MSP430 RF Applications with the MRF1611CC1100

and

API Reference

Written by

Tom Baugh

Published by

SoftBaugh

Custom Software, Firmware, Hardware, and Project Management Consulting

5400 Laurel Springs Parkway
Suite 1001
Suwanee, GA 30024
Copyrights

Unless otherwise stated, software, text and diagrams written and copyrighted by SoftBaugh or its authors remains the sole property of SoftBaugh or its authors, respectively. You may not modify this material or distribute it, in whole or in part, to any other party, without the specific written permission of SoftBaugh.

Trademark Notices

Texas Instruments, MSP430, MSP430F1611, CC1100, TPS60210, Chipcon, SmartRF Studio, and other MSP430 and device-specific nomenclature are trademarks or registered trademarks owned by Texas Instruments, Inc.

SoftBaugh, MRF1611CC1100, MRF1611, HMRF1611, BRS232, USBL, PrgUSBP, and USBP are trademarks or registered trademarks owned by SoftBaugh, Inc.

IAR Systems, IAR KickStart, EW430, IAR Embedded Workbench, C-SPY and VisualSTATE are trademarks or registered trademarks owned by IAR Systems AB.

HyperTerminal is a trademark or registered trademark owned by Hilgraeve, Inc.

All other trademarks not specifically listed are trademarks or registered trademarks of their respective owners.

No Warranty Provided

All software, hardware, firmware and related documentation is provided "AS IS" and without warranty or support of any kind and SoftBaugh expressly disclaims all other warranties, express or implied, including, but not limited to, the implied warranties of merchantability and/or fitness for a particular purpose. Under no circumstances shall SoftBaugh be liable for any incidental, special or consequential damages that result from the use or inability to use the software, firmware, hardware or related documentation, even if SoftBaugh has been advised of the liability.
# Table of Contents

1. Overview .................................................................................................................. 13
   1.1 MRF1611CC1100 Features .................................................................................. 13
   1.2 Prerequisites ....................................................................................................... 13
   1.3 Source Code Licensing ....................................................................................... 14
   1.4 References .......................................................................................................... 14
   1.5 MRF1611 Host Interface ...................................................................................... 15
   1.6 On-Chip Resources ............................................................................................. 17

2. System Preparation ..................................................................................................... 19
   2.1 Software Installation ........................................................................................... 19
      2.1.1 IAR Embedded Workbench ........................................................................ 20
      2.1.2 USBP Drivers and Applications ................................................................... 20
      2.1.3 MRF1611CC1100 Demo Project ................................................................. 20
   2.2 Terminal Setup ..................................................................................................... 21
   2.3 Hardware Preparation .......................................................................................... 22
      2.3.1 BR232 ......................................................................................................... 22
      2.3.2 Highly Calibrated Antenna ........................................................................ 22
      2.3.3 Install MRF1611CC1100s ........................................................................... 22
   2.4 Jumper Settings .................................................................................................... 23
      2.4.1 Power Options Header ................................................................................ 23
      2.4.2 Analog Options Header ............................................................................... 23
      2.4.3 Digital Options Header ............................................................................... 24
   2.5 Loading the Demo Firmware .............................................................................. 25
      2.5.1 Loading Firmware with PrgUSBP ............................................................... 26
      2.5.2 Loading Firmware with IAR Embedded Workbench ................................... 29
      2.5.3 Using Embedded Workbench to Generate Binaries ................................... 31

3. Terminal Commands ................................................................................................... 33
   3.1 Main Menu Commands ......................................................................................... 33
      3.1.1 ? - Show Menu ........................................................................................... 34
      3.1.2 A - Toggle Address Mode ............................................................................ 34
      3.1.3 C - Toggle CRC Mode ................................................................................ 34
      3.1.4 D - Display Channel Parameters ............................................................... 34
      3.1.5 E - Enter Echo Mode ................................................................................ 35
      3.1.6 F - Set Fixed Length Packets .................................................................... 35
      3.1.7 G - Set Fixed Length Packets With FEC ................................................. 35
      3.1.8 H - Enter Hub Mode ................................................................................... 35
      3.1.9 I - Get CC1100 Register ............................................................................ 36
      3.1.10 K - Change Address ................................................................................ 36
      3.1.11 N - Enter Node Mode ............................................................................... 37
      3.1.12 O - Set CC1100 Register ......................................................................... 37
      3.1.13 P - Enter Ping Mode ............................................................................... 38
      3.1.14 Q - Toggle RSSI/LQI Mode ..................................................................... 38
      3.1.15 R - Enter Rx Mode ................................................................................... 38
      3.1.16 S - Toggle Sequence Number Mode ....................................................... 39
      3.1.17 T - Enter Tx Mode ................................................................................... 39
      3.1.18 U - Enter User Mode ............................................................................... 39
      3.1.19 V - Set Variable Length Packets ............................................................ 40
      3.1.20 W - Toggle Whitening .............................................................................. 40
      3.1.21 Y - Get CC1100 Chip Status ................................................................. 40
      3.1.22 Z - Reset Digital Radio .......................................................................... 41
      3.1.23 0-7 - Set Channel 0 to 7 ........................................................................ 41
   3.2 Hub Menu Commands .......................................................................................... 42
      3.2.1 ? - Show Hub Menu ................................................................................... 43
# MSP430 RF Applications with the MRF1611CC1100

## 4. Firmware Architecture

### 4.1 State Machine Design for MSP430s

- **4.1.1 State Overview**
- **4.1.2 Tasks**
- **4.1.3 State Transitions**
- **4.1.4 State Functions**
- **4.1.5 Client Callbacks, or Hooks**

### 4.2 Demo Modules

- **4.2.1 DigitalRadio_ResetTable.s43**
- **4.2.2 main.c**
- **4.2.3 SerialNumber.c**
- **4.2.4 System.s43**
- **4.2.5 UserApp.c**
- **4.2.6 Library_MRF1611CC1100.r43**

### 4.3 Licensed Modules

- **4.4 Packet Formats**
  - **4.4.1 Basic CC1100 Packet Format**
  - **4.4.2 Variable-Length Packet Formats**
  - **4.4.3 Fixed-Length Packet Formats**

### 5. User Application Demo

- **5.1 Default User Application Firmware**
  - **5.1.1 Application Execution**
  - **5.1.2 State Machine Implementation**

- **5.2 Custom State Machine Application**
  - **5.2.1 Encapsulating the LEDs**
  - **5.2.2 Switch Debouncing**
  - **5.2.3 Freezing the Application**
  - **5.2.4 Encapsulating the ESC Exit**
  - **5.2.5 Reversing the Scroll Direction**

### 6. Tx and Rx Demo

- **6.1 Tx / Rx Architecture**
- **6.2 Default Operation**
  - **6.2.1 Materials**
  - **6.2.2 Preparation**
  - **6.2.3 First Execution**
  - **6.2.4 Packet Format**
  - **6.2.5 Sequence Numbers**
  - **6.2.6 Addressing**
  - **6.2.7 CRC**
  - **6.2.8 Channels**
  - **6.2.9 RSSI/LQI**
  - **6.2.10 Whitening**
  - **6.2.11 Variable Length Packets**
  - **6.2.12 Fixed Length Packets**
  - **6.2.13 Fixed Length Packets with Forward Error Correction (FEC)**

- **6.3 Handheld Mode**

- **6.4 Custom Application Samples**
  - **6.4.1 User Application Callbacks**
  - **6.4.2 Exercising the Callbacks**
  - **6.4.3 OnIdle Callbacks**
  - **6.4.4 Custom Transmit Functionality**
10. Info Flash Drive ................................................................. 195

10.1 Theory of Operation .......................................................... 195
10.1.1 Flash File Criteria ......................................................... 195
10.1.2 MSP430 Flash Primer ................................................... 195
10.1.3 MSP430 Info Flash and File Slots ................................. 196
10.1.4 File Slot Organization ................................................... 197

10.2 Hub File Example .............................................................. 198

10.3 Relocating To Main Flash .................................................. 201
10.3.1 Main Flash Segments ..................................................... 201
10.3.2 Updating the Linker Command File ............................... 202
10.3.3 Rename Files ............................................................... 202
10.3.4 Rename Declarations and References ........................... 203
10.3.5 Change the File Size ...................................................... 203
10.3.6 Update the Documentation .......................................... 203
10.3.7 Update the Flash Drive Parameters ............................... 203
10.3.8 Update the Slot Table .................................................. 204
10.3.9 Check the Erase Handlers ............................................. 206
10.3.10 Build and Test ............................................................ 206

11. Licensed Module Reference ............................................... 207

11.1 CalibrateDCO.s43 ............................................................. 207
11.1.1 Function CalibrateDCO_SetFrequency .......................... 207
11.1.2 Function CalibrateDCO_UpdateTracking ....................... 208
11.1.3 Function CalibrateDCO_StopTracking ............................ 208

11.2 CC1100.s43 ................................................................. 209
11.2.1 Function CC1100_GlobalResetSequence ....................... 209
11.2.2 Function CC1100_SetConfigurationRegister ................. 209
11.2.3 Function CC1100_GetConfigurationRegister ................. 210
11.2.4 Function CC1100_GetStatusRegister ............................ 210
11.2.5 Function CC1100_WriteBurstRegisters ......................... 211
11.2.6 Function CC1100_ReadBurstRegisters ............................ 212
11.2.7 Function CC1100_ReadRxFIFO ..................................... 212
11.2.8 Function CC1100_WriteTxFIFO ..................................... 213
11.2.9 Function CC1100_StrobeRES ....................................... 213
11.2.10 Function CC1100_StrobeFSTXON ................................. 214
11.2.11 Function CC1100_StrobeXOFF ..................................... 214
11.2.12 Function CC1100_StrobeCAL ..................................... 215
11.2.13 Function CC1100_StrobeRX ....................................... 215
11.2.14 Function CC1100_StrobeTX ....................................... 216
11.2.15 Function CC1100_StrobeIDLE ..................................... 216
11.2.16 Function CC1100_StrobeWOR .................................... 217
11.2.17 Function CC1100_StrobePWD .................................... 217
11.2.18 Function CC1100_StrobeFRX ..................................... 218
11.2.19 Function CC1100_StrobeFTX ..................................... 218
11.2.20 Function CC1100_StrobeWORST ................................ 219
11.2.21 Function CC1100_StrobeNOP ..................................... 219
11.2.22 Function CC1100_QueryChipStatus ............................. 220
11.2.23 Function CC1100_TweakIdle ..................................... 220
11.2.24 Function CC1100_QueryPARTNUM ............................. 221
11.2.25 Function CC1100_QueryVERSION ............................... 221
11.2.26 Function CC1100_QueryREQUEST ............................... 222
11.2.27 Function CC1100_QueryLOI ....................................... 222
11.2.28 Function CC1100_QueryRSSI ..................................... 223
11.2.29 Function CC1100_QueryMARSTATE ............................ 223
11.2.30 Function CC1100_QueryWORST ................................ 224
11.2.31 Function CC1100_QueryPKTSTATUS ......................... 224
11.2.32 Function CC1100_QueryVCO_VC_DAC ....................... 225
11.2.33  Function CC1100_QueryTXBYTES ................................................................. 225
11.2.34  Function CC1100_QueryRXBYTES .............................................................. 226
11.3  CHANNELPARAMETERS.C .............................................................................. 227
  11.3.1  Function ChannelParameters_GetRxAbortTicks .............................................. 227
  11.3.2  Function ChannelParameters_GetFeatures ...................................................... 228
  11.3.3  Function ChannelParameters_ReportFeatures ................................................. 228
  11.3.4  Function ChannelParameters_UpdateAddressFromTerminal ................................ 228
  11.3.5  Function ChannelParameters_EnableAddress .................................................. 229
  11.3.6  Function ChannelParameters_DisableAddress .................................................. 230
  11.3.7  Function ChannelParameters_ToggleAddress ................................................... 231
  11.3.8  Function ChannelParameters_SetAddress ...................................................... 231
  11.3.9  Function ChannelParameters_GetAddress ...................................................... 232
  11.3.10 Function ChannelParameters_EnableCRC ...................................................... 232
  11.3.11 Function ChannelParameters_DisableCRC ...................................................... 233
  11.3.12 Function ChannelParameters_ToggleCRC ...................................................... 233
  11.3.13 Function ChannelParameters_EnableRSSI_LQI .............................................. 234
  11.3.14 Function ChannelParameters_DisableRSSI_LQI ............................................. 234
  11.3.15 Function ChannelParameters_ToggleRSSI_LQI .............................................. 235
  11.3.16 Function ChannelParameters_EnableWhiten .................................................. 235
  11.3.17 Function ChannelParameters_DisableWhiten .............................................. 236
  11.3.18 Function ChannelParameters_ToggleWhiten .................................................. 237
  11.3.19 Function ChannelParameters_EnableSequenceNumber .................................... 237
  11.3.20 Function ChannelParameters_DisableSequenceNumber .................................... 238
  11.3.21 Function ChannelParameters_ToggleSequenceNumber .................................... 238
  11.3.22 Function ChannelParameters_SetChannel ..................................................... 239
  11.3.23 Function ChannelParameters_GetChannel ..................................................... 239
  11.3.24 Function ChannelParameters_SetLengthFixed .............................................. 240
  11.3.25 Function ChannelParameters_SetLengthFixedFEC ......................................... 240
  11.3.26 Function ChannelParameters_SetLengthVariable .......................................... 241
11.4  DIGITALRADIO.S43 .......................................................................................... 242
  11.4.1  Function DigitalRadio_ResetHW ................................................................. 242
  11.4.2  Function DigitalRadio_SetPATABLE ............................................................. 243
  11.4.3  Function DigitalRadio_Lock .......................................................................... 243
  11.4.4  Function DigitalRadio_Unlock ...................................................................... 244
  11.4.5  Function DigitalRadio_IsLocked ................................................................. 244
  11.4.6  Function DigitalRadio_SetChannel ............................................................. 245
  11.4.7  Function DigitalRadio_GetChannel ............................................................. 245
  11.4.8  Function DigitalRadio_SetFeatures ............................................................. 246
  11.4.9  Function DigitalRadio_GetFeatures ............................................................. 247
  11.4.10 Function DigitalRadio_StartTx ................................................................. 247
  11.4.11 Function DigitalRadio_StartRx ................................................................. 248
  11.4.12 Structure SDigitalRadioOperation ............................................................. 249
  11.4.13 Structure SDigitalRadioRxParams .............................................................. 250
  11.4.14 Structure SDigitalRadioRx ........................................................ ................. 251
  11.4.15 Structure SDigitalRadioTxParams .............................................................. 252
  11.4.16 Structure SDigitalRadioTx ........................................................ ................. 253
11.5  DISPLAY.C ............................................................................................................. 254
  11.5.1  Function Display_Open .................................................................................. 254
  11.5.2  Function Display_Close .................................................................................. 255
  11.5.3  Function Display_Tick .................................................................................. 255
  11.5.4  Function Display_RegisterEventHandlers ................................................... 256
  11.5.5  Function Display_Off ................................................................................. 257
  11.5.6  Function Display_Flash .............................................................................. 257
  11.5.7  Function Display_IsIdle .............................................................................. 258
11.6  HUBMODE.C ........................................................................................................... 259
  11.6.1  Function HubMode_Open .............................................................................. 259
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.6.2</td>
<td>Function HubMode_Close</td>
</tr>
<tr>
<td>11.6.3</td>
<td>Function HubMode_Start</td>
</tr>
<tr>
<td>11.7</td>
<td>INFOFLASHDRIVE.S43</td>
</tr>
<tr>
<td>11.7.1</td>
<td>Function InfoFlashDrive_WriteFile</td>
</tr>
<tr>
<td>11.7.2</td>
<td>Function InfoFlashDrive.ReadFile</td>
</tr>
<tr>
<td>11.7.3</td>
<td>Function InfoFlashDrive_EraseFile</td>
</tr>
<tr>
<td>11.8</td>
<td>MENU.C</td>
</tr>
<tr>
<td>11.8.1</td>
<td>Function Menu_GetInitialState</td>
</tr>
<tr>
<td>11.9</td>
<td>NODEMODE.C</td>
</tr>
<tr>
<td>11.9.1</td>
<td>Function NodeMode_Open</td>
</tr>
<tr>
<td>11.9.2</td>
<td>Function NodeMode_Close</td>
</tr>
<tr>
<td>11.9.3</td>
<td>Function NodeMode_Start</td>
</tr>
<tr>
<td>11.10</td>
<td>PACKET RADIO.C</td>
</tr>
<tr>
<td>11.10.1</td>
<td>Function PacketRadio_StartTx</td>
</tr>
<tr>
<td>11.10.2</td>
<td>Function PacketRadio_StartRx</td>
</tr>
<tr>
<td>11.10.3</td>
<td>Structure SPacketRadioParams</td>
</tr>
<tr>
<td>11.10.4</td>
<td>Structure SPacketRadio</td>
</tr>
<tr>
<td>11.11</td>
<td>RXTXHOST.S43</td>
</tr>
<tr>
<td>11.11.1</td>
<td>Function RxTxHost_QueueErasingBackspace</td>
</tr>
<tr>
<td>11.11.2</td>
<td>Function RxTxHost_QueueTxChar</td>
</tr>
<tr>
<td>11.11.3</td>
<td>Function RxTxHost_QueueTxCRLF</td>
</tr>
<tr>
<td>11.11.4</td>
<td>Function RxTxHost_QueueTxString</td>
</tr>
<tr>
<td>11.11.5</td>
<td>Function RxTxHost_QueueTxHexBYTE</td>
</tr>
<tr>
<td>11.11.6</td>
<td>Function RxTxHost_QueueTxHexWORD</td>
</tr>
<tr>
<td>11.11.7</td>
<td>Function RxTxHost_QueueTxHexDWORD</td>
</tr>
<tr>
<td>11.11.8</td>
<td>Function RxTxHost_QueueTxHexColonDWORD</td>
</tr>
<tr>
<td>11.11.9</td>
<td>Function RxTxHost_TickTx</td>
</tr>
<tr>
<td>11.11.10</td>
<td>Function RxTxHost_GetRx</td>
</tr>
<tr>
<td>11.11.11</td>
<td>Function RxTxHost_GetRxEcho</td>
</tr>
<tr>
<td>11.11.12</td>
<td>Function RxTxHost_GetHexBYTE</td>
</tr>
<tr>
<td>11.11.13</td>
<td>Function RxTxHost_GetCountRxInQueue</td>
</tr>
<tr>
<td>11.11.14</td>
<td>Function RxTxHost_GetCountRxInQueue</td>
</tr>
<tr>
<td>11.11.15</td>
<td>Function RxTxHost_GetCountRxCapacity</td>
</tr>
<tr>
<td>11.11.16</td>
<td>Functions RxTxHost_GetCountRxCapacity</td>
</tr>
<tr>
<td>11.12</td>
<td>RXTXMODES.C</td>
</tr>
<tr>
<td>11.12.1</td>
<td>Function RxTxModes_Open</td>
</tr>
<tr>
<td>11.12.2</td>
<td>Function RxTxModes_Close</td>
</tr>
<tr>
<td>11.12.3</td>
<td>Function RxTxModes_StartTx</td>
</tr>
<tr>
<td>11.12.4</td>
<td>Function RxTxModes_StartRx</td>
</tr>
<tr>
<td>11.12.5</td>
<td>Function RxTxModes_StartPing</td>
</tr>
<tr>
<td>11.12.6</td>
<td>Function RxTxModes_StartEcho</td>
</tr>
<tr>
<td>11.13</td>
<td>STAR PACKET.H</td>
</tr>
<tr>
<td>11.13.1</td>
<td>Structure SStarPacketSearch</td>
</tr>
<tr>
<td>11.13.2</td>
<td>Structure SStarPacketSearchAck</td>
</tr>
<tr>
<td>11.13.3</td>
<td>Structure SStarPacket</td>
</tr>
<tr>
<td>11.13.4</td>
<td>Structure SStarPacketAck</td>
</tr>
<tr>
<td>11.14</td>
<td>SWITCH.C</td>
</tr>
<tr>
<td>11.14.1</td>
<td>Function Switch_Open</td>
</tr>
<tr>
<td>11.14.2</td>
<td>Function Switch_Close</td>
</tr>
<tr>
<td>11.14.3</td>
<td>Function Switch_Tick</td>
</tr>
<tr>
<td>11.14.4</td>
<td>Function Switch_RegisterEventHandlers</td>
</tr>
<tr>
<td>11.14.5</td>
<td>Function Switch_SetRepeatParameters</td>
</tr>
<tr>
<td>11.15</td>
<td>TASK.S43</td>
</tr>
<tr>
<td>11.15.1</td>
<td>Function Task_TickTask</td>
</tr>
<tr>
<td>11.15.2</td>
<td>Function Task_InitializeTask</td>
</tr>
<tr>
<td>11.15.3</td>
<td>Function Task_GetCurrentTask</td>
</tr>
</tbody>
</table>

MSP430 RF Applications with the MRF1611CC1100
Table of Tables

Table 1-1 - J1 Host Signals .................................................. 15
Table 1-2 - J2 Host Signals .................................................. 16
Table 2-1 - Terminal Setup .................................................. 21
Table 2-2 - Hardware Units .................................................. 22
Table 2-3 - Unit A Jumper Settings ...................................... 25
Table 6-1 RX Mode Payload Data Bytes .............................. 90
Table 7-1 - Unit B Jumper Settings for ECHO ...................... 132
Table 7-2 - Unit A Jumper Settings for Ping ......................... 133
Table 8-1 Hub and Node Serialization ................................. 168
Table 8-2 - Unit B Jumper Settings as Node ....................... 169
Table 8-3 - Unit C, D, and E Jumper Settings as Node .......... 169
Table 8-4 - Unit A Jumper Settings as Hub ......................... 170
Table 10-1 File Slot Organization ...................................... 197
Table 10-2 Slot Table Entry Organization .......................... 204

Table of Figures

Figure 2-1 Terminal Session ............................................. 21
Figure 2-2 Power Options .................................................. 23
Figure 2-3 Analog Options ............................................... 23
Figure 2-4 Communication Options ................................. 24
Figure 2-5 Mode Options .................................................. 24
Figure 2-6 Opening PrgUSBP .......................................... 26
Figure 2-7 PrgUSBP Conflict ............................................ 27
Figure 2-8 PrgUSBP Project Opened ................................. 27
Figure 2-9 Operation Succeeded ....................................... 28
Figure 2-10 Open Workspace .......................................... 29
Figure 2-11 Debugger Loaded .......................................... 30
Figure 2-12 Linker Output Format ..................................... 31
Figure 4-1 Generic State Machine Diagram ...................... 45
Figure 4-2 Code Function Execution ................................. 49
Figure 4-3 Reused Code Function .................................... 50
Figure 4-4 State Function Example .................................. 52
Figure 4-5 Reused State Function .................................... 53
Figure 4-6 Callback Interaction Diagram ......................... 55
Figure 4-7 Packet Radio Application Layers .................... 61
Figure 4-8 Default CC1100 Packet Format ....................... 63
Figure 4-9 Variable-Length Packet Formats .................... 65
Figure 4-10 Fixed-Length Packet Formats ....................... 66
Figure 5-1 Default User Application ............................... 67
Figure 6-1 TX to RX Mode .............................................. 85
Figure 7-1 Ping and Echo Mode ...................................... 131
Figure 7-2 Custom Ping Application ............................... 138
Figure 8-1 Star Topology ................................................ 159
Figure 8-2 Sensor Network ............................................. 160
Figure 8-3 Hub/Node Search Mode ................................ 160
Figure 8-4 Hub/Node Active Mode ................................ 162
Figure 8-5 Node Mode States ......................................... 163
Figure 8-6 Hub Mode States .......................................... 165
Figure 9-1 Run to Calibrate NOP .................................... 186
Figure 9-2 Calibration Array Addresses .......................... 187
FCC Compliance Statement

This device complies with Part 15 of the FCC Rules. 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 undesired operation.

Application Limitations

All hardware supplied by SoftBaugh is intended for laboratory, experimental, manufacturing and/or research purposes only, and is not intended for commercial resale or commercial or residential use. Medical usage on humans is specifically prohibited. Alien autopsies are fine. Send us some of their equipment. Especially the things that make stuff go poof.

Standards Notice

We do not voluntarily comply with any standards to which we are not statutorily obligated. So don’t ask, and don’t send us procurement forms asking us to sign up for any! All we care about is shipping you great stuff, helping you use it, making you look good in the process, and making your boss happy he spent the money, while recognizing that in your world you are enough of an expert to know whether things work as they should.

You would love to work here…
This page intentionally left blank.
1. Overview

The MRF1611CC1100, RF Module with MSP430F1611 and CC1100, allows digital radio experiments in the unlicensed 915 MHz band. Refer to documentation from Texas Instruments for detailed information for the MSP430F1611 and the CC1100 devices.

1.1 MRF1611CC1100 Features

The key features of this product are:

- Self-contained 915 MHz RF packet radio, with reprogrammable MSP430
- 4 user LEDs
- Up to 36 user IOs available at 50-mil sockets
- Support for SMA, MCX, and MMCX antennae
- 32 kHz watch crystal on XT1
- Optional HC49US crystal on XT2, with capacitors provided suitable for 18 pF operation
- 100k 1% ROSC for up to 8MHz operation via the internal DCO and DCO tracking algorithm
- Sample code for packet-based transfers
- Access to JTAG and BSL functionality
- On-board analog supply conditioning

Also available is an optional HMRF1611 host board to simplify development. The features of the HMRF1611 include:

- 4 user switches mapped to P1.4 to P1.7
- JTAG+ header, including optional BSL support
- I2C header and optional I2C pull-up resistors
- Dual AAA supply with TPS60210 for 3.3v operation
- External supply option via screw terminals
- Powered prototyping area
- Layout for BRS232 module for RS232 communication option
- Convenient jumper options for analog configuration

1.2 Prerequisites

This document assumes some familiarity with embedded systems development with the MSP430, including the use of debuggers and the IAR Embedded Workbench development environment. Knowledge of the C language is also assumed.
1.3 Source Code Licensing

The demonstration firmware provided for the MRF1611CC1100 allows experimentation with the various ways in which a simple RF point-to-point and star networks can be made to work with very little coding when using our MRF1611CC1100 libraries. The free version of these libraries is limited to four nodes and one hub, but this limitation is due only to the fact that the hub has only four LEDs. The licensed version can be modified in source form to support additional nodes as desired by the licensee.

Licensed users, defined as those users who have purchased a license to the source code from SoftBaugh, and agreeing to the Non Disclosure Agreement accompanying this license, gain access to the source code used to develop the demo library LibraryMRF1611CC1100.r43, included as part of the demo project for this board.

To use the source code for the modules contained within the demo library LibraryMRF1611CC1100.r43, the user must also have access to or purchase a license for the full version of IAR Embedded Workbench (EW430). Both licenses are available for purchase from SoftBaugh.

Unlicensed users can build and modify the demo project at will using the free IAR Kickstart tool available from TI's website, but do not have access to all source code.

1.4 References

Much of the material in this document that was not developed at the expense of SoftBaugh was derived from documentation produced by Texas Instruments. The following references are essential for understanding the operation of the MSP430F1611 and the CC1100, and are available for download from www.ti.com:

SLAU049 - MSP430x1xx Family User’s Guide. This essential document discusses the architecture of the MSP430 processor and operating details of the peripheral modules for all MSP430s contained in the x1xx family, including the MSP430F1611.

SLAS368 - Datasheet for the MSP430x15x, MSP430x16x, MSP430x161x Mixed Signal Microcontroller. This document contains device-specific details complementing the information contained in the Family User’s Guide.

SWRS038 - Datasheet for the CC1100 Single Chip Low Cost Low Power RF Transceiver. This document provides essential details for the CC1100 found on the MRF1611CC1100.
1.5 MRF1611 Host Interface

The MRF1611CC1100 is to be connected to a host board via two 24-pin 50 mil headers, J1 and J2, and two mounting holes. The module connectors J1 and J2 are Samtec #CLP-112-02-F-D-PA. A recommended connector for the host side is Oupin #2215-2X12G00S/023/030B/SP, or Samtec #FTS-112-03-F-D-x or -F-DV-x. The signal definitions and physical layout are invariant across all MRF1611 module versions, allowing convenient exchange of modules for different applications.

The physical layout of the board, including headers and mounting holes, is shown in the separate document *MRF1611_MountingHoles.pdf*, available at [www.softbaugh.com](http://www.softbaugh.com) on the MRF1611CC1100 page.

The signal definitions of J1 and J2, as viewed from above the host board, are shown below:

<table>
<thead>
<tr>
<th>Group</th>
<th>Name</th>
<th>Pin</th>
<th>Pin</th>
<th>Name</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port5</td>
<td>P5.5</td>
<td>1</td>
<td>2</td>
<td>P5.4</td>
<td>Port5</td>
</tr>
<tr>
<td>Port5</td>
<td>P5.6</td>
<td>3</td>
<td>4</td>
<td>P4.7</td>
<td>Port4</td>
</tr>
<tr>
<td>Port5</td>
<td>P5.7</td>
<td>5</td>
<td>6</td>
<td>P2.2 (1)</td>
<td>Port2/BSL</td>
</tr>
<tr>
<td>JTAG</td>
<td>TDO</td>
<td>7</td>
<td>8</td>
<td>TMS</td>
<td>JTAG</td>
</tr>
<tr>
<td>JTAG</td>
<td>TDI</td>
<td>9</td>
<td>10</td>
<td>TCK</td>
<td>JTAG/BSL</td>
</tr>
<tr>
<td>Port6</td>
<td>P6.0</td>
<td>11</td>
<td>12</td>
<td>RST/NMI (2)</td>
<td>JTAG/BSL</td>
</tr>
<tr>
<td>Port6</td>
<td>P6.1</td>
<td>13</td>
<td>14</td>
<td>P1.1 (3)</td>
<td>Port1/BSL</td>
</tr>
<tr>
<td>Port6</td>
<td>P6.2</td>
<td>15</td>
<td>16</td>
<td>VREF-/VeREF- (4)</td>
<td>VREF</td>
</tr>
<tr>
<td>Ground</td>
<td>GND</td>
<td>17</td>
<td>18</td>
<td>VeREF+ (4)</td>
<td>VREF</td>
</tr>
<tr>
<td>Port6</td>
<td>P6.5</td>
<td>19</td>
<td>20</td>
<td>VREF+</td>
<td>VREF</td>
</tr>
<tr>
<td>Port6</td>
<td>P6.4</td>
<td>21</td>
<td>22</td>
<td>P6.7</td>
<td>Port6</td>
</tr>
<tr>
<td>Port6</td>
<td>P6.3</td>
<td>23</td>
<td>24</td>
<td>P6.6</td>
<td>Port6</td>
</tr>
</tbody>
</table>

Note 1: P2.2 is provided primarily for BSL operation. For some MRF1611 models this signal is consumed for control of the radio IC and should not be used on the host board. Consult the MRF1611 schematics for details. If the signal is not used on the MRF1611 for radio control, it is free for use as an I/O pin.

Note 2: The MRF1611 supplies a 0.1uF capacitor to ground and a 100k ohm pullup resistor to VCC. No additional conditioning is required on the host board for this signal.

Note 3: P1.1 is provided primarily for BSL operation. This signal is connected to D2 on the MRF1611. If desired as a user I/O, be aware that high levels on this signal will result in illumination of D2 and increased current consumption.

