DLP-RFS1231

LEAD FREE

915MHz DATA RADIO

****PRELIMINARY DOCUMENT-SUBJECT TO CHANGE****

FEATURES:

- 31-Channel FHSS
- +17dBm Output Power
- Up to 1-Mile Range
- u.fl Antenna Connector
- On-Board Chip Antenna
- FCC/IC/CE Modular Approvals in Place
- Permanent, Unique Serial Number Built In
- Single 2.4- to 3.6-Volt Supply
- Development Kit Available

APPLICATION AREAS:

- Real-Time Security
- Body-Worn Medical Telemetry
- Battery-Powered Home Automation
- Electric/Water/Gas Automated Meter Reading
- Industrial Monitoring and Control
- Active RFID
- Long Range, Battery-Powered, Multi-Hop Sensor Networks

1.0 INTRODUCTION

The DLP-RFS1231 is a low-cost module for transmitting and receiving digital data via radio frequency. All of the DLP-RFS1231's electronics (including an antenna) reside on a single PCB, and all operational power is derived from a single supply voltage.

The transceiver design is made up of a Renesas RL78 low-power microcontroller (R5F100EEANA), a Semtech SX1231 low-power, integrated UHF transceiver and an antenna switch for selecting between the on-board antenna and an optional external antenna. The hardware is designed for maximum range and optimum battery life.

2.0 ELECTRICAL SPECIFICATIONS

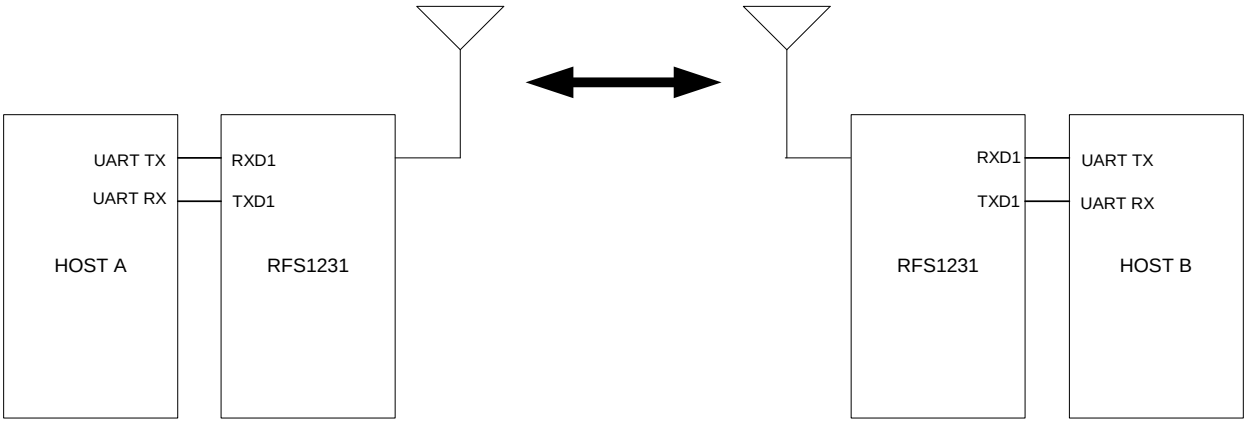
Supply Voltage	2.4-3.6V
Reader Frequency	902-928MHz
Output Power	50mW (17dBm) MAX
Range	Up to 1 Mile (depends upon the antenna used)
Protocol	Frequency Hopping; Spread Spectrum
Communications Interface	TTL Serial – 38KBPS
Operational Power – Transmit	Depends on Setup
Operational Power – Receive	~35mA
Operational Power – Sleep	5uA
Antenna Connector	u.fl*
Operating Temperature	0-70°C

*Please see the Antenna Section for important regulatory details.

2.1 SERIAL NUMBER

Each DLP-RFS1231 contains a unique, 48-bit, hard-coded serial number that cannot be altered by any means. The serial number can be read via the microcontroller and used to identify the transceiver in the transmitted packets.

2.2 HOST CONNECTION



2.3 QUICK-START GUIDE

This Quick-Start Guide is designed to work with our DLP-RFS1231 transceiver and DLP-RFS-BATT battery board Development Kit. (The part number for this kit is DLP-RFS-DK.) The kit contains two DLP-RFS1231 transceivers, two DLP-RFS-BATT battery boards and a Tag-Connect adapter for use with the Renesas E1 debugger (purchased separately).

DLP-RFS-DK

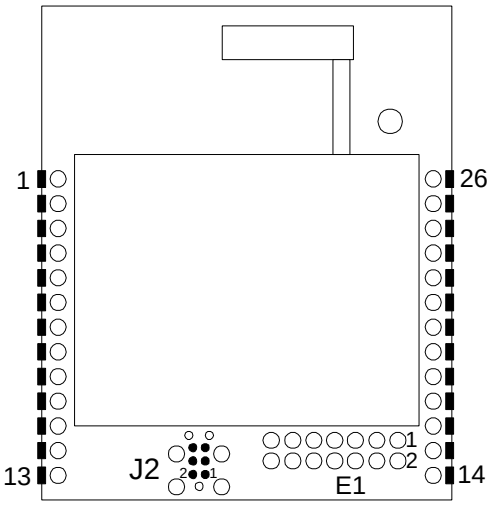
2.3.1 Mount the two transceiver modules to the battery boards via either the pin headers supplied with the transceiver or by surface-mount soldering.

2.3.2 Place two jumpers on J5 and J6. J5 connects the battery pack to the voltage regulator on the DLP-RFS-BATT. J6 connects regulated voltage to the transceiver module. If you want to measure current consumption into the transceiver module, simply remove J6 and connect a current meter to the two pins of J6. Doing so will place the meter in series only with the current flowing into the transceiver.

2.3.3 The sample application that is provided will perform one of two functions based upon a switch setting on the battery board. A Logic 1 on Switch 1 of the dip switch means that the node will be a sensor node. If it is a sensor node, the app will read the MAC ID chip and set the MAC ID to the value read from the MAC ID chip. The sensors (temperature sensor, light sensor and battery voltage) are read via the ADC. The result is inserted into a fixed-length message and transmitted to the gateway/master for display.

2.3.4 A Logic 0 on Switch 1 of the dip switch means that the node will be a gateway or master node. The gateway or master node will set the radio for reception. When a valid packet is received, the packet is output to the USB port, the buzzer is sounded and the green LED on the battery board is flashed.

2.4 PIN SIGNALS



J2 Tag-Connect Pins	
1	RESET#
2	VBAT
3	No Connect
4	Ground
5	TOOL0
6	No Connect
E1 Debugger Connection	
1	No Connect
2	Ground
3	No Connect
4	No Connect
5	TOOL0
6	No Connect
7	No Connect
8	VBAT
9	VBAT
10	RESET#
11	No Connect
12	Ground
13	RESET#
14	Ground
DLP-RFS1231 Module IO Pins	
1	Ground
2	TXD1
3	RXD1
4	P20 / ANI0
5	P61
6	P51 / INTP2 / SO11
7	P30 / INTP3 / SCK11
8	P60
9	P120 / ANI19

10	P72 / ANI19
11	P31 / INTP4 / PCLBUZ0
12	Ground
13	VBAT
14	VBAT
15	Ground
16	SPIMOSI / TXD0
17	P22 / ANI2
18	SPIMISO/RXD0
19	SPICLK
20	P24 / ANI4
21	P71 / SI21
22	P26 / ANI6
23	P73
24	P23 / ANI3
25	P21 / ANI1
26	Ground

2.5 PROGRAM / DEBUG INTERFACE

The Renesas E1 Device Programmer is required for programming and debugging the DLP-RFS1231 module. The E1 can be connected to the module via either the Tag-Connect interface or the standard 14-pin header. [Note that a male header (purchased separately) will need to be soldered to the PCB if the 14-pin header is to be used.]

The E1 can be purchased from one of several distributors. One easy way to determine who has stock is to search for Part Number R0E000010KCE00 on www.findchips.com:

http://www.findchips.com/avail?part=R0E000010KCE00+

2.6 C COMPILER

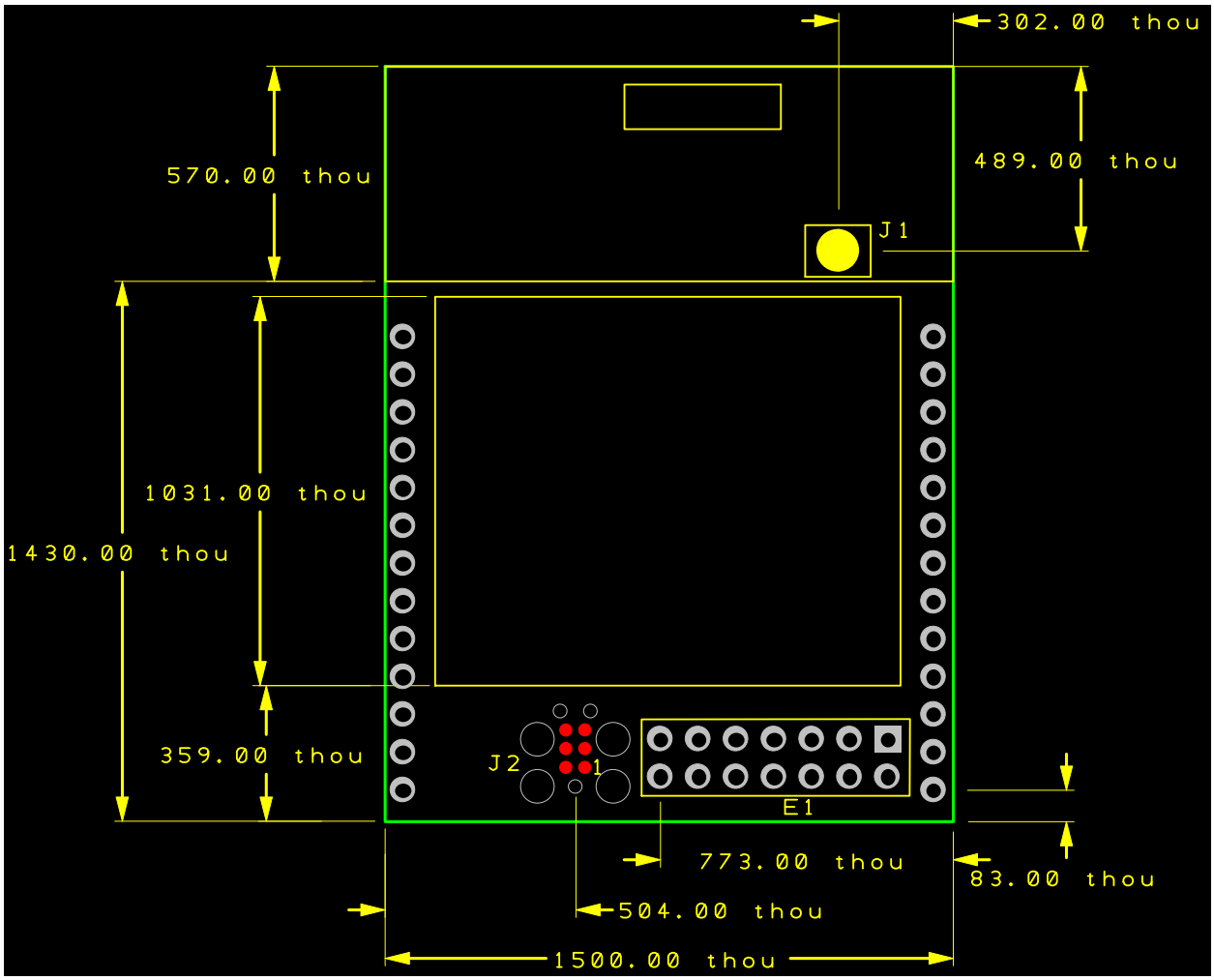
A free C compiler is available for download from IAR to help the user get started in firmware development:

http://www.iar.com/en/Products/IAR-Embedded-Workbench/Renesas-RL78/

The downloads available through this location offer either a Kickstart version or a 30-day evaluation version. It is best to select the Kickstart version since it carries no time limit and allows for up to 16K of object code.

3.0 MECHANICAL

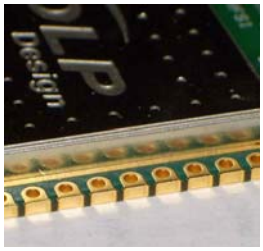
3.1 MECHANICAL DRAWING (Overall dimensions: 1.5 x 2.0 x 1.53 Inches)



Note: thou = mils or 0.001 inches.

3.2 MOUNTING OPTIONS

The DLP-RFS1231 module can be either surface mounted to a printed circuit board or connected using 0.1-inch spaced headers (0.025 sq-inch posts).



4.0 APPLICATION DEVELOPMENT – SimpleRF™

A software library is available for free download upon purchase of the DLP-RFS1231 that demonstrates reading the serial number, transmitting and receiving data packets and setting up simple point-to-point and star networks.

The software design is divided into task-specific “managers” and a hardware-specific API. A manager provides specific functions related to the operation of the radio protocol. The overall design consists of layers that interact through callback functions. There is no RTOS, although the software can be configured to accommodate a small RTOS if desired.

4.1 RF PROTOCOL STACK OVERVIEW

The SimpleRF™ RF Protocol Stack is divided up into three layers. The lowest-level layer is the PHY layer. An application could only use the PHY layer to provide point-to-multipoint (broadcast) RF communications, or it might use the PHY layer in conjunction with the MAC layer to provide higher-level message delivery services. Details of the operation of the various layers are outlined below:

4.1.1 RF MESSAGE MANAGER

The RF Message Manager is responsible for constructing messages. The RF Message Manager decouples message management from protocol components. It builds a message as a reverse linked list of pointers to buffers. The RF Message Manager operates on messages using unique message handles. All message storage is external; this manager does not store message data locally.

4.1.2 PHY MANAGER

The PHY Manager (layer) provides control of the physical RFIC and the low-level protocol for basic packet transceiver services. The PHY Manager can operate in single-channel DTS mode or in 26-channel FHSS mode.

4.1.3 MAC MANAGER

The MAC Manager (layer) is responsible for sending and receiving packets using the PHY. The MAC Manager is analogous to the MAC layer in the OSI standard 7-layer stack. It provides the ability to send messages to specifically addressed nodes and, optionally, to receive acknowledgements for those messages. If an acknowledgement is expected and not received, the MAC layer will retry the message.

The MAC Manager supports three packet types:

1. kMACUniAck - Acknowledged Packet

2. kMACUniNoAck - Unacknowledged Packet

3. kMACMulti - Multicast Packet

4.1.4 EVENT MANAGER

The Event Manager is used to notify the application layer of events that occur inside the PHY and MAC layers. This mechanism can be used to either drive the application layer or for diagnostic purposes. The following are valid events:

4.1.5 APPLICATION LAYER

The application layer is user defined, but it typically does the following:

- Initializes the hardware and software managers,
- Initializes application-layer variables and data structures,
- Reads the MAC ID chip and sets the MAC ID of the system,
- Passes the address of the application layer master loop; and
- Calls the MAC layer main loop (if a MAC-layer application).

The MAC layer main loop will run to completion then call the application layer main loop ad infinitum. The application layer main loop is completely user defined. A typical sensor node application will set a timer and periodically read sensors and transmit the sensor data to another node using MAC-layer function calls. The application layer may also listen for messages from other nodes. The MAC Manager will notify the application layer via the Event Manager when packet transmissions are complete, packet reception has occurred or some type of error event occurs.

The sample application that is provided will do one of two functions based upon a switch setting on the battery board. A Logic 1 on Switch 1 of the dip switch means that the node will be a sensor node. A Logic 0 means that the node will be a gateway or master node. If it is a sensor node, the app reads the MAC ID chip and sets the MAC ID to the value read from the MAC ID chip. If it is a gateway/master node, then the MAC ID is set to 0.

The gateway or master node will set the radio for reception. When a valid packet is received, the packet is output to the USB port on the battery board, the buzzer is sounded and the green LED on the battery board is flashed.

The sensor node sets a timer event using the Timer Manager. When a timer event occurs, the Timer Manager will call the application callback which, in turn, causes the start of a sensor read. The sensors (temperature, light and battery voltage) are read via the ADC. The result is inserted into a fixed-length message and transmitted to the gateway/master through a MAC Manager function call.

This is only one example of how to implement a wireless sensor application. The potential applications are limitless. Most of the resources of the RL78 MCU are available for the application to use.

4.1.6 MICROCONTROLLER API

The various managers communicate with the hardware through an abstraction layer called the Microcontroller API. Functions that are specific to the microcontroller are contained in this API. Those functions include firmware for initializing MCU hardware such as timers, communication ports, IO ports, interrupt handlers, etc. The API also has code for controlling these MCU devices during operation such as handling timer interrupts, sending and receiving data over the communication ports

(UARTS, SPI) and facilities for implementing callback functions for the higher-level managers. The design philosophy of this API is to provide the ability to move across the various MCU devices that Renesas has to offer without significantly changing the high-level managers and application software. The microcontroller API maintains a data structure called ucPrivateData that stores the addresses for all Microcontroller API callback functions as well as timer and IO port "shadow" registers (used to accommodate the fact that IO port control registers on the H8 38076 are write only). For details on the API function calls, refer to Section 5.0.

4.1.7 UART MANAGER

The UART Manager provides an interrupt-driven, circular-buffered, transmit-and-receive interface for UART0. The UART Manager API provides functions to configure the UART and to access the transmit and receive buffers.

The transmit buffer is 255 bytes. To send a byte to the UART, call uartSendByte(). That byte is put into the transmit buffer, and the transmit interrupt is enabled. The UART0 ISR will then remove that byte from the transmit buffer and send it to the UART. To send a null-terminated string, call uartSendString().

The receive buffer is 32 bytes. When the UART0 ISR receives a byte, it puts that byte into the receive buffer. To determine if there are bytes in the buffer, call uartKBHit(). To retrieve the next byte in the buffer, call uartGetByte().

Like all managers, the UART Manager has an initialization function that must be called at reset to put the manager in a default state.

4.1.8 TIMER MANAGER

The Timer Manager provides the application and other managers with timer functionality by providing a method of establishing timer events and registering callbacks when those events reach their terminal count. The Timer Manager provides 500-microsecond resolution (although this resolution is configurable via the Microcontroller API). The Timer Manager uses a single timer unit on the MCU. The Timer manager is capable of registering 16 event timers. Each event timer is 16 bits for a maximum of 32,768 seconds. An event timer can either be a continuous timer or a single-shot timer. A continuous timer event is started by "registering" the event using a function call. The function call provides the timer count and a callback function that is called when the event reaches the terminal count. The function returns a timer "handle" which is used to refer to the timer event and remove the timer event. Similar functions exist to register single-shot and elapsed-time events. For details on the Timer Manager functions, refer to Section 5.0.10.

4.1.9 INTERRUPT MANAGER

The Interrupt Manager is used to control the enabling and disabling of interrupts, and it ensures that interrupts cannot be inadvertently enabled in nested function calls.

4.1.10 RADIO MANAGER

The radio abstraction API provides low-level access to the XE1231 radio IC. Generally, these functions would be used by the protocol stack to control the radio. The application would not likely need to call these functions directly, except for the case of automated testing in the manufacturing stage.

5.0 RF PROTOCOL STACK DETAILS

The following sections document the details of the protocol stack.

