO_CURRENT[3:0]	1100	-	Control of current in VCO core for TX and RX 0000 : 150µA 0001 : 250µA 0010 : 350µA 0011 : 450µA 0100 : 950µA, use for RX, f= 400 - 500 MHz 0101 : 1050µA 0110 : 1150µA 0111 : 1250µA 1000 : 1450µA, use for RX, f<400 MHz and f>500 MHz; and TX, f= 400 - 500 MHz 1001 : 1550µA, use for TX, f<400 MHz 1010 : 1650µA 1011 : 1750µA 1100 : 2250µA 1101 : 2350µA 1110 : 2450µA 1111 : 2550µA, use for TX, f>500 MHz
CURRENT[3:2]	LO_DRIVE[1:0]	10		Control of current in VCO buffer for LO drive 00 : 0.5mA, use for TX 01 : 1.0mA, use for RX, f<500 MHz 10 : 1.5mA, 11 : 2.0mA, use for RX, f>500 MHz
CURRENT[1:0]	PA_DRIVE[1:0]	10		Control of current in VCO buffer for PA 00 : 1mA, use for RX 01 : 2mA, use for TX, f<500 MHz 10 : 3mA 11 : 4mA, use for TX, f>500 MHz

FRONT_END Register (0Ah)

REGISTER	NAME	Default value	Active	Description
FRONT_END[7:6]	-	00	-	Not used
FRONT_END[5]	BUF_CURRENT	0	-	Control of current in the LNA_FOLLOWER 0 : 520uA, use for f<500 MHz 1 : 690uA, use for f>500 MHz
FRONT_END[4:3]	LNA_CURRENT [1:0]	01	-	Control of current in LNA 00 : 0.8mA, use for f<500 MHz 01 : 1.4mA 10 : 1.8mA, use for f>500 MHz 11 : 2.2mA
FRONT_END[2:1]	IF_RSSI[1:0]	00	-	Control of IF_RSSI pin 00 : Internal IF and demodulator, RSSI inactive 01 : RSSI active, RSSI/IF is analog RSSI output 10 : External IF and demodulator, RSSI/IF is mixer output. Internal IF in power down mode. 11 : Not used
FRONT_END[0]	XOSC_BYPASS	0	-	0 : Internal XOSC enabled 1 : Power-Down of XOSC, external CLK used

Initial Release

PA POW Register (0Bh)

REGISTER	NAME	Default value	Active	Description
PA_POW[7:4]	PA_HIGHPOWER[3:0]	0000	-	Control of output power in high power array. Should be 0000 in PD mode . See Table 10 page 29 for details.
PA_POW[3:0]	PA_LOWPOWER[3:0]	1111	-	Control of output power in low power array. Should be 0000 in PD mode. See Table 10 page 29 for details.

PLL Register (0Ch)

REGISTER	NAME	Default value	Active	Description
PLL[7]	EXT_FILTER	0	-	1 : External loop filter 0 : Internal loop filter 1-to-0 transition samples F_COMP comparator when BREAK_LOOP=1 (TEST3)
PLL[6:3]	REFDIV[3:0]	0010	-	Reference divider 0000 : Not allowed 0001 : Not allowed 0010 : Divide by 2 0011 : Divide by 3 1111 : Divide by 15
PLL[2]	ALARM_DISABLE	0	h	0 : Alarm function enabled 1 : Alarm function disabled
PLL[1]	ALARM_H	-	-	Status bit for tuning voltage out of range (too close to VDD)
PLL[0]	ALARM_L	-	-	Status bit for tuning voltage out of range (too close to GND)

Initial Release

LOCK Register (0Dh)