Note 4: If using VREF-/VeREF- or VeREF+, then the host board must supply a 4.7 uF ceramic capacitor on each used signal. A 0.1uF ceramic capacitor is already supplied on the MRF1611.
### Table 1-2 - J2 Host Signals

<table>
<thead>
<tr>
<th>Group</th>
<th>Name</th>
<th>Pin</th>
<th>Pin</th>
<th>Name</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port5</td>
<td>(1)  P5.3</td>
<td>1</td>
<td>2</td>
<td>P5.2 (1), (2)</td>
<td>Port5</td>
</tr>
<tr>
<td>Port5</td>
<td>(1)  P5.1</td>
<td>3</td>
<td>4</td>
<td>P4.6</td>
<td>Port4</td>
</tr>
<tr>
<td>Port4</td>
<td>P4.5</td>
<td>5</td>
<td>6</td>
<td>P4.4</td>
<td>Port4</td>
</tr>
<tr>
<td>Port4</td>
<td>P4.3</td>
<td>7</td>
<td>8</td>
<td>P4.2</td>
<td>Port4</td>
</tr>
<tr>
<td>Port4</td>
<td>P4.1</td>
<td>9</td>
<td>10</td>
<td>P4.0</td>
<td>Port4</td>
</tr>
<tr>
<td>Port3</td>
<td>P3.5</td>
<td>11</td>
<td>12</td>
<td>P3.4</td>
<td>Port3</td>
</tr>
<tr>
<td>Ground</td>
<td>GND</td>
<td>13</td>
<td>14</td>
<td>VCC</td>
<td>Power</td>
</tr>
<tr>
<td>Port3</td>
<td>P3.3</td>
<td>15</td>
<td>16</td>
<td>P3.2 (3)</td>
<td>Port3</td>
</tr>
<tr>
<td>Port3</td>
<td>P3.1</td>
<td>17</td>
<td>18</td>
<td>P3.0</td>
<td>Port3</td>
</tr>
<tr>
<td>Port2</td>
<td>P2.7</td>
<td>19</td>
<td>20</td>
<td>GND</td>
<td>Ground</td>
</tr>
<tr>
<td>Port1</td>
<td>P1.7</td>
<td>21</td>
<td>22</td>
<td>P1.6</td>
<td>Port1</td>
</tr>
<tr>
<td>Port1</td>
<td>P1.5</td>
<td>23</td>
<td>24</td>
<td>P1.4</td>
<td>Port1</td>
</tr>
</tbody>
</table>

Note 1: USART1 is used by the MRF1611 for SPI control of the radio IC. This port is unavailable to user applications except as additional slave devices on this bus. This port is exposed on the header for convenience of operating additional slaves, but priority should be given to control of the radio IC.

Note 2: If P5.2 is used for additional SPI slave devices, then the host board may supply a pullup resistor. Be aware that in some radio control modes, this signal is driven by the radio IC to emit additional immediate status information. It may be necessary to deconflict this status information with slave SPI data in some applications.

Note 3: If P3.2 is used for SPI, then the host board should supply a pullup resistor if required for the application.
1.6 On-Chip Resources

At the time of this writing, the demo firmware consumes the following on-chip resources:

**RAM**: 1 kB, including redundant Tx and Rx buffers for many operating modes, plus demo menu and user terminal feedback buffers that are not required for a live application.

**Flash**: 19.2 kB, including redundant code for many operating modes, plus demo menu and user terminal feedback code that will not be required for a live application.

Once much of the demo code is stripped away to the essentials, the footprint of this firmware source in your application could be as little as a few hundred bytes of RAM and a few kilobytes of flash!

**DCO**: The DCOR option is used for this firmware with REXT set to a 100k 1% resistor (R3). It is set to 6.0 MHz nominally on startup by the DCO tracking algorithm.

**XT2**: Not used for this demo. Two 33 pF capacitors are provided externally for the user’s convenience in adding an 18 pF high-frequency crystal in the HC49US format, if desired.

**ACLK**: For this firmware, ACLK is set to XT1 / 8, where XT1 is a 32.768 kHz watch crystal. This frequency is used by the DCO stabilization algorithms available in source form for the licensed version of the firmware.

ACLK could be sourced with a lower divisor, if desired, at the expense of reduced DCO tracking resolution.

**SMCLK**: Sourced from the DCO, at approximately 6.0 MHz. Because the 115.2 kbps UART requires SMCLK to operate properly, no low power mode greater than LPM1 is used in this demo.

In a production application, it would be a simple matter to switch to LPM3 if neither DCO tracking is required nor UART operation was desired during sleep.

**MCLK**: Sourced from the DCO, at approximately 6.0 MHz.

**USARTs**: USART0 is used in UART mode to provide interaction with the user via the terminal. USART1 is used in SPI mode to control the CC1100. In a live application USART0 would be free and USART1 could be shared with the CC1100.

**TimerA3**: TimerA3 is implemented as a continuous mode timer based on SMCLK, which is in turn derived from the stabilized DCO. TA3.CCR1 implements an 8192 Hz state tick.

TA3.CCR2 is used for the DCO tracking algorithm, and triggers from ACLK to perform SMCLK measurements.
TA3.CCR0 is available to the user, as long as the continuous mode operation of this module is left undisturbed.

**TimerB7**: Not used.

**Hardware Multiplier**: Not used.

**DMA Controller**: Not used. In a production application, CC1100 performance could be enhanced through the use of this module if desired.

**POR/SVS/Brownout**: Only used in the default startup mode.

**Watchdog Timer**: Not used. The state-based operation of this system is well-suited for watchdog timer integration in the main idle loop.

**ADC12**: ADC12 channel 5 is used for potentiometer measurements for demo purposes. All ADC12 channels would be available in a live application.

**DAC12**: Not used.

**ComparatorA**: Not used.

**Port1**: P1.4 through P1.7 are used for HMRF1611 switches.

**Port2**: P2.2 and P2.7 are not used.

**Port3**: P3.0 is used for power control, and P3.3 is used for disabling the terminal Rx in some modes. P3.4 and P3.5 are used for terminal communication. P3.1 and P3.2 are unused.

**Port4**: P4.0 through P4.5 are used for selection of the startup mode. P4.6 and P4.7 are unused.

**Port5**: P5.1 through P5.3 are used for UART1 SPI communication with the CC1100, but this operation can be shared with user devices. P5.4 through P5.7 are unused.

**Port6**: P6.0 is used for a diagnostic twiddle to show state machine operation. P6.5 is used for the potentiometer demo. All other P6.x pins are unused.

In a live application using a derivative of this firmware, all I/O pins listed in Section 1.5, *MRF1611 Host Interface*, would be available, subject to the notes listed there.
2. System Preparation

The previous chapter gave an overview of the MRF1611CC1100 system. This chapter details the preparations needed to operate the demo examples. The next chapter will list all the features of the terminal software menus incorporated into the demo firmware.

The demonstration firmware provided for the MRF1611CC1100 allows experimentation with a variety of radio control modes. The features available with this software include:

- IAR Kickstart and/or EW430 Embedded Workbench application in C and assembly.
- Fixed 915 MHz operation, including fixed channel spacing
- Terminal-based user interface
- Six demonstration operating modes, including point-to-point and hub-star
- Optional selection of channels and addresses
- Blinky LED, potentiometer, and switch features in most modes
- Optional selection of variable-length, fixed length, and fixed length with forward error correction packet formats
- Optional selection of CRC-based packets
- Optional selection of packet sequence numbers
- Jumper-based startup for stand-alone demos
- Flash file storage of hub-star network configuration parameters for stand-alone demos

The recommended hardware for operation of all demo modes and features is the MRF1611CC1100X5-UC-HX5RX2-N. This part number includes:

- Five MRF1611CC1100 modules
- One USBP flash programming adapter and USB cable
- Five HMRF1611 host modules
- Two BRS232 RS232 interfaces (assembly to two HMRF1611s required)

In addition, you will need two PCs, known here as PC A (a development machine), and PC B (a support machine). Also, obtain two DB9 serial cables for connection to the PCs’ serial ports. Alternatively, one PC could host both connections, but throughout this book we assume two separate PCs.

2.1 Software Installation

To perform the demonstrations in this document, you will need three sets of software:

- IAR Kickstart or Embedded Workbench
- USBP Drivers and Applications, including the PrgUSBP Programming Utility
- MRF1611CC1100 Demo Firmware

The next sections discuss each of these items.
2.1.1 IAR Embedded Workbench

IAR Systems, AB, provides this excellent development tool for MSP430 development. If you do not already have IAR Embedded Workbench installed, get a copy of the free Kickstart version of this tool from Texas Instruments’ website:

www.ti.com/msp430

Follow the links to third party software development tools to get the latest version. Due to frequent changes to that website, it is not practical to publish specific instructions here.

Follow the IAR Kickstart and/or EW430 Embedded Workbench instructions to install the development software on PC A at a minimum.

2.1.2 USBP Drivers and Applications

Users of versions of IAR prior to December 2006 will need to follow the instructions available at on the SoftBaugh USBP page to install the USBP and drivers, paying careful attention to the instructions found in ThirdPartyIDE.pdf. This software is available at:

www.softbaugh.com/ProductPage.cfm?strPartNo=USBP

Part of this installation includes our PrgUSBP programming utility. This utility allows stand-alone programming of MSP430 target boards, and is excellent for production use. The free Lite version of this tool allows full control of the USBP for programming, while the full version, licensed separately, adds the ability to serialize units and perform custom flash overlays.

The experiments in this document which require serial numbers will be implemented manually using the Lite version. Install this software on both PC A and PC B.

2.1.3 MRF1611CC1100 Demo Project

Also, obtain a copy of the MRF1611CC1100 demo software from the SoftBaugh site at:

www.softbaugh.com/ProductPage.cfm?strPartNo=MRF1611CC1100

Extract the demo software into the folder C:\Dev\MRF1611, keeping all folder names, including the DemoMRF1611CC1100 root folder. If you choose to extract the demo software elsewhere, make sure you make the appropriate mental conversions when we make reference to this folder later.

This folder also includes DemoMRF1611CC1100.ht, a HyperTerminal terminal session file used as a command interface for Units A and B, as shown in the next session. This document assumes that these terminal sessions are running when required.

Install the demo firmware project on both PC A and PC B.
2.2 Terminal Setup

A HyperTerminal file, DemoMRF1611CC1100.ht, is provided for your convenience, and is located in the project folder C:\Dev\MRF1611\DemoMRF1611CC1100\, or wherever the demo firmware was extracted.

Whenever a terminal session is desired, double-click on the terminal file and the following should appear:

![Figure 2-1 Terminal Session](image)

If you wish to use a different terminal application, prepare the following settings:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baud</td>
<td>115.2 kbps</td>
</tr>
<tr>
<td>Data Bits</td>
<td>8</td>
</tr>
<tr>
<td>Parity</td>
<td>None</td>
</tr>
<tr>
<td>Stop Bits</td>
<td>1</td>
</tr>
<tr>
<td>Handshaking</td>
<td>None</td>
</tr>
</tbody>
</table>

If handshaking cannot be disabled by your application, choose *Hardware Handshaking* and jumper RTS to CTS and DSR to DTR on the HMRF1611.
2.3 Hardware Preparation

The hardware also requires preparation. When the preparations have been completed, the following units will be at your disposal:

Table 2-2 - Hardware Units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>HMRF1611 + MRF1611CC1100 + BRS232 + Antenna</td>
</tr>
<tr>
<td>B</td>
<td>HMRF1611 + MRF1611CC1100 + BRS232 + Antenna</td>
</tr>
<tr>
<td>C</td>
<td>HMRF1611 + MRF1611CC1100 + Antenna</td>
</tr>
<tr>
<td>D</td>
<td>HMRF1611 + MRF1611CC1100 + Antenna</td>
</tr>
<tr>
<td>E</td>
<td>HMRF1611 + MRF1611CC1100 + Antenna</td>
</tr>
</tbody>
</table>

2.3.1 BRS232

Prepare two HMRF1611 host modules by adding two BRS232 modules to the prototyping areas. See recommended assembly instructions at:

www.softbaugh.com/ProductPage.cfm?strPartNo=HMRF1611

2.3.2 Highly Calibrated Antenna

Also prepare each MRF1611CC1100 by attaching a Highly Calibrated Antenna. What is a Highly Calibrated Antenna? It is a wire soldered into the center hole of the MRF1611CC1100 which extends 3.25" above the top surface of the board.

It is not necessary to completely fill in the hole with solder, just apply enough solder to tack the wire to the inside of the hole. This makes removal of the wire and installation of an RF connector easier in the future.

Material choice giving you pause? Surely you have many scraps of solid CAT-4 or CAT-5 lying about. This makes excellent antenna wire for the purposes of this document.

See recommended assembly instructions at:

www.softbaugh.com/ProductPage.cfm?strPartNo=MRF1611CC1100

2.3.3 Install MRF1611CC1100s

Install an antennaed MRF1611CC1100 onto each HMRF1611. It is not possible to plug them in backwards, but it is very easy to skip a set of pins. Make sure the headers are plugged in correctly.
2.4 Jumper Settings

The jumpers on the HMRF1611 enable various features of the demo firmware. The jumper settings are summarized in the following sections:

2.4.1 Power Options Header

Configure the power options of the board as shown below:

![Figure 2-2 Power Options](image)

Jumper P3.0 to ONOFF to enable MSP430 control of the power supply IC U1.

When ready to use a system board, jumper VB to VIN to enable AAA power to the power supply IC. When not using the board, hang this jumper to the side to disable power.

2.4.2 Analog Options Header

Jumper VR+ to P+ to power the potentiometer from the internal reference VREF+, and jumper PW to 6.5 to route the potentiometer wiper to Port6.5 ADC channel on the MSP430:

![Figure 2-3 Analog Options](image)
2.4.3 Digital Options Header

Prepare the RS232 interface by jumpering the following positions, depending on whether the module will be connected to the PC or not:

<table>
<thead>
<tr>
<th>Units Connected to PC</th>
<th>Stand-Alone Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0 1 2 3.1</td>
<td>3.0 1 2 3.1</td>
</tr>
<tr>
<td>3.2 3 4 3.3</td>
<td>3.2 3 4 3.3</td>
</tr>
<tr>
<td>3.4 5 6 3.5</td>
<td>3.4 5 6 3.5</td>
</tr>
<tr>
<td>TX 1 2 RX</td>
<td>TX 1 2 RX</td>
</tr>
<tr>
<td>FOF 3 4 FON</td>
<td>FOF 3 4 FON</td>
</tr>
<tr>
<td>RTB 5 6 VLD</td>
<td>RTB 5 6 VLD</td>
</tr>
<tr>
<td>RTS 7 8 CTS</td>
<td>RTS 7 8 CTS</td>
</tr>
<tr>
<td>DSR 9 10 DTR</td>
<td>DSR 9 10 DTR</td>
</tr>
</tbody>
</table>

**Figure 2-4 Communication Options**

In the case of stand-alone units, P3.3 is set to high, and jumpered to P3.5 to ensure that spurious serial port input doesn’t modify the operation of the unit.

Prepare the startup mode of each board by setting the jumpers on Port4 to the following positions, depending on the demo performed:

<table>
<thead>
<tr>
<th>Menu Mode</th>
<th>Echo Mode</th>
<th>Hub Mode</th>
<th>Node Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0 1 2 4.1</td>
<td>4.0 1 2 4.1</td>
<td>4.0 1 2 4.1</td>
<td>4.0 1 2 4.1</td>
</tr>
<tr>
<td>4.2 3 4 4.3</td>
<td>4.2 3 4 4.3</td>
<td>4.2 3 4 4.3</td>
<td>4.2 3 4 4.3</td>
</tr>
<tr>
<td>4.4 5 6 4.5</td>
<td>4.4 5 6 4.5</td>
<td>4.4 5 6 4.5</td>
<td>4.4 5 6 4.5</td>
</tr>
<tr>
<td>4.6 7 8 4.7</td>
<td>4.6 7 8 4.7</td>
<td>4.6 7 8 4.7</td>
<td>4.6 7 8 4.7</td>
</tr>
</tbody>
</table>

**Figure 2-5 Mode Options**

These options will be explicitly stated as part of the individual demo instructions later.
2.5 Loading the Demo Firmware

Throughout the following sections, you must load various versions of the demo code onto the MRF1611CC1100 assemblies in order to successfully run the samples.

There are two techniques which can be used to load this firmware. The first technique makes use of the SoftBaugh PrgUSBP programming utility to load a .txt file directly, while the second technique makes use of the IAR Embedded Workbench to load a compiled and linked project on the fly.

During development, the Embedded Workbench technique will be the most useful of the two, as this technique also loads the debugger, allowing you to step through the code. We will use this approach most often while working with Unit A.

However, during production, and also while preparing the remainder of the units for the demonstrations in this document, we will use PrgUSBP to prepare the units for work, which to some extent emulates a miniature production process. We will also demonstrate how to use IAR Embedded Workbench to prepare .txt files for use with the PrgUSBP.

To perform the work in this section, you will need Unit A, as prepared earlier with an HMRF1611, BRS232, MRF1611CC1100, and antenna.

Configure the HMRF1611 jumpers for Unit A as follows:

<table>
<thead>
<tr>
<th>Header</th>
<th>Option</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Power Off</td>
<td>Unit powered from USBP</td>
</tr>
<tr>
<td>Analog</td>
<td>Default</td>
<td>n/a</td>
</tr>
<tr>
<td>Port3 / Comm</td>
<td>Unit Connected to PC</td>
<td>For terminal operation</td>
</tr>
<tr>
<td>Port4</td>
<td>Menu Mode</td>
<td></td>
</tr>
</tbody>
</table>

Refer back to Section 2.4, Jumper Settings, for descriptions and diagrams for these options.

You will also need IAR Embedded Workbench installed on PC A, which should be running the DemoMRF1611CC1100.ht terminal session. Connect the Unit A hardware to PC A with one of the 9-pin serial cables.

It is important that you work through both of the following sections, in order, as both techniques will be used throughout. And, by verifying that the software components have been installed correctly, you can more easily work through the experiments with a minimal amount of frustration later.
2.5.1 Loading Firmware with PrgUSBP

In this section, we demonstrate use of the PrgUSBP programming utility installed with the USBP drivers and applications. You will use this technique later in this document to directly program units as required.

With a USBP attached to the USB cable, and Unit A attached to the USBP via the 14-pin JTAG cable, start the PrgUSBP utility:

![Figure 2-6 Opening PrgUSBP]

If the USBP is installed correctly, you will see a hardware version number in the upper right corner. Your observed number will vary, but the important feature here is that the software started, and a version number is presented. This means that the USBP is properly installed, and communicating with the host PC.

This utility provides access to all the programming features of the USBP, and allows use of project files to define various programming operations. We will use a predefined project file in a moment.
Note: When using the PrgUSBP utility, make sure other USBP-related applications are closed, including IAR Embedded Workbench, or otherwise an error message will appear:

The PrgUSBP makes use of project files ending with the `.usbprj` extension. The demo firmware includes a `DemoMRF1611CC1100.usbprj` sample project, located in the `C:\Dev\MRF1611\DemoMRF1611CC1100\` folder. Using the PrgUSBP File | Open menu, open this project file:
This project file defines several project operations, including Default Demo, Hub, etc. Normally, these different operations would define steps in a production process, but in our case they define the various builds of the demo code that provide different features.

The Default Demo will be used for most demonstrations, while the Hub and Node versions have unique serial numbers used for a star topology.

Note that if you extracted the demo firmware to any folder other than the default C:\Dev\MRF1611\DemoMRF1611CC1100, then you will have to repair the target file links for each of the operations defined in this file by clicking the … (Browse) button.

Make sure the Default Demo operation is selected as shown above, and click Execute. If all is well, the firmware will be loaded onto the target, and released to run, resulting in:

Figure 2-9 Operation Succeeded

displayed on the PrgUSBP, and the message Operation succeeded shown on the status bar at the bottom. If a failure was encountered, correct this before proceeding.

The demo software supports terminal operation via a menu system. Upon startup, the firmware displays a version number and a copyright notice, as shown:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006
Resetting...

If you do not see the above on your terminal session, make sure that all the previous steps in this chapter have been followed. Get the above working before moving on.

This software should be installed on both PC A and PC B. Try running this exercise on both PCs, using Unit A for both, as Unit B will have a slightly different hardware configuration as described later.

When done, close PrgUSBP on PC A and return Unit A back to PC A.
2.5.2 Loading Firmware with IAR Embedded Workbench

Now open the IAR Kickstart or Embedded Workbench software and open the DemoMRF1611CC1100.eww workspace. You should see something similar to the following:

![Figure 2-10 Open Workspace](image)

Make sure that the *debugger* is selected as the output for debugging, rather than the *simulator*. This selection is made via the *Project | Options* menu, and then choosing *Debugger | Setup | Driver* from the dialog that appears.

If you do not see *Debugger* listed in the dialog, close the dialog and make sure that DemoMRF1611CC1100 - Debug project name is highlighted in the *Workspace*, then try again.

Press *Debug* to load the firmware into the target board. If all the software components have been installed correctly, you will see the debugger loading the firmware.

If an error occurs, make sure that the debugger is selected as the output, as mentioned above. Also, for newer versions of IAR from December 2006 forward, make sure that the *SoftBaugh USBP* is selected as the debugger. If you are using an older version of the IAR tools make sure that the *FET Debugger* is selected and that on the *FET Debugger | Setup* page the *TI USB FET* is selected as the *Connection*.
Once the debugger has been loaded, the following screen appears:

![Figure 2-11 Debugger Loaded](image)

At this point, the firmware has been loaded, but is not executing as the debugger is waiting for the user to apply debugging commands, which we will in a moment.

Close the terminal session, and reopen it to ensure you are working with a clean session and not seeing artifacts left over from the previous exercise with the PrgUSBP.

Now press Run to execute the firmware on the target.

As before, the firmware displays a version number and a copyright notice, as shown:

```
MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...
```

If you do not see the above on your terminal, make sure that all the previous steps in this chapter have been followed. Get the above working before moving on.

Close the debugger with the Stop Debugging button. The target will continue to run and respond to commands.
2.5.3 Using Embedded Workbench to Generate Binaries

To load code with the PrgUSBP, a .txt binary file is required. IAR Embedded Workbench can be used to generate these binaries by changing the output format.

With the debugger session closed, open the Project | Options dialog and choose the Linker category and the Output tab:

![Figure 2-12 Linker Output Format](image)

The Format option shown, Debug information for C-SPY, is the format used for debugging a target.

However, by choosing the Other option and Output format set to msp430-txt, the corresponding DemoMRF1611CC1100.txt file will be created in the \Debug\Exe\ subfolder underneath the project.

Leave the Format set as-is for the debugger, and hit Cancel to close the dialog.
This page intentionally left blank.
3. Terminal Commands

The previous chapter described the preparations necessary for using the MRF1611CC1100 hardware and demo firmware. This chapter describes the menu options available with the demo firmware. The next chapter will describe the firmware architecture used to build the demo firmware.

Throughout this manual, user entries are denoted in **bold** and terminate with `<Enter>`, in bold italics, when `Enter` is required. Escape (ESC) is denoted by `<ESC>`, in bold italics.

Ensure that Unit A is connected to the PC, and is executing the demo software and that the unit is reporting to the terminal session, as described in the previous chapter.

### 3.1 Main Menu Commands

From the main prompt, type `?` for a menu overview:

```
-?
Menu options:
? - this list
A - toggle address mode
C - toggle CRC mode
D - display channel parameters
E - enter Echo mode
F - set fixed length packets
G - set fixed length packets with FEC
H - enter Hub mode
I - get CC1100 register
K - change address
N - enter Node mode
O - set CC1100 register
P - enter Ping mode
Q - toggle RSSI/LQI mode
R - enter Rx mode
S - toggle sequence number mode
T - enter Tx mode
U - enter User mode
V - set variable length packets
W - toggle whitening
Y - get CC1100 chip status
Z - reset digital radio
0-7 - set channel 0 to 7
-?
```

The following sections briefly discuss each of these main menu commands.
3.1.1 ? - Show Menu

As mentioned in the previous section, this menu command displays the menu.

3.1.2 A - Toggle Address Mode

Selectively adds the address specified with the 'K' command to the packet framing.

Examples are shown below for enabling and disabling addresses:

- \texttt{a}
  Address (0x33) enabled

- \texttt{a}
  Address (0x33) disabled

3.1.3 C - Toggle CRC Mode

Selectively adds CRC to packet framing encoding (Tx) and decoding (Rx).

Examples are shown below for enabling and disabling CRC:

- \texttt{c}
  CRC disabled

- \texttt{c}
  CRC enabled

3.1.4 D - Display Channel Parameters

Displays the channel parameters supported by \textit{ChannelParameters.c}.

An example is shown below:

- \texttt{d}
  Current channel parameters:
  Addresses disabled
  CRC enabled
  RSSI/LQI enabled
  Whitening enabled
  Sequence numbers enabled
  Using channel 0x00 for variable length packets

-
3.1.5  E - Enter Echo Mode

Enters Echo mode, ESC to cancel. Requires a remote unit in Ping mode.

An example is shown below:

- e
  Echo started
  <ESC>

More examples of Echo mode are given in Chapter 7, Ping/Echo Demo.

3.1.6  F - Set Fixed Length Packets

Sets packets for fixed length format.

An example is shown below.

- f
  Set fixed length packets

3.1.7  G - Set Fixed Length Packets With FEC

Sets packets for fixed length format, and enables hardware Forward Error Correction (FEC).

An example is shown below:

- g
  Set fixed length packets with FEC

3.1.8  H - Enter Hub Mode

Enters Hub mode, ESC to cancel. Requires 1 to 4 remote units in Node mode. See next section for Hub mode commands.

An example is shown below:

- h
  Hub mode started, serial #8888:6700

  Loading hub file: failed 0x0004, file not found
  Writing hub file to flash
3.1.9 I - Get CC1100 Register

Reads the indicated CC1100 register and presents this value to the user.

An example of reading the ADDR (Device Address) register (0x09) is shown below:

```
-i
adr: 0x09 = 0x33
ok
```

If the specified address is below 0x30, the register read is a configuration register. If the address is 0x30 or higher, the indicated register is a status register.

In the example below, the VERSION (0x31) register is read:

```
-i
adr: 0x31 = 0x03
ok
```

Note that the write/read and single/burst bits are set internally, and do not have to be specified by the user.

3.1.10 K - Change Address

Collects a packet address value from the user. If addressed packets are enabled, this value is inserted into Tx packets and provides a filter for Rx packets.

This setting is not relevant for Hub and Node modes, which use fixed addresses.

An example of changing the address to 0x21, and then back to the default 0x33, is shown below:

```
-k
Change address (0x33): 0x21
Address set to 0x21
ok
-k
Change address (0x21): 0x33
Address set to 0x33
ok
```
3.1.11 N - Enter Node Mode

Enters Node mode, ESC to cancel. Requires a remote unit in Hub mode.

An example is shown below, in which the node does not currently have a registered hub:

```
-n
Node mode started, serial #8888:6700
Loading node file: failed 0x0004, file not found
Listening for hub as node 01
<ESC>
User cancelled operation
Node mode stopped
```

Further examples of Node mode are given in Chapter 8, Hub/Node Demo.

3.1.12 O - Set CC1100 Register

Collects a CC1100 register value from the user and stores it in the indicated CC1100 register.

The following example reads the PKTLEN register (0x06), writes it with 0x47, and reads it again:

```
-i
adr: 0x06= 0xFF
ok
-o
adr: 0x06
dat: 0x47
ok
-i
adr: 0x06= 0x47
ok
```

Warning: This command should be used only as a curiosity, as it bypasses many of the shadow mechanisms inherent in the demo code and could cause unexpected side effects.
3.1.13 P - Enter Ping Mode

Enters Ping mode, ESC to cancel. Requires a remote unit in Echo mode.

An example is shown below:

```
P
Ping started
Ping 00: failed: 0008 (0001): timed out
Attempting retry for packet #00
Ping 00: failed: 0008 (0001): timed out
Attempting retry for packet #00
Ping 00: <ESC>
```

Error count = 0000
Retry count = 0002

In the example above, no remote unit was available. More examples of Ping mode will be given in Chapter 7, Ping/Echo Demo.

3.1.14 Q - Toggle RSSI/LQI Mode

Toggles RSSI/LQI mode for received packets. If RSSI/LQI is disabled in Hub and Node modes, the RSSI and LQI are extracted from the CC1100 status registers instead.

Examples are shown below for enabling and disabling RSSI/LQI mode:

```
-q
RSSI/LQI disabled

-q
RSSI/LQI enabled
```

3.1.15 R - Enter Rx Mode

Enters Rx mode, ESC to cancel. Requires a remote unit in Tx mode.

An example is shown below.

```
r
Rx started
<ESC>
Error count = 0000
```

In the example above, no remote Tx unit was available. Further examples of Rx mode will be presented in Chapter 6, Tx and Rx Demo.

3.1.16 S - Toggle Sequence Number Mode

Toggles sequence number filtering. If set, a scrolling sequence number is inserted into Tx packets, and Rx packets are scanned for this sequence number, with mismatched sequence numbers reported as warnings.

Examples are shown below for enabling and disabling sequence numbers:

- `s`
  Sequence numbers disabled

- `s`
  Sequence numbers enabled

Sequence numbers are forced to enabled for Hub and Node modes, without regard to and without changing this option.

3.1.17 T - Enter Tx Mode

Enters Tx mode, ESC to cancel. Requires a remote unit in Rx mode.

An example of entering Tx mode is shown below.

- `t`
  Tx started
  Tx 00:
  Tx 01:
  Tx 02:
  Tx 03:
  Tx 04:
  Tx 05:
  Tx 06:
  Tx 07:
  Tx 08:
  Tx 09:
  Tx 0A:
  `<ESC>`

More examples of Tx mode are shown in Chapter 6, Tx and Rx Demo.

3.1.18 U - Enter User Mode

Enters User mode, and executes code in UserApp.c. Examples throughout this text will use this mode for fun.
An example of the default User mode code is shown below:

```
-u
User app started
<ESC>
ESC!
ok
-
```

More examples for this mode follow in later chapters.

### 3.1.19 V - Set Variable Length Packets

Sets packets for variable length format.

An example of this command follows:

```
-v
Set variable length packets
-
```

### 3.1.20 W - Toggle Whitening

Toggles whitening of packets.

Examples are shown below for enabling and disabling whitening:

```
-w
Whitening disabled
-w
Whitening enabled
-
```

### 3.1.21 Y - Get CC1100 Chip Status

Reads the indicated CC1100 chip status byte and presents this value to the user.

An example is shown below:

```
-y
status = 0x0F
ok
-
```
3.1.22  Z - Reset Digital Radio

Resets the CC1100 and then loads the values contained in `DigitalRadio_ResetTable.s43`. Use sparingly as this reset bypasses many of the user options selected from the menu. Called automatically when the firmware starts.

An example is shown below:

```
-z
Resetting...
ok
```

3.1.23  0-7 - Set Channel 0 to 7

Sets the channel to the indicated selection. This value is written to a shadow variable, and only updated on the CC1100 hardware when the next radio operation is started.