5.0.1 APPLICATION LAYER

Detailed Description:

The Application Layer is user defined, but it typically does the following:

- Initializes the hardware and software managers
- Initializes Application-Layer variables and data structures
- Reads the MAC ID chip and sets the MAC ID of the system
- Passes the address of the Application Layer master loop to MacMain:
  Do Forever MacMain() PhyMain() AppMain() End Do

The MAC Layer main loop will run to completion then call the Application Layer main loop ad infinitum. The Application Layer main loop is completely user defined. A typical sensor node application will set a timer and periodically read sensors and transmit the sensor data to another node using MAC-Layer function calls. The Application Layer may also listen for messages from other nodes. The MAC Manager will notify the Application Layer via the Event Manager when packet transmissions are complete, packet reception has occurred or some type of error event occurs.

The sample application that is provided will do one of two functions based upon a switch setting on the battery board. A Logic 1 on Switch 1 of the dip switch means that the node will be a sensor node. A Logic 0 means that the node will be a gateway or master node. If it is a sensor node, the app reads the MAC ID chip and sets the MAC ID to the value read from the MAC ID chip. If it is a gateway/ master node, the MAC ID is set to 0.

The gateway or master node will set the radio for reception. When a valid packet is received, the packet is output to the USB port on the battery board, the buzzer is sounded and the green LED on the battery board is flashed.

The sensor node sets a timer event using the Timer Manager. When a timer event occurs, the Timer Manager will call the application callback which, in turn, causes the start of a sensor read. The sensors (temperature, light and battery voltage) are read via the ADC. The result is inserted into a fixed -length message and transmitted to the gateway/master through a MAC Manager function call.

This is only one example of how to implement a wireless sensor application. The potential applications are limitless. Most of the resources of the RL78 MCU are available for the application to use.

Macros:

#define ADC_ARRAY_SIZE 4
#define ADC_READ_ITERATIONS 20
#define ADC_ARRAY_INDEX_MASK 0x03
#define ADC_MULTIPLIER 3.0/256

Enumerations:

Enum
 { AppActionNone, AppActionTransmit, AppActionReceive, AppActionProcessReceiver }

Functions:

void updateLCD (void)
double calcADCVoltage (uint16_t rawADCReading)
void extractFractionalFromFloat (double adcVoltage, uint16_t *integerPart, uint16_t
  *fractionalPart)
void intToDecString (unsigned int val, char *stringOut, char decPlaces, char isSigned)
void AppPacketSniff (void)
void AppException (TException e)
void AppEvent (TEventType event)
void AppTransmitPacket (void)
void AppAdcRead (void)
void AppMain (void)
void AppInitialize (void)
void main (void)

Variables:

__root const uint8_t opbyte0 = 0xEFU
__root const uint8_t opbyte1 = 0x7FU
__root const uint8_t opbyte2 = 0xE8U
__root const uint8_t opbyte3 = 0x04U
uint8_t appMessageNumber
TMessageComponent appMessageComponent
uint8_t appMessage [64]
uint8_t incomingMessageBuffer [64]
enum { ... } appAction
uint8_t sniffPackets
uint16_t txPacketCount
uint16_t rxPacketCount
uint8_t adcValsIndex
uint8_t adcCount
uint8_t adcConversionCount
uint8_t adcIndexMask
uint8_t adcComplete
uint16_t adcVals [ADC_ARRAY_SIZE]

5.0.2 MESSAGE MANAGER API

Detailed Description:

The Message Manager is responsible for assembling messages for the different components of the protocol, including the application. Generally, the function that owns the message would request a message ID from the Message Manager, and that ID would be passed to subordinate layers which would use that ID to add additional information to the message.

Data Structures:

struct node
Defines the nodes for a linked list of data buffers.

Typedefs:

typedef struct node TMessageComponent
Type definition for buffer list item for the Message Manager.

Enumerations:

enum TMessageComponentType
{
kMACMessageComponent = 0,
kNETMessageComponent = 1,
kAppMessageComponent = 2
}
Defines the types of message components. This enumeration can be expanded to support user-defined types.

Functions:

void msgClear (uint8_t msgNo)
Clears the message list by setting the list head to NULL.
void msgAdd (TMessageComponent *node, uint8_t msgNo)
Adds a new item to the beginning of the message list.
TMessageComponent * msgGetHead (uint8_t msgNo)
Gets the address to the head of the message list.
TMessageComponent * msgGetNext (TMessageComponent *currentNode)
Gets the address of the next message in the list (front to back).
uint8_t msgGetCount (uint8_t msgNo)
Gets the total number of bytes to send.
void msgInitialize (void)
Initializes the Message Manager.
void msgRelease (uint8_t msgNo)
Releases a managed message.
uint8_t msgGetNew (void)
Gets the handle to a new message.

Function Documentation:

void msgAdd (TMessageComponent * node, uint8_t msgNo)
Adds a new item to the beginning of the message list.

Parameters:

Node A pointer to the TMessageComponent to be added
msgNo Indicates which managed message to perform the operation on

This function is called to add a list item to the head of the list. The caller is responsible for providing storage for the list item. The Message Manager only stores a pointer to the list item.

void msgClear (uint8_t msgNo)
Clears the message list by setting the list head to NULL.

Parameters:

msgNo Indicates which managed message to perform the operation on

This function is called to clear the message list.
uint8_t msgGetCount (uint8_t msgNo)
Gets the total number of bytes to send.

Parameters:

msgNo Indicates which managed message to perform the operation on

This function traverses the message tree and calculates the size of the entire message as it is currently configured.

TMessageComponent* msgGetHead (uint8_t msgNo)
Gets the address to the head of the message list.

Parameters:

msgNo Indicates which managed message to perform the operation on

This function is called to get the head of the message list.

uint8_t msgGetNew (void)
Gets the handle to a new message.

This function is called to allocate one of the managed messages to the caller. A very simple management system is used that allows the caller to call ReleaseMessage to free the message up for another use. If all messages are being used, this function returns 0xFF. A return value of 0 is a valid message handle.

NOTE: TESTS FOR FAILURE MUST TEST AGAINST A RETURN VALUE OF 0xFF.

TMessageComponent* msgGetNext (TMessageComponent * currentNode)
Gets the address of the next message in the list (front to back).

Parameters:

currentNode Current node to use as a reference to get the next node

Return Values:

Pointer to the next node in the list (NULL if at list end).

This function is called to get the address of the next message in the list. Typically, this is used in conjunction with MessageGetHead() to traverse the message list. First MessageGetHead is called to get the pointer to the head of the list. Then MessageGetNext is called in a loop to get the next node based upon the last one returned by either MessageGetNext or MessageGetHead. When MessageGetNext() returns a NULL, the end of the list is reached.

void msgInitialize (void)
Initializes the Message Manager.

This should be called before any calls to the Message Manager component are made. Otherwise, the internal data structures are not initialized, and their values are not guaranteed.

void msgRelease (uint8_t msgNo)
Releases a managed message.

Parameters:

msgNo Indicates which managed message to perform the operation on

This function is called to release one of the managed messages.

5.0.3 MAC LAYER MANAGER API

Detailed Description:

The MAC Manager (layer) is responsible for sending and receiving packets using the PHY. The MAC Manager is analogous to the MAC layer in the OSI standard 7-layer stack. It provides the ability to send messages to addressed nodes and to optionally receive acknowledgements for those messages. If an acknowledgement is expected and not received, the MAC layer will retry the message.

Data Structures:

struct TMACMsgObject
Defines the internal properties of a TMACMsg. union TMACMsg.
Defines a MAC-layer message.

Typedefs:

typedef uint32_t TMACAddress
Defines the MAC-layer address type.

Enumerations:

enum TMACPacketType { kMACUniAck = 0, kMACUniNoAck = 1, kMACMulti = 2, kMACACK = 3 }
Defines the MAC-layer message types.

enum TMACCallbackType { kMACRxDoneCallback, kMACTxDoneCallback, kMACErrorCallback }
Defines the callback functions that can be assigned.
enum TMACErrorType { kMACErrorNone, kMACErrorTX, kMACErrorPHY, kMACErrorNoAck }
Defines the MAC error types.
enum TMACAction { kMACActionNone, kMACActionSendACK, kMACActionReceiveACK, kMACActionReceivePacket, kMACActionSendPacket }
Defines the MAC action types.
enum TMACPacketState { kMACPacketIdle, kMACPacketTxWait, kMACPacketRxWait, kMACPacketSendingACK, kMACPacketAckWait }
Defines the MAC packet engine status types.

Functions:

void macRegisterErrorCB (TErrorCallback func)
Error callback.

Configuration Functions:

void macRegisterCB (TMACCallbackType cbno, TCallback func)
Registers a MAC-layer callback function.
TCallback macInitialize (TCallback appfunc)
Initializes the MAC layer.
void macGetAddress (TMACAddress *addr)
Gets the local MAC address.
void macSetAddress (TMACAddress addr)
Sets the local MAC address.
TMACPacketState macGetEngineStatus (void)
Gets the internal packet engine status.
void macPacketMgrAckTimeout (void)
Callback function for ACK timeout.
void macPacketMgrRxDone (void)
Callback function for received packet.
void macPacketMgrTxDone (void)
Callback function for packet sent.
void macPacketMgrError (TEventType error)
Callback function for error generated by PHY layer.
void macMain (void)
macMain

Packet Functions:

uint8_t macSendPacket (TMessageComponent *node, TMACPacketType pkttype, TMACAddress destaddr, uint16_t ackTimeout, uint8_t ackRetries, uint8_t msgNo, TTxMode mode)
Sends a generic packet using the MAC layer.
void macSendAppPacket (uint8_t *payloadBuffer, TMACPacketType pktType, TMACAddress destAddress, uint16_t ackTimeout, uint8_t ackRetries, TTxMode mode)
Sends an application packet using the MAC layer.
void macReceivePacket (unsigned char *pkt, uint8_t bufferLength, TRxMode mode)
Puts the MAC layer into receive mode.
uint8_t macGetRXBufferIndex (void)
Returns an index to the next byte in the receive buffer after the MAC bytes.

uint8_t macIsReadyToTransmit (void)
Returns a 1 if it is OK to transmit; a 0 if not.
uint8_t macIsReadyToReceive (void)
Returns a 1 if it is OK to receive; a 0 if not.