REGISTER	NAME	Default value	Active	Description
LOCK[7:4]	LOCK_SELECT[3:0]	0000	-	Selection of signals to CHP_OUT (LOCK) pin 0000 : Normal, pin can be used as CHP_OUT 0001 : LOCK_CONTINUOUS (active high) 0010 : LOCK_INSTANT (active high) 0011 : ALARM_H (active high) 0100 : ALARM_L (active high) 0101 : CAL_COMPLETE (active high) 0110 : IF_OUT 0111 : REFERENCE_DIVIDER Output 1000 : TX_PDB (active high, activates external PA when TX_PD=0) 1001 : Manchester Violation (active high) 1010 : RX_PDB (active high, activates external LNA when RX_PD=0) 1011 : Not defined 1100 : Not defined 1101 : LOCK_AVG_FILTER 1110 : N_DIVIDER Output 1111 : F_COMP
LOCK[3]	PLL_LOCK_ACCURACY	0	-	0 : Sets Lock Threshold = 127, Reset Lock Threshold = 111. Corresponds to a worst case accuracy of 0.7% 1 : Sets Lock Threshold = 31, Reset Lock Threshold = 15. Corresponds to a worst case accuracy of 2.8%
LOCK[2]	PLL_LOCK_LENGTH	0	-	0 : Normal PLL lock window 1 : Not used
LOCK[1]	LOCK_INSTANT	-	-	Status bit from Lock Detector
LOCK[0]	LOCK_CONTINUOUS	-	-	Status bit from Lock Detector

CAL Register (0Eh)

REGISTER	NAME	Default value	Active	Description
CAL[7]	CAL_START	0	↑	↑ 1 : Calibration started 0 : Calibration inactive CAL_START must be set to 0 after calibration is done
CAL[6]	CAL_DUAL	0	H	1 : Store calibration in both A and B 0 : Store calibration in A or B defined by MAIN[6]
CAL[5]	CAL_WAIT	0	H	1 : Normal Calibration Wait Time 0 : Half Calibration Wait Time The calibration time is proportional to the internal reference frequency. 2 MHz reference frequency gives 14 ms wait time.
CAL[4]	CAL_CURRENT	0	H	1 : Calibration Current Doubled 0 : Normal Calibration Current
CAL[3]	CAL_COMPLETE	0	H	Status bit defining that calibration is complete
CAL[2:0]	CAL_ITERATE	101	H	Iteration start value for calibration DAC 000 - 101: Not used 110 : Normal start value 111 : Not used

Initial Release

MODEM2 Register (0Fh)

REGISTER	NAME	Default value	Active	Description
MODEM2[7]	PEAKDETECT	1	H	Peak Detector and Remover disabled or enabled 0 : Peak detector and remover is disabled 1 : Peak detector and remover is enabled
MODEM2[6:0]	PEAK_LEVEL_OFFSET[6:0]	0010110	-	Threshold level for Peak Remover in Demodulator. Correlated to frequency deviation, see note.

Note: $PEAK_LEVEL_OFFSET[6:0] = \frac{F_x}{IF_{low}} - \frac{F_x}{IF_{low} + \frac{\Delta f}{2}} \cdot \frac{5}{8}$ where $F_x = \frac{f_{XOSC}}{XOSC_FREQ + 1}$

and $IF_{low} = 150kHz - 2 \cdot f_{rf} \cdot XTAL_accuracy$ and Δf is the separation

MODEM1 Register (10h)

REGISTER	NAME	Default value	Active	Description
MODEM1[7:5]	MLIMIT	011	-	Sets the limit for the Manchester Violation Flag. A Manchester Value = 14 is a perfect bit and a Manchester Value = 0 is a constant level (an unbalanced corrupted bit) 000 : No Violation Flag is set 001 : Violation Flag is set for Manchester Value < 1 010 : Violation Flag is set for Manchester Value < 2 011 : Violation Flag is set for Manchester Value < 3 100 : Violation Flag is set for Manchester Value < 4 101 : Violation Flag is set for Manchester Value < 5 110 : Violation Flag is set for Manchester Value < 6 111 : Violation Flag is set for Manchester Value < 7
MODEM1[4]	LOCK_AVG_IN	0	H	Lock control bit of Average Filter 0 : Average Filter is free-running 1 : Average Filter is locked
MODEM1[3]	LOCK_AVG_MODE	0	-	Automatic lock of Average Filter 0 : Lock of Average Filter is controlled automatically 1 : Lock of Average Filter is controlled by LOCK_AVG_IN
MODEM1[2:1]	SETTLING[1:0]	11	-	Settling Time of Average Filter 00 : 11 baud settling time, worst case 1.2dB loss in sensitivity 01 : 22 baud settling time, worst case 0.6dB loss in sensitivity 10 : 43 baud settling time, worst case 0.3dB loss in sensitivity 11 : 86 baud settling time, worst case 0.15dB loss in sensitivity
MODEM1[0]	MODEM RESET N	1	L	Separate reset of MODEM