In the following example, the CHANNR register (0x0A) is read. The channel is set to 3, then the CHANNR register is read again. The old value is still present:

```
-i
adr: 0x0a = 0x00
ok
-3
Set channel 03

-i
adr: 0x0a = 0x00
ok
```

To flush this channel number to the CC1100 hardware from the shadow, an Rx operation is started and stopped. The CHANNR register is read again, and is shown to have the desired contents:

```
-r
Rx started
<ESC>
Error count = 0000

-i
adr: 0x0a = 0x03
ok
```

It is left as an exercise for the reader to try the same approach with changing the CC1100 address (register ADDR, 0x09) via the `k` command, and flushing the shadow with an `r` command, while observing the results before and after the flush with the `i` command.
3.2 Hub Menu Commands

If hub mode (H) is selected from the main menu, then hub mode is started. In this mode, the prompt is a ‘+’, as shown below, for the case of an unconfigured hub:

```
-h
Hub mode started, serial #8765:4321
Loading hub file: failed 0x0004, file not found
Writing hub file to flash
+
```

If the hub already has a file defined in info flash, then the following will be seen instead:

```
-h
Hub mode started, serial #8888:6700
Loading hub file: success
Listening for nodes as hub
```

If this is the case, hit <ESC> to get back to the hub prompt:

```
-h
Hub mode started, serial #8888:6700
Loading hub file: success
Listening for nodes as hub
<ESC>
User cancelled operation
+
```

From the hub prompt, type ? for a hub menu overview:

```
+
Hub options:
? - this list
A - start active mode
D - display hub file
Z - erase hub file
1-4 - search for node N
<ESC> - exit to Main Menu
+
```

These commands are discussed in the following sections.
3.2.1 ? - Show Hub Menu

As mentioned in the previous section, this hub menu command displays the hub menu.

3.2.2 A - Start Active Mode

This menu option starts the active hub mode, in which the hub is waiting for messages from nodes. To exit this mode, press ESC.

An example is shown below:

```
+a
Listening for nodes as hub
<ESC>
User cancelled operation
+ 
```

More examples of hub mode operation are shown in Chapter 8, Hub/Node Demo.

3.2.3 D - Display Hub File

This menu option displays the current contents of the hub file.

An example of an empty hub file is shown below:

```
+d
Hub file contents:
wMagicNumber:      0x9387
wFeatures:         0x0036
byRxAddress:       0x40
byChannel:         0x02
byNodeRecordCount: 0x04

Node Record 0x01:  Unused
Node Record 0x02:  Unused
Node Record 0x03:  Unused
Node Record 0x04:  Unused

+ 
```

More examples of hub mode operation are shown in Chapter 8, Hub/Node Demo.

3.2.4 Z - Erase Hub File

The hub file will be erased with this menu option.

An example is shown below:
Erasing hub file...
Writing hub file to flash

Note that even though the file has been erased, a clean hub file has been initialized and written to flash. The hub always maintains a valid, if empty, file.

More examples of hub file operations are shown in Chapter 8, Hub/Node Demo.

3.2.5 1-4 - Search For Node N

The demo software supports up to four nodes, numbered 1 to 4. The hub can be commanded to search for nodes by pressing a key 1 to 4.

An example of searching for Node #3 is shown below.

+3
Searching for node 03
....................Search failed: 0008 (0001): timed out
+

In this case, the desired node could not be found.

More examples of hub mode operation are shown in Chapter 8, Hub/Node Demo.

3.2.6 <ESC> - Exit To Main Menu

To return to the main menu, hit ESC on the keyboard.

An example is shown below:

+<ESC>

User cancelled operation

Hub mode stopped
4. Firmware Architecture

The previous chapter described the menu operations available in the default demo firmware. This chapter describes the architecture used to build the demo firmware. The next chapter demonstrates portions of this architecture with applications using the UserApp.c module.

The demo firmware is a state-based application, and implements a miniature operating system framework developed by SoftBaugh for taking advantage of the low-power features of the MSP430. Our forthcoming MSP430 books will describe this state machine approach to firmware development in detail.

4.1 State Machine Design for MSP430s

The implementation of the demo firmware reflects a state machine architecture developed by SoftBaugh for the MSP430 since 1998 by adapting to the MSP430 software techniques that have been in existence for nearly forty years and which placed men on the Moon. The notation, however, is taken from various flavors of UML (Unified Modeling Language), for which gobs of literature exist and is condensed here into a thick salve.

A generic state machine diagram consistent (-enough) with UML notation is shown below:

![Figure 4-1 Generic State Machine Diagram](image-url)
The traditional notation for state machines is to represent states, or unique operating conditions of the system, as rounded boxes, with the state name contained within the box. The above diagram contains four states: InitialState, MyState, MyOtherState, and YetAnotherState.

The initial state of the system, or the state into which the system “boots”, is indicated by the anarchist’s bomb. The initial state of the system above is defined to be InitialState.

The system moves from state to state via transitions, shown as solid directional arrows. Some state transitions are unconditional, in which case the transition is not labeled. Conditional state transitions are indicated by the condition stated as text near the launching point of a transition. In the example above the transition between InitialState and MyState is unconditional, while the transition between MyState and MyOtherState is caused by ConditionA, whatever that is.

In some cases, a state transitions back to itself, as shown by the loops on MyState and YetAnotherState. This represents the default case in which none of the specified conditions hold true.

The transitions themselves are driven in our designs by a system tick, or discrete intervals of time for which the MSP430 is remarkably capable of measuring given the variety of timers available.

Representing one’s system in such a graphical notation allows some design aspects to be almost read off directly into code, or at least into an application framework.

IAR has an excellent tool, VisualSTATE, which can perform this conversion into code, and is a worthwhile investment for large, complex projects. In this document, however, we hand craft a state machine design.

4.1.1 State Overview

In code, a state is a function which has the following prototype, defined in Task_Structure.h:

```c
typedef void ( * LPFN_STATE_VOID_VOID )( void );
```

As an example, the following functions are properly implemented states:

```c
void MyState( void )
{
}

void MyOtherState( void )
{
}
```
4.1.2 Tasks

A task in our model consists of a single thread of execution through a group of related states. Multiple simultaneous tasks may exist in a system, and may overlay and share states. The task manager is responsible for maintaining the current state for each task, and is defined by code in Task.s43, available in source form in the licensed framework.

This demo firmware only supports one task, the main task, which is declared and initialized in main.c:

static STask m_tMainTask;

The states in a task are called during each system tick, defined in this demo as a resumption of the main thread based on the 8192 Hz TA.CCR1 interrupts:

```
while(1)
{
    __low_power_mode_0();
    P6OUT |= BIT0;
    Task_TickTask( &m_tMainTask );
    RxTxHost_TickTx();
    P6OUT &= ~BIT0;
}
```

Upon resumption of execution, P6.0 is set, then the main task is ticked. The RxTxHost terminal communication module is ticked before the P6.0 twiddle is cleared and the processor put to sleep again.

The framework of the application, particularly the Task module located in the demo library, manages the state stack and lightweight context switching. Most of the details of this architecture can be ignored by the user.

4.1.3 State Transitions

The important features of the state machine design will be illustrated by code samples in later chapters. Two design details are important to the user, however.

First, to switch states, use the Task_UpdateState API function, as shown below:

```c
void MyState( void )
{
    Task_UpdateState( MyOtherState );
}
```

Upon the next system tick, MyOtherState will be called.

The second design detail is the use of the task’s state stack. Using this stack, one can easily implement reusable state chains, or state functions.

To push states onto the stack, use the Task_PushState() API function, as shown below:
void MyFirstState( void )
{
    Task_PushState( MyLaterState );
    Task_UpdateState( MyNextState );
}

void MyNextState( void )
{
    
}

In the example above, the state *MyFirstState* executes. During execution of this state, *MyLaterState* is pushed onto the task’s state stack. Next, the task is updated to execute *MyNextState*. During execution of *MyNextState*, the previously pushed *MyLaterState* is patiently waiting on the task’s state stack.

To pop states from the stack, use the *Task_PopState()* API function, as shown below:

void MyNextState( void )
{
    Task_UpdateState( MyNextState2 );
}

void MyNextState2( void )
{
    Task_PopState();
}

Continuing the example, the state to execute after *MyNextState* is *MyNextState2*. During execution of *MyNextState2*, the return state *MyLaterState* is popped from the state stack, just as if the following call was implemented:

    Task_UpdateState( MyLaterState );

Note that the *MyNextState2* function needs no direct knowledge of what state should execute next, as this information is stored in the task’s state stack. This approach allows the chain of states to be independent of the terminal state.

This approach allows the implementation of what is known as a *state function*. Another example of a simple state function, and one which is less trivial, is given in the next section.
4.1.4 State Functions

Most readers are familiar with the use of code functions to encapsulate desired behavior. Use of code functions allow the implementation details to be hidden from the client code, in addition to allowing this implementation to be called, or reused, from various portions of the code.

Consider the following function, and two sets of client code which call it:

```c
void MyClient( void )
{
    ... other code
    MyFunction();
    ... more code
}

void MyOtherClient( void )
{
    ... other code
    MyFunction();
    ... more code
}

void MyFunction( void )
{
    ... Some code here
    return;
}
```

Of course, meaningful functions would probably pass parameters in and return values out, but for the discussion here the above simple examples suffice.

The diagrams below show the execution of this code graphically:

![Figure 4-2 Code Function Execution](image-url)
In this example, when entering *MyClient* the only information on the stack is the return address for this function. Upon hitting *MyFunction*, the return address is pushed onto the stack, and the PC updated to the function address, or 0x6094. Upon returning from the *MyFunction*, uhh, function, the return address is popped from the stack and execution continues accordingly.

The key reason for which the call-return paradigm was created lo these many years ago is for the case in which you wish to reuse the services of *MyFunction*:

```c
0x6024 void MyClient( void )
{
    ... Other code
    03E0:03E0
    03DE:03DE
    579A
    602A
    03E0:03E0
    03DE:03DE
    579A
    6072
    03E0:03E0
    03DE:03DE
    579A
    6094
    SP:03DE
    03E0:03E0
    03DE:03DE
    579A
    6098
    return;
}
```

**Figure 4-3 Reused Code Function**

As before, the stack is used to store the return address inside *MyOtherClient* when execution of *MyFunction* completes. The essential detail is that *MyFunction* need not know where the return should occur, it merely executes the implicit stack pop when done.

Why did we go to the trouble of giving an almost content-free description of call and return, when anyone likely reading this will certainly already understand this? And take the time to draw all those little dottie pointers?

Because...

*State functions* are analogous to code functions, in that a particular sequence of states is designed to be *called* from any point in a state stream, with control *returning* to the next state in the client chain.

Similar to code functions, such a *stream of server states* allows the implementation details of the server to be hidden from the user. In addition, these streams can be called from a variety of state chains and tasks. State functions also imply that the lower level state chains information about where to go once execution is complete.

Consider as an example the requirement to insert 100 delay states between some state and the next state. A naïve implementation would be as shown below:

```c
__no_init static WORD m_wTickCount;
```
void NaiveSomeState( void )
{
    // Go to NaiveNextState, but only after a delay
    // Prepare the counter
    m_wTickCount = wDelayTicks;

    // Transition states
    Task_UpdateState( NaiveDelayState );
}

void NaiveDelayState( void )
{
    // Update counter
    m_wTickCount--;
    if( m_wTickCount > 0 )
    {
        return;
    }

    // Count has expired, transition states
    Task_UpdateState( NaiveNextState );
}

void NaiveNextState( void )
{
    // Do something after the delay
}

Using state functions, a cleaner example can be presented, as shown in the following code fragment:

__no_init static WORD m_wTickCount;

void SomeState( void )
{
    // Go to NextState, but only after a delay
    EnterDelay( NextState, 100 );
}

void NextState( void )
{
    // Do something after the delay
}

Continuing the example, the delay entry can be implemented as follows:

void EnterDelay( LPFN_STATE_VOID_VOID pfnNextState, WORD wDelayTicks )
{
    // Store the following state
    Task_PushState( pfnNextState );

    // Prepare the counter
    m_wTickCount = wDelayTicks;

    // Transition states
    Task_UpdateState( DelayState );
}
Completing the delay features requires implementing a special delay state, as shown below:

```c
void DelayState( void )
{
    // Update counter
    m_wTickCount--; // Update counter
    if( m_wTickCount > 0 )
    {
        return;
    }
    // Count has expired, transition states
    Task_PopState();
}
```

A graphical representation of this approach is shown below:

![State Function Example](image)

**Figure 4-4 State Function Example**

In this example, *SomeState* prepares the use of the *DelayState* state function by preparing the count and pushing *NextState*, the return state, onto the task's state stack. *DelayState* ticks off the delay, and then, without needing any special knowledge of the higher-level state flow, pops *NextState* from the task's state stack. Execution then continues in *NextState*, as expected.
In our notation here, the double-wall around `DelayState` indicates a lower-level state function that is capable of reuse and which uses the state stack, as shown here.

In the previous example, 100 state ticks of delay were inserted between `SomeState` and `NextState`. Now, without reimplementing another set of delay code, a 200 tick delay can be created between `OtherState` and `AnotherNextState` as below:

```c
void OtherState( void )
{
    // Go to AnotherNextState, but only after a delay
    EnterDelay( AnotherNextState, 200 );
}

void AnotherNextState( void )
{
    // Do something after the delay
}
```

Using this technique, the delay feature has become a reusable state function that can be inserted, and parameterized, at will between any two states, just as code functions can be called, and parameterized, by various code portions.

![Diagram](image)

**Figure 4-5 Reused State Function**
As before, the *DelayState* is an independent stream of states that can be used from any state chain. As long as the calling state chain pushes the return state onto the state stack and prepares the state variables accordingly, the state function will provide its services to the caller. These last two criteria are handled nicely and in an encapsulated fashion by the *EnterDelay* function, which would usually be exported by the module hosting the state function.

Note also the fact that the terminal state differs from the previous example, as well as the delay interval. The need to terminate in different states is analogous to *branching* versus *calling* with code modules.

A *branch* flows into whatever terminal code (or state) is hard-coded into the program, while a *call-return* can arbitrary transfer code (or state) control to any desired reusable code (or state) fragment, with control returned cleanly back to the next logical step in the sequence.

The radio control library makes heavy use of this state-based approach to task management.

In fact, each of the key radio features consists of state functions to implement key reusable features, such as sending, receiving, and acknowledging packets. The user code is drawn into the state chain through the strategic use of handler functions, or callbacks, hooked into the radio control library.
4.1.5 Client Callbacks, or Hooks

Many times a state chain requires the client code to intervene and make decisions at critical points in the execution of this chain. Often, these decisions must be made based on immediate data available mid-execution, and cannot be pre-determined and thus parameterized upon entry.

In these circumstances, the recommended approach is to implement a client callback, or hook. The following diagram shows a representative chain of execution for such callbacks:

![Callback Interaction Diagram]

In the diagram above, the server exports some functionality supported by function $FnS$. $FnS$ takes, among others, a parameter which defines the address of a callback function $pfnCB$. This callback function must be implemented by any client wishing to use the services of $FnS$.

In the diagram, there are two clients, Client A and Client B, each of which implements callback functions $CB_A$ and $CB_B$, respectively.

When Client A executes its own function $FnA$, which calls $FnS$, the client supplies the address of its callback function $CB_A$. After some processing is done by the server, it then calls the Client A callback function $CB_A$, which supplies the needed data or action, and then returns back to the server. When the server completes executing, it returns to the caller, in this case $FnA$.

Sometime later, Client B calls the same server, and this time supplies its own callback function, $CB_B$, for use by the server. As a result, the server code is reusable across different clients, even though each client may supply different data or actions, thus customizing the action of the server function $FnS$ at runtime.
For example, assume your sensor host needs to send back different acknowledgement packet data, depending on the sensor data and/or unit reporting. This is a perfect example of the need for a client callback.

The user app code provided by the *RxTxModes* module within the demo library provides client callbacks after receipt of an incoming packet and before transmission of the acknowledgment packet in echo mode. The stubs of these callbacks, found in *UserApp.c* are presented below:

```c
BOOL UserApp_OnUpdatePacket_PostEchoRx(
    SDigitalRadioRxParams* ptDigRadRxParams
)
{
    return FALSE;
}

BOOL UserApp_OnUpdatePacket_PreEchoTx(
    SDigitalRadioTxParams* ptDigRadTxParams
)
{
    return FALSE;
}
```

The *SDigitalRadioRxParams* and *SDigitalRadioTxParams* structures are defined in *DigitalRadio_Structure.h* as follows:

```c
typedef struct StructDigitalRadioRxParams
{
    // Address of buffer to be populated with received data
    LPBYTE pabyRxData;

    // Size of bytes to be received into buffer
    BYTE byRxDataSize;

    // Count of bytes read upon completion
    BYTE byRxCountRead;

    // Sequence number received, if enabled
    BYTE byRxSequenceNumber;

    // Address for receive packet filtering, if applicable
    BYTE byRxAddress;

    // Ticks allowed prior to declaring a receive timeout
    // Set this to WAIT_RX_INFINITE to wait until cancelled
    WORD wRxTimeoutTicks;

    // Ticks allowed prior to aborting a receive packet in progress
    after sync

    // RSSI value to be populated upon completion, if enabled
    BYTE byRSSI;

    // LQI value to be populated upon completion, if enabled
```

... Additional lines omitted for brevity...

```c
    WORD wRxAbortTicks;

    // RSSI value to be populated upon completion, if enabled
    BYTE byRSSI;

    // LQI value to be populated upon completion, if enabled
```
BYTE byLQI;
} SDigitalRadioRxParams;

and

typedef struct StructDigitalRadioTxParams
{
  // Address of buffer populated with transmit data
  LPBYTE pabyTxData;

  // Size of transmit buffer contents
  BYTE byTxDataSize;

  // Count of bytes transmitted upon completion
  BYTE byTxCountSent;

  // Sequence number received, if enabled
  BYTE byTxSequenceNumber;

  // Address value to transmit in packet preamble, if enabled
  BYTE byTxAddress;

  // Ticks allowed prior to declaring a transmit (CCA) timeout
  WORD wTxTimeoutTicks;
} SDigitalRadioTxParams;

Clearly, by examination and manipulation of the data buffers and parameters exposed
by these structures, a client callback can dramatically influence the operation of the
packet radio at run-time and packet-time, without having to drill down into the internal
operation at all.

An excellent example of these callbacks would be to examine the incoming packet for
specific data, and then insert custom data fields into the outgoing acknowledgement
packet based on the data received. Such an example is provided in the chapter on
Echo/Ping modes.

The implementation of the details of the state machine system is provided in source form
to licensed users.

To summarize, the combination of task management functions, which allow creation of
encapsulated state streams, and client callbacks allow the creation of robust and simple
state machines which can implement surprisingly complex state patterns.

Variations on the theme of this state machine design will be provided in a forthcoming
series of SoftBaugh books on MSP430 design, including sample architectural patterns
used to solve a variety of MSP430 design problems.
4.2 Demo Modules

The demo software is intended to allow the user to build custom versions of the demo using the IAR KickStart 4kB limited version of the IAR Embedded Workbench, while still enjoying enough features of the licensed packet radio software to make a realistic evaluation of the library.

The modules available to the unlicensed user are:

- DigitalRadio_ResetTable.s43
- main.c
- SerialNumber.c
- System.s43
- UserApp.c
- Library_MRF1611CC1100.r43

Each of these modules is discussed in the following sections.

4.2.1 DigitalRadio_ResetTable.s43

This assembly language module contains a table used for preloaded CC1100 register settings. A fragment of this file is shown below:

```plaintext
PUBLIC DigitalRadio_abyResetTable
DigitalRadio_abyResetTable
// This table is used during execution of the DigitalRadio_ResetHW call
// Each entry will be used in the order given
// The first entry of each pair is the configuration register address
// The second entry of each pair is the configuration register value

// The following entries are for completeness
DC8 CC1100_FIFOTHR, FIFO_THR2 | FIFO_THR1 | FIFO_THR0
DC8 CC1100_SYNC1, 0xD3
DC8 CC1100_SYNC0, 0x91
DC8 CC1100_WOREVT1, 0x87
DC8 CC1100_WOREVT0, 0x6B
DC8 CC1100_WORCTRL, 0xF8

// The following entries are specified by Chipcon
DC8 CC1100_FSCTRL1, 0x0B
... Additional lines omitted for brevity
DC8 CC1100_PKTCTRL1, APPEND_STATUS
DC8 CC1100_PKTCTRL0, WHITE_DATA|CRC_EN|LENGTH_CONFIG0
DC8 CC1100_ADDR, 0x00
DC8 CC1100_PKTLEN, 0xFF

// The last entry must be 0xFF, 0xXX
DC8 0xFF, 0xFF

PUBLIC DigitalRadio_abyDefaultPATABLE
```

Page 58 of 301
The first section, the `DigitalRadio_abyResetTable` entries, define the CC1100 registers loaded on startup. Note that these entries are presented as pairs of ordered bytes, representing the configuration register addresses and values, respectively. Also note that the last entry in the table must be start with 0xFF to flag termination. The register names and constants are documented in the file `CC1100_Registers.h`.

The second section, `DigitalRadio_abyDefaultPATABLE`, defines the PATABLE settings. An informed RF expert can likely select different parameters with more optimum performance for your unique situation.

These settings are documented in the CC1100 documentation, SWRS038, from Texas Instruments. There is also a PC-based design tool, SmartRF Studio, available from Texas Instruments to assist with the design process.

### 4.2.2 main.c

This C language module contains the startup and initialization code for the system, as well as the main task loop. This module also initiates the state stream by making a one-way transition into the menu application.

### 4.2.3 SerialNumber.c

This C language module contains the serial number variable for the unit.

```c
#pragma constseg=SERNUM
static const DWORD dwSerial = 0x87654321;
#pragma constseg=default
```

It will be modified during the Hub / Node star network demos later. This serial number is placed into the fixed address 0xFFD8.

As a fixed address, the units can be serialized in production by the full version of the SoftBaugh PrgUSBP, if desired, provided the template value of this number is first changed to 0xFFFFFFFF, as shown below:

```c
#pragma constseg=SERNUM
static const DWORD dwSerial = 0xFFFFFFFF;
#pragma constseg=default
```
4.2.4 System.s43

This assembly language module contains various setup functions necessary to get the hardware into the proper operating condition. It is provided to allow the evaluator an opportunity to support user hardware while still using the demo library.

A later chapter discusses modifications to this file that can be made by a licensed user to switch to an external high-frequency crystal on XT2, if desired. An unlicensed user can also change to an external crystal, but the frequency choice is limited to 6 MHz.

4.2.5 UserApp.c

This C language module contains a sample state stream, launched from the menu, as well as sufficient hooks into the Tx, Rx, Echo, and Ping modes to allow the evaluator to build fairly interesting demo projects prior to purchasing a license to the library source code.

Later chapters will include sections in which modifications to this module are presented.

4.2.6 Library_MRF1611CC1100.r43

This library contains the compiled object code for the licensed portions of the system. Licensed users will receive the source code for this library. The library contains additional modules that, in source form, give the licensed user full control over the operation of the packet radio system.

The modules in the library which available to the licensed user in source code form are:

CalibrateDCO.s43
CC1100.s43
ChannelParameters.c
DigitalRadio.s43
Display.c
HubMode.c
InfoFlashDrive.s43
Menu.c
NodeMode.c
PacketRadio.c
RxTxHost.s43
RxTxModes.c
Switch.c
Task.s43
Utility.s43

Each of these modules, and their usage, is discussed in detail in the following chapters.
4.3 Licensed Modules

The relationship among the various licensed software components relevant to the CC1100 device is shown in the following diagram:

```
+-----------------+       +-----------------+
|                 |       |                 |
|  CC1100 IC      |       |  CC1100.s43     |
|                 |       |                 |
|                 |       |  DigitalRadio.s43|
|                 |       |                 |
|  PacketRadio.c  |       |                 |
|                 |       |                 |
|  in RxTxModes.c |       |  HubMode.c      |
|  Tx Mode        |       |  NodeMode.c     |
|  Ping Mode      |       |                 |
|  Rx Mode        |       |                 |
|  Echo Mode      |       |                 |
|                 |       |                 |
+-----------------+       +-----------------+

Figure 4-7 Packet Radio Application Layers
```

As shown in the above diagram, the CC1100.s43 module provides a software driver layer around the CC1100 SPI interface. This module is used in turn by various application components to provide a consistent means of interacting with the radio hardware.

DigitalRadio.s43 adds channel and other radio features to the low-level features of the CC1100.s43 module by synthesizing black box features from discrete register operations. In addition, the DigitalRadio.s43 module adds data block Rx and Tx operations, including optional addressing, raw sequence numbers, and timeout monitoring.

PacketRadio.c builds on the foundation of DigitalRadio.s43 to supply a variety of packet formats, depending on the specific features selected, and adds optional sequence number validation, link quality measurements, and optional automatic handling of acknowledgements and other handshaking. PacketRadio.c may also interact directly with the CC1100.s43 module to synthesize features not appropriate for DigitalRadio.s43.

The demo software provides three modules, RxTxModes.c, HubMode.c, and NodeMode.c, which interact with PacketRadio.c to demonstrate the six most common packet radio modes. Four of these modes, Rx Only, Tx Only, Echo, and Ping, are
implemented directly by the PacketRadio.c module, and controlled by selection of various options for packet Rx and Tx via this module.

The last two modes demonstrated in this firmware, Hub, and Node, are synthesized by grouping packet radio modes with star topology logic, serial numbers, and a flash drive system to implement a persistent star network consisting, in this demo, of one hub unit and up to four node units.

YourApp.c is a fictional placeholder module that represents custom packet radio applications written by the licensed user. The licensed code modules can be used to create packet radio applications either by calling existing functionality in the PacketRadio.c module, the CC1100.s43 driver module, and, to a much lesser extent, the DigitalRadio.s43 module, typically through the modification of code found in RxTxModes.c, HubMode.c, and NodeMode.c.

The sample code provided at the end of Chapter 7, Ping/Echo Demo, shows use of the packet radio functionality in detail. Licensed users will be able to modify the source for the packet radio and digital radio modules, and thus receive even more benefit from the packet radio functionality.
4.4 Packet Formats

The demo application uses the services of PacketRadio.c and DigitalRadio.s43 to implement packet exchanges between CC1100s. The packet formats relevant to the operation of the demo software is discussed in this section.

4.4.1 Basic CC1100 Packet Format

The basic packet format defined by the CC1100, and described in Section 15 of SWRS038, is as shown below:

![Default CC1100 Packet Format Diagram](image_url)

- **Preamble**: Inserted automatically in Tx, processed and removed by Rx
- **byLength**: Optional user fields processed in Tx, processed but not removed by Rx
- **byAddress**: Unprocessed user data

**Figure 4-8 Default CC1100 Packet Format**

Note that this packet format is absolutely incompatible with Zigbee, which is why this document does not discuss that protocol.

Not shown are the extra bytes inserted by forward error correction (FEC) or the effect of whitening, as these features are transparent to both the transmitter and the receiver.

This default packet format has three optional fields, `byLength`, `byAddress`, and `wCRC16`. As we shall see in following chapters, each of these fields control specific features on the receiver. In all cases, the settings of the receiver and transmitter must be consistent with each other to properly interpret any received data fields. The only exception to this rule is that in variable-length modes the presence of the `byLength` field allows the receiver to dynamically adjust its expectation of received packet size.

`byLength` is used by the transmitter to send variable length packets, and receipt and interpretation of this value instructs the receiver how many bytes to expect to extract from the channel following this value. This value must be the first BYTE transmitted in variable-length mode. This value is not used in fixed-length modes, and unfortunately variable-length packets are incompatible with forward error correction. These bytes are automatically inserted into the transmit stream, and interpreted by the receive stream, by
the *DigitalRadio.s43* module, based on the setting of the variable-length or fixed-length features.

*byAddress* must be the second BYTE transmitted in variable-length mode, and the first BYTE transmitted in fixed-length mode, if the receiver is to properly filter for addresses. Otherwise, a value sent in this position will be interpreted by the receiver as additional payload data. The *DigitalRadio.s43* module automatically inserts addresses into the transmitter stream, and enables address filtering on the receiver, based on the setting of the `DIGITAL_RADIO_FEATURE_ADDRESS` feature bit.

`wCRC16` is shown for clarity, but it is never seen directly by either the transmitter or receiver application software. It either exists or not based on the `CRC_EN` bit of the `PKTCTRL0` register. The success or failure of the CRC can be examined by receiver firmware through examination of the `CRC_OK` bit in the `LQI` status register. The *DigitalRadio.s43* module handles CRC enabling and filtering automatically based on the setting of the `DIGITAL_RADIO_FEATURE_CRC` feature bit, reporting errors as encountered to the application software.

This packet format is used by our demo software to implement higher-level packet formats, based on the various feature selections. These higher-packet formats can be divided into two classes, variable-length and fixed-length, and are discussed in the following sections.

In the following sections, the following key is used to discriminate among the various fields:

- **PPPP**: Preamble bytes
- **SS**: Sync bytes
- **L**: Length byte, representing the total length of the A:s:Dn fields
- **A**: Address byte
- **#:** Sequence number
- **KK**: Checksum (CRC-16) word
4.4.2 Variable-Length Packet Formats

The defining feature of variable-length packets is the presence of the length byte. The demo firmware supports eight different variable-length packet formats, as shown below:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPP</td>
<td>User Payload Data [Dn]</td>
</tr>
<tr>
<td>SS</td>
<td>L</td>
</tr>
<tr>
<td>PPPP</td>
<td>SS</td>
</tr>
<tr>
<td>PPPP</td>
<td>SS</td>
</tr>
<tr>
<td>PPPP</td>
<td>SS</td>
</tr>
<tr>
<td>PPPP</td>
<td>SS</td>
</tr>
<tr>
<td>PPPP</td>
<td>SS</td>
</tr>
<tr>
<td>PPPP</td>
<td>SS</td>
</tr>
</tbody>
</table>

Note that in each case, the amount of user data is the same. The DigitalRadio.s43 module handles support for each of the optional fields transparently to the user code. In all cases, the length byte is adjusted accordingly for the fields which follow.
### 4.4.3 Fixed-Length Packet Formats

The defining feature of fixed-length packets is the absence of the length byte. The demo firmware supports eight different fixed-length packet formats, as shown below:

- **Fixed-length, addressed, sequence numbers, and checksum**
  - PPPP
  - SS
  - A
  - #
  - User Payload Data [Dn]
  - KK

- **Fixed-length, addressed, sequence numbers, and no checksum**
  - PPPP
  - SS
  - A
  -#
  - User Payload Data [Dn]

- **Fixed-length, addressed, no sequence numbers, and checksum**
  - PPPP
  - SS
  - A
  - #
  - User Payload Data [Dn]
  - KK

- **Fixed-length, addressed, no sequence numbers, and no checksum**
  - PPPP
  - SS
  - A
  - User Payload Data [Dn]

- **Fixed-length, not addressed, sequence numbers, and checksum**
  - PPPP
  - SS
  - #
  - User Payload Data [Dn]
  - KK

- **Fixed-length, not addressed, sequence numbers, and no checksum**
  - PPPP
  - SS
  - #
  - User Payload Data [Dn]

- **Fixed-length, not addressed, no sequence numbers, and checksum**
  - PPPP
  - SS
  - User Payload Data [Dn]
  - KK

- **Fixed-length, not addressed, no sequence numbers, and no checksum**
  - PPPP
  - SS
  - User Payload Data [Dn]

**Figure 4-10 Fixed-Length Packet Formats**

Note that in each case, the amount of user data is the same. The `DigitalRadio.s43` module handles support for each of the optional fields transparently to the user code.

These formats apply to both the Fixed-Length, and Fixed-Length with FEC options.
5. User Application Demo

The previous chapter described the architecture used to build the demo firmware. This chapter demonstrates portions of this architecture with applications using the UserApp.c module. The next chapter will exercise the functionality of the simple Tx and Rx modes available in the demo firmware.