The MAC Manager supports three valid packet types:

-# kMACUniAck - Acknowledged Packet
-# kMACUniNoAck - Unacknowledged Packet
-# kMACMulti - Multicast Packet

Function Documentation:

TCallback macInitialize (TCallback appfunc)
Initializes the MAC layer.

Parameters:

Appfunc	Pointer to the appMain() function

This function is called to initialize the MAC layer. It stores the appfunc pointer in PrivateData.appMain, which is called in the macMain function.

This function returns a pointer to macMain. When the application layer has finished initialization on reset, it transfers control to that function. In this way, we provide loose coupling.

void macReceivePacket (unsigned char * pkt, uint8_t bufferLength, TRxMode mode)
Puts the MAC layer into receive mode.

Parameters:

*pkt	Pointer to the packet structure that contains the packet info
bufferLength	Length of the receive buffer mode–specifies frequency hopping or single-channel receive mode

This function initiates the MAC layer packet reception machine. It is a non-blocking call. When it returns, the SX1231 will be in receive mode, scanning channels looking for a transmission. The MAC layer state will be kMACPacketRxWait. The protocol and radio will continue in this state until one of the following termination events occurs:

Termination Events:

1. Valid Packet Received - The registered callback function for a received packet is called, and the radio is set to IDLE.

2. Packet Error - A packet start was detected, but the packet failed CRC. The registered callback for exceptions is called, and the radio is set to IDLE.

3. Transmission - If macSendPacket or macSendAppPacket is called, the reception is terminated.

Once reception is terminated, macReceivePacket must be called again to restart reception. If you are using the NET layer, macReceivePacket() is called automatically when you call netReceivePacket().

void macRegisterCB (TMACCallbackType cbno, TCallback func)
Registers a MAC-layer callback function.

Parameters:

cbno	Identifies which event callback is being provided
*func	Pointer to the actual callback function

This function registers a callback function for the MAC layer. Callback functions are identified by (cbno) and point to the function (*func). These callback functions are used to notify other layers/ classes that an event has happened in the PHY layer/class.

void macSendAppPacket (uint8_t * payloadBuffer, TMACPacketType pktType, TMACAddress destAddress, uint16_t ackTimeout, uint8_t ackRetries, TTxMode mode)
Sends an application packet using the MAC layer.

Parameters:

*payloadBuffer	Pointer to the buffer containing app data
pktType	Packet type (see description)
destAddress	Destination address for the packet
ackTimeout	The amount of time to wait for an acknowledgement
ackRetries	The maximum number of times to retry an ACK packet
mode	Specifies frequency hopping or single-channel receive mode

This function is called to send an application packet using the MAC layer. The main difference between this function and macSendPacket() is that the message number and message component are not provided. Internally, the MAC layer reserves a component and number for application messages. This greatly simplifies sending packets from the application because now the application does not need to make any calls to the Message Manager. In fact, the Message Manager operation is completely hidden from the application now.

The MAC layer provides point-to-point connectivity with other nodes in the network using a unique 32-bit address.

The local MAC message is configured based on the information passed to the function. Then the PHY is called to transmit the complete packet. When the PHY is done, macPacketMgrRxDone() is called by the PHY to notify the MAC that the packet is sent. At that point, the packet engine will take over and manage the rest of the MAC's responsibility regarding the packet. If the packet requires an ACK; a timer is started, and automatic retransmissions will be attempted when the timer expires. If no ACK is required or the ACK is received, the MAC will call maccbPacketTxDone() to notify the registered subscriber that the packet was successfully sent.

Valid pktType values:

kMACUniAck - Acknowledged Packet
kMACUniNoAck - Unacknowledged Packet
kMACMulti - Multicast Packet

The function is blocking and will maintain control of the processor until the packet transmission is complete.

uint8_t macSendPacket (TMessageComponent * node, TMACPacketType pkttype, TMACAddress destaddr, uint16_t ackTimeout, uint8_t ackRetries, uint8_t msgNo, TTxMode mode)
Sends a generic packet using the MAC layer.

Parameters:

node	Caller maintained entry for the message list
pkttype	Packet type (see description)
destaddr	Destination address for the packet
ackTimeout	The amount of time to wait for an acknowledgement
ackRetries	The maximum number of times to retry an ACK packet
msgNo	The managed message that is being sent
mode	Specifies frequency hopping or single-channel receive mode

Return Values:

Always returns 1 in current revision.

This function is called to send a packet using the MAC layer. The MAC layer provides point-to-point connectivity with other nodes in the network using a unique 32-bit address.

The local MAC message is configured based on the information passed to the function. Then, the PHY is called to transmit the complete packet. When the PHY is done, macPacketMgrRxDone() is called by the PHY to notify the MAC that the packet is sent. At that point, the packet engine will take over and manage the rest of the MAC's responsibility regarding the packet. If the packet requires an ACK; a timer is started, and automatic retransmissions will be attempted when the timer expires. If no ACK is required or the ACK is received, the MAC will generate a packet sent event.

Valid pktType values:

kMACUniAck - Acknowledged Packet
kMACUniNoAck - Unacknowledged Packet
kMACMulti - Multicast Packet

The function is blocking and will maintain control of the processor until the packet transmission is complete.

5.0.4 PHY LAYER MANAGER API

Detailed Description:

The PHY layer is responsible for sending and receiving packets using the radio. It is analogous to the PHY layer in the OSI 7-layer protocol reference stack. It does not provide assured delivery. It can be used directly by the application when a very simple protocol is being built or when the application has legacy requirements that aren't filled by the higher protocol stack elements.

The current distribution contains a receiver-synchronized, 26-channel FHSS algorithm that can be used in addition to a single-frequency DTS mode of operation.

Macros:

#define SYNC_PACKET_ID 0x10
#define RENDEZVOUS_PACKET_ID 0x11
#define SYNC_COMMAND_TIMEOUT 400

Typedefs:

typedef uint8_t PHYAPI
PHYAPI calls return uint8_t

Enumerations:

enum TPHYPacketState { pesRXWAIT, pesSYNCSTART, pesRXSTART, pesTXSTART, pesOFF }
Packet engine states.
enum TSyncMode { SYNC_MODE_STAR_NETWORK, SYNC_MODE_POINT_TO_POINT, SYNC_MODE_SLEEPER_CELL }
Frequency hopping synchronization method.
enum TSyncState { SYNC_NOT_IN_SYNC, SYNC_IN_SYNC }
enum TPreambleType { preSHORT, preLONG }
RF Preamble types.
enum TRxMode { rxmFHSS, rxmDTS }
Receive modes (FHSS = Frequency Hopping, DTS = Single Channel)
enum TTxMode { txmFHSS, txmDTS }
Transmit mode (FHSS = Frequency Hopping, DTS = Single Channel)
enum TPHYCallbackType { kPHYRxDoneCallback, kPHYTxDoneCallback }
Defines the callback functions that can be assigned.
enum TTxPower { kTxPower0dBm, kTxPower5dBm, kTxPower10dBm, kTxPower13dBm, kTxPower17dBm }
Defines available transmit powers.

Configuration Functions:

Functions used to configure the PHY layer:

PHYAPI phyIdle (void)
This function sets the radio mode to idle.
void phyRegisterCB (TPHYCallbackType cbno, TCallback func)
This function registers the phyTxDone and phyRxDone callbacks.
PHYAPI phySleep (void)
This function sets the radio mode to sleep.
PHYAPI phyOff (void)
This function sets the radio mode to off.
PHYAPI phyWake (void)
This function wakes the radio from an off state.
PHYAPI phyInitialize (void)
This function initializes the PHY layer.
PHYAPI phySetTxChannel (uint8_t channel)
Sets the radio transmit channel.
PHYAPI phySetRxChannel (uint8_t channel)
Sets the radio receive channel.
void phySetHopTable (uint8_t tableIndex)
Selects one of 256 possible hop tables.

void phySelectRadioPattern (uint8_t patternNo)
Sets the radio DTS pattern to one of four (0-3) preset byte sequences.
void phyDefineRadioPattern (uint8_t patternNo, uint8_t *bytes, uint8_t length)
Defines a custom DTS pattern in one of the four available slots.
void phySetTransmitPower (TTxPower power)
Sets the radio power.
uint8_t phyReadDIPSwitch (void)
Reads the DIP switches on the evaluation board.
void phySetManufID (uint32_t ID)
Sets the manufacturer ID.
void phyDisableMIDFiltering (void)
Disables manufacturer ID filtering.
void phyEnableMIDFiltering (void)
Enables manufacturer ID filtering.

Send/Receive Functions:

Functions used to send and receive packets, and to obtain information about the packet engine are as follows:

uint8_t * phyGetRxBuffer (void)
Gets a pointer to the receive buffer.
uint8_t * phyGetSniffBuffer (void)
Gets a pointer to the buffer containing the sniffed packet.
uint8_t phyGetSniffCount (void)
Returns current sniff count.
PHYAPI phySendPacket (TMessageComponent *node, uint8_t msgNo, uint8_t encryptionOn)
Sends a packet over the radio.
PHYAPI phyReSendPacket (uint8_t msgNo, uint8_t encryptionOn)
Sends the last packet over the network.
PHYAPI phyReceivePacket (uint8_t *PacketSDU, uint8_t bufferLength)
Begins listening for a packet on the network.
TPHYPacketState phyGetPktState (void)
Gets the current packet engine state.
uint8_t phyGetRxPacketLen (void)
Gets the packet length of a received packet.
uint8_t phyGetRxBufferIndex (void)
Returns an index to the next byte in the receive buffer after the PHY bytes.
uint8_t phyGetRxChannel (void)
Gets the current receive channel number.
uint8_t phyGetTxChannel (void)
Gets the current transmit channel number.
uint8_t phyIsReadyToTransmit (void)
Identifies if the PHY is ready to send a message.
uint8_t phyIsReadyToReceive (void)
Identifies if the PHY is ready to receive a message.