Initial Release

MODEM0 Register (11h)

REGISTER	NAME	Default value	Active	Description
MODEM0[7]	-	-	-	Not used
MODEM0[6:4]	BAUDRATE[2:0]	010	-	000 : 0.6 kBaud 001 : 1.2 kBaud 010 : 2.4 kBaud 011 : 4.8 kBaud 100 : 9.6 kBaud 101 : 19.2, 38.4 and 76.8 kBaud 110 : Not used 111 : Not used
MODEM0[3:2]	DATA_FORMAT[1:0]	01	-	00 : NRZ operation. 01 : Manchester operation 10 : Transparent Asynchronous UART operation 11 : Not used
MODEM0[1:0]	XOSC_FREQ[1:0]	00	-	Selection of XTAL frequency range 00 : 3MHz - 4MHz crystal, 3.6864MHz recommended Also used for 76.8 kBaud, 14.7456MHz 01 : 6MHz - 8MHz crystal, 7.3728MHz recommended Also used for 38.4 kBaud, 14.7456MHz 10 : 9MHz - 12MHz crystal, 11.0592 MHz recommended 11 : 12MHz - 16MHz crystal, 14.7456MHz recommended

MATCH Register (12h)

REGISTER	NAME	Default value	Active	Description
MATCH[7:4]	RX_MATCH[3:0]	0000	-	Selects matching capacitor array value for RX, step size is 0.4 pF 0010: Use for RF frequency > 500 MHz 0111: Use for RF frequency < 500 MHz
MATCH[3:0]	TX_MATCH[3:0]	0000	-	Selects matching capacitor array value for TX, step size is 0.4 pF

FSCTRL Register (13h)

REGISTER	NAME	Default value	Active	Description
FSCTRL[7:4]	-	-	-	Not used
FSCTRL[3]	DITHER1	0	H	Enable dithering when transmitting '1'
FSCTRL[2]	DITHER0	0	H	Enable dithering during RX, and when transmitting '0'
FSCTRL[1]	SHAPE	0	H	Enable data shaping
FSCTRL[0]	FS_RESET_N	1	L	Separate reset of shaping sequencer

FSHAPE7 Register (14h)

REGISTER	NAME	Default value	Active	Description
FSHAPE7[7:5]	--	-	-	Not used
FSHAPE7[4:0]	FSHAPE7	00001	-	Frequency shape register 1, used when SHAPE in FSCTRL is active.

Initial Release

FSHAPE6 Register (15h)

REGISTER	NAME	Default value	Active	Description
FSHAPE6[7:5]	-	-	-	Not used
FSHAPE6[4:0]	FSHAPE6	00011	-	Frequency shape register 2, used when <i>SHAPE</i> in <i>FCTRL</i> is active.

FSHAPE5 Register (16h)

REGISTER	NAME	Default value	Active	Description
FSHAPE5[7:5]	-	-	-	Not used
FSHAPE5[4:0]	FSHAPE5	00110	-	Frequency shape register 3, used when <i>SHAPE</i> in <i>FCTRL</i> is active.

FSHAPE4 Register (17h)

REGISTER	NAME	Default value	Active	Description
FSHAPE4[7:5]	-	-	-	Not used
FSHAPE4[4:0]	FSHAPE4	01010	-	Frequency shape register 4, used when <i>SHAPE</i> in <i>FCTRL</i> is active.

FSHAPE3 Register (18h)

REGISTER	NAME	Default value	Active	Description
FSHAPE3[7:5]	-	-	-	Not used
FSHAPE3[4:0]	FSHAPE3	10000	-	Frequency shape register 5, used when <i>SHAPE</i> in <i>FCTRL</i> is active.