Prior to the chapters on RF experiments with the MRF1611CC1100, we will first experiment with the demo software to gain some familiarity with the state machine concepts discussed in the previous chapter. Note that the original UserApp.c source file is also provided as UserApp_Original.c, as we will revert to this original file from time to time.

The default original UserApp.c contains code to support a state machine demonstration, as well as callback functions for the Tx and Rx packet radio modes. The remainder of this chapter is devoted to state machine sample for the user application. Discussion of the other major feature set in UserApp.c, the callback handlers, is reserved for portions of the next chapter.

5.1 Default User Application Firmware

The default user application implements the state machine design shown below:

![State Machine Diagram]

**Figure 5-1 Default User Application**
The user application is entered when the user types “U” at the main menu. The Menu module calls UserApp_Start() to start the user application. Upon entry, the state UserApp_State_One is prepared.

The states cycle through _One, _Two, _Three, and _Four, unless an escape character is typed, or SW1 is pressed. If either terminal conditions occur, the terminating condition is printed to the terminal, and the application stops with the call to UserApp_Stop().

Otherwise, in each state the corresponding LED is illuminated, and all other LEDs turned off. Between each state is a delay of approximately one-eighth of a second.

### 5.1.1 Application Execution

To watch the default user application executing, enter the U command to start the user application from the main menu:

```
-u
User app started
```

Note that the LEDs are now cycling at 1/8th second intervals.

To exit the user application, either press ESC:

```
-u
User app started
<ESC>
ESC!
ok
-
```

or press SW1 on the HMRF1611:

```
-u
User app started
SW1!
ok
-
```

### 5.1.2 State Machine Implementation

Now, let’s examine each portion of this state machine in detail, referring to the source code contained in UserApp.c.

The entry point into the default user application is the UserApp_Start function:

```c
void UserApp_Start( LPFN_STATE_VOID_VOID pfnNextState )
{
```
// Save the next client state on the state stack
Task_PushState( pfnNextState );

RxTxHost_QueueTxString( "\nUser app started\n" );

// Clear LEDs
P1OUT &= ~(BIT3|BIT2|BIT1|BIT0);

// Prepare SW1 to generate IFG, but not interrupts, on a press
P1IES |= (BIT4);
P1IE &= ~(BIT4);
P1IFG &= ~(BIT4);

// Prepare next state
Task_UpdateState( UserApp_State_One );
}

As before with our fictional state function example, this function first saves the following state on the task stack, and then prints a message for the user on the host terminal.

The LEDs are then cleared, and SW1 is prepared for polling the interrupt flag on any press. This approach is superior to simply polling the input pin, as testing the interrupt flag ensures that any switch event will be captured. This is important as the switch test occurs only during one state out of many thousands, and these states execute for only a very thin slice of time.

The last line of code in the function prepares the next state for execution during the next tick.

Before considering the first true state in the user application, consider the UserApp_Stop function:

void UserApp_Stop( void )
{
    // Clear LEDs
    P1OUT &= ~(BIT3|BIT2|BIT1|BIT0);
    Task_PopState();
}

This function resets all LEDs, and then pops the state saved during the UserApp_Start function, effectively returning from the state function.

This stop function can be called any time it is desired to exit the user application. We highly recommend a stop function in your own applications, even if it only contains a call to pop the next state. Not only is this approach a graceful way to ensure symmetry with the start function and provides a single place to collect uninitialization code, it avoids littering the application with pop calls. This is reason enough to use a dedicated stop function, as only one missed, or repeated, pop will have disastrous consequences, much as a mismatched processor pop would.
The first true state in the application is the *UserApp_State_One* function, shown below:

```c
void UserApp_State_One( void )
{
    WORD wRxKey;

    // Check whether user pressed "ESC" on terminal
    wRxKey = RxTxHost_GetRxEcho();
    if( CHAR_ESCAPE == wRxKey )
        // User hit escape
        { RxTxHost_QueueTxString( "$ESC!" );
          UserApp_Stop();
          return;
        }

    // Also, check whether SW1 has been pressed
    if( 0 != (P1IFG & BIT4) )
        { RxTxHost_QueueTxString( "$SW1!" );
          UserApp_Stop();
          return;
        }

    P1OUT &= ~(BIT3|BIT2|BIT1|BIT0);
    P1OUT |= (BIT0);

    // Prepare next state
    Task_DelayTicks( UserApp_State_Two, LED_DEMO_DELAY_TICKS );
}
```

This workhorse function first checks whether the user has pressed the ESC key, and if so, prints a diagnostic message and exits the application.

Next, if the SW1 interrupt flag is set, then another diagnostic message is printed and the application exits.

Finally, the LEDs are updated to show D1, and the next state is prepared to execute after a 125 millisecond delay.

The next three states are all similar to the second state, shown below:

```c
void UserApp_State_Two( void )
{
    P1OUT &= ~(BIT3|BIT2|BIT1|BIT0);
    P1OUT |= (BIT1);

    // Prepare next state
    Task_DelayTicks( UserApp_State_Three, LED_DEMO_DELAY_TICKS );
}
```

This function, and the two which follow, updates the appropriate LED and prepares an additional eighth-second delay. The last such state feeds back into the first state, resulting in a state machine that matches *Figure 5-1 Default User Application*. 
5.2 Custom State Machine Application

In the previous section, we saw the inner workings of the default user application. In this section, we will evolve this default application into a more comprehensive user application, taking advantage of the switch debouncing services available in Switch.s43. This code sample is available in its entirety as UserApp_Switches.c.

5.2.1 Encapsulating the LEDs

The first change we will make is to encapsulate the LED functionality inside a black box function.

Near the top of UserApp.c, add the following #defines and function declaration:

```c
#define LED_DEMO_DELAY_TICKS 1024
#define LED_MASK (BIT3|BIT2|BIT1|BIT0)
#define LED_D1 BIT0
#define LED_D2 BIT1
#define LED_D3 BIT2
#define LED_D4 BIT3

void UserApp_Stop( void );
void UserApp_ShowLEDs( BYTE byLEDs );
```

Just beyond UserApp_State_Four, in the middle of UserApp.c, add the following function:

```c
void UserApp_ShowLEDs( BYTE byLEDs )
{
    BYTE byMaskedLEDs;

    // Turn off all LEDs
    P1OUT &= ~(LED_MASK);

    // Mask the input
    byMaskedLEDs = byLEDs & LED_MASK;

    // Write the LEDs
    P1OUT |= byMaskedLEDs;
}
```

In UserApp_Start, change the following lines of code:

```c
RxTxHost_QueueTxString( "\nNew user app started\n" );
// Clear LEDs
P1OUT &= ~(LED_MASK);
```
Similarly, in UserApp_Stop, change the following line of code:

```c
    // Clear LEDs
    P1OUT &= ~(LED_MASK);
```

In UserApp_State_One, make the following change:

```c
    // Also, check whether SW1 has been pressed
    if( 0 != (P1IFG & BIT4) )
    {
        RxTxHost_QueueTxString( "\nSW1!" );
        UserApp_Stop();
        return;
    }

    UserApp_ShowLEDs( LED_D1 );

    // Prepare next state
    Task_DelayTicks( UserApp_State_Two, LED_DEMO_DELAY_TICKS );
```

Similarly, change UserApp_State_Two:

```c
    void UserApp_State_Two( void )
    {
        UserApp_ShowLEDs( LED_D2 );

        // Prepare next state
        Task_DelayTicks( UserApp_State_Three, LED_DEMO_DELAY_TICKS );
    }
```

and UserApp_State_Three:

```c
    void UserApp_State_Three( void )
    {
        UserApp_ShowLEDs( LED_D3 );

        // Prepare next state
        Task_DelayTicks( UserApp_State_Four, LED_DEMO_DELAY_TICKS );
    }
```

and finally UserApp_State_Four:

```c
    void UserApp_State_Four( void )
    {
        UserApp_ShowLEDs( LED_D4 );

        // Prepare next state
        Task_DelayTicks( UserApp_State_One, LED_DEMO_DELAY_TICKS );
    }
```
This is an excellent example of refactoring code to encapsulate functionality once you know what portions can be reused, and how they should be parameterized. Excellent example, of course, if it works!

Rebuild the project with the Make button. Correct any errors that appear. Now press Debug to load the firmware onto the target board. Once the target has been programmed, press Go to execute the firmware. The following will appear on the host terminal:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...
ok
-

Use the U command to start the user application:

ok
-u
New user app started

Convince yourself that the user application is cycling the LEDs and that a SW1 or ESC stops the application.

Stop the debugger by pressing Stop Debugging.

5.2.2 Switch Debouncing

Switch.s43 contains debounce algorithms that support the four switches on the HMRF1611. These algorithms ultimately result in events fired as client callbacks. Although integrating the switch module requires a certain amount of initial work, adding additional switch features is almost trivial once this investment is made. As before, we will evolve the application rather than drop a completed application out of the blue.

First, stub in support for the switch debouncing found in Switch.s43. Add an #include at the top of UserApp.h:

#include "System.h"
#include "Switch.h"

Next, declare some switch event handlers and add a table of these handlers near the top of UserApp.h:

void UserApp_State_Four( void );

// Switch event handlers
void UserApp_OnSwitchDown1( void );
void UserApp_OnSwitchIgnore( void );

// Switch handler arrays
static const LPFN_SWITCH_EVENT_HANDLER_VOID_VOID mUserApp_cahSwitchHandlers[] =
{ UserApp_OnSwitchDown1,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore
};

This table defines which handler will be called by the switch module upon the indicated switch events.

Note that the “\” symbol simply means that this line wrapped in the display above. Your code should show the first two lines as one line, without the “\”, although it would compile as shown above.

In UserApp_Start, replace the switch port initialization with the switch module calls:

    // Clear LEDs
    P1OUT &= ~(LED_MASK);

    // Open the switch module
    Switch_Open();

    // Turn off repeat
    Switch_SetRepeatParameters( 0, 0 );

    // Register switch handlers
    Switch_RegisterEventHandlers( mUserApp_cahSwitchHandlers );

    // Prepare next state
    Task_UpdateState( UserApp_State_One );

Similarly, add a call to close the switch module near the top of UserApp_Stop:

void UserApp_Stop( void )
{
    // Close the switch module
    Switch_Close();
}

Just below UserApp_Stop, add the following two functions:

    // Switch event handlers
    void UserApp_OnSwitchDown1( void )
    {
    
    
Page 74 of 301
MSP430 RF Applications with the MRF1611CC1100

```c
RxTxHost_QueueTxString( "\nSW1!" );
UserApp_Stop();
}

void UserApp_OnSwitchIgnore( void )
{
}

Modify UserApp_State_One by removing the code that checks the switch:

// Check whether user pressed "ESC" on terminal
wRxKey = RxTxHost_GetRxEcho();
if( CHAR_ESCAPE == wRxKey )
// User hit escape
  {RxTxHost_QueueTxString( "\nESC!" );
    UserApp_Stop();
    return;
  }

UserApp_ShowLEDs( LED_D1 );
```

The switch module operates by having the `Switch_Tick` function called periodically. By so doing, the switch module is fed cycles with which to operate the debouncing algorithms and fire events. Fortunately, these ticks are not time-critical, and so can be called when the processor is otherwise idle.

An alert reader may wish to port these algorithms to a timer-based scheme. But here, we wanted to show a technique by which a timer resource is not required for some systems that already use a periodic tick of some sort.

We could sprinkle a bunch of calls to `Switch_Tick` around our application, but in many real-time systems there is a better way.

The best time to call the switch module tick is when the user application state machine is otherwise idle. When is this? Well, the state machine is technically idle while waiting out the state delays between LED changes. So, let’s insert switch module ticks into the LED delays by writing our own delay function.

First, add declarations for the delay entry function and the delay state near the top of UserApp.h:

```c
void UserApp_ShowLEDs( BYTE byLEDs );
void UserApp_StartDelay( LPFN_STATE_VOID_VOID pfnNextState, WORD wDelayCount );
void UserApp_State_Delay( void );
void UserApp_State_One( void );
```

Add a delay counter variable and a state variable just before the state handler table:
Add the following state function after UserApp_State_Four:

```c
void UserApp_State_Delay( void )
{
    // Decrement the counter
    mUserApp_wDelayCount--;

    if( 0 == mUserApp_wDelayCount )
    // Count expired
    { Task_UpdateState( mUserApp_pfnStateAfterTickDelay );
        return;
    }

    // Count has not expired
    // Now tick the switch module
    Switch_Tick();
}
```

And, add the following function after UserApp_ShowLEDs:

```c
void UserApp_StartDelay( LPFN_STATE_VOID_VOID pfnNextState,
                        WORD wDelayCount )
{
    // Save the following state
    mUserApp_pfnStateAfterTickDelay = pfnNextState;

    // Prepare the delay count
    mUserApp_wDelayCount = wDelayCount;

    // Change to the delay state
    Task_UpdateState( UserApp_State_Delay );
}
```

Finally, replace each call to Task_DelayTicks with a call to UserApp_StartDelay, as shown below for UserApp_State_One:

```c
// Prepare next state
UserApp_StartDelay( UserApp_State_Two, LED_DEMO_DELAY_TICKS );
```

and UserApp_State_Two:

```c
// Prepare next state
UserApp_StartDelay( UserApp_State_Three, LED_DEMO_DELAY_TICKS );
```

and UserApp_State_Three:
// Prepare next state
UserApp_StartDelay( UserApp_State_Four, LED_DEMO_DELAY_TICKS );

and UserApp_State_Four.

// Prepare next state
UserApp_StartDelay( UserApp_State_One, LED_DEMO_DELAY_TICKS );

Note that in the case of this state function, we are NOT using the task’s state stack. This is because the switch callback must execute in the context of the user application state machine. If we pushed the return state on the state stack, then when a switch event handler is called, we would have to contend with a state higher up on the stack rather than the state at our current level. The implication would be the requirement to insert an extra Task_PopState in the switch handler.

In some applications that approach would be fine, but to keep our example simple we have implemented it as shown.

And, WOW, that is a lot of stuff for just having a switch make our application exit! Clearly, in the case of just a simple one-time-only switch event, watching the interrupt flag is a lot simpler.

But we’re not done yet, and before proceeding further build and debug the application, making sure that the SW1 event still causes the application to exit.

Note, however, the change in switch operation. Unlike the previous version, in which the switch event was not registered until the UserApp_State_One executed, in this version the switch event is registered immediately and the application exits right away.

5.2.3 Freezing the Application

Now that we’ve laid the framework for using switch events in our application, let’s make the application do something more interesting.

First, we shall make SW4 “freeze” the application in the current state. To do so, add the following switch event handler declarations to the top of UserApp.c:

```c
void UserApp_OnSwitchDown1( void );
void UserApp_OnSwitchDown4( void );
void UserApp_OnSwitchUp4( void );
void UserApp_OnSwitchIgnore( void );
```

and update the switch event handler table:

```c
static const LPFN_SWITCH_EVENT_HANDLER_VOID_VOID mUserApp_cahSwitchHandlers[] =
{ UserApp_OnSwitchDown1,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
  UserApp_OnSwitchIgnore,
```
Recall that the "\" symbol simply means that this line wrapped in the display above. Your code should show the first two lines as one line, without the "\".

Also, let's add a flag for determining whether the app is frozen or not. The non-MSP430 approach would be to add a BOOL variable:

__no_init BOOL mUserApp_bFreezeApp;

But, the MSP430 is very efficient at handling bit flags, especially those located in the bottom four bits of a BYTE. So, we will instead create the following #define near the top of UserApp.c:

#define LED_D4 BIT3
#define USER_APP_FLAG_FROZEN BIT0

and define the following static flags variable near the top of UserApp.c:

__no_init static WORD mUserApp_wDelayCount;
__no_init static LPFN_STATE_VOID_VOID mUserApp_pfnStateAfterTickDelay;
__no_init static BYTE mUserApp_byUserAppFlags;
#pragma diag_suppress=Pe177
__no_init static BYTE mUserApp_byDummy;
#pragma diag_default=Pe177

The dummy byte ensures WORD alignment. Although the compiler and linker will normally manage this alignment just fine, it helps remove some warnings seen in some programming tools later.

Also, by leaving this dummy BYTE in place, you can later transform it into a second flags BYTE and get an extra four high speed flags, for a total of eight, by using two BYTES. Packing all flags into a single WORD would yield instead only four high speed flags but twelve low speed flags.

The diag Suppress pragma turns off the compiler warning for the fact that this dummy variable isn’t referenced anywhere, while the diag_default pragma turns the compiler warning back on.

Add the following initialization to the top of UserApp_Start:

// Clear flags
mUserApp_byFlags = 0x00;

// Clear LEDs
P1OUT &= ~(LED_MASK);
Since the user application can be re-entered, we explicitly initialize on each entry, rather than counting on the zero-initialization services of the C framework.

Add the following two switch event handler functions prior to `UserApp_OnSwitchIgnore`:

```c
void UserApp_OnSwitchDown4( void )
{
    RxTxHost_QueueTxString( "\nSW4!" );
    mUserApp_byFlags |= (USER_APP_FLAG_FROZEN);
}

void UserApp_OnSwitchUp4( void )
{
    RxTxHost_QueueTxString( "~SW4!" );
    mUserApp_byFlags &= ~(USER_APP_FLAG_FROZEN);
}
```

Add the following conditional wrapper around the decrementing counter inside `UserApp_State_Delay`:

```c
// Decrement the counter, but only if the app is not frozen
if( 0 == (USER_APP_FLAG_FROZEN & mUserApp_byFlags) )
{
    mUserApp_wDelayCount--;
}
```

Now build the app with the `Make` button, and correct any errors that are reported. Debug the application with the `Debug` button, and execute with the `Go` button. Close the debugger with the `Stop Debugging` button.

Start the user application from the terminal, and test the new SW4 functionality. Note that you can still shut down the user application using SW1 even if SW4 is held down:
But, as you may have noticed, the ESC command has no effect until the state machine hits UserApp_State_One after being unfrozen. This command is still suffering from the latency problem that SW1 had earlier. But, this is easy enough to fix.

5.2.4 Encapsulating the ESC Exit

First, at the top of UserApp.c declare a function in which to encapsulate the ESC check:

```c
void UserApp_StartDelay( LPFN_STATE_VOID_VOID pfnNextState ... 
void UserApp_CheckESC( void );
```

Now, stub in this function after UserApp_StartDelay:

```c
void UserApp_CheckESC( void )
{
}
```

Next, cut the wRxKey variable declaration and related ESC code from the top of UserApp_State_One:

```c
void UserApp_State_One( void )
{
    UserApp_ShowLEDs( LED_D1 );
    // Prepare next state
    UserApp_StartDelay( UserApp_State_Two, LED_DEMO_DELAY_TICKS );
}
```

and paste it into UserApp_CheckESC, removing the now spurious return statement:

```c
void UserApp_CheckESC( void )
{
    WORD wRxKey;
    // Check whether user pressed "ESC" on terminal
    wRxKey = RxTxHost_GetRxEcho();
    if( CHAR_ESCAPE == wRxKey )
        // User hit escape
        {RxTxHost_QueueTxString( "\nESC!" );
         UserApp_Stop();
        };
}
```

Finally, add a call to this function at the top of UserApp_State_Delay:

```c
void UserApp_State_Delay( void )
{
    UserApp_CheckESC();
}
// Decrement the counter, but only if the app is not frozen
if( 0 == (USER_APP_FLAG_FROZEN & mUserApp_byFlags) )
{
    mUserApp_wDelayCount--;
}

Rebuild and retest with the debugger. Note that the ESC functionality is immediate now, and works even if the application is frozen.

Also note that this was an easy change to make since the delay is executing in the same state stack context as the main states. Otherwise, if the main states were stacked, this change would have required some housekeeping to make sure that the task state stack was not corrupted.

The downside of this change is that asking the RxTxHost module whether a character was received during each idle tick can add up to an expensive form of latency prevention!

To mitigate the effect of this increased power consumption, we will only test for an ESC a) during each frozen tick, and b) during each active tick. To do so, change UserApp_State_Delay as follows:

void UserApp_State_Delay( void )
{
    UserApp_CheckESC();

    // Decrement the counter, but only if the app is not frozen
    if( 0 == (USER_APP_FLAG_FROZEN & mUserApp_byFlags) )
        // Not frozen
        { mUserApp_wDelayCount--; }
    else
        // Frozen
        { // Check for ESC while frozen
            UserApp_CheckESC();
        }

    if( 0 == mUserApp_wDelayCount )
        // Count expired
        { Task_UpdateState( mUserApp_pfnStateAfterTickDelay );

            // Check for ESC prior to the next active tick
            UserApp_CheckESC();
            return;
        }

    // Count has not expired
    // Now tick the switch module
    Switch_Tick();
}

This change avoids checking during idle ticks, and save a significant amount of cycles, and thus current.
The one downside of this approach is that the switch module is always ticked after an ESC is detected. So, if a switch event changes the state of the system on the same tick that an ESC is detected, then the switch event would take precedence and the ESC would be ignored.

There are multiple ways around this that we leave as an exercise to the reader! The bottom line is that you must analyze your actual system to decide which event should have priority.

### 5.2.5 Reversing the Scroll Direction

Let's make one more change to the user application and call it a day. Suppose we use SW3 to change the direction in which the display scrolls. Add the following `#define` at the top of `UserApp.c`:

```c
#define USER_APP_FLAG_FROZEN BIT0
#define USER_APP_FLAG_FORWARD BIT1
```

Declare a new switch event handler:

```c
void UserApp_OnSwitchDown1( void );
void UserApp_OnSwitchDown3( void );
void UserApp_OnSwitchDown4( void );
```

and update the switch event handler table:

```c
static const LPFN_SWITCH_EVENT_HANDLER_VOID_VOID mUserApp_cahSwitchHandlers[] = {
    UserApp_OnSwitchDown1,
    UserApp_OnSwitchIgnore,
    UserApp_OnSwitchIgnore,
    UserApp_OnSwitchIgnore,
    UserApp_OnSwitchDown3,
    UserApp_OnSwitchIgnore,
    UserApp_OnSwitchDown4,
    UserApp_OnSwitchUp4
};
```

Update the initialization code for the flags in `UserApp_Start`:

```c
    // Initialize flags
    mUserApp_byFlags = USER_APP_FLAG_FORWARD;
```

Note that leaving this initialization at 0x00 would have resulted in the state machine starting off cycling in reverse!

Add the new handler function after `UserApp_OnSwitchDown1`:

```c
void UserApp_OnSwitchDown3( void )
```
RxTxHost_QueueTxString( "\nSW3!" );
mUserApp_byFlags ^= (USER_APP_FLAG_FORWARD);

Now add a conditional wrapper to the state change in UserApp_State_One:

```c
// Prepare next state
if( mUserApp_byFlags & USER_APP_FLAG_FORWARD ){
    UserApp_StartDelay( UserApp_State_Two,
                        LED_DEMO_DELAY_TICKS );
}
else {
    UserApp_StartDelay( UserApp_State_Four,
                        LED_DEMO_DELAY_TICKS );
}
```

and UserApp_State_Two:

```c
// Prepare next state
if( mUserApp_byFlags & USER_APP_FLAG_FORWARD ){
    UserApp_StartDelay( UserApp_State_Three,
                        LED_DEMO_DELAY_TICKS );
}
else {
    UserApp_StartDelay( UserApp_State_One,
                        LED_DEMO_DELAY_TICKS );
}
```

and UserApp_State_Three:

```c
// Prepare next state
if( mUserApp_byFlags & USER_APP_FLAG_FORWARD ){
    UserApp_StartDelay( UserApp_State_Four,
                        LED_DEMO_DELAY_TICKS );
}
else {
    UserApp_StartDelay( UserApp_State_Two,
                        LED_DEMO_DELAY_TICKS );
}
```

and UserApp_State_Four:

```c
// Prepare next state
if( mUserApp_byFlags & USER_APP_FLAG_FORWARD ){
    UserApp_StartDelay( UserApp_State_One,
                        LED_DEMO_DELAY_TICKS );
}
else {
    UserApp_StartDelay( UserApp_State_Three,
                        LED_DEMO_DELAY_TICKS );
}
```
Build, debug and test. Note that the reverse command still works even while the application is frozen, and that the debounce functionality encapsulated inside the switch module defeats any reasonable (or even best) effort to confuse it!

In this chapter we presented an example of working with state-based applications without any radio components. In the following chapters, we shall apply some of these same techniques to packet radio operations.
6. Tx and Rx Demo

The previous chapter demonstrated portions of the demo firmware architecture with applications using the UserApp.c module. This chapter exercises the functionality of the simple Tx and Rx modes available in the demo firmware. The next chapter will exercise the more advanced point-to-point functionality available in the Ping and Echo modes.

Before proceeding further, overwrite UserApp.c with the default UserApp_Original.c.

6.1 Tx / Rx Architecture

The simplest form of packet radio operations with the MRF1611CC1100 is the use of Rx-only and Tx-only modes. Internally, these operations are implemented as only the first half of an Echo or Ping mode, respectively:

![Tx to Rx Mode Diagram](image)

Figure 6-1 Tx to Rx Mode

Note the missing returned acknowledge packet, which will appear in the next chapter.

6.2 Default Operation

This section describes the default operation of Tx-only and Rx-only modes of the DemoMRF1611CC1100 project.

6.2.1 Materials

To perform the experiments in this section, you will need the following materials:

- Unit A and Unit B, each consisting of HMRF1611 host boards with installed BRS232s and MRF1611CC1100 modules.

- PC A and PC B, each running the DemoMRF1611CC1100.ht session.

Make sure both of the MRF1611CC1100 modules have the highly calibrated antenna wire attached, as discussed in Subsection 2.3.2, Highly Calibrated Antenna.
6.2.2 Preparation

Prepare the two HMRF1611 host board assemblies by installing the jumpers for Menu Mode and Connection to PC. Refer to Subsection 2.4.3, Digital Options Header, for details about these jumper settings.

One unit will be connected to and powered by your USBP JTAG debugger. We will refer to this unit as Unit A. The second unit, Unit B, will be battery powered.

Program Unit B with the demo firmware from within the PrgUSBP application using the pre-linked DemoMRF1611CC1100.txt file available as the Default Demo operation, discussed previously in Section 2.5.1, Loading Firmware with PrgUSBP.

Install batteries into Unit B. Make sure each cell has a no-load voltage of 1.2v or higher for the purposes of these demos.

Also, ensure that the Power Options header for Unit B is set to Power On, as shown in Subsection 2.4.1, Power Options Header.

Plug Unit B into the second PC’s serial port. Cycle power on Unit B (unplug and reinstall the VB:VIN jumper), and the following should appear on its terminal:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

ok
-

If you see nothing, make sure that the serial port cable is installed, that the batteries individually are at 1.2v or higher, that the BRS232 is installed correctly, that the demo firmware has been loaded onto Unit B, and that the jumpers are configured as detailed above.

If you see the copyright notice, but not the ok message followed by the main menu prompt -, then make sure the jumpers are in Menu mode and try again.

Now, attach Unit A to the USBP, and plug Unit A into the first PC’s serial port. Program Unit A with the demo firmware from within IAR Embedded Workbench by use of the Debug button, and execute the firmware by pressing the Go button.

The same copyright notice and prompt should appear on Unit A’s terminal. If not, debug the setup until it does.

Once both terminals show the above, you are ready to proceed to the next section.
6.2.3 First Execution

We will place Unit B into transmit mode and Unit A into receive mode, but first let us make sure that both units are in the proper software configuration. From each terminal, display the channel parameters with the D command:

- d
  Current channel parameters:
  Addresses disabled
  CRC enabled
  RSSI/LQI enabled
  Whitening enabled
  Sequence numbers enabled
  Using channel 0x00 for variable length packets

Become familiar with the use of the D command to show the channel parameters, and the commands which affect each of these parameters, as you will be using this technique often through the next few chapters.

Make sure both units are configured as shown above before proceeding. If they are not so configured, then use the menu options discussed in that previous chapter until they are as shown above.

Next, place Unit A into receive mode via the R command:

- r
  Rx started

The prompt disappears, and the terminal is now waiting for traffic. Some error messages may appear if the environment is noisy and the unit receives, filters, and rejects unrecognized traffic. For example:

  Rx failed: 000A (0000): invalid CRC
  Rx failed: 0012 (0000): RXFIFO overflow

This is OK.

Now, place Unit B into transmit mode via the T command:

- t
  Tx started
  Tx 00:
  Tx 01:
  Tx 02:
  Tx 03:
  Tx 04:
  Tx 05:
Unit B begins continuous packet transmission, and Unit A will now show a scrolling packet summary:

... other stuff
Rx 00: 00 2D 0D 00 00 00 06 07 ... 2.06v [0A:1B]
Rx 01: 00 2D 0D 00 00 00 06 07 ... 2.06v [0C:0A]
Rx 02: 00 2C 0D 00 00 00 06 07 ... 2.06v [0C:11]
Rx 03: 00 2C 0D 00 00 00 06 07 ... 2.06v [0B:0B]
Rx 04: 00 2C 0D 00 00 00 06 07 ... 2.06v [0A:15]
Rx 05: 00 2D 0D 00 00 00 06 07 ... 2.06v [0B:10]
Rx 06: 00 2D 0D 00 00 00 06 07 ... 2.06v [0A:0B]
Rx 07: 00 2C 0D 00 00 00 06 07 ... 2.06v [0A:13]
Rx 08: 00 2C 0D 00 00 00 06 07 ... 2.06v [0A:10]
... more stuff

If you do not see a display from Unit A similar to that above, debug your setup and channel parameters before proceeding.

For fun, press some switches on the transmitter, and watch the LEDs on the receiver. Note that the LEDs on the receiver echo the switches on the transmitter.

We shall see in a following section how the switches are communicated to the receiver, but can you determine how this happens by watching the display on the receiver’s terminal?
6.2.4 Packet Format

The Tx display simply shows that a packet has been sent and the sequence number of that packet, but the Rx display is much more interesting.

The first byte after the Rx tag, but before the colon, is the sequence number of the received packet:

```
Rx 00: 00 2D 0D 00 00 00 06 07 ... 2.06v [0A:1B]
```

If a packet is skipped, you will see a message similar to the following:

```
Rx 09: 00 2D 0D 00 00 00 06 07 ... 2.06v [06:0A]
Skipped packets 0A to FF
Rx 00: 00 2D 0D 00 00 00 06 07 ... 2.06v [08:0A]
```

You can force such an error by stopping and starting the transmitter.

Recall from the main menu chapter that ESC stops the Tx mode (and Rx, and Ping, and Echo modes also). So, stop, restart, and stop again the transmit mode by using the following key sequence (ESC T ESC), entered as quickly as practical while the unit is in transmit mode:

```
Tx 85:
Tx 86:
Tx 87:
<ESC> t
Tx started
Tx 00:
Tx 01:
Tx 02:
Tx 03:
Tx 04:
Tx 05:
Tx 06:
Tx 07:
Tx 08:
Tx 07:
Tx 08:
Tx 09:
Tx 0A:
Tx 0B:
<ESC>
```

If you examine the Rx display, you will see a skipped packets message, unless you were remarkably fortunate in stopping the original sequence at packet 0xFF!

After the colon, the next eight bytes displayed at the receiver are the first eight bytes of the packet payload data:
For all the Rx, Tx, Ping and Echo examples in this document, the payload data consists of 32 bytes, organized as follows:

<table>
<thead>
<tr>
<th>Table 6-1 Rx Mode Payload Data Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6-31</td>
</tr>
</tbody>
</table>

The Hub and Node examples use different and context-dependent payload data, which will be explained in a later chapter. Also, the Tx and Rx modes do not acknowledge receipt, so offsets 4 and 5 will always be set to 0x00 in this chapter.

The ellipses represent hidden payload data at offsets 8 to 31, while the next value in the receiver display is the transmitter’s potentiometer setting, as a voltage from 0.00v to 2.50v:

Rx 00: 00 2D 0D 00 00 00 06 07 ... 2.06v [0A:1B]

For fun, twirl the transmitter’s potentiometer and watch the displayed voltage change.

Rx 00: 00 3C 08 00 00 00 06 07 ... 1.29v [11:15]

The last portion of the receiver display is the RSSI/LQI measurement at the receiver for the incoming packet:

Rx 00: 00 3C 08 00 00 00 06 07 ... 1.29v [11:15]

These values provide an estimate of how well the transmitted packet is getting through. Refer to the CC1100 datasheet for a detailed description of how these values are to be interpreted. For our purposes here, just remember that the RSSI is a signed 8-bit value that ranges from 0x80 (horrible signal strength) through 0x00 (barely OK) to 0x7F (screaming), and that LQI is an unsigned 7-bit value that ranges from 0x00 (unhappy) to 0x7F (delighted).

We will see in a moment how the ability to view RSSI and LQI values on the fly will help determine how your particular circumstances affect the channel. But first, we will play with the channel parameters a bit to get some familiarity with the options available in the demo firmware.
6.2.5 Sequence Numbers

The first bit of fun we’ll have with the Rx/Tx modes is to play with the sequence numbers.

A sequence number is an incrementing number applied to the framing of a packet to allow the receiver to determine if previous packets might have been lost. In addition, this number can be turned back around to the transmitter in the form of an acknowledgement to tell the transmitter whether a packet might have been lost.

Programmatically, sequence numbers are defined in transmitted packets by setting the byTxSequenceNumber member of the SDigitalRadioTxParams structure:

```c
m_byLastSequenceNumber = m_bySequenceNumber++;  
// Update the Tx packet with the new sequence number  
m_tDigRadTxParams.byTxSequenceNumber = m_byLastSequenceNumber;
```

At the receiver, the sequence number received in the current packet can be examined by viewing the byRxSequenceNumber member of the SDigitalRadioRxParams structure:

```c
// Get the current sequence number  
m_byThisSequenceNumber = m_tDigRadRxParams.byRxSequenceNumber;

// Make sure the sequence numbers match  
if( m_byLastSequenceNumber != m_byThisSequenceNumber )  
{  
    // Do something  
}

m_byLastSequenceNumber = m_byThisSequenceNumber;
```

The sequence number feature is controlled directly for both the transmitter and receiver by manipulating the DIGITAL_RADIO_FEATURE_SEQUENCE bit with the DigitalRadio_SetFeatures API function:

```c
m_wFeatures |= DIGITAL_RADIO_FEATURE_SEQUENCE;  
DigitalRadio_SetFeatures( m_wFeatures );
```

Alternatively, the sequence numbers are enabled indirectly by calling the API function:

```c
// Enable sequence numbers  
ChannelParameters_EnableSequenceNumbers();
```

or

```c
// Disable sequence numbers  
ChannelParameters_DisableSequenceNumbers();
```
or

```c
// Toggle sequence numbers
ChannelParameters_ToggleSequenceNumbers();
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

From the main menu, sequence numbers can be enabled or disabled through the use of the `S` command, which toggles the sequence number feature on and off, as discussed in Subsection 3.1.16.