Function Documentation:

uint8_t* phyGetRxBuffer (void)
Gets a pointer to the receive buffer.

Return Values:

Pointer to Receiver Buffer -This routine is called to retrieve a pointer to the buffer that the PHY uses to store a received packet. Using this pointer together with index offsets achieved from the individual layer managers themselves, the caller can access the context-specific application in the buffer. (See also: phyGetRXBufferIndex.)

uint8_t* phyGetSniffBuffer (void)
Gets a pointer to the buffer containing the sniffed packet.

Pointer to Buffer - This routine is called to retrieve a pointer to the buffer that the PHY uses to store a transmitted or received packet. Using this pointer and index offsets achieved from the individual layer managers themselves, the caller can access the context-specific application in the buffer.

PHYAPI phyIdle (void)
This function sets the radio mode to IDLE.

In this mode, the radio is off, but the oscillator is still running. The ISR packet machine is disabled.

PHYAPI phyInitialize (void)
This function initializes the PHY layer.

All pointers are set to NULL. The packet engine state is set to pesOFF. Local callback functions are registered with the ucAPI layer.

PHYAPI phyReceivePacket (uint8_t * PacketSDU, uint8_t bufferLength)
Begins listening for a packet on the network.

Parameters:

*PacketSDU	Pointer to the buffer that holds the payload data
bufferLength	Length of the buffer; used to prevent buffer overwrites

Return Values:

Returns a 0 if there was a problem starting the receiver and a 1 if the receiver started successfully.

This function is called to receive a packet. It puts the radio in receive mode, and enables the pattern interrupt and the ISR-driven RX packet machine. The packet is received, and the SDU is decoded and CRC verified. Once a valid packet is received by the packet machine, cbMACRXDone() is called.

PacketSDU is a variable that is maintained in the MAC layer. Pointers to those variables are passed in this function so that the packet machine can populate them when the packet is complete and valid.

The way that the receiver works depends upon the preamble type. If the preamble type is short, then the receiver is operated in DTS (single-channel) mode. If the preamble type is long, then the receiver is operated in FHSS mode.

In DTS mode, the receiver will operate on a single channel. When this function returns to the caller, the receiver will be programmed to look for a preamble of 4 bytes. When that preamble is detected, an interrupt will occur. In the pattern ISR, the radio will be reprogrammed to look for the actual start of packet sequence, and then the ISR will go into a hard loop looking for the start sequence. If the start sequence is found, a packet is received. If it is not found within the preset period of time, the radio is

reprogrammed to look for the preamble, and the ISR returns.

In FHSS mode, the receiver will operate the same as in DTS mode except that an additional interrupt will be occurring every 1.5mS to force the radio to scan a new channel. The application programmer should keep in mind that the pattern ISR will keep control of the processor once a preamble is detected until either a fixed time has elapsed with no start-of-packet detection or a packet is fully received. During this time, no other interrupt functions will work, including event timers. [See also: phySendPacket() and phyReSendPacket()]

void phyRegisterCB (TPHYCallbackType cbno, TCallback func)
This function registers the phyTxDone and phyRxDone callbacks.

Parameters:

cbno	Determines the type of callback to be registered
func	Pointer to the callback function

This function is called by the higher-level protocol to register callbacks for phyTxDone and phyRxdone.

PHYAPI phyReSendPacket (uint8_t msgNo, uint8_t encryptionOn)
Sends the last packet over the network.

Parameters:

msgNo	Indicates which managed message we are sending
uint8_t	encryptionOn indicates the use of AES encryption

This function is called to resend the last packet sent by the PHY layer. Calls to this function are only valid after phySendPacket() has been called at least once. The preamble type determines how the packet is sent. A short preamble will cause the packet to be sent in DTS (single-channel) mode. A long preamble will cause the packet to be sent in FHSS mode. [See also: phyReceivePacket() and phySendPacket()]

PHYAPI phySendPacket (TMessageComponent * node, uint8_t msgNo, uint8_t encryptionOn)
Sends a packet over the radio.

Parameters:

node	Caller-maintained entry for the message list
msgNo	Indicates which managed message we are sending
uint8_t	encryptionOn indicates the use of AES encryption

When called, this function will initiate the transmission of a complete packet which is comprised of message fragments coordinated by the Message Manager. The preamble type determines how the packet is sent. A short preamble will cause the packet to be sent in DTS (single-channel) mode. A long preamble will cause the packet to be sent in FHSS mode. [See also: phyReceivePacket()]

void phySetHopTable (uint8_t tableIndex)
Selects one of 256 possible hop tables.

Parameters:

tableIndex Indicates which hop table to use

Hop tables are generated procedurally at runtime using a linear feedback shift register. Our algorithm allows up 256 different tables to be generated with minimal cross-correlation.

PHYAPI phySetRxChannel (uint8_t channel)
Sets the radio receive channel.

Parameters:

channel Determines the receive channel of the radio

Returns:

1 if successful; 0 if the channel is not valid. This function sets the radio channel. Valid channels depend upon the operating band. (For the US band, valid channels are 0-31. For the European band, valid channels are 0-35.)

PHYAPI phySetTxChannel (uint8_t channel)
Sets the radio transmit channel.

Parameters:

channel Determines the transmit channel of the radio

Returns:

1 if successful; 0 if the channel is not valid. This function sets the radio transmit channel. Valid channels depend on the operating band. (For the US band, valid channels are 0-31. For the European band, valid channels are 0-35.)

PHYAPI phySleep (void)
This function sets the radio mode to sleep.

In this mode, the radio is off—including the oscillator. The ISR packet machine is disabled.

5.0.5 RADIO ABSTRACTION API

Detailed Description:

This manager is provided in library form. The radio abstraction API provides low-level access to the SX1231 radio IC. Generally, these functions would be used by the protocol stack to control the radio. The application would likely not need to call these functions directly, except in the case of automated testing in the manufacturing stage.

Data Structures:

struct freq_t

Macros:

#define NUM_CHANNELS 32

Enumerations:

enum TRFRate { xer9_6K, xer19_2K, xer38_4K }
Defines all possible data rates for the radio.
enum TRFTxPower { kRFTxPower0dBm, kRFTxPower5dBm, kRFTxPower10dBm, kRFTxPower15dBm }
Defines possible transmit power settings available.

RF Configuration Functions:

These functions are used to initialize and configure the radio:

void rfSetMode (uint8_t mode)
Sets the operating mode off.
void rfInitialize (void)
Initializes the SX1231.
void rfSetTXDeviation (unsigned char dev)
Sets the SX1231 deviation.
void rfSetDataRate (TRFRate rate)
Sets the SX1231 data rate.
TRFRate rfGetDataRate (void)
Gets the SX1231 data rate.
void rfSetupRx (unsigned char channel)
Sets the RX channel operating parameters. Now it is just the channel.
void rfSetChannel (uint8_t channel)
Sets the receiver channel and bandwidth.
uint8_t rfSetTxChannel (uint8_t channel)
Sets the receiver channel and transmitter deviation.
void rfSetupTx (uint8_t channel, TRFTxPower pwr)
Sets the TX channel operating parameters. For now, they are channel and power.
void rfSetTxPower (sint8_t power)
void rfSetPA (uint8_t onOff)
Turns the power amp on and off.
void rfUseAlternatePins (void)
Forces RadioAPI to use P6.1-P6.2 instead of P9.1 and P9.2.
void rfUseNormalPins (void)
Forces RadioAPI to use P9.1-P9.2 instead of P6.1 and P6.2.
void rfDoNotUsePowerPins (void)
Forces RadioAPI not to use P7.0, P7.1, and P9.3. Disables power control. Do not use with PA.
void rfUsePowerPins (void)
Forces RadioAPI to use P7.0, P7.1, and P9.3 for power control.
freq_t rfGetChannelFreq (uint8_t channel)
Gets the channel frequency in Hz.
void rfSetDataPattern (uint8_t patternCount, uint8_t *patternBuffer)
Sets up the pattern used for detection by the SX1231 receiver.
void rfSendDataByte (unsigned char byte2send)
Sends a data byte over the SX1231 radio.

Register Access Functions:

void rfWriteRegister (uint8_t reg, uint8_t regdata)
Writes a SX1231 register.
unsigned char rfReadRegister (unsigned char reg)
Reads a SX1231 register.
uint8_t rfGetMode (void)
Gets the operating mode of the SX1231.

Function Documentation:

uint8_t rfGetMode (void)
Gets the operating mode of the SX1231.

Valid modes are:

xeSLEEP - SX1231 is in Sleep Mode
xeIDLE - SX1231 is in IDLE Mode
xeTX - SX1231 is in TX Mode
xeTXNOMOD - SX1231 is in TX Mode with no modulation (used for testing)
xeRX - SX1231 is in RX Mode

void rfInitialize (void)
Initializes the SX1231.

This function initializes the registers in the SX1231 that are always the same, regardless of the function of the design. This function also configures the IO interface of the radio and initializes it to an inactive state.

unsigned char rfReadRegister (unsigned char reg)
Reads a SX1231 register.

Parameters:

reg	Register to be programmed

This function reads a register in the SX1231. No checking is done on "reg" or "regdata", so it is imperative that the programmer ensures that these variables are valid. The value of the register is returned by the function.

void rfSetChannel (uint8_t channel)
Sets the receiver channel and bandwidth.