FSHAPE2 Register (19h)

REGISTER	NAME	Default value	Active	Description
FSHAPE2[7:5]	-	-	-	Not used
FSHAPE2[4:0]	FSHAPE2	10110	-	Frequency shape register 6, used when <i>SHAPE</i> in <i>FCTRL</i> is active.

FSHAPE1 Register (1Ah)

REGISTER	NAME	Default value	Active	Description
FSHAPE1[7:5]	-	-	-	Not used
FSHAPE1[4:0]	FSHAPE1	11100	-	Frequency shape register 7, used when <i>SHAPE</i> in <i>FCTRL</i> is active.

FSDELAY Register (1Bh)

REGISTER	NAME	Default value	Active	Description
FSDELAY[7:0]	FSDELAY[7:0]	00101111	-	Sets the number of clock cycles delay between the use of the FSHAPE registers during frequency shaping

Initial Release

PRESCALER Register (1Ch)

REGISTER	NAME	Default value	Active	Description
PRESCALER[7:6]	PRE_SWING[1:0]	00	-	Prescaler swing. Fractions for PRE_CURRENT[1:0] = 00 00 : 1 * Nominal Swing 01 : 2/3 * Nominal Swing 10 : 7/3 * Nominal Swing 11 : 5/3 * Nominal Swing
PRESCALER[5:4]	PRE_CURRENT [1:0]	00	-	Prescaler current scaling 00 : 1 * Nominal Current 01 : 2/3 * Nominal Current 10 : 1/2 * Nominal Current 11 : 2/5 * Nominal Current
PRESCALER[3]	IF_INPUT	0	-	0 : Nominal setting 1 : RSSI/IF pin is input to IF-strips
PRESCALER[2]	IF_FRONT	0	-	0 : Nominal setting 1 : Output of IF_Front_amp is switched to RSSI/IF pin
PRESCALER[1:0]	-	00	-	Not used

TEST6 Register (for test only, 40h)

REGISTER	NAME	Default value	Active	Description
TEST6[7]	LOOPFILTER_TP1	0	-	1 : Select testpoint 1 to CHP_OUT 0 : CHP_OUT tied to GND
TEST6 [6]	LOOPFILTER_TP2	0	-	1 : Select testpoint 2 to CHP_OUT 0 : CHP_OUT tied to GND
TEST6 [5]	CHP_OVERRIDE	0	-	1 : use CHP_CO[4:0] value 0 : use calibrated value
TEST6[4:0]	CHP_CO[4:0]	10000	-	Charge_Pump Current DAC override value

TEST5 Register (for test only, 41h)

REGISTER	NAME	Default value	Active	Description
TEST5[7:6]	-	-	-	Not used
TEST5[5]	CHP_DISABLE	0	-	1 : CHP up and down pulses disabled 0 : normal operation
TEST5[4]	VCO_OVERRIDE	0	-	1 : use VCO_AO[2:0] value 0 : use calibrated value
TEST5[3:0]	VCO_AO[3:0]	1000	-	VCO_ARRAY override value

TEST4 Register (for test only, 42h)

REGISTER	NAME	Default value	Active	Description
TEST4[7:6]	-	-	-	Not used
TEST4[5:0]	L2KIO[5:0]	100101	h	Constant setting charge pump current scaling/rounding factor. Sets Bandwidth of PLL. Use 3Fh for 38.4 and 76.8 kBaud

Initial Release

TEST3 Register (for test only, 43h)

REGISTER	NAME	Default value	Active	Description
TEST3[7:5]	-	-	-	Not used
TEST3[4]	BREAK_LOOP	0	-	1 : PLL loop open 0 : PLL loop closed
TEST3[3:0]	CAL_DAC_OPEN	0100	-	Calibration DAC override value, active when BREAK_LOOP =1

TEST2 Register (for test only, 44h)