Now let us experiment with the sequence numbers.

On Unit B, make sure sequence numbers are enabled. Recall that you can always check the status of the channel parameters by using the `D` command, but first you have to stop transmission with `ESC`, and then restart transmission with the `T` command:

```plaintext
... other stuff
Tx AE:
Tx AF:
Tx B0:
Tx B1:
<ESC>
-d
Current channel parameters:
Addresses disabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets
-t
Tx started
Tx 00:
Tx 01:
Tx 02:
... more stuff
```

On Unit A, stop reception with `ESC`, and disable sequence numbers with the `S` command:

```plaintext
... other stuff
Rx A2: 00 3C 08 00 00 00 06 07 ... 1.29v [11:15]
Rx A3: 00 3C 08 00 00 00 06 07 ... 1.29v [11:11]
Rx A4: 00 3C 08 00 00 00 06 07 ... 1.29v [0F:12]
Rx A5: 00 3C 08 00 00 00 06 07 ... 1.29v [11:0D]
```
Rx A6: 00 3D 08 00 00 00 06 07 ... 1.29v [0F:17]

<ESC>
Error count = 0003

-s
Sequence numbers disabled

- 

Note that the receiver has reported how many errors have been encountered since it was started. The actual number displayed here for you will differ.

Now restart reception with the \textit{R} command:

- r
Rx started
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
... more stuff

This error means that the receiver expected 20h (32 decimal) bytes of packet data, but instead received 21h (33 decimal bytes). This is because the transmitter was sending a sequence number, but the receiver was not prepared to receive it. This error occurs because the CC1100 does not natively support sequence numbers, but instead the sequence number is prepended to the payload data as a kind of “meta-payload”.

Let’s swap how sequence numbers are enabled at the receiver and transmitter and see what happens. So, stop reception at Unit A with \textit{ESC} and enable sequence numbers:

... other stuff
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx
<ESC>
-s
Sequence numbers enabled

- 

Next, \textit{ESC} to the main menu on Unit B (transmitter) and disable sequence numbers:

... other stuff
Tx 9A:
Tx 9B:
Tx 9C:
Tx 9D:
<ESC>
-s
Sequence numbers disabled
and restart reception at Unit A with the R command:

```
-r
Rx started
Rx 00: 3C 08 00 00 00 06 07 08 ... 0.00v [16:0A]
Rx 00: 3C 08 00 00 00 06 07 08 ... 0.00v [13:15]
Skipped packets 01 to FF
Rx 00: 3C 08 00 00 00 06 07 08 ... 0.00v [13:11]
Skipped packets 01 to FF
Rx 00: 3C 08 00 00 00 06 07 08 ... 0.00v [14:11]
Skipped packets 01 to FF
Rx 00: 3C 08 00 00 00 06 07 08 ... 0.00v [12:13]
Skipped packets 01 to FF
Rx 00: 3C 08 00 00 00 06 07 08 ... 0.00v [12:11]
Skipped packets 01 to FF
```

In this case, the receiver was expecting 32+1 bytes of meta-payload, but instead only received 32 bytes. But, since the packet itself contained length information, and the received data could fit in the buffer, it was properly received at the low level of the `DigitalRadio.s43` module.

Note, however, that the first byte, 0x00, was improperly interpreted as the sequence number suffix, and all bytes in the packet were then shifted to the left one position.

Note also that every two to six packets, the following line may appear:

```
Rx 00: +++
```

The +++ symbols are an indication that there was a packet to be displayed, but insufficient room existed to queue it to the terminal.

The demo software is timed to allow full streaming of the packets to the terminal, but only if there aren’t error messages to be displayed. The constant stream of skipped packet messages can cause the outgoing terminal buffer on the MRF1611CC1100 to be overflowed. So, if there isn’t room for a packet plus error message, the packet isn’t displayed except for the sequence number and the +++ symbols.

Now, disable sequence numbers at both the transmitter and receiver, and try again:

```
Rx 00: 00 3C 08 00 00 06 07 ... 1.29v [11:11]
```
Skipped packets 01 to FF
Rx 00: 00 3D 08 00 00 00 06 07 ... 1.29v [11:0F]
Skipped packets 01 to FF
Rx 00: 00 3D 08 00 00 00 06 07 ... 1.29v [0F:15]
Skipped packets 01 to FF
Rx 00: 00 3C 08 00 00 00 06 07 ... 1.29v [14:13]
Skipped packets 01 to FF
... more stuff

This time, while the skipped packet message remains, at least the data is intact once again because the transmitted data and the received data are re-aligned.

An alert reader may wonder why, if the sequence numbers are disabled at the receiver, that the receiver shows a warning if they are missing. This is a valid question. It would be a straight-forward process to disable the skipped packet warning by modifying the PrintSequenceNumberWarnings function within RxTxModes.c in the licensed library:

```c
void PrintSequenceNumberWarnings( void )
{
    WORD wFeatures;

    ... Other code

    wFeatures = ChannelParameters_GetFeatures();
    if( 0 == ( wFeatures & DIGITAL_RADIO_FEATURE_SEQUENCE ) )
        // Ignore sequence numbers
        { return;
        }

    ... More code
}
```

This is left as an exercise for the licensed reader.

But, since sequence numbers are so important to robust communication, and are an essential element required for communicating the need for retries, we elected to leave this warning in place as a alert for the unwary.
6.2.6 Addressing

In this section, we will experiment with addressing. First, make sure both units are in the default mode (remember to hit ESC to return to the main menu). Recall that you can view the current channel settings with the D command, as shown below:

`-d`

Current channel parameters:
Addresses disabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

Programmatically, addresses are enabled with the `DigitalRadio_SetFeatures` function by setting the `DIGITAL_RADIO_FEATURE_ADDRESS` bit:

```c
m_wFeatures |= DIGITAL_RADIO_FEATURE_ADDRESS;
DigitalRadio_SetFeatures( m_wFeatures );
```

Alternatively, addresses are enabled indirectly by calling the API function:

```c
// Enable address
ChannelParameters_EnableAddress();
```

or

```c
// Disable address
ChannelParameters_DisableAddress();
```

or

```c
// Toggle address
ChannelParameters_ToggleAddress();
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

The transmitter’s address is directly set through the use of the `byTxAddress` member of the `SDigitalRadioTxParams` structure:

```c
m_tDigRadTxParams.byTxAddress = 0x22;
```

while the receiver’s address is directly set through the use of the `byRxAddress` member of the `SDigitalRadioRxParams` structure:
m_tDigRadRxParams.byRxAddress = 0x22;

The address is set indirectly by calling the API function:

    // Store the current address
    ChannelParameters_SetAddress( 0x22 );

and later setting the transmitter address:

    m_tDigRadTxParams.byTxAddress = ChannelParameters_GetAddress();

or receiver address:

    m_tDigRadRxParams.byRxAddress = ChannelParameters_GetAddress();

From the menu, enable addresses for the transmitter with the A command, and leave addresses disabled in the receiver. Restart transmission at the transmitter with the T command, and reception at the receiver with the R command. The following should appear at the receiver's terminal:

... other stuff
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
Rx failed: 000E (2120): invalid packet size
... more stuff

As before, the address is prepended to the payload data by the transmitter. However, because addresses are turned off at the receiver, the hardware filtering does not strip the address from the incoming byte stream, so it appears at the receiver as an additional byte of payload.

Next, ESC to the main menu on both units, and turn addressing off at the transmitter, and on at the receiver.

Transmitting a packet stream now results in a page full of:

... other stuff
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... more stuff

with an occasional:

... other stuff
Rx failed: 000C (0000): filtered out
Rx 00: 2C 0D 00 00 00 06 07 08 ... 0.01v [16:0D] 
Skipped packets 01 to FF 
Rx failed: 000C (0000): filtered out
as the sequence number happens to hit 0x00, the broadcast address, and the following:

... other stuff
Rx failed: 000C (0000): filtered out
Rx 00: 2C 0D 00 00 00 06 07 08 ... 0.01v [18:09]
Skipped packets 01 to FF
Rx failed: 000C (0000): filtered out
... more stuff

as the sequence number happens to hit 0x33, the active address at the receiver.

Next, enable addressing at the transmitter, leaving them enabled at the receiver, and you should now observe at the receiver:

... other stuff
Rx 24: 00 2D 0D 00 00 00 06 07 ... 2.06v [12:07]
Rx 25: 00 2D 0D 00 00 00 06 07 ... 2.06v [14:17]
Rx 26: 00 2C 0D 00 00 00 06 07 ... 2.06v [14:0E]
Rx 27: 00 2D 0D 00 00 00 06 07 ... 2.06v [13:10]
Rx 28: 00 2C 0D 00 00 00 06 07 ... 2.06v [14:09]
Rx 29: 00 2D 0D 00 00 00 06 07 ... 2.06v [12:0B]
Rx 2A: 00 2D 0D 00 00 00 06 07 ... 2.06v [11:15]
Rx 2B: 00 2C 0D 00 00 00 06 07 ... 2.06v [13:0A]
Rx 2C: 00 2D 0D 00 00 00 06 07 ... 2.06v [13:13]
Rx 2D: 00 2B 0D 00 00 00 06 07 ... 2.06v [12:0D]
Rx 2E: 00 2D 0D 00 00 00 06 07 ... 2.06v [14:0A]
... more stuff

The packets are aligned once again, and all is well.

Now, at the transmitter, stop transmission, change the address to 0x22, and restart transmission:

... other stuff
Tx 65:
Tx 66:
Tx 67:
Tx 68:
Tx 69:
<ESC>
-k
Change address (0x33): 0x22
Address set to 0x22
ok
-t
Tx started
Tx 00:
Tx 01:
Tx 02:
Tx 03:
Tx 04:
... more stuff
From this point forward, we will take off the training wheels for stopping transmission or reception with `ESC`, changing the channel parameters with the appropriate menu commands, and restarting transmission or reception with the `T` or `R` commands, respectively.

Instead, we will assume the user is now proficient in those areas and just state the procedure to be performed instead of the keystrokes, unless there is a compelling reason to show the entire sequence.

The receiver will now show continuously:

```
... other stuff
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... more stuff
```

Finally, change the transmitter’s address to 0x00, the broadcast address, and restart transmission. The receiver will now show the properly sequenced stream:

```
... other stuff
Rx 94: 00 2D 0D 00 00 00 06 07 ... 2.06v [12:11]
Rx 95: 00 2C 0D 00 00 00 06 07 ... 2.06v [18:0A]
Rx 96: 00 2C 0D 00 00 00 06 07 ... 2.06v [16:18]
Rx 97: 00 2D 0D 00 00 00 06 07 ... 2.06v [17:0D]
Rx 98: 00 2D 0D 00 00 00 06 07 ... 2.06v [15:15]
... more stuff
```

Since the transmitter is now using the broadcast address, all receivers on this channel will now process those packets.
6.2.7 CRC

Before proceeding further, ensure that both the transmitter (remember to set the address back to 0x33) and receiver are configured as follows:

Current channel parameters:
Address 0x33 enabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

The CRC calculation and check is a hardware feature of the CC1100, and helps ensure that received packets are correct, with only a small performance penalty imposed by sending effectively two more bytes of meta-payload appended to the end. The CRC is injected into the data stream at the transmitter, and removed at the receiver, transparently to the user.

Programmatically, the CRC feature is controlled with the `DigitalRadio_SetFeatures` function by setting or clearing the `DIGITAL_RADIO_FEATURE_CRC` bit:

```c
m_wFeatures |= DIGITAL_RADIO_FEATURE_CRC;
DigitalRadio_SetFeatures( m_wFeatures );
```

or indirectly by calling the API function:

```c
// Enable the CRC feature
ChannelParameters_EnableCRC();
```

or

```c
// Disable the CRC feature
ChannelParameters_DisableCRC();
```

or

```c
// Toggle the CRC feature
ChannelParameters_ToggleCRC();
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

There are no digital radio structure references to this feature as it is transparent to the user.
From the menu, turn CRC (the C command) off at the transmitter, leaving it on at the receiver. The receiver will now show:

... other stuff
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
... more stuff

because the transmitted packets are missing the CRC value, so the hardware reports a CRC failure.

Now, turn CRC on at the transmitter, and off at the receiver:

... other stuff
Rx 13: 00 2D 0D 00 00 00 06 07 ... 2.06v [17:0D]
Rx 14: 00 2D 0D 00 00 00 06 07 ... 2.06v [10:15]
Rx 15: 00 2C 0D 00 00 00 06 07 ... 2.06v [16:10]
... more stuff

The receiver is working correctly, because it is simply ignoring the CRC bytes as they arrive.

Now turn CRC off at the transmitter, and leave it off at the receiver:

... other stuff
Rx D8: 00 2C 0D 00 00 00 06 07 ... 2.06v [17:11]
Rx D9: 00 2D 0D 00 00 00 06 07 ... 2.06v [0F:0E]
Rx DA: 00 2C 0D 00 00 00 06 07 ... 2.06v [12:11]
... more stuff

With CRC off at both ends, reception works fine. However, you completely lose the packet integrity protection that CRC provides, with no additional software load. We recommend turning CRC on, and leaving it on.

6.2.8 Channels

Before proceeding further, ensure that both units configured as follows:

Current channel parameters:
Address 0x33 enabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

We now drop the D command training wheel, assuming that the user is familiar with this command, and will just show the command result from now forward.
Programmatically, the transmitter’s channel is set by directly calling the function `DigitalRadio_SetChannel`:

```c
// Set the channel to 3
DigitalRadio_SetChannel( 0x03 );
```

or indirectly by calling the API function:

```c
// Store the channel setting
ChannelParameters_SetChannel( 0x03 );
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_byChannel = ChannelParameters_GetChannel();
DigitalRadio_SetChannel( m_byChannel );
```

From the menu, change the transmitter to channel 1 with the `1` command. The receiver will show no activity.

Change the receiver to channel 1 also, and reception continues:

```c
... other stuff
Rx D5: 00 2D 0D 00 00 00 06 07 ... 2.06v [12:0C]
Rx D6: 00 2C 0D 00 00 00 06 07 ... 2.06v [12:0F]
Rx D7: 00 2D 0D 00 00 00 06 07 ... 2.06v [17:0B]
Rx D8: 00 2C 0D 00 00 00 06 07 ... 2.06v [17:11]
Rx D9: 00 2D 0D 00 00 00 06 07 ... 2.06v [0F:0E]
... more stuff
```

It is an interesting exercise to have one unit jabbering away on one channel, while a pair of units communicate on an adjacent channel. In the next chapter, we will setup this situation while testing the Ping and Echo modes, since Echo can be started without using the terminal.

Note also that the channel spacing supported by the demo code can be changed by modifying the CC1100 setup table as described in Subsection 4.2.1, `DigitalRadio_ResetTable.s43`. This module allows an expert user to modify the channel frequency and spacing of the CC1100, without requiring modification of the licensed modules.

We highly recommend the use of `SmartRF Studio` to assist with these changes.
6.2.9 RSSI/LQI

Be sure to set both units back to channel 0 before proceeding.

The Received Signal Strength Indicator (RSSI) and the Link Quality Indicator (LQI) are useful parameters to monitor while determining how the RF channel behaves. These are a receiver-only feature, and enabling or disabling merely controls whether they are appended by the receiver into the incoming data in the FIFO (enabled) or retrieved directly from the status registers after the packet has been received (disabled).

Programmatically, this feature is directly accessed via the DigitalRadio_SetFeatures function and the DIGITAL_RADIO_FEATURE_RSSI_LQI bit:

```
m_wFeatures |= DIGITAL_RADIO_FEATURE_RSSI_LQI;
DigitalRadio_SetFeatures( m_wFeatures );
```

or indirectly by calling the API function:

```
// Enable the RSSI/LQI feature
ChannelParameters_EnableRSSI_LQI();
```

or

```
// Disable the RSSI/LQI feature
ChannelParameters_DisableRSSI_LQI();
```

or

```
// Toggle the RSSI/LQI feature
ChannelParameters_ToggleRSSI_LQI();
```

and later calling the DigitalRadio_SetFeatures API function:

```
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

Convince yourself that the RSSI/LQI setting (Q command) makes no difference for the transmitter, or receiver, or both, by twiddling this feature on and off at both units. Again, this setting merely affects how the receiver firmware retrieves these values from the CC1100 hardware.

When done, re-enable RSSI/LQI at both units, as retrieving these values from the FIFO provides a slight performance improvement over individual register accesses.
6.2.10 Whitening

Before proceeding further, ensure that both the transmitter and receiver are configured as follows:

Current channel parameters:
Address 0x33 enabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

The data whitening option is a hardware feature of the CC1100, and tends to spread the energy of the signal throughout the allocated channel. If enabled, whitening is applied to the data stream at the transmitter, and removed at the receiver, transparently to the user.

Programmatically, the data whitening feature is directly controlled with the API function `DigitalRadio_SetFeatures` with the `DIGITAL_RADIO_FEATURE_WHITEN` bit:

```c
m_wFeatures |= DIGITAL_RADIO_FEATURE_WHITEN;
DigitalRadio_SetFeatures( m_wFeatures );
```

or indirectly by calling the API function:

```c
// Enable the data whitening feature
ChannelParameters_EnableWhiten();
```

or

```c
// Disable the data whitening feature
ChannelParameters_DisableWhiten();
```

or

```c
// Toggle the data whitening feature
ChannelParameters_ToggleWhiten();
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

There are no digital radio structure references to this feature as it is transparent to the user.

From the menu, turn whitening (the W command) off at the receiver, leaving it on at the transmitter. The receiver will now show:
because the receiver is not removing the whitening sequence, resulting in jibber for the
length and other meta-packet framing, so the hardware filters out the packet as the
address doesn’t match. Other combinations of packet data may result in other results.

Next, turn whitening off at the transmitter, and on at the receiver:

Finally, turn whitening off at the receiver, and leave it off at the transmitter:

With whitening on or off at both ends, reception works fine. However, you lose some
advantages that whitening provides, with no additional software load. We recommend
turning whitening on, and leaving it on.

One exception to always whitening might be if you want to make a jammer to test your
packet algorithms. In that case, you could place one unit in Tx mode transmitting whiten
data, and use that to test other units’ ability to retransmit and work through the noise. If
the units under test did not use whitening, the payload of the jammer would appear to be
white noise in the channel.
6.2.11 Variable Length Packets

All of our tests so far have involved variable-length packets. With variable-length packets, a length byte is embedded into the meta-packet to allow the receiver to determine how many bytes to expect from the FIFO.

Programmatically, variable-length packets are supported by the demo firmware by directly clearing the bits `DIGITAL_RADIO_FEATURE_FIXED_LENGTH` and `DIGITAL_RADIO_FEATURE_FEC` when using the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures &= ~( DIGITAL_RADIO_FEATURE_FIXED_LENGTH | DIGITAL_RADIO_FEATURE_FEC );
DigitalRadio_SetFeatures( m_wFeatures );
```

or indirectly by calling the API function:

```c
// Set variable length packets by updating the channel features
ChannelParameters_SetLengthVariable();
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

After setting the variable-length mode, the packet size for the transmitter is determined by the contents of the `byTxDataSize` member of the `SDigitalRadioTxParams` structure:

```c
m_tDigRadTxParams.byTxDataSize = 0x20;
```

and for the receiver by the contents of the `byRxDataSize` member of the `SDigitalRadioRxParams` structure:

```c
m_tDigRadRxParams.byRxDataSize = 0x20;
```

From the terminal menu, variable-length packets are set with the `V` command:

```
-v
Set variable length packets
```

We’ve already seen variable-length packets in operation, so we won’t show a live demo of them.

Variable-length packets have the advantage of allowing the system to receive dynamically sized packets from a variety of sources. However, when using variable length packets, Forward Error Correction (FEC) is prohibited. To use FEC with the CC1100, fixed-packets must be enabled.
6.2.12 Fixed Length Packets

Fixed-length packets allow the use of Forward Error Correction (FEC), but can also be used without FEC. The CC1100 supports fixed-length packets via the PKTLEN register, rather than embedding a length byte into the outgoing packet as with variable length packets.

Programmatically, fixed-length packets are supported by the demo firmware by directly setting the bit DIGITAL_RADIO_FEATURE_FIXED_LENGTH and clearing the bit DIGITAL_RADIO_FEATURE_FEC when using the DigitalRadio_SetFeatures API function:

```c
m_wFeatures |= DIGITAL_RADIO_FEATURE_FIXED_LENGTH;
m_wFeatures &= ~DIGITAL_RADIO_FEATURE_FEC;
DigitalRadio_SetFeatures( m_wFeatures );
```

or indirectly by calling the API function:

```c
// Set fixed-length packets by updating the channel features
ChannelParameters_SetLengthFixed();
```

and later calling the DigitalRadio_SetFeatures API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

After setting the fixed-length mode, the packet size for the transmitter is determined by the contents of the byTxDataSize member of the SDigitalRadioTxParams structure:

```c
m_tDigRadTxParams.byTxDataSize = 0x20;
```

and for the receiver by the contents of the byRxDataSize member of the SDigitalRadioRxParams structure:

```c
m_tDigRadRxParams.byRxDataSize = 0x20;
```

From the terminal menu, fixed-length packets are set with the F command:

```
-f
Set fixed length packets
```

Before proceeding, make sure the channel parameters on both units are set as shown below:

Current channel parameters:
Address 0x33 enabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

Now, set both the transmitter and receiver to fixed-length mode, and test:

```
... other stuff
Rx CE: 00 34 OD 00 00 00 06 07 ... 2.06v [12:0C]
Rx CF: 00 36 OD 00 00 00 06 07 ... 2.06v [12:1D]
Rx D0: 00 35 OD 00 00 00 06 07 ... 2.06v [13:0B]
Rx D1: 00 35 OD 00 00 00 06 07 ... 2.06v [15:0E]
... more stuff
```

Note that there is no observable change in operation from the variable-length mode, or in the interpretation of any of the operating parameters.

Next, set the transmitter to variable-length mode, and leave the receiver in fixed-length mode:

```
... other stuff
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... more stuff
```

Now set the transmitter to fixed-length mode, and set the receiver in variable-length mode, leaving all the other parameters unchanged:

```
... other stuff
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000E (3120): invalid packet size
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... 45 packets removed for brevity
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000E (3120): invalid packet size
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... more stuff
```

Most of the packets are filtered out because the addresses do not match. However, outgoing sequence numbers 0x00 and, 51 packets later, 0x33 are interpreted incorrectly as matching the broadcast address and unit address, respectively. In these cases, the DigitalRadio module then rejects the packets as being too large for the supplied buffer, since the outgoing address value of 0x33 is adjusted for meta-packet framing and interpreted as a packet length of 0x31.
Now, turn addressing off at the transmitter, and try again:

... other stuff
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 000C (0000): filtered out
Rx failed: 000E (FF20): invalid packet size
Rx failed: 000E (0020): invalid packet size
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
... 25 packets removed for brevity
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000E (2120): invalid packet size
... 25 packets removed for brevity
Rx failed: 000E (3B20): invalid packet size
Rx failed: 000E (3C20): invalid packet size
Rx failed: 0008 (0000): timed out
Rx failed: 0008 (0000): timed out
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
... more stuff

The peculiarity of the packet format causes this pattern to repeat every time the sequence number rolls over because of how the data pattern is interpreted as correctly formatted packets, or packets too large for the buffer, etc.

The lesson to be learned from this example is to be sure to test packets with impossible data, to see whether unexpected, but repeatable, variations can cause seemingly legitimate data to creep through, or report unexpected errors.
6.2.13 Fixed Length Packets with Forward Error Correction (FEC)

Fixed-length packets with FEC are identical to the fixed-length mode without, with the exception that a forward error correction algorithm is performed on the packet data, and extra bytes added to the packet data containing redundancy information so that some errors can be corrected.

The redundant data added to the packet causes the data rate to drop by half. However, the removal of the necessity for some retries may cause the effective data rate to go up, when the delays of handshaking, interpreting incorrect or missing packets, or waiting for filtered packets to go away.