Parameters:

channel	Receiver channel; this is an indexed channel of 0-35

Return Values:

1=success, 0=bad Channel specified

This function sets the receiver channel and configures the receiver bandwidth based upon the channel and band. If the channel is not valid for the band, then the function returns a 0; otherwise it returns a 1.

void rfSetDataPattern (uint8_t patternCount, uint8_t * patternBuffer)
Sets up the pattern used for detection by the SX1231 receiver.

Parameters:

patternCount	The number of pattern bytes to use (valid values are 1-4)
*patternBuffer	Pointer to the buffer that holds the pattern bytes

This function sets the pattern byte count and the pattern bytes in the XE1203.

void rfSetDataRate (TRFRate rate)
Sets the SX1231 data rate.

Parameters:

rate	RF data rate for the SX1231 (type is xeRATE).

This function sets the data rate of the radio. To do this, both the SX1231 and the SPI port have to be configured. The following data rates are supported:

xer9_6K
xer19_2K
xer38_4K

void rfSetMode (uint8_t mode)
Sets the operating mode of the SX1231.

Parameters:

mode	SX1231 operating mode (type is TRFMode)

This function sets the operating mode of the radio. Valid modes are:

xeSLEEP - SX1231 is in Sleep Mode
xeIDLE - SX1231 is in IDLE Mode
xeTX - SX1231 is in TX Mode
xeTXNOMOD - SX1231 is in TX Mode with no modulation (used for testing)
xeRX - SX1231 is in RX Mode

void rfSetPA (uint8_t onOff)
Turns the power amp on and off.

Parameters:

onOff 0=Power Amp Off; 1= Power Amp On

This function will set the power state of the power amp.

void rfSetTXDeviation (unsigned char dev)
Sets the SX1231 deviation.

Parameters:

dev Deviation in 1kHz steps

This function sets the transmit deviation of the radio. Deviation is determined as follows:

deviation(khz) = dev * 1kHz

void rfSetTxPower (sint8_t power)
Sets the transmitter power level from -18 to + 17db in 1db steps.
void rfWriteRegister (uint8_t reg, uint8_t regdata)
Writes a SX1231 register.

Parameters:

reg Register to be programmed
regdata Data to be programmed

This function writes a register in the SX1231. No checking is done on "reg" or "regdata", so it is imperative that the programmer ensures that these variables are valid.

5.0.6 EVENT MANAGER

Detailed Description:

This manager is provided in source code form. This very simple manager provides a service container that allows an application to receive event notifications from the various API's by registering a callback function via evtRegisterGlobalEventCallback(). An application may also fire events using evtEvent().

If a callback has not been registered, events will be ignored. This very simple manager provides a service container that allows an application to receive event notifications from the various API's by registering a callback function via evtRegisterGlobalEventCallback(). An application may also fire events using evtEvent().

If a callback has not been registered, events will be ignored.

Enumeration Type Documentation:

enum TEventType

Event Types - Not all events will propagate up to the application level. Many of the PHY Rx events are handled internally by the protocol stack.

Enumerator:

kEventPacketSent - A packet was successfully sent.
kEventPacketReceived - A packet was successfully received.
kEventNETMultiPacketReceived - A multicast packet was successfully received.
kEventNETFormationPingReceived - A formation ping was received.
kEventNETErrorRxNETType - A packet was received with an invalid NET message type.
kEventNETErrorRouteUnavailable - A packet cannot be sent because no up-tree route is available.
kEventMACMultiPacketReceived - A multicast packet was successfully received.
kEventMACErrorNoAck - No ACK was received for a previously sent packet which requested an ACK.
kEventPHYBeginReceive - Receiver is set up and about to begin receiving.
kEventPHYEndReceive - Receiver finished receiving (does not imply packet was received).
kEventPHYBeginTransmit - A packet transmission is about to begin.
kEventPHYEndTransmit - A packet transmission has ended (does not imply packet was sent).
kEventPHYRxPacketSniff - A packet has been received at the PHY level.
kEventPHYTxPacketSniff - A packet has been sent at the PHY level.
kEventPHYErrorRxTimeout - Receive timeout occurred while waiting for bytes.
kEventPHYErrorRxBadCRC - A packet was received with an invalid CRC.
kEventPHYErrorRxBadLength - A packet was received with an invalid Length.
kEventPHYErrorRxBadMID - A packet was received with a non-matching Manufacturer ID.
kEventPHYErrorTxTimeout - Transmit timeout occurred during a packet send.
kEventUser - Starting value for user-defined events.
kEventPacketSent - A packet was successfully sent.
kEventPacketReceived - A packet was successfully received.
kEventNETMultiPacketReceived - A multicast packet was successfully received.
kEventNETFormationPingReceived - A formation ping was received.
kEventNETErrorRxNETType - A packet was received with an invalid NET message type.
kEventNETErrorRouteUnavailable - A packet cannot be sent because no up-tree route is available.
kEventMACMultiPacketReceived - A multicast packet was successfully received.
kEventMACErrorNoAck - No ACK was received for a previously sent packet which requested an ACK.
kEventPHYBeginReceive - Receiver is set up and about to begin receiving.
kEventPHYEndReceive - Receiver finished receiving (does not imply packet was received).
kEventPHYBeginTransmit - A packet transmission is about to begin.
kEventPHYEndTransmit - A packet transmission has ended (does not imply packet was sent).
kEventPHYRxPacketSniff - A packet has been received at the PHY level.
kEventPHYTxPacketSniff - A packet has been sent at the PHY level.
kEventPHYErrorRxTimeout - Receive timeout occurred while waiting for bytes.
kEventPHYErrorRxBadCRC - A packet was received with an invalid CRC.
kEventPHYErrorRxBadLength - A packet was received with an invalid Length.
kEventPHYErrorRxBadMID - A packet was received with a non-matching Manufacturer ID.
kEventPHYErrorTxTimeout - Transmit timeout occurred during a packet send.
kEventUser - Starting value for user-defined events.

5.0.7 MICROCONTROLLER ABSTRACTION API

Detailed Description:

The microcontroller API encompasses all functions necessary to interface with the managed microcontroller resources that are available to the application. This API provides low-level access to the interrupts, SPI, UART and timer. For the most part, these functions are used by other managers to provide higher-level control.

The UART Manager, for example, provides buffered and interrupt-driven transmit and receive services. (Generally, there should not be a need for the application to call these functions directly.)

Enumerations:

enum TInterrupt { intPATTERN, intUART0_TX, intUART0_RX, intUART1_TX, intUART1_RX, intSPIALL, intSPITX, intSPIRX, intTMRF, intTMRFH, intINTP0, intINTP5 }
Defines all possible interrupts that are managed by the API.
enum { spidr9_6K, spidr19_2K, spidr38_4K }
Defines all possible data rates for the SPI.
enum { spimOFF, spimMASTER, spimSLAVE }
Defines all possible modes of the SPI port.
enum TBaudRate { br300, br1200, br2400, br4800, br9600, br19200, br38400 }
Defines all possible data rates of the UART.
enum INTERRUPT_PRIORITIES { PRIORITY_LEVEL_0, PRIORITY_LEVEL_1, PRIORITY_LEVEL_2, PRIORITY_LEVEL_3 }
enum TADCMode { ADC_MODE_SINGLE_ONE_SHOT, ADC_MODE_SINGLE_CONTINUOUS, ADC_MODE_SCAN_ONE_SHOT, ADC_MODE_SCAN_CONTINUOUS }

Interrupt Functions:

These functions are used to manage the interrupts of the microcontroller:

UCAPI ucDisableInterrupt (TInterrupt intno)
Disables hardware interrupt.
UCAPI ucEnableInterrupt (TInterrupt intno)
Enables hardware interrupt.
UCAPI ucINTP0RegisterCallback (TCallback callback)
Registers callback function for INTP0.
UCAPI ucINTP5RegisterCallback (TCallback callback)
Registers callback function for INTP5.
void ucConfigureINTP0 (uint8_t pedge, uint8_t nedge, uint8_t priority)
Configures INTP0 as an interrupt.
void ucConfigureINTP5 (uint8_t pedge, uint8_t nedge, uint8_t priority)
Configures INTP5 as an interrupt.

Function Documentation:

void ucConfigureINTP0 (uint8_t pedge, uint8_t nedge, uint8_t priority)
Configures INTP0 as an interrupt.

Parameters:

edge	1=Rising Edge; 0=Falling Edge
priority	Priority level of interrupt (valid values are 0,1,2)

This function is called to configure INTP0 as an interrupt with the desired edge and priority settings. To enable this interrupt, call ucEnableInterrupt(intINTP0). To disable this interrupt, call ucDisableInterrupt(intINTP1).

void ucConfigureINTP5 (uint8_t pedge, uint8_t nedge, uint8_t priority)
Configures INTP5 as an interrupt.

Parameters:

edge 1=Rising Edge; 0=Falling Edge
priority Priority level of interrupt (valid values are 0,1,2)

This function is called to configure INTP5 as an interrupt with the desired edge and priority settings. To enable this interrupt, call ucEnableInterrupt(intINTP1). To disable this interrupt, call ucDisableInterrupt(INTP5).

UCAPI ucDisableInterrupt (TInterrupt intno)
Disables hardware interrupt.

Parameters:

intno Specifies the interrupt that is to be disabled (type is TInterrupt)

This function is called to disable a microcontroller interrupt.

-Valid intno values:

intUART_TX - UART data transmit interrupt
intUART_RX - UART data receive interrupt
intSPI - Used by the SX1231 for data transfer
intTMRF - TimerF overflow low-byte interrupt

UCAPI ucEnableInterrupt (TInterrupt intno)
Enables hardware interrupt.