REGISTER	NAME	Default value	Active	Description
TEST2[7:5]	-	-	-	Not used
TEST2[4:0]	CHP_CURRENT [4:0]	-	-	Status vector defining applied CHP_CURRENT value

TEST1 Register (for test only, 45h)

REGISTER	NAME	Default value	Active	Description
TEST1[7:4]	-	-	-	Not used
TEST1[3:0]	CAL_DAC[3:0]	-	-	Status vector defining applied Calibration DAC value

TEST0 Register (for test only, 46h)

REGISTER	NAME	Default value	Active	Description
TEST0[7:4]	-	-	-	Not used
TEST0[3:0]	VCO_ARRAY[3:0]	-	-	Status vector defining applied VCO_ARRAY value

Initial Release

Appendix B: Channel Tables

915MHz Band Tables

There are three pre-defined channel tables that you may choose to have compiled into your code memory. Including a table in your code memory involves two steps. The first is to ensure that `#define OPTIMAL_915_ARRAY 1` is in your `rfapi.h` header file. By default, it is. This tells the compiler to include the channel functions, as well as a 915MHz table. The second setting only has effect if `OPTIMAL_915_ARRAY` is 1. This setting, `#define USE_915_MHZ_TABLE_INDEX`, tells the pre-compiler to include one of the three tables listed below.

Original "Legacy" 49-channel Table (USE_915_MHZ_TABLE_INDEX = 1)					
Channel	Frequency (MHz)	Channel	Frequency (MHz)	Channel	Frequency (MHz)
1	902.265	18	911.217	35	920.170
2	902.791	19	911.744	36	920.697
3	903.318	20	912.271	37	921.223
4	903.845	21	912.797	38	921.750
5	904.371	22	913.324	39	922.277
6	904.898	23	913.851	40	922.803
7	905.425	24	914.377	41	923.330
8	905.951	25	914.904	42	923.857
9	906.478	26	915.430	43	924.383
10	907.004	27	915.957	44	924.910
11	907.531	28	916.484	45	925.436
12	908.058	29	917.010	46	925.963
13	908.584	30	917.537	47	926.490
14	909.111	31	918.064	48	927.016
15	909.638	32	918.590	49	927.543
16	910.164	33	919.117		
17	910.691	34	919.643		

Table 6: 915MHz Legacy Channel Table

Initial Release

Enhanced 50-channel table, 500kHz (USE 915_MHZ_TABLE_INDEX = 2)					
Channel	Frequency (MHz)	Channel	Frequency (MHz)	Channel	Frequency (MHz)
1	902.013	18	910.917	35	919.749
2	902.491	19	911.396	36	920.221
3	903.018	20	911.865	37	920.713
4	903.545	21	912.376	38	921.450
5	904.023	22	912.848	39	921.977
6	904.493	23	913.340	40	922.503
7	905.003	24	914.077	41	923.030
8	905.476	25	914.604	42	923.557
9	905.967	26	915.130	43	924.083
10	906.704	27	915.657	44	924.610
11	907.231	28	916.184	45	925.136
12	907.758	29	916.710	46	925.663
13	908.284	30	917.237	47	926.142
14	908.811	31	917.764	48	926.611
15	909.338	32	918.290	49	927.121
16	909.864	33	918.769	50	927.594
17	910.391	34	919.238		