Parameters:

intno Specifies the interrupt that is to be enabled (type is TInterrupt)

This function is called to enable a microcontroller interrupt:

-Valid intno values:

intUART_TX - UART data transmit interrupt
intUART_RX - UART data receive interrupt
intSPI - Used by the SX1231 for data transfer
intTMRF - TimerF overflow low-byte interrupt

UCAPI ucINTP0RegisterCallback (TCallback callback)
Registers callback function for INTP0.

Parameters:

callback Pointer to function to be called when the IRQ0 fires

This function is called by an external module to subscribe to the IRQ0 event. The caller provides a pointer to an internal function that will handle the event.

UCAPI ucINTP5RegisterCallback (TCallback callback)
Registers callback function for the INTP5.

Parameters:

callback Pointer to function to be called when the IRQ3 fires

This function is called by an external module to subscribe to the IRQ3 event. The caller provides a pointer to an internal function that will handle the event.

5.0.8 DEBUG MANAGER API

Detailed Description

This manager is provided in source code form. This manager provides a very simple interface by managing the main interrupt mask using a counting semaphore. This allows individual components to enable and disable interrupts at will without having to synchronize to other modules. By using this manager, we always ensure that interrupts are never enabled inadvertently by properly operating code. The semaphore is incremented when interrupts are disabled and decremented when interrupts are enabled. If the semaphore reaches zero in EnableInterrupts(), the global interrupt mask is cleared.

To understand how the interrupt semaphore works, consider the following example:

Typedefs:

typedef void(* ExceptionCallback)(TException)
Exception callback function definition.

Enumerations:

enum TException { kExMessageListClear, kExMessageListAdd, kExMessageGetCount, kExGetNewMessage, kExReleaseMessage, kExPHYTxBufferOverflow, kExPHYRegisterCBInvalid, kExPHYInvalidRadioPattern, kExPHYInvalidTxPower, kExPHYTxInvalidPacketState, kExPHYTxMessageTooLong, kExPHYInvalidPatternLength, kExPHYInvalidTXChannel, kExPHYInvalidRXChannel, kExPHYHoppingNotSupported, kExMACRegisterCBInvalid, kExMACInvalidMACTypeReceived, kExMACPacketEngineStatus, kExMACInvalidAction, kExMACAckTimeoutInvalidState, kExNETPacketEngineStatus, kExNETInvalidAction, kExNETInvalidErrorFromMAC, kExNETInvalidErrorFromPHY, kExUARTRxBufferOverflow, kExUARTTxBufferOverflow, kExUARTTxPacketBufferOverflow, kExEnableInterruptsInvalid, kExDisableInterruptsInvalid, kExInterruptsAlreadyEnabled, kExUnhandledInterrupt, kExRadioInvalidTxPower, kExSPIConfiguration, kExReleasedInvalidTimer, kExNoFreeTimerEvents, kExNoFreeElapsedTimers, kExInvalidElapsedTimer, kExUser }

Exception Types:

Functions

void dbgThrowException (TException ex)
Throws an exception.
void dbgRegisterExceptionCallback (ExceptionCallback func)
Registers a callback function to receive notification of exceptions.

Detailed Description:

This manager is provided in source code form. This very simple manager provides a service container that allows an application to receive notifications of API exceptions by registering a callback function via dbgRegisterExceptionCallback(). An application may also throw exceptions using dbgThrowException().

If a callback has not been registered, exceptions will be ignored.

Enumeration Type Documentation:

enum TException
Exception types.

Enumerators:

kExMessageListClear - Message number exceeded maximum message count.
kExMessageListAdd - Message number exceeded maximum message count.
kExMessageGetCount - Message number exceeded maximum message count.
kExGetNewMessage - Maximum number of messages have been allocated.
kExReleaseMessage - Message number exceeded maximum message count.
kExPHYTxBufferOverflow - Message content overflowed transmit buffer.
kExPHYRegisterCBInvalid - Invalid callback type specified.
kExPHYInvalidRadioPattern - Invalid radio pattern number selected.
kExPHYInvalidTxPower - Invalid transmit power selected at the PHY level.
kExPHYTxInvalidPacketState - Invalid packet state during PHY transmit.
kExPHYTxMessageTooLong - Message being sent is too long.
kExPHYInvalidPatternLength - Pattern length does not match required value (4 by default).
kExPHYInvalidTXChannel - PhySetChannel was called with an invalid TX Channel.
kExPHYInvalidRXChannel - PhySetChannel was called with an invalid RX Channel.
kExPHYHoppingNotSupported - Frequency hopping not supported in the currently selected band.
kExMACRegisterCBInvalid - Invalid callback type specified.
kExMACInvalidMACTypeReceived - Invalid MAC packet type was received in a packet.
kExMACPacketEngineStatus - Invalid packet engine status detected.
kExMACInvalidAction - Invalid MAC action.
kExMACAckTimeoutInvalidState - ACK timeout fired but not waiting for ACK.
kExNETPacketEngineStatus - Invalid packet engine status detected.
kExNETInvalidAction - Invalid NET action.
kExNETInvalidErrorFromMAC - Invalid error sent up from MAC layer.
kExNETInvalidErrorFromPHY - Invalid error sent up from PHY layer.
kExUARTRxBufferOverflow - Receive buffer overflow reading byte from UART.
kExUARTTxBufferOverflow - Transmit buffer overflow sending byte to UART.
kExUARTTxPacketBufferOverflow - Transmit buffer overflow sending packet to UART.
kExEnableInterruptsInvalid - Attempted to enable interrupts within an active ISR.
kExDisableInterruptsInvalid - Attempted to disable interrupts within an active ISR.
kExInterruptsAlreadyEnabled - Attempted to enable interrupts when they are already enabled.
kExUnhandledInterrupt - An unhandled interrupt was triggered. Use call stack to determine which one.
kExRadioInvalidTxPower - Invalid raw radio transmit power selected.
kExSPIConfiguration - Internal error detected during SPI configuration.
kExReleasedInvalidTimer - An invalid timer handle was released.
kExNoFreeTimerEvents - No more timer event slots are available.
kExNoFreeElapsedTimers - No more elapsed timer slots are available.
kExInvalidElapsedTimer - Invalid elapsed timer handle specified or not in use.

kExUser - Starting value for user-defined exceptions.

Function Documentation:

void dbgRegisterExceptionCallback (ExceptionCallback func)
Registers a callback function to receive notification of exceptions.

Parameters:

func Application callback function called when an exception occurs

During development it is highly recommended that you register an exception handler in your application so that you can diagnose them if they appear. Exceptions are considered critical and are usually not recoverable from software. They may result as a consequence of not using the API functions as intended.

Your application exception callback should be defined as follows:

```
void AppException(TException ex)
{
  // set a breakpoint in here to inspect the exception
}
```

5.0.9 INTERRUPT MANAGER API

Functions:

void EnableInterrupts (void)
Globally enables interrupts.
void DisableInterrupts (void)
Globally disables interrupts.
uint16_t GetInterruptCount (void)
Gets the current count of nested interrupt disables.
void EnterISR (void)
Called on entry to an ISR. When in an ISR, the Interrupt Manager will not manipulate the semaphore.
void ExitISR (void)
Called on exit from an ISR. Allows the semaphore to behave normally.
uint8_t inISR (void)
Returns the flag indicating whether we are currently in an ISR.

```
void func1(void)
{
  DisableInterrupts();
  doSomething();
  EnableInterrupts();
}

void main(void)
{
  DisableInterrupts();
  func1();
  doSomethingElse();
```

```
  EnableInterrupts();
}
```

Without the semaphore, func1() would leave interrupts enabled when it exits. Therefore, when doSomethingElse() executes, interrupts are enabled—which is clearly not the intention.

Using a semaphore, let's look at what happens:

1. DisableInterrupts() is called in main(). Interrupts are disabled, and the semaphore is incremented to 1.
2. func1() is called, which calls DisableInterrupts() again. The semaphore is incremented to 2.
3. EnableInterrupts() is called at the end of func1(). The semaphore is decremented to 1, but interrupts are left disabled.
4. doSomethingElse() is called with interrupts disabled, which is the intent.
5. Now that it is zero, and interrupts are enabled, the last call to EnableInterrupts() decrements the semaphore again.

5.0.10 TIMER MANAGER API

Detailed Description:

This manager is provided in library form. The Timer Manager API provides the functions necessary to interface with the built-in timer system. It allows other managers and the application to register cyclic and one-time events that are to be triggered at a programmed delay.

The Timer Manager provides two types of timers: event timers and elapsed timers. The timer is driven from TimerF and the 10MHz main clock. It generates a count every 500uSec. Event timers are interrupt driven. These are used when a piece of code needs an asynchronous notification that a period of time has expired. Event timers can be cyclic or one-time. Event timers can only be used when interrupts are enabled at least part of the time. Elapsed timers are not interrupt driven. Their value can be read using the API at any time. These timers are used to provide timeout functions for code which can operate with interrupts disabled.

Functions:

void tmrInitialize (void)
Initializes the Timer Manager.
uint8_t tmrRegisterEvent (uint16_t ticks, TCallback TCallbackfunc)
Registers a repeating timer event.
uint8_t tmrRegisterOneTimeEvent (uint16_t ticks, TCallback TCallbackfunc)
Registers a one-time timer event.
void tmrReleaseEvent (uint8_t eventno)
Releases a previously registered event.
uint8_t tmrRegisterElapsedTimer (void)
Registers an elapsed timer.
uint16_t tmrGetElapsedTime (uint8_t handle)
Gets the elapsed time in milliseconds for the specified elapsed timer handle.
void tmrReleaseElapsedTimer (uint8_t handle)
Releases a previously registered elapsed timer.
void tmrWait (uint16_t milliseconds)
Waits a specified number of milliseconds.

Function Documentation:

void tmrInitialize (void)
Initializes the Timer Manager.

This function initializes the Timer Manager; setting up the hardware timer and registering the local callback function with the microcontroller layer.

uint8_t tmrRegisterElapsedTimer (void)
Registers an elapsed timer.

Return Values:

Handle	Handle to the newly created elapsed timer (or 255 if no timers are available)

When an elapsed timer is created, it begins counting from 0. Successive calls to tmrGetElapsedTime will return the amount of time in milliseconds which has elapsed since the elapsed timer was created.

uint8_t tmrRegisterEvent (uint16_t ticks, TCallback TCallbackfunc)
Registers a repeating timer event.

Parameters:

ticks	The time in ticks between each event (each tick is 500us)
TCallbackfunc	The function to be called when the timer event occurs

Return Values:

Handle	Handle to the newly created elapsed timer (or 255 if no timers are available)

This function is called to register a repeating timer event. If there are no timer handles available, the function will return 255. Otherwise, the function will return a handle to the newly registered timer event.

Event timers will not work inside of callback functions or ISR's because interrupts are generally not enabled in either of those cases. If a timer is needed in a callback function or ISR, an elapsed timer should be used instead. (See also: tmrRegisterElapsedTimer.)

uint8_t tmrRegisterOneTimeEvent (uint16_t ticks, TCallback TCallbackfunc)
Registers a one-time timer event.

Parameters:

ticks	The time in ticks between each event (each tick is 500uS)
TCallbackfunc	The function to be called when the timer event occurs

Return Values:

Handle	Handle to the newly created elapsed timer (or 255 if no timers are available)

This function is called to register a one-time timer event. If there are no timer handles available, the function will return 255. Otherwise, the function will return a handle to the newly registered timer event.

Event timers will not work inside of callback functions or ISR's because interrupts are generally not enabled in either of those cases. If a timer is needed in a callback function or ISR, an elapsed timer should be used instead. (See also: tmrRegisterElapsedTimer.)

void tmrReleaseEvent (uint8_t eventno)
Releases a previously registered event.

Parameters:

eventno The handle assigned to the event by tmrRegisterEvent()

This function is called to release an event registered with tmrRegisterEvent().

5.0.11 UART MANAGER API

Detailed Description:

This manager is provided in source code form. The UART Manager provides an interrupt-driven, circular-buffered transmit and receive interface for UART0. The UART Manager API provides functions to configure the UART and to access the transmit and receive buffers.

The transmit buffer is 255 bytes. To send a byte to the UART, call uartSendByte(). That byte is put into the transmit buffer, and the transmit interrupt is enabled. The UART0 ISR will then remove that byte from the transmit buffer and send it to the UART. To send a null terminated string, call uartSendString().

The receive buffer is 32 bytes. When the UART0 ISR receives a byte, it puts that byte into the receive buffer. To determine if there are bytes in the buffer, call uartKBHit(). To retrieve the next byte in the buffer, call uartGetByte().

Like all managers, the UART Manager has an initialization function that must be called at reset to put the manager into a default state.

Functions:

void uart0SetBaud (TBaudRate br)
Sets the UART baud rate.
void uart1SetBaud (TBaudRate br)
uint8_t uart0SendByte (uint8_t data)
Sends a byte using the interrupt driven UART.
uint8_t uart1SendByte (uint8_t data)
void uart0StringOut (char *str)
Sends a null terminated string to the UART.
void uart1StringOut (char *str)
uint8_t uart0KBHit (void)
Determines if a byte has been received via the UART.
uint8_t uart1KBHit (void)
uint8_t uart0GetByte (void)
Gets a byte from the UART receive buffer.
uint8_t uart1GetByte (void)
void uart0Initialize (void)
Initializes the UART to a known state.

void uart1Initialize (void)
uint8_t uart0PktOut (uint8_t *str, uint8_t len)
Special function to send a packet to the UART for packet sniffing.
uint8_t uart1PktOut (uint8_t *str, uint8_t len)
Special function to send a packet to the UART for packet sniffing.

Function Documentation:

void uart0Initialize (void)
Initializes the UART to a known state.

This function is called at the beginning of the application (and must be called before any other UART functions) to initialize the UART to a known state. Currently, the UART is configured to the following state by this routine:

Baud rate is 38.4 kbaud
Flow control is none

See also:

uartKBHit()
uartSendByte()
uartReceiveByte()
uartStringOut()

uint8_t uart0KBHit (void)
Determines if a byte has been received via the UART.

The UART receive interrupt service routine uses a ring buffer to store incoming bytes. This function is called to determine if there are any new bytes in that buffer. It returns a 1 if there is at least one byte in the buffer, and a 0 if there are no bytes in the buffer. uartGetByte can be used to retrieve a byte once uartKBHit determines that a byte is available. [See also: uartGetByte()]

uint8_t uart0PktOut (uint8_t * str, uint8_t len)
Special function to send a packet to the UART for packet sniffing.

Parameters:

*str	Pointer to data packet
len	Length of packet in bytes

This function is designed to support packet-sniffing applications. The caller passes a pointer to a raw packet buffer along with the packet length. In typical use, this packet buffer is retrieved from the PHY API—along with its length—using phyGetSniffBuffer() and phyGetSniffCount().

This function will send a "start-of-packet" (0x02) byte, the entire contents of the packet and an "end-of-packet" byte (0x03). The contents of the packet will be encoded to remove special marker byte values (0x00-0x03). [See also: phyGetRxBuffer()]

uint8_t uart0SendByte (uint8_t data)
Sends a byte using the interrupt-driven UART.

Parameters:

data Byte to send

This function is called to send a byte via the UART. The UART is interrupt driven, and it employs a ring buffer to store data waiting to be sent. This function will place the data byte in the ring buffer. No checks are made for overlap of the buffer, so the newest byte always overwrites the oldest. If the oldest byte hasn't been sent yet, then it will be lost.

void uart0SetBaud (TBaudRate br)
Sets the UART baud rate.

Parameters:

br Baud rate (type is TBaudRate)

NOTE: See TBaudRate documentation for details.

This function should be called before the UART is used. It configures the baud rate of the UART.

void uart0StringOut (char * str)
Sends a null-terminated string to the UART.

Parameters:

*str Pointer to null-terminated string

This function is called to send a null-terminated (c-string) string to the UART. It waits for the buffer to clear (i.e. head and tail pointers are the same), and then copies the string into the buffer. Because the function waits for the buffer to clear, it is a blocking function. If, for some reason, the UART interrupts become disabled before this function is called, it will hang waiting for the buffer to clear. No checks are made to ensure that the string is smaller than the buffer size; therefore, the caller should ensure that the string is less than 255 characters.

See also:

uartKBHit()
uartInitialize()
uartReceiveByte()
uartStringOut()

6.0 REGULATORY AGENCY CONSIDERATIONS

6.1 AGENCY IDENTIFICATION NUMBERS

Compliance with the appropriate regulatory agencies is essential in the deployment of all transceiver devices. DLP Design has obtained modular approval for this RF product. As such, an OEM need only meet a few basic requirements in order to utilize their end product under this approval. Corresponding agency identification numbers are listed below:

PART NUMBER	US / FCC	CANADA / IC
DLP-RFS1231	SX9RFS1	5675A-RFS1

6.2 EXTERNAL ANTENNAS

The DLP-RFS1231 is preapproved for use with both its on-board chip antenna and an external antenna (Part No. 0600-00048 made by Laird Technologies). Connection to the external antenna is made via a u.fl connector.

6.3 FCC/IC REQUIREMENTS FOR MODULAR APPROVAL

Any changes or modifications to the DLP-RFS1231's printed circuit board, on-board antenna or pre-approved external antenna could void the user's authority to operate the equipment.

6.4 WARNINGS

Operation is subject to the following two conditions: (1) This device may not cause harmful interference; and (2), this device must accept any interference received, including interference that may cause undesirable operation.

This device is intended for use under the following conditions:

1. The transmitter module may not be co-located with any other transmitter or antenna; and,

2. The module will be approved using the FCC's "unlicensed modular transmitter approval" method.

As long as these two conditions are met, further transmitter testing will not be required. However, the OEM integrator is still responsible for testing their end product for any additional compliance measures necessitated by the installation of this module (i.e. digital device emissions, PC peripheral requirements, etc.).

Note: In the event that these conditions cannot be met (i.e. co-location with another transmitter), then the FCC authorization is no longer valid, and the corresponding FCC ID may *not* be used on the final product. Under these circumstances, the OEM integrator will be responsible for re-evaluating the end product (including the transmitter) and obtaining a separate FCC authorization.

6.5 OEM PRODUCT LABELING

The final end product must be labeled in a visible area with the following text:

"Contains TX FCC ID: SX9RFS1"

6.6 RF EXPOSURE

In order to comply with FCC RF exposure requirements, the antenna used for this transmitter must not be co-located or operating in conjunction with any other antenna or transmitter.

6.7 ADDITIONAL INFORMATION FOR OEM INTEGRATORS

The end user should NOT be provided with any instructions on how to remove or install the DLP-RFS1231. This device will be pre-certified to operate with the antenna models listed below:

- On-board Chip Antenna
- Laird Technologies P/N 0600-00048

7.0 DISCLAMER

Neither the whole nor any part of the information contained herein nor the product described in this datasheet may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder.

This product and its documentation are supplied on an as-is basis, and no warranty as to their suitability for any particular purpose is either made or implied. DLP Design will not accept any claim for damages whatsoever arising as a result of the use or failure of this product. Your statutory rights are not affected.

This product or any variant of it is not intended for use in any medical appliance, device or system in which the failure of the product might reasonably be expected to result in personal injury.

This document provides preliminary information that may be subject to change without notice.

8.0 CONTACT INFORMATION

DLP Design, Inc.
1605 Roma Lane
Allen, TX 75013

Phone: 469-964-8027
Fax: 415-901-4859
Email: support@dlpdesign.com
Internet: http://www.dlpdesign.com