Table 7: Enhanced 915MHz 50-Channel Table

Initial Release

197-channel table, 50kHz (USE 915 MHZ TABLE INDEX = 3)					
Channel	Frequency (MHz)	Channel	Frequency (MHz)	Channel	Frequency (MHz)
1	902.013	67	910.800	133	919.439
2	902.096	68	910.917	134	919.607
3	902.167	69	911.005	135	919.749
4	902.281	70	911.128	136	919.870
5	902.404	71	911.242	137	919.975
6	902.491	72	911.312	138	920.109
7	902.608	73	911.396	139	920.221
8	902.683	74	911.620	140	920.316
9	902.734	75	911.809	141	920.397
10	903.018	76	911.865	142	920.528
11	903.302	77	911.971	143	920.631
12	903.428	78	912.066	144	920.713
13	903.545	79	912.234	145	920.836
14	903.632	80	912.376	146	920.923
15	903.755	81	912.497	147	921.450
16	903.869	82	912.603	148	921.977
17	903.940	83	912.737	149	922.064
18	904.023	84	912.848	150	922.187
19	904.247	85	912.943	151	922.269
20	904.436	86	913.024	152	922.372
21	904.493	87	913.156	153	922.503
22	904.598	88	913.258	154	922.584
23	904.694	89	913.340	155	922.679
24	904.861	90	913.463	156	922.791
25	905.003	91	913.551	157	922.925
26	905.125	92	914.077	158	923.030
27	905.230	93	914.604	159	923.151
28	905.364	94	914.692	160	923.293
29	905.476	95	914.814	161	923.461
30	905.570	96	914.896	162	923.557
31	905.651	97	914.999	163	923.662
32	905.783	98	915.130	164	923.719
33	905.885	99	915.211	165	923.908
34	905.967	100	915.306	166	924.083
35	906.090	101	915.418	167	924.215
36	906.178	102	915.552	168	924.286
37	906.704	103	915.657	169	924.399
38	907.231	104	915.779	170	924.522
39	907.319	105	915.920	171	924.610
40	907.442	106	916.088	172	924.727
41	907.524	107	916.184	173	924.801
42	907.626	108	916.289	174	924.853
43	907.758	109	916.346	175	925.136
44	907.839	110	916.535	176	925.420
45	907.933	111	916.710	177	925.546
46	908.045	112	916.842		925.663
47	908.179	113	916.913	179	925.751
48	908.284	114	917.026	180	925.874

Initial Release

197-channel table, 50kHz (USE 915 MHZ TABLE INDEX = 3)					
Channel	Frequency (MHz)	Channel	Frequency (MHz)	Channel	Frequency (MHz)
49	908.406	115	917.149	181	925.987
50	908.548	116	917.237	182	926.058
51	908.715	117	917.354	183	926.142
52	908.811	118	917.428	184	926.365
53	908.916	119	917.480	185	926.554
54	908.973	120	917.764	186	926.611
55	909.162	121	918.047	187	926.716
56	909.338	122	918.173	188	926.812
57	909.469	123	918.290	189	926.980
58	909.540	124	918.378	190	927.121
59	909.654	125	918.501	191	927.243
60	909.776	126	918.614	192	927.348
61	909.864	127	918.685	193	927.482
62	909.981	128	918.769	194	927.594
63	910.056	129	918.992	195	927.689
64	910.107	130	919.181	196	927.770
65	910.391	131	919.238	197	927.901
66	910.674	132	919.343		

Table 8: Expanded 915MHz 197Channel Table

Initial Release

868MHz Band Tables

You may include an 868MHz channel table in your code memory, as well. Currently, there is only one 868MHz table defined. Others will be added as customer demand dictates. To include the channel functions and the 868MHz table, ensure that that `#define OPTIMAL_868_ARRAY 1` is uncommented and in your `rfapi.h` header file. By default, it is. You may include both 915MHz and 868MHz tables in your code simultaneously.

Original 34-channel Table					
Channel	Frequency (MHz)	Channel	Frequency (MHz)	Channel	Frequency (MHz)
1	868.034	13	869.321	25	<i>868.787</i>
2	868.130	14	869.403	26	<i>868.919</i>
3	868.297	15	869.470	27	<i>869.021</i>
4	868.439	16	869.526	28	<i>869.103</i>
5	868.502	17	869.573	29	<i>869.170</i>
6	868.561	18	<i>868.139</i>	30	<i>869.226</i>
7	868.666	19	<i>868.202</i>	31	<i>869.273</i>
8	868.800	20	<i>868.261</i>	32	<i>869.314</i>
9	868.912	21	<i>868.366</i>	33	<i>869.614</i>
10	869.006	22	<i>868.500</i>	34	<i>869.840</i>
11	869.087	23	<i>868.612</i>	* Italics indicate data is negated.	
12	869.219	24	<i>868.706</i>		

Table 9: Original 868MHz Channel Table

Initial Release