Programmatically, fixed-length packets with FEC are supported by the demo firmware by directly setting the bits `DIGITAL_RADIO_FEATURE_FIXED_LENGTH` and `DIGITAL_RADIO_FEATURE_FEC` when using the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures |= ( DIGITAL_RADIO_FEATURE_FIXED_LENGTH | DIGITAL_RADIO_FEATURE_FEC );
DigitalRadio_SetFeatures( m_wFeatures );
```

or indirectly by calling the API function:

```c
// Set fixed-length packets with FEC
ChannelParameters_SetLengthFixedFEC();
```

and later calling the `DigitalRadio_SetFeatures` API function:

```c
m_wFeatures = ChannelParameters_GetFeatures();
DigitalRadio_SetFeatures( m_wFeatures );
```

After setting the fixed-length with FEC mode, the packet size for the transmitter is determined by the contents of the `byTxDataSize` member of the `SDigitalRadioTxParams` structure:

```c
m_tDigRadTxParams.byTxDataSize = 0x20;
```

and for the receiver by the contents of the `byRxDataSize` member of the `SDigitalRadioRxParams` structure:

```c
m_tDigRadRxParams.byRxDataSize = 0x20;
```

From the terminal menu, fixed-length packets with FEC are set with the `G` command:

```
-g
Set fixed length packets with FEC
```

Before proceeding, ensure both units are set as follows:
Current channel parameters:
Address 0x33 enabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

Now set both the transmitter and receiver to fixed-length with FEC mode, and test:

... other stuff
Rx 40: 00 35 0D 00 00 00 06 07 ... 2.06v [13:1B]
Rx 41: 00 35 0D 00 00 00 06 07 ... 2.06v [13:0E]
Rx 42: 00 35 0D 00 00 00 06 07 ... 2.06v [12:11]
Rx 43: 00 35 0D 00 00 00 06 07 ... 2.06v [13:13]
... more stuff

Note that there is no observable change in operation from the fixed-length mode, or in the interpretation of any of the operating parameters, except that the bit rate over the air is cut in half.

Next set the transmitter to variable-length mode, and the receiver to fixed-length with FEC:

... other stuff
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... more stuff

Because the transmitted packet did not have the FEC applied, it appears to be invalid.

Next set the transmitter to fixed-length mode, without FEC:

... other stuff
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
Rx failed: 000C (0000): filtered out
... more stuff

Disabling addresses at the receiver results in:

... other stuff
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
... more stuff

since without address filtering the garbled data comes through, but is caught by the CRC.

Turning off CRC protection results in:
MSP430 RF Applications with the MRF1611CC1100

... other stuff
Rx 5E: +++
Skipped packets 5F to 5F
Rx 60: 51 41 FD 25 01 69 C3 DC ... 2.07v [18:10]
Skipped packets 61 to 37
Rx 38: 29 41 FD 25 01 69 C3 DC ... 2.07v [18:0E]
Skipped packets 39 to 6D
Rx 6E: E9 41 FD 25 01 69 C3 DC ... 2.07v [18:0E]
Skipped packets 6F to 5B
Rx 5C: +++
Skipped packets 5D to 6D
Rx 6E: 29 41 FD 25 01 69 C3 DC ... 2.07v [18:12]
Skipped packets 6F to 6D
Rx 6E: 29 41 FD 25 01 69 C3 DC ... 2.07v [17:18]
Skipped packets 6F to 5D
Rx 5E: 29 41 FD 25 01 69 C3 DC ... 2.07v [18:0D]
Skipped packets 5F to 5D
... more stuff

which is an unmitigated disaster, but passes through since most of the protective filtering is now turned off!

Finally, set the transmitter to fixed length with FEC, and the receiver to variable length:

... other stuff
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 000E (2E20): invalid packet size
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx 4B: 27 69 42 F2 54 3D DB 08 ... 0.38v [11:15]
Skipped packets 4C to FA
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx FB: E7 69 42 F1 57 3F DB 08 ... 0.38v [18:12]
Skipped packets FC to BA
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx failed: 0012 (0000): RXFIFO overflow
Rx BB: 67 A9 42 F1 57 3C DB 08 ... 0.42v [11:12]
Skipped packets BC to 3A
Rx failed: 0012 (0000): RXFIFO overflow
... more stuff

Different failures are reported, depending on what bits resulted from the redundancy encoding. This variety of exciting failures intermingled with almost reasonable-looking data is an argument for production applications using the maximum amount of protective filtering possible, which means addressing, sequence numbers, fixed-length packet with FEC, and CRCs, all working in concert.
6.3  Handheld Mode

We next exploit the fact that the HMRF1611 is designed to be a battery-powered host by converting Unit B into a mobile transmitter. This handheld mode allows us to experiment with a variety of disturbances in the environment.

First, ensure that the channel parameters of both units are set as shown below, using the power-up defaults:

Current channel parameters:
Addresses disabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

Next, use the terminals to set Unit B into transmit mode, and Unit A into receive. Verify that Unit B is transmitting by watching the receiver, then disconnect the terminal. Twirl the potentiometer and twiddle some switches:

... other stuff
Rx EE: 00 FD 05 00 00 00 06 07 ... 0.94v [1B:17]
Rx EF: 00 FD 05 00 00 00 06 07 ... 0.94v [1B:12]
Rx F0: 00 FD 05 00 00 00 06 07 ... 0.94v [1B:12]
Rx F1: 00 FC 05 00 00 00 06 07 ... 0.93v [1A:11]
Rx F2: 00 FD 05 00 00 00 06 07 ... 0.94v [1B:0D]
... more stuff

Now approach Unit A, and set the two units near each other:

... other stuff
Rx 0E: 00 FE 05 00 00 00 06 07 ... 0.94v [72:0E]
Rx 0F: 00 FD 05 00 00 00 06 07 ... 0.94v [72:0E]
Rx 10: 00 FD 05 00 00 00 06 07 ... 0.94v [73:0B]
Rx 11: 00 FE 05 00 00 00 06 07 ... 0.94v [74:11]
Rx 12: 00 FE 05 00 00 00 06 07 ... 0.94v [73:0A]
... more stuff

Notice how the RSSI has greatly increased, and is near the maximum.

Next, as you put the units closer and closer, eventually they stop communicating:

... other stuff
Rx B0: 00 FE 05 00 00 00 06 07 ... 0.94v [76:14]
Skipped packets B1 to B1
Rx failed: 000A (0000): invalid CRC
Rx B2: 00 FE 05 00 00 00 06 07 ... 0.94v [75:0A]
Skipped packets B3 to B3
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 000A (0000): invalid CRC
Rx failed: 0012 (0000): RXFIFO overflow
... more stuff

But then as you bring Unit B farther away, they start talking again:

... other stuff
Rx CE: 00 FD 05 00 00 00 06 07 ... 0.94v [49:0D]
Rx CF: 00 FD 05 00 00 00 06 07 ... 0.94v [49:0E]
Rx D0: 00 FE 05 00 00 00 06 07 ... 0.94v [4A:13]
... more stuff

Here are some other experiments you can try, watching the effect on RSSI:

- Build a foil shield between the units.
- Touch the antennae with your hand or various objects.
- Change the orientation of the antennae relative to each other.
- Fold the antennae into different paper clip shapes.
- Try different antenna sizes.
- Hold the handheld near different motors or other EMI sources like grill starters, cats, fuzzy sweaters (the last two must be stroked correctly to generate EMI), etc. Can you tell which motors are brushless? Or try that old Mr. Sparky drill from 1954.
- Bring the EMI sources near the fixed unit. What differences, if any, do you observe?
- How far away can you place the units and still make the LEDs change?
- What is the combined effect of interior walls and range? Exterior walls? Windows?
- What happens when you put the handheld in a closed box like a safe or filing cabinet? Was the result what you expected? Hint: Avoid microwave ovens as the temptation will be too great. We know from personal experience.
- Try different values for the PATALBE in DigitalRadio_ResetTable.s43.
- Try many of the above cases with FEC and without FEC, and try to get a feel for how effective various disturbances might be.

Remember that if you accidentally touch the crystals or the MSP430 you may restart the board, and that would cause the Tx mode to be lost as the jumpers are set to enter the main menu on restart.
6.4 Custom Application Samples

In the previous chapter, we modified the user application module by adding custom code to exercise the switch module. In this section, we extend the Tx/Rx functionality by handling client callbacks embedded into UserApp.c.

Before proceeding further, overwrite UserApp.c with the default UserApp_Original.c. The complete source code for this project is provided as UserApp_TxRx.c.

6.4.1 User Application Callbacks

The user application supports ten callbacks, described below:

**UserApp_Tx_OnStart**

This callback executes when the Tx mode starts operation. The user application can use this callback to initialize any resources required for Tx mode.

**UserApp_Tx_OnStop**

This callback executes when the Tx mode stops operation. The user application can use this callback to un-initialize any resources required for Tx mode.

**UserApp_OnUpdatePacket_PreTx**

This callback executes when the Tx mode is about to send a packet. Return FALSE to prevent the default Tx display.

**UserApp_Tx_OnIdle**

This callback executes when any the Tx packet radio mode is ticking through any idle sequence. Returning FALSE cancels the packet radio operation.

**UserApp_Tx_OnNextPacket**

This callback executes when the Tx packet radio mode is preparing to restart transmission of a new packet. Returning FALSE cancels the packet radio operation.

**UserApp_Rx_OnStart**

This callback executes when the Rx mode starts operation. The user application can use this callback to initialize any resources required for Rx mode.
**UserApp_Rx_OnStop**

This callback executes when the Rx mode stops operation. The user application can use this callback to un-initialize any resources required for Rx mode.

**UserApp_OnUpdatePacket_PostRx**

This callback executes when the Rx mode has just received a packet. Return FALSE to prevent the default Rx display.

**UserApp_Rx_OnIdle**

This callback executes when the Rx packet radio mode is ticking through any idle sequence. Returning FALSE cancels the packet radio operation.

**UserApp_Rx_OnNextPacket**

This callback executes when the Rx packet radio mode is preparing to restart reception of a new packet. Returning FALSE cancels the packet radio operation.

The remainder of this chapter will use these callbacks to create a user application that does not require any detailed knowledge of the packet radio interface.
6.4.2 Exercising the Callbacks

First, we add user interface alerts to the terminal display for the OnStart and OnClose callbacks. Add the following code for the Tx mode:

```c
void UserApp_Tx_OnStart( void )
{
    RxTxHost_QueueTxString( "\nTx OnStart\n" );
}

void UserApp_Tx_OnStop( void )
{
    RxTxHost_QueueTxString( "\nTx OnStop\n" );
}
```

and for the Rx mode:

```c
void UserApp_Rx_OnStart( void )
{
    RxTxHost_QueueTxString( "\nRx OnStart\n" );
}

void UserApp_Rx_OnStop( void )
{
    RxTxHost_QueueTxString( "\nRx OnStop\n" );
}
```

Let us also add a user interface alert as a plus symbol for the OnNextPacket callback for the Tx mode:

```c
BOOL UserApp_Tx_OnNextPacket( void )
{
    RxTxHost_QueueTxString( "+" );
    return TRUE;
}
```

and for the Rx mode:

```c
BOOL UserApp_Rx_OnNextPacket( void )
{
    RxTxHost_QueueTxString( "+" );
    return TRUE;
}
```

Finally, add a question mark as a user interface alert for the PreTx handler:

```c
BOOL UserApp_OnUpdatePacket_PreTx(
    SDigitalRadioTxParams* ptDigRadTxParams )
{
```
and the PostRx handler:

```c
BOOL UserApp_OnUpdatePacket_PostRx(SDigitalRadioRxParams* ptDigRadRxParams) {
    RxTxHost_QueueTxString( "?" );
    return FALSE;
}
```

Make sure Unit B is operating with the following default settings:

Current channel parameters:
- Addresses disabled
- CRC enabled
- RSSI/LQI enabled
- Whitening enabled
- Sequence numbers enabled
- Using channel 0x00 for variable length packets

Build the project, and load it onto Unit A. Execute the code with Go.

Let us test the callbacks by first setting Unit B into Tx mode:

```markdown
-t
Tx started
Tx 00:
Tx 01:
Tx 02:
Tx 03:
Tx 04:
Tx 05:
... more stuff
```

Then, execute Rx mode on Unit A:

```markdown
-r
Rx started
Rx OnStart
+?Rx 79: 00 5F 06 00 00 00 06 07 ... 1.00v [0F:10]
+?Rx 7A: 00 5F 06 00 00 00 06 07 ... 1.00v [0E:0A]
+?Rx 7B: 00 5F 06 00 00 00 06 07 ... 1.00v [0D:0A]
+?Rx 7C: 00 5F 06 00 00 00 06 07 ... 1.00v [0E:0C]
+?Rx 7D: 00 5F 06 00 00 00 06 07 ... 1.00v [0D:0E]
... more stuff
```
Note the addition of the OnStart message, the “+” before each packet, and the “?” before the data for each packet is displayed.

Similarly, when Rx mode is stopped, the following will be displayed:

… other stuff
+?Rx F0: 00 5F 06 00 00 00 06 07 ... 1.00v [0F:0D]
+?Rx F1: 00 5F 06 00 00 00 06 07 ... 1.00v [0D:0E]
+?Rx F2: 00 5F 06 00 00 00 06 07 ... 1.00v [0D:14]
+<ESC>
   Error count = 0000
Rx OnStop

Note the addition of the OnStop message when the Rx mode is terminated.

Now place Unit B into Rx mode with ESC followed by the R command:

… other stuff
Tx 12:
Tx 13:
Tx 14:<ESC>
-r
Rx started
Rx D8: 00 FD 05 00 00 00 06 07 ... 0.94v [06:0E]
Rx D9: 00 FD 05 00 00 00 06 07 ... 0.94v [07:14]
Rx DA: 00 FD 05 00 00 00 06 07 ... 0.94v [07:10]
… more stuff

And place Unit A into Tx mode with the T command:

-t
Tx started

Tx OnStart
+?Tx 00:
+?Tx 01:
+?Tx 02:
+?Tx 03:
+?Tx 04:
+?Tx 05:
… more stuff

The OnStart message is displayed before packets start, and as before the OnNextPacket alert is displayed before each packet, and that the PreTx alert is displayed before a packet goes out.
Similarly, when transmission is stopped with $ESC$:

... other stuff
+?Tx 0D:
+?Tx 0E:
+?Tx 0F:
+?Tx 10:
+?Tx 11:
+?Tx 12: <ESC>
          Tx OnStop

The $OnStop$ message is displayed as we would expect, although slightly offset as this terminal implementation swallows the $CR$ portion of the $CRLF$ when $ESC$ is hit.

Also note that in each case, the $OnNextPacket$ callback executes even before the first packet.
6.4.3 OnIdle Callbacks

In the previous example, we did not implement an alert for the idle callback, as this would have resulted in a busy display. However, the idle callback is useful for providing some non-critical task, or for canceling radio operation.

Let us implement a cancel based on SW4 for Tx mode. First, update the code in the Tx OnStart and OnStop callbacks as follows:

```c
void UserApp_Tx_OnStart( void )
{
    // Prepare SW4 to generate IFG, but not interrupts, on a press
    P1IES |= (BIT7);
    P1IE &= ~(BIT7);
    P1IFG &= ~(BIT7);
}
void UserApp_Tx_OnStop( void )
{
}
```

Similarly, implement code in the Tx OnIdle callback to check SW4:

```c
BOOL UserApp_Tx_OnIdle( void )
{
    // Check whether SW4 has been pressed
    if( 0 != (P1IFG & BIT7) )
    {
        RxTxHost_QueueTxString( "\nSW4!" );
        return FALSE;
    }
    return TRUE;
}
```

Build, load onto the target board, and execute. Place Unit B into receive mode. Now start Tx mode on Unit A:

```
tx
Tx started
+?Tx 00:
+?Tx 01:
+?Tx 02:
+?Tx 03:
+?Tx 04:
+?Tx 05:
... more stuff
```

and test SW4:
As shown, pressing the switch terminates transmit mode.

Also, note that by an ESC not being echoed to the terminal, the CRLF works as desired.

Similarly, let us add a kill switch feature to Rx. First, update the OnStart and OnStop callbacks as shown below:

```c
void UserApp_Rx_OnStart( void ) {
   // Prepare SW4 to generate IFG, but not interrupts, on a press
   P1IES |= (BIT7);
   P1IE &= ~(BIT7);
   P1IFG &= ~(BIT7);
}

void UserApp_Rx_OnStop( void ) {
}
```

And, implement the following OnIdle code:

```c
BOOL UserApp_Rx_OnIdle( void ) {
   // Check whether SW4 has been pressed
   if( 0 != (P1IFG & BIT7) ) {
      RxTxHost_QueueTxString( "\nSW4!" );
      return FALSE;
   }
   return TRUE;
}
```

Build, load onto the target board, and execute. Place Unit B into transmit mode, and start receive mode on Unit A:

```
-r
Rx started
+?Rx DA: 00 5F 06 00 00 00 06 07 ... 1.00v [08:0F]
+?Rx DB: 00 5E 06 00 00 00 06 07 ... 0.99v [06:0E]
+?Rx DC: 00 5F 06 00 00 00 06 07 ... 1.00v [07:0F]
+?Rx DD: 00 5F 06 00 00 00 06 07 ... 1.00v [07:12]
```
and sometime later press SW4:

... other stuff
+?Rx E8: 00 5F 06 00 00 00 06 07 ... 1.00v [06:09]
+?Rx E9: 00 5F 06 00 00 00 06 07 ... 1.00v [07:0A]
+?Rx EA: 00 5E 06 00 00 00 06 07 ... 0.99v [06:13]
+?Rx EB: 00 5F 06 00 00 00 06 07 ... 1.00v [08:11]
+?Rx EC: 00 5E 06 00 00 00 06 07 ... 1.00v [06:0C]
+?Rx ED: 00 5F 06 00 00 00 06 07 ... 1.00v [0A:0B]
+?Rx EE: 00 5F 06 00 00 00 06 07 ... 1.00v [06:08]
+?Rx EF: 00 5E 06 00 00 00 06 07 ... 0.99v [07:19]

SW4!
Error count = 0000
### 6.4.4 Custom Transmit Functionality

The user callbacks include `UserApp_OnUpdatePacket_PreTx`, which fires before a packet is sent, and `UserApp_OnUpdatePacket_PostRx`, which fires after a packet is received. These two callbacks have as a parameter a pointer to the structures `SDigitalRadioTxParams` and `SDigitalRadioRxParams`, respectively. These structures contain information about the packet radio operation.

Before we work with these structures, let us first convert our Tx and Rx operations into a one-shot mode. That is, each Tx command will send exactly one packet, and each Rx command will receive exactly one packet.

To do so, first add a flag define, a flags BYTE and a dummy BYTE to the top of `UserApp.c`:

```c
// 1/8 second delays, at a system tick rate of 8192 Hz
#define LED_DEMO_DELAY_TICKS 1024
#define USER_APP_FLAG_FIRST_PACKET BIT0

__no_init static BYTE mUserApp_byFlags;
#pragma diag_suppress=Pe177
__no_init static BYTE mUserApp_byDummy;
#pragma diag_default=Pe177

void UserApp_Stop( void );
```

Now, update the Tx `OnStart` callback as shown below:

```c
void UserApp_Tx_OnStart( void )
{
    // Prepare SW4 to generate IFG, but not interrupts, on a press
    P1IES |= (BIT7);
    P1IE &= ~(BIT7);
    P1IFG &= ~(BIT7);

    // Prepare flags
    mUserApp_byFlags = USER_APP_FLAG_FIRST_PACKET;
}
```

and the `OnNextPacket` callback:

```c
BOOL UserApp_Tx_OnNextPacket( void )
{
    if( 0 == (USER_APP_FLAG_FIRST_PACKET & mUserApp_byFlags) )
        // This is not the first packet
        { return FALSE;
    }
```
mUserApp_byFlags &= ~(USER_APP_FLAG_FIRST_PACKET);

return TRUE;
}

Now, place Unit B in receive mode, and build, load, and execute this code on Unit A. Now Unit A only sends out one packet each time the T command is issued:

-t
Tx started
?Tx 00:

-t
Tx started
?Tx 00:

-t
Tx started
?Tx 00:

-t
Tx started
?Tx 00:

The only problem with this approach is now the sequence numbers are always 0x00, so the receiver shows:

-r
Rx started
Rx 00: 00 7E 08 00 00 00 06 07 ... 1.33v [17:0F]
Rx 00: 00 80 08 00 00 00 06 07 ... 1.33v [17:0F]
Skipped packets 01 to FF
Rx 00: 00 7E 08 00 00 00 06 07 ... 1.33v [16:0E]
Skipped packets 01 to FF
Rx 00: 00 7F 08 00 00 00 06 07 ... 1.33v [16:0E]
Skipped packets 01 to FF

To solve this, let's move the sequence number to the user application. First, change the name of the dummy variable at the top of UserApp.c, and remove the pragmas:

__no_init static BYTE mUserApp_byFlags;
__no_init static BYTE mUserApp_bySequenceNumber;

Now, add the following initialization code to UserApp_InitializeApp:
void UserApp_InitializeApp( void )
// Initialize any application items required by this module
// This function is equivalent to class-initialization in C++ (static)
{
    mUserApp_bySequenceNumber = 0x00;
}

Now let's examine the parameter passed to UserApp_OnUpdatePacket_PreTx. This parameter is a pointer to a SDigitalRadioTxParams structure, which is detailed in Section 11.4.15, Structure SDigitalRadioTxParams.

The byTxSequenceNumber member of this structure contains the sequence number to be transmitted with the packet. So, let's update this sequence number on each transmit:

BOOL UserApp_OnUpdatePacket_PreTx(
    SDigitalRadioTxParams* ptDigRadTxParams )
{
    ptDigRadTxParams->byTxSequenceNumber = \n        mUserApp_bySequenceNumber++;
    return FALSE;
}

As before, the “\” character indicates a wrapped line in this text, but this code statement should be on a single line in the actual source.

Rebuild, load, and execute the code on Unit A, and send a few single packets:

- t
  Tx started
  Tx 00:

- t
  Tx started
  Tx 01:

- t
  Tx started
  Tx 02:

- t
  Tx started
  Tx 03:
The sequence number is incrementing properly, and note that the received packet data is now as you would expect:

Rx 00: 00 7E 08 00 00 00 06 07 ... 1.33v [17:0F]
Rx 01: 00 7F 08 00 00 00 06 07 ... 1.33v [16:0F]
Rx 02: 00 7F 08 00 00 00 06 07 ... 1.33v [17:10]
Rx 03: 00 7F 08 00 00 00 06 07 ... 1.33v [17:0E]

As a curiosity, we are returning FALSE from `UserApp_OnUpdatePacket_PreTx`, even though we have affected the digital radio parameters. This return value determines whether the packet is populated with the default switch and potentiometer information, as well as displayed to the terminal. So, for now we return FALSE to let the demo application handle these details for us.

The first BYTE in the packet is unused, as are many following the sixth byte. We can now use the first one for our purposes, by working with the `pabyTxData` member of `SDigitalRadioTxParams`. Update `UserApp_OnUpdatePacket_PreTx` to set that value to 0x37:

```c
BOOL UserApp_OnUpdatePacket_PreTx( SDigitalRadioTxParams* ptDigRadTxParams )
{
    ptDigRadTxParams->byTxSequenceNumber = \n        mUserApp_bySequenceNumber++;
    ptDigRadTxParams->pabyTxData[0] = 0x37;
    return FALSE;
}
```

Build, load and execute, and the following will be observed at the receiver when the T command is given from Unit A:

Rx 00: 37 7F 08 00 00 00 06 07 ... 1.33v [16:0E]
6.4.5 Custom Receive Functionality

We can also modify Rx mode to only receive one packet per \textit{R} command. Update the Rx \textit{OnStart} callback as shown below:

```c
void UserApp_Rx_OnStart( void )
{
    // Prepare SW4 to generate IFG, but not interrupts, on a press
    P1IES |= (BIT7);
    P1IE &= ~(BIT7);
    P1IFG &= ~(BIT7);

    // Prepare flags
    mUserApp_byFlags = USER_APP_FLAG_FIRST_PACKET;
}
```

And, edit the Rx \textit{OnNextPacket} callback as follows:

```c
BOOL UserApp_Rx_OnNextPacket( void )
{
    if( 0 == (USER_APP_FLAG_FIRST_PACKET & mUserApp_byFlags) )
    // This is not the first packet
    { return FALSE;
    }

    mUserApp_byFlags &= ~(USER_APP_FLAG_FIRST_PACKET);

    return TRUE;
}
```

Place Unit B into Tx mode. Build, load, and execute the code on Unit A. Now the receiver will handle one packet at a time, including errors if any are encountered:

- `r` Rx started
  \texttt{?Rx 27: 00 5F 06 00 00 00 06 07 ... 1.00v [10:10]}

- `r` Rx started
  \texttt{?Rx 98: 00 5F 06 00 00 00 06 07 ... 1.00v [10:09]}

- `r` Rx started
  \texttt{?Rx 53: 00 5F 06 00 00 00 06 07 ... 1.00v [11:0B]}

- `r` Rx started
  \texttt{?Rx B1: 00 5F 06 00 00 00 06 07 ... 1.00v [10:0B]}
Rx started
?Rx 04: 00 5F 06 00 00 00 06 07 ... 1.00v [0E:0F]

Now that we are receiving individual packets, we can write a custom display function for each received packet, without worry of overrunning the serial port. To do so, we will edit the PostRx callback, as shown below:

```c
BOOL UserApp_OnUpdatePacket_PostRx( 
    SDigitalRadioRxParams* ptDigRadRxParams )
{
    BYTE byCount, byValue;

    // Show sequence number
    RxxTxHost_QueueTxString( "\nbySeqNum: " );
    RxxTxHost_QueueTxHexBYTE( ptDigRadRxParams->byRxSequenceNumber );

    // Show RSSI
    RxxTxHost_QueueTxString( "\nbyRSSI: " );
    RxxTxHost_QueueTxHexBYTE( ptDigRadRxParams->byRSSI );

    // Show LQI
    RxxTxHost_QueueTxString( "\nbyLQI: " );
    RxxTxHost_QueueTxHexBYTE( ptDigRadRxParams->byLQI );

    // Show count
    RxxTxHost_QueueTxString( "\nbyCount: " );
    RxxTxHost_QueueTxHexBYTE( ptDigRadRxParams->byRxCountRead );

    // Show packet contents
    RxxTxHost_QueueTxString( "\nPacket:\n" );
    for( byCount = 0; byCount < ptDigRadRxParams->byRxCountRead; byCount++ )
    {
        if( 0 != byCount )
        {
            if( 0 == (byCount%8) )
                // Add a newline
                RxxTxHost_QueueTxCRLF();
        }
        byValue = ptDigRadRxParams->pabyRxData[byCount];
        RxxTxHost_QueueTxHexString( byValue );
        RxxTxHost_QueueTxChar( ' ' );
    }

    return TRUE;
}
```

This time, we set the return value to TRUE, to replace the default display code.
Build the code, load it onto the target, and execute it. Make sure Unit B is in Tx mode. Then, receive several packets with the \texttt{R} command:

\begin{verbatim}
   -r
Rx started

bySeqNum: C1
byRSSI: FA
byLQI: 12
byCount: 20

Packet:
00 5F 06 00 00 00 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F

   -r
Rx started

bySeqNum: 86
byRSSI: F8
byLQI: 10
byCount: 20

Packet:
00 5F 06 00 00 00 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F

   -r
Rx started

bySeqNum: 24
byRSSI: 05
byLQI: 0C
byCount: 20

Packet:
00 5F 06 00 00 00 06 07
08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17
18 19 1A 1B 1C 1D 1E 1F
\end{verbatim}

In this chapter, we exercised the callback approach to packet radio firmware. This model is appropriate for projects in which one team provides the packet radio framework, yet another team is responsible for the packet content.

At the end of the next chapter, we drill further down into packet radio operations by calling the packet radio module directly, rather than through the demo intermediary. This approach is more appropriate for one integrated development team.
7. Ping/Echo Demo

The previous chapter exercised the functionality of the simple Tx and Rx modes available in the demo firmware. This chapter exercises the more advanced point-to-point functionality available in the Ping and Echo modes. The next chapter describes the Hub and Node functionality available as a star network topology in the demo firmware.

This chapter assumes the reader is familiar with, and builds upon, the material in the previous chapter. It is highly recommended that the exercises in the previous chapter be followed through in detail. At a minimum, be sure to read the previous chapter for understanding before moving on.

Before proceeding further, overwrite UserApp.c with the default UserApp_Original.c.

7.1 Ping/Echo Architecture

The following diagram shows the functional components of a Ping-Echo point-to-point network:

This diagram has been enhanced from the simple Tx to Rx mode, in that a returned acknowledge (ack) packet is now provided. This ack packet informs the transmitter (now called the Ping unit) that the receiver (now called the Echo unit) received the original packet. Lack of an ack packet is a clue to the Ping unit that it should retry the original packet.

The use of sequence numbers in the original packet, and echoed back in the ack packet, allows the two units to handshake on agreeing whether or not a packet has made it through to the Echo unit.

In the following sections, we will show how the parameters exercised in the last chapter affect the Ping/Echo network.
7.2 Default Operation

This section describes the default operation of the Ping and Echo modes of the DemoMRF1611CC1100 project.

7.2.1 Materials

To perform the experiments in this section, you will need the same materials from the Tx and Rx chapter, which are:

- Units A and B, each consisting of HMRF1611 host boards with installed BRS232s and MRF1611CC1100 modules.

- PC A and PC B running the DemoMRF1611CC1100.ht session.

7.2.2 Preparation

On PC A, close Embedded Workbench, if running, and move the USBP to PC B.

Check the batteries in Unit B. Make sure each cell has a no-load voltage of 1.2v or higher for the purposes of these demos.

Configure the HMRF1611 jumpers for Unit B as follows:

<table>
<thead>
<tr>
<th>Header</th>
<th>Option</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Power On</td>
<td>Unit battery powered</td>
</tr>
<tr>
<td>Analog</td>
<td>Default</td>
<td>n/a</td>
</tr>
<tr>
<td>Port3 / Comm</td>
<td>Unit Connected to PC</td>
<td>For terminal operation</td>
</tr>
<tr>
<td>Port4</td>
<td>Echo Mode</td>
<td>in echo mode</td>
</tr>
</tbody>
</table>

Refer back to Section 2.4, Jumper Settings, for descriptions and diagrams for these options.

Reprogram Unit B with the default demo firmware DemoMRF1611CC1100.txt from within PrgUSBP.

Close the PrgUSBP application, disconnect Unit B from the USBP and return the USBP to PC A.

Connect Unit B to PC B with a serial cable, and ensure the terminal application is running. Cycle power on Unit B (unplug and reinstall the VB:VIN jumper), and the following should appear on its terminal:
Note that Unit B should proceed directly into echo mode upon startup. If not, debug the preparation of Unit B until the above appears on PC B.

On PC A, copy UserApp_Original.c over UserApp.c to restore the default user application, and restart Embedded Workbench.

Configure the HMRF1611 jumpers for Unit A as follows:

<table>
<thead>
<tr>
<th>Header</th>
<th>Option</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Power Off</td>
<td>Unit powered from USBP</td>
</tr>
<tr>
<td>Analog</td>
<td>Default</td>
<td>n/a</td>
</tr>
<tr>
<td>Port3 / Comm</td>
<td>Unit Connected to PC</td>
<td>For terminal operation</td>
</tr>
<tr>
<td>Port4</td>
<td>Menu Mode</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Rebuild the project and load onto Unit A via the Debug button, and hit the Go button to execute the firmware.

Ensure that the terminal on PC A is reporting the following:

Resetting...

ok

If not, check the configuration until it does before proceeding.
7.2.3 First Execution

Start Ping mode on Unit A, and the following should appear on the terminal:

```
Ping started
Ping 00: 00 3C 08 00 19 0F 06 07 ... 1.29v [16:0A]
Ping 01: 00 3C 08 00 17 0A 06 07 ... 1.29v [13:19]
Ping 02: 00 3C 08 00 18 12 06 07 ... 1.29v [15:0E]
Ping 03: 00 3C 08 00 18 07 06 07 ... 1.29v [15:08]
Ping 04: 00 3C 08 00 18 10 06 07 ... 1.29v [16:06]
Ping 05: 00 3C 08 00 18 11 06 07 ... 1.29v [14:0B]
Ping 06: 00 3C 08 00 18 0E 06 07 ... 1.29v [15:0C]
Ping 07: 00 3C 08 00 19 0C 06 07 ... 1.29v [13:0D]
... more stuff
```

Meanwhile, on PC B, the Echo side, the following will be scrolling by:

```
... other stuff
Echo 75: 00 78 07 00 00 00 06 07 ... 1.17v [19:13]
Echo 76: 00 78 07 00 00 00 06 07 ... 1.17v [1A:0B]
Echo 77: 00 78 07 00 00 00 06 07 ... 1.17v [1A:0D]
Echo 78: 00 78 07 00 00 00 06 07 ... 1.17v [1A:09]
Echo 79: 00 78 07 00 00 00 06 07 ... 1.17v [19:10]
Echo 7A: 00 79 07 00 00 00 06 07 ... 1.17v [18:10]
Echo 7B: 00 78 07 00 00 00 06 07 ... 1.17v [19:05]
Echo 7C: 00 78 07 00 00 00 06 07 ... 1.17v [1A:12]
Echo 7D: 00 78 07 00 00 00 06 07 ... 1.17v [19:0B]
... more stuff
```

Note several differences from the Tx and Rx modes. First, both units report packets now, as there is bi-directional packet traffic.

Secondly, while the Echo side does not show values in the payload at offset 4, and 5, the Ping side does. The values at those locations in the Ping report are the RSSI and LQI values observed for the Ping’s packets at the Echo station. This allows the Ping side to now measure both the outgoing and the incoming signal strength.

Also, experiment with LEDs and the pot from each unit, and notice their effect on the other unit. Since we have bi-directional packets, we can now see some nice two-way traffic.
7.2.4 Error Reporting and Retry

For purposes of this section, kill the Echo by pressing ESC on Unit B:

... other stuff
Echo 13: 00 78 07 00 00 00 06 07 ... 1.17v [25:0B]
Echo 14: 00 78 07 00 00 00 06 07 ... 1.17v [24:08]
Echo 15: 00 78 07 00 00 00 06 07 ... 1.17v [26:10]
Echo 16: 00 78 07 00 00 00 06 07 ... 1.17v [24:06]
Echo 17: 00 78 07 00 00 00 06 07 ... 1.17v [25:0B]
Echo 18: 00 78 07 00 00 00 06 07 ... 1.17v [25:0A]
Echo 19: 00 78 07 00 00 00 06 07 ... 1.17v [26:0B]
Echo 1A: 00 78 07 00 00 00 06 07 ... 1.17v [25:0D]
Echo 1B: 00 79 07 00 00 00 06 07 ... 1.17v [27:0A]
<ESC>

Since there is no Echo unit functioning, the Ping side will now be showing:

... other stuff
Ping 04: failed: 0008 (0001): timed out
Attempting retry for packet #04
Ping 04: failed: 0008 (0001): timed out
Attempting retry for packet #04
Ping 04: failed: 0008 (0001): timed out
Attempting retry for packet #04
Ping 04: failed: 0008 (0001): timed out
Attempting retry for packet #04
Ping 04: failed: 0008 (0001): timed out
Attempting retry for packet #04
Ping 04: failed: 0008 (0001): timed out
Attempting retry for packet #04
Ping 04: Ping failed: 0008 (0001): timed out
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: failed: 0008 (0001): timed out
Attempting retry for packet #05
Ping 05: Ping failed: 0008 (0001): timed out
Ping 06: failed: 0008 (0001): timed out
What we see in action here is the retry feature of Ping mode, in which it tries to resend a packet for up to ten times before moving on. The “moving on” portion of this exercise is an artifact of the demo, for in an actual application it would be important to resolve the failure in a more robust fashion!

Any reported errors on the Ping side, and not just timeouts, will result in a retry. As we have seen before, in most circumstances the packet failures are intermittent, so retry often allows an acceptable level of communication even if there are failures of individual packets.

To reflect this, the Ping side now reports not only the number of errors encountered, but also the number of retries attempted.

To see this report, stop the Ping side with ESC:

A new report line has been added to show the Retry count, which is equal to the total number of individual packet errors encountered, while the Error count now means the number of packets which failed to get through, even with retries.

Now restart first Echo and then Ping, and let the units run for a while. You should be able to observe some intermittent failures on the Ping side by watching for “bobbles” in the scrolling data (particularly if you let your fuzzy cat rub up against one of them, etc.).

After several packet errors, stop the Ping again and look at the error report:

... other stuff
Ping C3: 00 78 07 00 1E 0C 06 07 ... 1.17v [1A:07]
Ping C4: 00 78 07 00 1E 09 06 07 ... 1.17v [1A:09]
Ping C5: 00 78 07 00 1E 13 06 07 ... 1.17v [19:0C]
Ping C6: 00 78 07 00 1F 0E 06 07 ... 1.17v [19:0A]
Ping C7: 00 78 07 00 1E 0B 06 07 ... 1.17v [19:0F]
Ping C8: 00 78 07 00 1E 0C 06 07 ... 1.17v [1A:0E]
Ping C9: 00 78 07 00 1E 0A 06 07 ... 1.17v [1B:0B]
Ping CA: 00 78 07 00 1E 0E 06 07 ... 1.17v [19:07]
Ping CB: \(\text{ESC}\)

Error count = 0000
Retry count = 001D

Note that even after many packet errors, in this case 29, the count of complete packet failures will likely still be zero, or close to it, unless there is a total channel failure.

And, in extreme cases, as long as there is still some channel capacity available for packets at all, an increased retry count, or more stretched and randomized retry interval, would still allow reasonably robust communication.

### 7.2.5 Forward Error Correction

The previous section gave examples of communication using variable length packets. Now, change both sides to fixed length packets with FEC using the \(G\) command and try again.

Can you notice any difference in the apparent ease of generating packet errors?

Note that FEC is a double-edged sword. In some cases, FEC helps by removing some burst errors. However, in other cases, as it doubles the air-time for the data, intermittent packets may become more exposed to error.

All things considered, outside of compelling evidence to the contrary, our recommendation is to enable FEC for production applications.

### 7.2.6 Channel Parameters

We leave it as an exercise for the reader to experiment with varying the channel parameters for Ping and Echo, as was done previously for the Tx and Rx modes.

### 7.2.7 Handheld Mode

It is a simple matter to detach Unit B from the serial cable and walk around with it. Repeat some of the exercises in the previous chapter and note how the retry feature of the \(PacketRadio\) module, combined with sequence numbers, greatly increases robustness of packet communication, even though individual packets may be blocked.

And, as the jumpers on Unit B are already set for Echo mode, it is also possible to remove the battery jumper from Unit B, and re-install it. Note how the unit immediately begins to answer Pings from Unit A.

And, since the communication is bi-directional, now both sets of LEDs respond to switches on the opposite unit.
7.3 Custom Ping Application Sample

In the previous chapter, we only skimmed the surface of the demo application packet radio functionality. In this chapter, we will implement a custom ping application. By so doing, we will demonstrate the core functionality of the packet radio modules. In addition, licensed users will receive the source code for these modules.

Before proceeding further, overwrite UserApp.c with the default UserApp_Original.c. The complete source code for this project is provided as UserApp_Ping.c.

7.3.1 Features

This custom ping application will implement the following features:

- Sends the local potentiometer value to the remote unit as a bar-graph.
- Displays the remote potentiometer value as a bar-graph on the local unit.
- Transmits only when SW4 is pressed.
- Retries each packet up to 20 times.
- Exits when SW1 is pressed or ESC is sent from the host terminal.

These features are reflected in the simplified state machine diagram below:

![State Machine Diagram](image)

**Figure 7-2 Custom Ping Application**

As before, we will evolve the application toward this goal, explaining as we go, rather than dump the entire project on you all at once!
7.3.2 Remove Default User Application

The first step in our custom ping is to rip out all the current user application code, leaving only the initialization framework and the Tx/Rx callbacks. The only reason the Tx/Rx callbacks will remain is because the project will not link without them.

Remove the LED timing `#define` and the declarations for the four LED states from the top of `UserApp.c`:

```c
// 1/8 second delays, at a system tick rate of 8192 Hz
#define LED_DEMO_DELAY_TICKS                    1024

void UserApp_Stop( void );
void UserApp_State_One( void );
void UserApp_State_Two( void );
void UserApp_State_Three( void );
void UserApp_State_Four( void );
```

In `UserApp_Start`, remove the LED and switch code, change the message, and change the terminal state:

```c
void UserApp_Start( LPFN_STATE_VOID_VOID pfnNextState )
{
    // Save the next client state on the state stack
    Task_PushState( pfnNextState );

    RxTxHost_QueueTxString( "\nCustom ping app started\n" );

    // Prepare next state
    Task_UpdateState( pfnNextState );
}
```

Remove the LED code from `UserApp_Stop`:

```c
void UserApp_Stop( void )
{
    Task_PopState();
}
```

At this point, the remaining `Task_PopState` call does nothing as this function is not yet hit. But, because of the unmatched `Push` and `Pop`, do not try to execute this application yet.

Delete the implementation for all four states `UserApp_State_One`, `Two`, `Three`, and `Four`. At this point there should be no code between `UserApp_Stop` function and the `Tx/Rx Client Callbacks` comment.

`Build` the project and correct any errors found. Do not load the code at this point.
7.3.3 Add User Application Framework

Now we will add the framework for the user application that handles the switches and optionally launches a packet operation. We will also impose a 5 millisecond delay between outgoing ping operations.

At the top of UserApp.c, add the following #defines:

```c
#define CHAR_ESCAPE 0x001B
// Prepare a post-ping delay of about 5 milliseconds
#define POST_PING_DELAY_TICKS (5*(SYSTEM_TICK_RATE/1000))
#define USER_APP_SW1 BIT4
#define USER_APP_SW4 BIT7
```

While not precise, the indicated delay math avoids overflow. Since we don’t care whether the delay is exactly five milliseconds, it is good enough.

Also near the top of UserApp.c, add the following function and state declarations:

```c
void UserApp_Stop( void );
BOOL UserApp_CheckForStop( void );

// Ping states
void UserApp_State_Idle( void );
void UserApp_State_StartPing( void );
```

Update UserApp_Start as follows:

```c
void UserApp_Start( LPFN_STATE_VOID_VOID pfnNextState )
{
    // Save the next client state on the state stack
    Task_PushState( pfnNextState );

    RxTxHost_QueueTxString( "\nCustom ping app started\n" );

    // Prepare SW1 to generate IFG, but not interrupts, on a press
    P1IES |= (USER_APP_SW1);
    P1IE &= ~(USER_APP_SW1);
    P1IFG &= ~(USER_APP_SW1);

    // SW4 is polled directly rather than using the interrupt flag

    // Prepare next state
    Task_UpdateState( UserApp_State_Idle );
}
```

Add the following function after UserApp_Stop:
void UserApp_CheckForStop( void )
{
    WORD wRxKey;

    // Check whether user pressed "ESC" on terminal
    wRxKey = RxTxHost_GetRxEcho();
    if( CHAR_ESCAPE == wRxKey )
    // User hit escape
    {
        RxTxHost_QueueTxString( "\nESC!" );
        return TRUE;
    }

    // Also, check whether SW1 has been pressed
    if( 0 != (P1IFG & USER_APP_SW1) )
    {
        RxTxHost_QueueTxString( "\nSW1!" );
        return TRUE;
    }

    return FALSE;
}

Add the two state functions after UserApp_CheckForStop:

void UserApp_State_Idle( void )
{
    BOOL bStop;

    // Check whether to stop
    bStop = UserApp_CheckForStop();
    if( FALSE != bStop )
    {
        UserApp_Stop();
        return;
    }

    // Also, check if SW4 is currently pressed
    if( 0 == (P1IN & USER_APP_SW4) )
    {
        // Start a packet
        Task_UpdateState( UserApp_State_StartPing );
    }
}

void UserApp_State_StartPing( void )
{
    // Send something to host
    RxTxHost_QueueTxString( "Packet!\n" );

    // For now, transition back to idle
    Task_DelayTicks( UserApp_State_Idle, POST_PING_DELAY_TICKS );
}

Build ☑️ the project, load it onto Unit A with Start Debugging 🔄, and execute the code with the Go 🐡 button. You should see the following on the terminal when you execute the U command:
Resetting...
ok
-u
Custom ping app started

Press SW4 briefly to generate some packet messages:
Packet!
Packet!
Packet!
Packet!
Packet!
Packet!

Now test ESC:

... other stuff
Packet!
Packet!
Packet!
Packet!
<ESC>
ESC!
ok
-

Restart the user application and try SW1:

-u
Custom ping app started
<SW1>
SW1!
ok
-

Note that even if SW4 is pressed, SW1 or ESC can still terminate the application because each iteration of the cycle passes through UserApp_State_Idle. Try this and convince yourself that this is the case.

Close the debugger with the Stop Debugging button.

In the next section we will blend in actual packet operations to our work thus far.
7.3.4 Add Packet Functionality

We shall now add actual packet functionality to our user application. This is going to be a VERY long section. Remember that all the code below is UserApp_Ping.c if you want to avoid the typing, but please read along to see all the remarks.

Add the following include statements near the top of UserApp.c:

```c
#include "System.h"
#include "CC1100.h"
#include "DigitalRadio.h"
#include "PacketRadio.h"
#include "RxTxModes.h"
#include "ChannelParameters.h"
#include "Utility.h"
```

Add the following define statements a little farther down:

```c
#define USER_APP_SW1 BIT4
#define USER_APP_SW4 BIT7
#define PING_PACKET_SIZE 0x20
#define DEFAULT_RETRY_COUNT 20
#define DEFAULT_RETRY_INTERVAL_TICKS 100
```

Add these function prototypes near the top of UserApp.c:

```c
void UserApp_Stop( void );
BOOL UserApp_CheckForStop( void );
void UserApp_PreparePacketParameters( void );
void UserApp_UpdateTxPacket( void );
```

These two new functions prepare the packet structures for operation before the first ping, and before each ping, respectively.

Add the following callback prototypes:

```c
// Packet radio callbacks
BOOL UserApp_Ping_OnIdle( void );
BOOL UserApp_Ping_OnPreAck( void );
BOOL UserApp_Ping_OnRetry( void );
BOOL UserApp_Ping_OnPacketComplete( void );
BOOL UserApp_Ping_OnError( void );
```

and declare the following additional state:

```c
void UserApp_State_Idle( void );
```
void UserApp_State_StartPing( void );
void UserApp_State_PingComplete( void );

Then add the following prototypes:

// Print formatting functions
void UserApp_PrintError( WORD wResult, WORD wResultParam );
void UserApp_PrintRxPacket( BYTE byCountRead,
                        BYTE byRSSI,
                        BYTE byLQI );

We next need to add several data elements used by the user application to manage the packet radio. The first set of these include sequence numbers, buffers, and various counters:

// Ping mode data
static BYTE mUserApp_bySequenceNumber, mUserApp_byLastSequenceNumber;
static BYTE mUserApp_abyTxData[PING_PACKET_SIZE];
static BYTE mUserApp_abyRxData[PING_PACKET_SIZE];
static WORD mUserApp_wRxPacketErrorCount;
static WORD mUserApp_wPingRetryCount;

Next, add the following RAM structures:

// Shared operation object
static SDigitalRadioOperation mUserApp_tOperation;

// Shared packet parameters object
static SPacketRadioParams mUserApp_tPacketParams;

// Digital radio structures used in packet Rx and Tx modes
// Digital radio Rx parameter object
static SDigitalRadioRxParams mUserApp_tDigRadRxParams;
// Digital radio Tx parameter object
static SDigitalRadioTxParams mUserApp_tDigRadTxParams;

The first of these structures is a collection of digital radio parameters defined in Section 11.4.12, Structure SDigitalRadioOperation, which receives result codes and can include user-specific data, if desired. This element can be shared between receive and transmit operations.

The second structure is a set of packet parameters defined in Section 11.10.3, Structure SPacketRadioParams, which controls acknowledge and retry operations.

The third structure is a collection of digital radio receive parameters, described in Section 11.4.13, Structure SDigitalRadioRxParams, which defines the incoming received packet and quality of reception.

The fourth structure is similar to the third, except that it defines the outgoing transmit packet, and is described in Section 11.4.15, Structure SDigitalRadioTxParams.
The next data element added is a ROM table that identifies the various parameters and callbacks used by the packet radio:

```c
static const SPacketRadio mUserApp_tPacketPing = {  
    &mUserApp_tOperation,  
    &mUserApp_tPacketParams,  
    &mUserApp_tDigRadRxParams,  
    &mUserApp_tDigRadTxParams,  
    UserApp_Ping_OnIdle,  
    UserApp_Ping_OnPreAck,  
    UserApp_Ping_OnRetry,  
    UserApp_Ping_OnPacketComplete,  
    UserApp_PingOnError  
};
```

This structure is described in Section 11.10.4, **Structure SPacketRadio**. This table first points to the four items previously added, and then to the five key client callbacks that handle the various radio events.

Next, update **UserApp_Start**, as shown below:

```c
void UserApp_Start( LPFN_STATE_VOID_VOID pfnNextState )
{
    BOOL bResult;

    // Attempt to lock the radio
    bResult = DigitalRadio_Lock();
    if( FALSE == bResult )
    {
        RxTxHost_QueueTxString( "\nUnable to lock radio for UserApp\n" );
        Task_UpdateState( pfnNextState );
        return;
    }

    // Save the next client state on the state stack
    Task_PushState( pfnNextState );

    RxTxHost_QueueTxString( "\nCustom ping app started\n" );

    // Prepare SW1 to generate IFG, but not interrupts, on a press
    P1IES |= (USER_APP_SW1);
    P1IE &= ~(USER_APP_SW1);
    P1IFG &= ~(USER_APP_SW1);

    // SW4 is polled directly rather than using the interrupt flag

    // Set the channel
    DigitalRadio_SetChannel( ChannelParameters_GetChannel() );

    // Set the features
    DigitalRadio_SetFeatures( ChannelParameters_GetFeatures() );

    // Prepare invariant parameters
    UserApp_PreparePacketParameters();
}
```
// Prepare next state
Task_UpdateState( UserApp_State_Idle );

The first addition tries to acquire a resource lock on the digital radio, which means that the locking task owns the resource exclusively until the lock is released. If the task cannot acquire the lock, the application exits after printing a warning message.

The second and third additions set the channel and features based on the settings entered by the user with the main menu. This means that this application can work in the various modes exercised previously, including channels, addresses, sequence numbers, packet formats, etc.

The fourth addition is a call to a function that performs a one-time preparation of the packet parameters.

Similarly, update the UserApp_Stop function as follows:

```c
void UserApp_Stop( void )
{
    // Report statistics
    RxTxHost_QueueTxString( "\n\nError count = " );
    RxTxHost_QueueTxHexWORD( mUserApp_wRxPacketErrorCount );
    RxTxHost_QueueTxString( "\nRetry count = " );
    RxTxHost_QueueTxHexWORD( mUserApp_wPingRetryCount );
    RxTxHost_QueueTxCRLF();

    // Unlock the radio
    DigitalRadio_Unlock();
    CC1100_StrobeIDLE();

    Task_PopState();
}
```

The first addition above prints the ping statistics seen when operation is cancelled. The second addition unlocks the radio resource and forces the CC1100 into idle.

After UserApp_CheckForStop add the following function:

```c
void UserApp_PreparePacketParameters( void )
// Prepares the Tx/Rx packets and packet radio parameters
{
    BYTE byIndex;

    // Pre-fill generic values
    for( byIndex = 0; byIndex < PING_PACKET_SIZE; byIndex++ )
    {
        mUserApp_abyTxData[byIndex] = byIndex;
    }

    // Fill in special values
    mUserApp_abyTxData[PACKET_OFFSET_ZERO] = 0x00;
```
This busy function performs the straight-forward process of initializing the packet parameters. First, the Tx packet is filled with dummy data, including the spaces that we will later fill with potentiometer data. Then, the sequence numbers and operation flags are initialized, followed by the operation result structure. Next, the Tx and then Rx parameters are prepared. Finally, the retry parameters are initialized. With the exception of offsets 0x04 and 0x05, which are both set to 0x00 in the demo application, this code mirrors exactly the functionality of the demo Ping application.

The above function is used on startup of the application, but another function is called each time a packet is about to be transmitted. This function is defined next:
void UserApp_UpdateTxPacket( void )
// Updates variable information in the Tx packet
{
    BYTE byFlags;
    WORD wADC0;

    // Get the conversion
    wADC0 = ADC12MEM0;
    mUserApp_abyTxData[PACKET_OFFSET_ADC_LO] = LOBYTE( wADC0 );
    mUserApp_abyTxData[PACKET_OFFSET_ADC_HI] = HIBYTE( wADC0 );

    // Get the switch states
    byFlags = P1IN & 0xF0;
    byFlags >>= 4;
    byFlags ^= 0x0F;
    mUserApp_abyTxData[PACKET_OFFSET_SWITCHES] = byFlags;
}

The first element to be updated is the potentiometer setting. Our application, thanks to code in System.s43, is constantly converting the potentiometer setting into ADC12MEM0. The code here just plucks the most recent value and shoves it in a packet.

In a production application, a cleaner approach would be to encapsulate the potentiometer measurement in a separate module, and merely return the most recent conversion at the time of the call, updating this value during ADC12 interrupts. Here we took the most expeditious route to constantly accessing the potentiometer value.

The second half of this new function updates the switches portion of the Tx packet based on the LED settings.

These two portions of code mirror exactly the packet data that is sent by the demo Ping application. We will change this code later to implement the bar-graph functionality.

Next, update UserApp_State_StartPing and add UserApp_State_PingComplete:

void UserApp_State_StartPing( void )
{
    // Update the Tx packet with switches and potentiometer
    UserApp_UpdateTxPacket();

    mUserApp_byLastSequenceNumber = mUserApp_bySequenceNumber++;

    // Update the Tx packet with the new sequence number
    mUserApp_tDigRadTxParams.byTxSequenceNumber = \mUserApp_byLastSequenceNumber;

    // Start a packet exchange
    PacketRadio_StartTx( &mUserApp_tPacketPing,
                        UserApp_State_PingComplete );
    RxTxHost_QueueTxString( "UserPing " );
    RxTxHost_QueueTxHexBYTE( mUserApp_byLastSequenceNumber );
RxTxHost_QueueTxString( ": " );
}

void UserApp_State_PingComplete( void )
{
    // For now, transition back to idle after a delay
    Task_DelayTicks( UserApp_State_Idle, POST_PING_DELAY_TICKS );
}

The StartPing state first updates the Tx packet as discussed previously. Then, the sequence number is updated and embedded into the packet parameters. The fourth statement above starts a state-based Tx operation using PacketRadio_StartTx, which uses the previously initialized packet structures, and identifies the state to follow this packet radio operation. This API function is described in Section 11.10.1. Finally, a user message is started.

Note also that the delay has been moved to PingComplete.

Next, below UserApp_State_PingComplete, start a new code section for the client callbacks and add:

    ///////////////////////////////////////////////////////////////////////////
    // Packet Radio Callbacks
    ///////////////////////////////////////////////////////////////////////////

BOOL UserApp_Ping_OnIdle( void )
// This is called each time the digital radio Ping idle ticks
{
    BOOL bStop;

    // Check whether to stop
    bStop = UserApp_CheckForStop();
    if( FALSE != bStop )
    {
        return FALSE;
    }

    return TRUE;
}

This first callback, OnIdle, is executed while the packet radio is executing various idle states. Returning FALSE will cancel operation, which this code does if a stop condition is detected.

Add the following OnPreAck callback next.

BOOL UserApp_Ping_OnPreAck( void )
// Return FALSE to cancel the Ack Rx, TRUE to allow Ack
// Almost all applications will allow Acks
{
    return TRUE;
}
This callback is merely a stub, as we want to enable acknowledgements, and the corresponding retry mechanism this provides.

Following *OnPreAck*, add:

```c
BOOL UserApp_Ping_OnRetry( void )
{
    // Disallow retries for cancel
    if( DIGITAL_RADIO_CANCEL == (mUserApp_tOperation.wResult & \
        PACKET_RADIO_RESULT_MASK) )
    {
        return FALSE;
    }

    mUserApp_wPingRetryCount++;

    UserApp_PrintError( mUserApp_tOperation.wResult, \
        mUserApp_tOperation.wResultParam );
    RxTxHost_QueueTxString( "Attempting retry for packet #" );
    RxTxHost_QueueHexBYTE( \
        mUserApp_tDigRadTxParams.byTxSequenceNumber );
    RxTxHost_QueueCRLF();
    RxTxHost_QueueString( "UserPing " );
    RxTxHost_QueueHexBYTE( mUserApp_byLastSequenceNumber );
    RxTxHost_QueueString( ": " );

    // Allow it
    return TRUE;
}
```

This callback executes prior to a retry in the case of a missing or erroneous received packet. The code above ensures that retries are halted in the case of a user cancel, as commanded in the *OnIdle* callback. Otherwise, all errors are formatted by *PrintError* and reported to the host terminal.

The fourth callback to add is the *OnComplete* callback:

```c
BOOL UserApp_Ping_OnPacketComplete( void )
// The next state has been updated when this is called
{
    // Make sure the sequence numbers match
    if( mUserApp_byLastSequenceNumber != \
        mUserApp_tDigRadRxParams.byRxSequenceNumber )
    {
        RxTxHost_QueueTxString( "Expected sequence " );
        RxTxHost_QueueHexBYTE( mUserApp_byLastSequenceNumber );
        RxTxHost_QueueTxString( " received sequence " );
        RxTxHost_QueueHexBYTE( \
            mUserApp_tDigRadRxParams.byRxSequenceNumber );
        RxTxHost_QueueCRLF();
        mUserApp_wRxPacketErrorCount++;
    }
    else
    {
        // Print the packet in parsed format
```
This callback is executed when an acknowledge packet has been received at the end of
the ping’s Tx+RxAck process has completed. The acknowledge packet may contain an
unexpected sequence number mismatch, particularly if multiple networks are operating
on the same channel. This case is the only error not handled by the retry mechanism,
and should be handled at the application level.

If the sequence numbers match, then the received packet is printed to the host terminal.

The final callback to be added is the OnError handler:

BOOL UserApp_Ping_OnError( void )
// The next state has been updated when this is called
{
    if( DIGITAL_RADIO_CANCEL == (mUserApp_tOperation.wResult & 
        PACKET_RADIO_RESULT_MASK) )
        // Cancelled during the OnIdle handler
        { UserApp_Stop();
        } else
        { mUserApp_wRxPacketErrorCount++;
            RxBFHost_QueueTxString( "UserPing " );
            UserApp_PrintError( mUserApp_tOperation.wResult,
                mUserApp_tOperation.wResultParam );
        }

        return TRUE;
}

This callback executes when the operation has been cancelled, or when the retry
mechanism has exhausted its efforts. In the former case, the application is stopped. In
the latter case, the last error is printed, as by definition it was not presented to the
OnRetry callback.

The final changes in this long effort is to add the two print formatting functions. Create a
new code section and add:

//////////////////////////////////////////////////////////
// Packet Print Functions
//////////////////////////////////////////////////////////
void UserApp_PrintError( WORD wResult, WORD wResultParam )
{
    RxTxHost_QueueTxString( "failed: " );
    RxTxHost_QueueTxHexWORD( wResult );
    RxTxHost_QueueTxString( " (" );
    RxTxHost_QueueTxHexWORD( wResultParam );
    RxTxHost_QueueTxString( ": " );

    switch( __even_in_range( (wResult & PACKET_RADIO_RESULT_MASK), \ 
                           DIGITAL_RADIO_MAX_ERROR ) )
    {
        case DIGITAL_RADIO_BAD_PARAMS:
            RxTxHost_QueueTxString( "bad parameters" );
            break;
        case DIGITAL_RADIO_RESOURCE_LOCKED:
            RxTxHost_QueueTxString( "resource locked" );
            break;
        case DIGITAL_RADIO_FAIL_TIMEOUT:
            RxTxHost_QueueTxString( "timed out" );
            break;
        case DIGITAL_RADIO_FAIL_CRC:
            RxTxHost_QueueTxString( "invalid CRC" );
            break;
        case DIGITAL_RADIO_FILTERED_OUT:
            RxTxHost_QueueTxString( "filtered out" );
            break;
        case DIGITAL_RADIO_INVALID_SIZE:
            RxTxHost_QueueTxString( "invalid packet size" );
            break;
        case DIGITAL_RADIO_CONTENT_MISMATCH:
            RxTxHost_QueueTxString( "content mismatch" );
            break;
        case DIGITAL_RADIO_RX_BUFFER_OVERFLOW:
            RxTxHost_QueueTxString( "RXFIFO overflow" );
            break;
        case DIGITAL_RADIO_TX_BUFFER_UNDERFLOW:
            RxTxHost_QueueTxString( "TXFIFO underflow" );
            break;
    }

    RxTxHost_QueueTxCRLF();
}

This lengthy function formats each error recognized by the system and prints a corresponding error message to the host terminal. These messages are identical to those reported by the demo Ping application.
The second print format function, and the last to be added during this phase is shown below:

```c
void UserApp_PrintRxPacket( BYTE byCountRead, BYTE byRSSI, BYTE byLQI )
{
    WORD wRxADC, wTxCapacity;
    BYTE byIndex, byDisplayed;
    char szVoltage[8];

    wTxCapacity = RxTxHost_GetCountTxCapacity();
    // Make sure room exists for the packet data
    if( wTxCapacity < 80 )
    // Skip the full display
    {
        RxTxHost_QueueTxString( "+++
" );
        return;
    }
    // Write the data
    if( byCountRead > 8 )
    {
        byDisplayed = 8;
    }
    else
    {
        byDisplayed = byCountRead;
    }
    for( byIndex = 0; byIndex < byDisplayed; byIndex++ )
    {
        RxTxHost_QueueTxHexBYTE( mUserApp_abyRxData[byIndex] );
        RxTxHost_QueueTxChar( ' ' );
    }
    if( byCountRead > 8 )
    {
        RxTxHost_QueueTxString( "... " );
    }
    // Update voltage display
    wRxADC = MAKEWORD( mUserApp_abyRxData[PACKET_OFFSET_ADC_LO], mUserApp_abyRxData[PACKET_OFFSET_ADC_HI] );
    Utility_FormatVoltage_2_50v( wRxADC, szVoltage );
    RxTxHost_QueueTxString( szVoltage );
    // Update LEDs
    P1OUT &= 0xF0;
    P1OUT |= (mUserApp_abyRxData[PACKET_OFFSET_SWITCHES] & 0x0F);
    // Write RSSI:LQI
    RxTxHost_QueueTxChar( ']' );
    RxTxHost_QueueTxHexBYTE( byRSSI );
    RxTxHost_QueueTxChar( ':' );
    RxTxHost_QueueTxHexBYTE( byLQI & 0x7F );
    RxTxHost_QueueTxString( "\n" );
}
```

As with the previous function, this function is identical to that used by the demo Ping application. It first check to see if sufficient room exists to print the packet to the terminal. If not, it prints a truncated message, and returns. Otherwise, the packet data is displayed in the now familiar format.
Now let's build, load, and execute the project on Unit A. Make sure that Unit B is operating in Echo mode, and then launch the user application with $U$:

```
-u
Custom ping app started
```

Press SW4 briefly, and packets will ping out to the echo unit:

```
UserPing 00: 00 07 08 00 16 09 06 07 ... 1.25v [14:0A]
UserPing 01: 00 07 08 00 15 12 06 07 ... 1.25v [11:12]
UserPing 02: 00 07 08 00 16 0E 06 07 ... 1.25v [11:11]
UserPing 03: 00 07 08 00 16 11 06 07 ... 1.25v [12:15]
UserPing 04: 00 07 08 00 15 0D 06 07 ... 1.25v [13:0F]
UserPing 05: 00 07 08 00 15 0B 06 07 ... 1.25v [13:14]
```

At Unit B, the following packets are seen:

```
Echo 00: 00 FE 08 00 04 05 06 07 ... 1.40v [16:09]
Echo 01: 00 FD 08 00 04 05 06 07 ... 1.40v [15:12]
Echo 02: 00 FE 08 00 04 05 06 07 ... 1.40v [16:0E]
Echo 03: 00 FF 08 00 04 05 06 07 ... 1.41v [16:11]
Echo 04: 00 04 09 00 04 05 06 07 ... 1.41v [15:0D]
Echo 05: 00 01 09 00 04 05 06 07 ... 1.41v [15:0B]
```

For fun, hold SW2 and twitch SW4 to send a few packets. Note that D2 on Unit B turns on and stays on. Release SW2, twitch SW4 again, and D2 turns off.

In the next section, we complete the custom ping user application by adding the bar-graph feature.
7.3.5 Add the Bar Graph

In the previous section, we demonstrated the custom ping user application, including bi-directional traffic. In the default ping application, the local switches are mapped onto the LEDs at the remote. In this section, we add the bar-graph feature by mapping the potentiometer data onto the LEDs instead. We will do this without touching the echo unit’s code whatsoever.

In each packet in the demo application, the switch settings are embedded at the offset defined by PACKET_OFFSET_SWITCHES, or offset 0x03 in the current demo. These switch settings determine which LEDs will be illuminated at the remote.

So, by mapping the potentiometer measurement into virtual switch presses, we can manipulate the remote’s LEDs at will. In addition, we will re-interpret the remote’s potentiometer value to create a LED bar-graph on the local LEDs also.

To perform this mapping, we need a function that maps potentiometer measurements into a 4-bit LED value. At the top of UserApp.c add the following prototype:

```c
void UserApp_Stop( void );
BOOL UserApp_CheckForStop( void );
void UserApp_PreparePacketParameters( void );
void UserApp_UpdateTxPacket( void );
BYTE UserApp_Map_ADC12_To_LED( WORD wADC12 );
```

MSP430s are excellent at table lookups, so we will implement this map as a sixteen-entry BYTE table. Add this table after the mUserApp_tPacketPing structure:

```c
static const BYTE mUserApp_awMapLED[] =
{
  0x00, // Offset 0
  0x00, // Offset 1
  0x00, // Offset 2
  0x01, // Offset 3
  0x01, // Offset 4
  0x01, // Offset 5
  0x03, // Offset 6
  0x03, // Offset 7
  0x03, // Offset 8
  0x03, // Offset 9
  0x07, // Offset A
  0x07, // Offset B
  0x07, // Offset C
  0x0F, // Offset D
  0x0F, // Offset E
  0x0F, // Offset F
};
```

This table is declared as a `const` to ensure that it is placed in flash. Note that by manipulating the entries in this table, we could synthesize a variety of non-linear and scaling functions very easily.
Now let's add the map function after `UserApp_UpdateTxPacket`:

```c
BYTE UserApp_Map_ADC12_To_LED( WORD wADC12 )
// Convert the 12-bit ADC12 value into a nybble suitable
// for display on the LEDs as a bar-graph
{ BYTE byIndex, byMappedValue;
    byIndex = HIBYTE(wADC12 & 0x0F00);
    // byIndex ranges from 0x00 to 0x0F

    byMappedValue = mUserApp_awMapLED[byIndex];
    return byMappedValue;
}
```

This conversion could have easily been implemented as a macro, but we chose to implement it as a function. This decision in some cases will pay off when additional functionality, such as rounding, may be required later.

Now modify `UserApp_UpdateTxPacket`:

```c
void UserApp_UpdateTxPacket( void )
// Updates variable information in the Tx packet
{ BYTE byFlags;
    WORD wADC0;
    // Get the conversion
    wADC0 = ADC12MEM0;
    mUserApp_abyTxData[PACKET_OFFSET_ADC_LO] = LOBYTE( wADC0 );
    mUserApp_abyTxData[PACKET_OFFSET_ADC_HI] = HIBYTE( wADC0 );

    // Convert the ADC12 value
    byFlags = UserApp_Map_ADC12_To_LED( wADC0 );
    mUserApp_abyTxData[PACKET_OFFSET_SWITCHES] = byFlags;
}
```

This handles the outgoing bar-graph, so all we need to do now is map the incoming potentiometer value into a local bar-graph. To do so, add a local variable at the top of `UserApp_PrintRxPacket`:

```c
void UserApp_PrintRxPacket( BYTE byCountRead, BYTE byRSSI, BYTE byLQI )
{ WORD wRxADC, wTxCapacity;
    BYTE byIndex, byDisplayed, byBarGraph;
    char szVoltage[8];

    // Update LEDs
    byBarGraph = UserApp_Map_ADC12_To_LED( wRxADC );
    P1OUT &= 0xF0;
    P1OUT |= (byBarGraph & 0x0F);
```

and modify the LED code near the bottom of this function:
Now build, load, and execute the application on Unit A:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

ok

Execute the custom ping application with the U command:

-u
Custom ping app started

and send a few packets to the remote unit with SW4:

UserPing 00: 00 07 08 00 09 0F 06 07 ... 1.25v [04:19]
UserPing 01: 00 07 08 00 09 0B 06 07 ... 1.25v [05:0C]
UserPing 02: 00 07 08 00 08 0E 06 07 ... 1.25v [05:05]
UserPing 03: 00 07 08 00 08 0D 06 07 ... 1.25v [05:11]

In this case, since the remote potentiometer was set at about mid-scale, the voltage was interpreted as a mid-scale setting on our local display, thus illuminating D1 and D2.

At the remote, the following packets were observed:

Echo 00: 00 FE 08 03 04 05 06 07 ... 1.40v [09:0F]
Echo 01: 00 04 09 03 04 05 06 07 ... 1.41v [09:0B]
Echo 02: 00 01 09 03 04 05 06 07 ... 1.41v [08:0E]
Echo 03: 00 FD 08 03 04 05 06 07 ... 1.40v [08:0D]

and similarly, D1 and D2 were illuminated on the remote.

Now experiment with the bar-graph on both ends by twirling the potentiometer while pressing SW4 to send packets.

As mentioned previously, the code for this chapter is provided in its entirety as *UserApp_Ping.c*.

Before proceeding further, overwrite *UserApp.c* with the default *UserApp_Original.c*. 
This page intentionally left blank.
8. Hub/Node Demo

The previous chapter exercised the more advanced point-to-point functionality available in the Ping and Echo modes. This chapter describes the Hub and Node functionality available as a star network topology in the demo firmware. The next chapter will provide more detail about the DCO tracking algorithms.

8.1 Hub / Node Architecture

Just as the Ping / Echo architecture built upon the foundation of the Tx / Rx architecture, so does the Hub / Node build upon the Ping / Echo architecture, as we shall see in a moment.

But first, we need to briefly discuss the concept of a Hub / Node network, also known as a star topology:

![Figure 8-1 Star Topology](image)

In a star topology, there is a central hub through which all communication passes. Nodes do not directly communicate with each other, but if they need to, then the network protocol, defined by the software, must provide a means to do so by passing messages through the hub. If nodes were to communicate directly, then that would be known as a mesh topology.

A mesh network provides some key advantages over a star, including a) no single point of failure, such as a hub, which can cause the entire system to fail, and b) self-healing and automatic recognition as nodes are added to and removed from a network.

The Zigbee protocol, trademarked and copyrighted by the Zigbee Alliance, defines a robust industry-standard way of designing a mesh network. Readers whose applications require a mesh network are encouraged to research Zigbee further.
However, a mesh network has the disadvantage of requiring all nodes to be in receive mode when not actively transmitting, or else beacon periodically to scan for activity. In MSP430 terms, this implies significant power consumption at all nodes in the mesh network.

In contrast, reconsider the star network, with the nodes renamed as *sensors*, and the hub as a *recorder*:

![Figure 8-2 Sensor Network](image)

In this example, a sensor may be collecting and storing data for a long period, completely off the air, with greatly reduced power consumption. There are many other examples in which a similar topology applies to other applications, with the discriminating factor being responsiveness to input and relative on-air time.

Periodically, or through some additional stimulus, the sensors (nodes) come alive, go on air, and burst their data to the recorder (hub). The recorder, typically connected to some relatively larger power source such as the mains, a machine’s power supply, a vehicle’s power supply, solar panel, etc., is not as power limited as the nodes, which may typically be battery powered. As such, the recorder has no limitations with being in continuous receive mode, available when needed to answer the call of a node.

The recorder may also be connected to some larger data pipe such as an intranet or a vehicle data bus, and so serves as a conduit for the sensor traffic to a downstream system.

Granted, a mesh network could be made to operate in an identical fashion, but the long off-air times required to conserve power would cripple to some extent the self-healing and automatic recognition modes of operation that recommend such a network.

Since the name of the MSP430 game is power consumption, the reader must carefully decide whether a mesh network is required, or if a star network is sufficient. If a star suffices, then read on. For the remainder of this chapter, we will not consider the mesh option further, and focus instead on the star topology.
8.1.1 Search Mode

Prior to the network performing useful work, a node and hub must agree that they belong to the same network. The process of associating nodes with hubs is called the search mode. There are many ways to accomplish this task, and the technique used with our demo software is described briefly in this section.

In the approach taken here, the hub is the master of search mode, unassociated nodes are slaves, and associated nodes are idle. The hub contains N slots with which to associate specific nodes. For our demo firmware, N is set to 4. The hub terminal application allows the user to specify to which node slot the node requesting that slot will be assigned. We will see an example of this assignment later.

The hub and each node are given unique serial numbers. In production, the PrgUSBP serializing feature, available in the full version of PrgUSBP, is useful for this purpose, but we will assign serial numbers at build time for our examples in this chapter.

For our demo software, search mode always operates on channel 0, with the hub responding to messages at address 0xF0.

Each node will send a search packet, containing the node’s serial number and its desired node address, to the hub at that address. Upon receipt of the search packet, the hub assigns the node to the desired slot, and communicates this assignment via the acknowledgement packet to the node at address 0xF0 + n, where n is the specific slot assignment (1 ≤ n ≤ N), along with the serial number of the hub.

![Figure 8-3 Hub/Node Search Mode](image-url)
After this process has been performed for each node, a star network then exists, and active mode can then be entered.

Note that both the hub and node were commanded by the user to be assigned a specific slot. We use that approach here to demonstrate the techniques involved, but nothing prohibits alternative approaches, such as dynamic assignment of nodes to hubs, dynamic slot assignments, etc. Based on the Ping-Echo paradigm, it would be a simple matter for a licensed user to change the algorithms of the Hub and Node to follow different assignment strategies than the one presented here.

### 8.1.2 Active Mode

Once a set of nodes has been associated with a hub, the active mode is used in which a hub waits for nodes to transmit, and then acknowledges the node activity. In active mode, the network essentially operates as a multi-master system, in which the nodes are the masters and the hub is the slave.

In our demo firmware, all active mode packets use channel 2, the hub responds to messages at address 0x40, and nodes at address 0x40 + n.

![Figure 8-4 Hub/Node Active Mode](image)

In case of conflict, nodes use a delay-then-retry strategy, with the delay randomized based on the slot number n.

In the next section, we will see an example of this approach, including the use of untethered nodes to allow configuration without a terminal. But first, let’s examine the state diagrams for a node and a hub.
8.1.3 Node States

Implemented in NodeMode.c, a node executes the following state machine, simplified for clarity:

**Figure 8-5 Node Mode States**

Upon startup, a node attempts to load a node file. If a valid file is found, the node enters *active mode*. If no valid node file is found, the node enters *search mode*.

In active mode, a node waits for a user switch event on SW4. If a switch event is encountered, then the node performs the process of sending a packet to the hub. The
double-wall state indicates that this state is composed of many smaller states which executes this process.

On the other hand, if SW3 is pressed and held for five or more seconds, then the node file is erased and the node enters search mode by waiting for a hub search message addressed to Node 1. While this command interval is being timed, D2 flashes with increasing urgency until the command interval expires. An example of this behavior is given in Section 8.5.5, Erasing a Node File.

Recall that search mode can also be entered if a valid node file is not found. In either case, pressing SW3 causes repeated transitions into successive states waiting as Node 2, Node 3, etc. to Node 6. A further switch press takes the node back to waiting as Node 1.

Waiting in any of these states also causes an LED sequence to flash on D1. This flash sequence indicates which of the waiting states the node is currently in: four flashes indicates the node is waiting as Node 4, five flashes as Node 5, etc.

Also, while in search mode, any SW3 press is indicated by a brief, timed flash on D2.

In any of these search mode waiting states, a valid hub message causes the node to register with the hub as the appropriate node ID. This configuration information is then written to the node file. After writing the file, the node enters active mode.
8.1.4 Hub States

The hub executes the following states, simplified for clarity:

Upon entry into hub mode, the hub file is loaded. If the file is invalid or does not exist, then search mode is entered. Otherwise, if there is a valid file, its contents are loaded into RAM and active mode is entered.

In active mode, the hub idles, waiting for incoming node packets. When a node packet arrives, a subset of states executes to handle that packet. Afterward, control is returned to the active idle state.

Active mode is exited by pressing ESC, whereupon search mode is entered.

Whether search mode is entered from failure to load a hub file, or from active mode via ESC, the hub idles at a menu prompt.

Figure 8-6 Hub Mode States
If $D$ is pressed, the hub displays the contents of the current hub file to the terminal host, and then re-enters the hub menu idle state.

If $Z$ is pressed, then the hub file is erased, and control returns to the menu idle state.

If 1, 2, 3, or 4 is pressed, then a node search message is sent, and the sub-states required to handle this process is executed. If an error is encountered, the hub returns directly to the menu idle state. However, if a node is found, it is logged in the file and the file is written to flash. After which, control returns to the menu idle state.

Finally, if $A$ is pressed, the hub re-enters active mode, using the most recently written hub file for the network configuration.
8.2 Serial Number Preparation

Before launching into the demonstration, we must first prepare the hardware units with builds of the firmware that contain serial numbers so that the hub and nodes can discriminate among themselves.

8.2.1 Serial Number Definition

Serial numbers are assigned in the demo firmware through code found in `SerialNumber.c`:

```c
// Info variables
#pragma constseg=SERNUM
static const DWORD dwSerial = 0x88886700;
#pragma constseg=default
```

This code places the serial number `0x88886700` into the constant segment named `SERNUM`.

Thanks to the following definition in `DemoMRF1611CC1100.xcl`:

```c
// Serial number space
-Z(CONST)SERNUM=FFD8-FFDF
```

the constant segment `SERNUM` is 8 bytes long and located at 0xFFD8. Our serial number only uses the first four bytes of this constant segment.

The generated binary, if the `msp430-txt` linker option is chosen, generates the following fragment of binary image in the `DemoMRF1611CC1100.txt`:

```
@FFD8
00 67 88 88
```

Note the byte-reversal as the MSP430 expects data to be laid out in memory least-significant-byte (LSB) first.

8.2.2 Serialization Techniques

There are three basic techniques for serializing each unit, and are discussed briefly in this section.

The first way of loading a unique serial number into the units would be to change this number in `SerialNumber.c`, rebuild, and burn the firmware, repeating the process for each unit. This approach would get old very fast, but might be suitable for setting up just a few units for test purposes.
The second and best approach would be to directly program in a serial number for each unit when the firmware is loaded by using the serializing feature of the full version of PrgUSBP. This would be accomplished by changing the serial number in SerialNumber.c to all 0xFF:

```c
// Info variables
#pragma constseg=SERNUM
static const DWORD dwSerial = 0xFFFFFFFF;
#pragma constseg=default
```

leaving an erased hole in memory at address 0xFFD8 for the serializer to update. However, without a license to the full version of PrgUSBP, this approach is not suitable for evaluating the demo firmware.

The third technique, and the one used there, changes the firmware binary directly to give each unit a unique file that gets programmed into a specific unit.

Recall that the PrgUSBP command file DemoMRF1611CC1100.usbprj defined a separate operation for the Hub and each of four Nodes. Each of these operations reference a specific build, each containing a specific serial number, in accordance with the table below:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Serial #</th>
<th>Operation</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x88886700</td>
<td>Hub</td>
<td>DemoMRF1611CC1100_Hub.txt</td>
</tr>
<tr>
<td>B</td>
<td>0x88886701</td>
<td>Node 1</td>
<td>DemoMRF1611CC1100_Node1.txt</td>
</tr>
<tr>
<td>C</td>
<td>0x88886702</td>
<td>Node 2</td>
<td>DemoMRF1611CC1100_Node2.txt</td>
</tr>
<tr>
<td>D</td>
<td>0x88886703</td>
<td>Node 3</td>
<td>DemoMRF1611CC1100_Node3.txt</td>
</tr>
<tr>
<td>E</td>
<td>0x88886704</td>
<td>Node 4</td>
<td>DemoMRF1611CC1100_Node4.txt</td>
</tr>
</tbody>
</table>

Each of these builds is identical to the default demo build, with the exception of the serial numbers embedded within them.

We shall use this table of serial numbers in the next step, along with PrgUSBP, to prepare each node unit accordingly.
8.2.3 Prepare Node Hardware

In this section, we will prepare the node hardware for operation with the demo firmware.

Using PrgUSBP on PC B, program Units B, C, D, and E with their respective node firmware as shown in Table 8-1 Hub and Node Serialization.

Prepare each unit with batteries, checking the individual cell voltages to make sure each is 1.2v or higher, but leave the VB:VIN jumper hanging for the moment.

Finally, configure the HMRF1611 jumpers for Unit B as follows:

<table>
<thead>
<tr>
<th>Header</th>
<th>Option</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Power Off</td>
<td>Waiting until Hub is prepared</td>
</tr>
<tr>
<td>Analog</td>
<td>Default</td>
<td>n/a</td>
</tr>
<tr>
<td>Port3 / Comm</td>
<td>Unit Connected to PC</td>
<td>For terminal operation</td>
</tr>
<tr>
<td>Port4</td>
<td>Node Mode</td>
<td>n/a</td>
</tr>
</tbody>
</table>

and configure the HMRF1611 jumpers for Units C, D, and E as follows:

<table>
<thead>
<tr>
<th>Header</th>
<th>Option</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Power Off</td>
<td>Waiting until Hub is prepared</td>
</tr>
<tr>
<td>Analog</td>
<td>Default</td>
<td>n/a</td>
</tr>
<tr>
<td>Port3 / Comm</td>
<td>Stand Alone</td>
<td>For handheld operation</td>
</tr>
<tr>
<td>Port4</td>
<td>Node Mode</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Close PrgUSBP, and connect Unit B to the serial port. Jumper power with VB:VIN, and ensure that the terminal application is reporting:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

Node mode started, serial #8888:6701
Loading node file: failed 0x0004, file not found
Listening for hub as node 01

If not, debug the setup until it does before proceeding.
8.2.4 Prepare Hub Hardware

On PC A, copy UserApp_Original.c over UserApp.c to restore the default user application, and restart Embedded Workbench.

Configure the HMRF1611 jumpers for Unit A as follows:

<table>
<thead>
<tr>
<th>Header</th>
<th>Option</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Power Off</td>
<td>Unit powered from USBP</td>
</tr>
<tr>
<td>Analog</td>
<td>Default</td>
<td>n/a</td>
</tr>
<tr>
<td>Port3 / Comm</td>
<td>Unit Connected to PC</td>
<td>For terminal operation</td>
</tr>
<tr>
<td>Port4</td>
<td>Hub Mode</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Rebuild the project and load onto Unit A via the Debug button, and hit the Go button to execute the firmware.

Ensure that the terminal on PC A is reporting either:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

Hub mode started, serial #8888:6700

Loading hub file: success

Listening for nodes as hub

or

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

Hub mode started, serial #8888:6700

Loading hub file: failed 0x0004, file not found
Writing hub file to flash

+

The former case will occur if a valid Hub file, even if empty, has been loaded, in which case the Hub will immediately go into Active Mode.

The latter case, Search Mode, occurs if no valid Hub file is loaded, as would be the case with erased flash, in which case the file’s sectors have not been formatted.
If you see neither message, debug the preparation until you do.

We want to force the Hub into Search Mode, so if it found a file and went straight into Active Mode, cancel that mode and go to Search Mode by hitting ESC:

```
MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

Hub mode started, serial #8888:6700

Loading hub file: success

Listening for nodes as hub

<ESC>

User cancelled operation

+

Any time you see the “+” prompt, the Hub is in Search Mode. If you see the following line:

Listening for nodes as hub

then the Hub is in Active Mode.

The Hub can be forced from Search Mode to Active Mode by using the A command:

+a

Listening for nodes as hub

Likewise, the Hub can be forced from Active Mode to Search Mode by hitting ESC:

Listening for nodes as hub

<ESC>

User cancelled operation

+

Force the Hub into Search Mode and leave it there for the next section.
8.3  Search Mode

Let us first experiment with the Unit B Node. Recall that it booted into Node Mode due to the jumper settings.

So, because the firmware did not find a valid Node file, Unit B is in Search Mode, but as a Node instead of a Hub. If there had been a valid Node file in flash, Unit B would have moved directly into Active Mode.

Observe that Unit B is flashing D1 about every 2.5 seconds. It is indicating that it wants to connect as Node 1. This connection preference has nothing whatsoever to do with the serial number we assigned, as the serial number and the Hub slot are completely independent, as we shall soon see.

We can change the search slot assigned to this node by successive brief presses of SW3:

Listening for hub as node 01<SW3>
User changed node ID

Listening for hub as node 02

Note that D2 echoes the switch press, while D1 now flashes twice to indicate it is waiting for the Hub to assign it slot 2.

Scroll the Node back around to slot 1:

Listening for hub as node 02<SW3>
User changed node ID

Listening for hub as node 03<SW3>
User changed node ID

Listening for hub as node 04<SW3>
User changed node ID

Listening for hub as node 05<SW3>
User changed node ID

Listening for hub as node 06<SW3>
User changed node ID

Listening for hub as node 01

As shown above, the available slots go all the way up to 6, and then rolls over to 1. But, with our current demo firmware, the Hub does not accept slot requests 5 and 6.
Ensure that Node 1 is flashing once, and command the Hub to connect to that Node in slot 1 by using the 1 command:

```
+1
Searching for node 01

Found node #8888:6701
Writing hub file to flash
```

And at the Node’s terminal the following is displayed:

```
Listening for hub as node 01

Found by hub #8888:6700
Writing node file to flash

Active as node 01
```

Also observe that the Node is no longer flashing D1.

Now, display the Hub’s flash file and examine the contents:

```
+d
Hub file contents:
wMagicNumber:    0x9387
wFeatures:       0x007F
byRxAddress:     0x40
byChannel:       0x02
byNodeRecordCount: 0x04

Node Record 0x01:  Unit #8888:6701 at address 0x41
Node Record 0x02:  Unused
Node Record 0x03:  Unused
Node Record 0x04:  Unused
```

Observe that the Node has now been assigned a slot, and the corresponding record has been associated with that Node in flash.

Although we can watch the Node’s actions on the terminal, the Node firmware is written such that it can be configured as a handheld device. To demonstrate this, apply power to Unit C by jumpering VB:VIN.

You should observe D1 flashing on this unit, indicating it is seeking assignment to slot 2. Using SW3 twice, scroll over to slot 3. Ensure that D1 is now flashing three times per cycle.
Now, ask the Hub to attempt to connect to a missing node by using the 2 command:

```
+2
Searching for node 02
.........................Search failed: 0008 (0001): timed out
+
```

The Hub made several attempts, and then gave up the hunt.

Now have the Hub connect to Unit C with the 3 command:

```
+3
Searching for node 03

Found node #8888:6702
Writing hub file to flash
+
```

Note that Unit C is also no longer flashing D1 as it also has transitioned to Active Mode.

Ask the hub for the new flash file contents:

```
+d
Hub file contents:
wMagicNumber:      0x9387
wFeatures:         0x007F
byRxAddress:       0x40
byChannel:         0x02
byNodeRecordCount: 0x04

Node Record 0x01:  Unit #8888:6701 at address 0x41
Node Record 0x02:  Unused
Node Record 0x03:  Unit #8888:6702 at address 0x43
Node Record 0x04:  Unused
+
```

This example shows that the serial numbers and slot numbers are independent, although a licensed user could certainly make that association if desired by editing the code accordingly.

We won’t yet associate the remaining two units in slots 2 and 4 as we have some other examples to present first.
8.4 Active Mode

In the last section, we associated two Node units with the Hub. During that exercise, the Nodes immediately transitioned to Active Mode after configuration, while the Hub stayed in Search Mode.

Force the Hub to Active Mode now with the A command:

```
+a
Listening for nodes as hub
```

Next, from Unit B, press and release SW4 and observe the terminal:

Active as node 01
Node Ping 00: 8888:6700 [FA:0B] 0.66v [FE:0F]

which indicates that the Node sent an active mode packet to the Hub, with a received RSSI:LQI at the Hub of 0xFA:0xB. Also, the Hub’s potentiometer was set to 0.66v and the acknowledge packet arrived with an RSSI:LQI of 0xFE:0x0F.

Also note that the Hub reported:

```
Hub Echo 00 from node 01 #8888:6701 [FA:0B] 0.93v
```

which indicates that the Hub responded to the packet with an acknowledge packet. Further, the RSSI:LQI received by the Hub was 0xFA:0x0B, and the Node’s potentiometer was set to 0.93v.

Also note that the Hub is displaying D1. Each packet from a node will result in a toggle of the respective LED.

Press SW4 on Unit B a few more times, and notice the sequence numbers incrementing and D1 toggling, and adjusting the potentiometer between presses. As seen at the Hub:

```
Hub Echo 01 from node 01 #8888:6701 [09:11] 0.93v
Hub Echo 02 from node 01 #8888:6701 [07:0D] 0.93v
Hub Echo 03 from node 01 #8888:6701 [07:12] 0.93v
Hub Echo 04 from node 01 #8888:6701 [08:0A] 0.93v
Hub Echo 05 from node 01 #8888:6701 [07:0D] 0.93v
Hub Echo 06 from node 01 #8888:6701 [0A:10] 0.93v
Hub Echo 07 from node 01 #8888:6701 [08:0F] 0.93v
Hub Echo 08 from node 01 #8888:6701 [08:0C] 0.93v
Hub Echo 09 from node 01 #8888:6701 [07:0F] 0.93v
Hub Echo 0A from node 01 #8888:6701 [0C:09] 0.71v
Hub Echo 0B from node 01 #8888:6701 [0C:0B] 1.31v
Hub Echo 0C from node 01 #8888:6701 [0C:12] 1.80v
Hub Echo 0D from node 01 #8888:6701 [0C:0B] 2.07v
```
Now, from the handheld Unit C, press SW4 and watch the Hub’s D3 toggle, and view the terminal output:

Hub Echo 00 from node 03 #8888:6702 [58:0E] 1.16v
Hub Echo 01 from node 03 #8888:6702 [58:14] 1.16v
Hub Echo 02 from node 03 #8888:6702 [59:13] 1.16v
Hub Echo 03 from node 03 #8888:6702 [57:10] 1.16v
Hub Echo 04 from node 03 #8888:6702 [59:1D] 1.44v
Hub Echo 05 from node 03 #8888:6702 [5A:15] 1.13v

Hub Echo 10 from node 01 #8888:6701 [09:09] 2.16v
Hub Echo 06 from node 03 #8888:6702 [24:14] 1.13v
Hub Echo 07 from node 03 #8888:6702 [23:0A] 1.13v
Hub Echo 11 from node 01 #8888:6701 [08:0A] 2.16v
Hub Echo 08 from node 03 #8888:6702 [21:0C] 1.13v
Hub Echo 12 from node 01 #8888:6701 [08:11] 2.16v
Hub Echo 09 from node 03 #8888:6702 [1F:10] 1.13v
Hub Echo 13 from node 01 #8888:6701 [09:11] 2.16v
Hub Echo 0A from node 03 #8888:6702 [21:12] 1.13v
Hub Echo 14 from node 01 #8888:6701 [08:0D] 2.16v
Hub Echo 15 from node 01 #8888:6701 [0A:08] 2.16v
Hub Echo 0B from node 03 #8888:6702 [1E:13] 1.13v
Hub Echo 0C from node 03 #8888:6702 [21:0A] 1.13v
Hub Echo 16 from node 01 #8888:6701 [08:09] 2.16v
Hub Echo 0D from node 03 #8888:6702 [1F:11] 1.13v
Hub Echo 17 from node 01 #8888:6701 [06:0C] 2.16v

If you attempt to press SW4 on both Nodes simultaneously, you may be able to observe a collision on the PC B terminal. As an example:

Node Ping 16: failed: 000C (0000): filtered out
Attempting retry for packet #16
Node Ping 16: 8888:6700 [08:09] 0.66v [0A:0F]

Note that the retry mechanism swallowed the failure, and recovered from it by resending the offending packet.
8.5 Network Operations

In this section we will manipulate the Hub and Node files to simulate field configuration issues that might arise while using a star network, as well as perform other network tasks.

8.5.1 Hub File on Reboot

Since the hub file is valid, the Hub will boot straight to active mode on power-up.

We will emulate a cold boot of the Hub by using the C-SPY debugger. Stop execution of the Hub by pressing the Break button. Now, restart the application by pressing the Reset button, followed by the Go button.

Notice now that the Hub immediately goes into Active Mode:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006
Resetting...
Hub mode started, serial #8888:6700
Loading hub file: success
Listening for nodes as hub

This would be the case of a unit in the field after having been configured by the user. Send a packet or two from Unit C:

Listening for nodes as hub
failed: 000C (0000): filtered out
Hub Echo 00 from node 03 #8888:6702 [54:0F] 1.13v
Hub Echo 01 from node 03 #8888:6702 [4C:0D] 1.13v

In this case, some noise in the environment was detected and filtered away.

8.5.2 Erasing the Hub File

It is possible to erase the Hub’s file using the erase command from the terminal. ESC to Search Mode, and take another look at the Hub’s file with the D command:

<ESC>
User cancelled operation
+d
Hub file contents:
wMagicNumber: 0x9387
wFeatures: 0x007F
byRxAddress: 0x40
byChannel: 0x02
byNodeRecordCount: 0x04

Node Record 0x01: Unit #8888:6701 at address 0x41
Node Record 0x02: Unused
Node Record 0x03: Unit #8888:6702 at address 0x43
Node Record 0x04: Unused

then use the Z command to erase the file:

+z
Erasing hub file...
Writing hub file to flash

and display the file again:

+d
Hub file contents:
wMagicNumber: 0x9387
wFeatures: 0x007F
byRxAddress: 0x40
byChannel: 0x02
byNodeRecordCount: 0x04

Node Record 0x01: Unused
Node Record 0x02: Unused
Node Record 0x03: Unused
Node Record 0x04: Unused

Note that the Hub still has a valid file, albeit empty. When the processor is booted again using Break, Reset, and Go, the Hub still goes straight to Active Mode:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

Hub mode started, serial #8888:6700

Loading hub file: success

Listening for nodes as hub
8.5.3 Orphaned Nodes

Since we erased the hub file, the two nodes previously registered with the hub have now been orphaned.

With the Hub in Active Mode, send a few packets now from Unit C:

Unregistered remote node ID 03
8888:6702 [59:14] 1.13v

Unregistered remote node ID 03
8888:6702 [58:0E] 1.13v

Unregistered remote node ID 03
8888:6702 [59:0F] 1.13v

Unregistered remote node ID 03
8888:6702 [59:14] 1.13v

... more stuff

The Hub recognizes these attempts as coming from an unregistered Node. The Node makes the attempt since it still thinks it is registered to a Hub, but the Hub does not answer. As a result, for each attempt, the Node exercises the retry algorithm and sends out several Active Mode pings.

8.5.4 Node File on Reboot

As with the Hub, a Node will boot into active mode if it contains a valid node file.

First, cycle power on Unit B, and observe its terminal:

MRF1611CC1100 Library v01.00.00
Copyright SoftBaugh 2006

Resetting...

Node mode started, serial #8888:6701

Loading node file: success

Active as node 01

Depending on the design of your PC, the serial port may supply enough power to the board to defeat an attempt at power cycling. If so, force a restart by briefly shorting the RST and - pins on the Power Options header.

Note that a node will also load a valid file if you ESC to the main menu, and then re-enter Node mode with the N command:

Active as node 01
8.5.5 Erasing a Node File

Let us now erase a Node file, which will place the erased Node back into Search Mode.

Since a node is designed to be operated in handheld mode and thus untethered to a terminal, the node file can be erased from hardware by pressing and holding SW3 for five seconds, or the *Erase Command interval*.

During this command interval D2 will flash with increasing urgency until the command interval expires, at which time the node file will be erased and Search Mode entered:

```
Active as node 01 <SW3 pressed for five seconds>
User commanded search mode
Listening for hub as node 01
```

Since the node’s file has been erased, a power cycle will bring this node back to search mode, as would an exit to the main menu and return with the *N* command.

Repeat this process for Unit C to erase its file also.

8.5.6 Associating Nodes

At this point, both Unit B and Unit C are configured as erased nodes, and should be blinking D1 in search mode. Use *SW3* to command Unit B to listen for the hub as Node 01, and command Unit C to listen for the hub as Node 02.

Now let us associate both nodes with the hub, so ESC to the hub menu:

```
Listening for nodes as hub
<ESC>
User cancelled operation
+
```

and command the hub to find both nodes with the *1* command followed by the *2* command:
Searching for node 01

Found node #8888:6701
Writing hub file to flash

Searching for node 02

Found node #8888:6702
Writing hub file to flash

Now display the hub file and review its contents:

```
Hub file contents:
wMagicNumber:    0x9387
wFeatures:       0x007F
byRxAddress:     0x40
byChannel:       0x02
byNodeRecordCount: 0x04

Node Record 0x01: Unit #8888:6701 at address 0x41
Node Record 0x02: Unit #8888:6702 at address 0x42
Node Record 0x03: Unused
Node Record 0x04: Unused
```

Transition to active mode and verify that the nodes work by pressing SW4 on each a few times:

```
Listening for nodes as hub
Hub Echo 00 from node 02 #8888:6702 [37:0C] 1.13v
Hub Echo 00 from node 01 #8888:6701 [13:0B] 2.16v
Hub Echo 01 from node 02 #8888:6702 [0A:0B] 1.13v
Hub Echo 01 from node 01 #8888:6701 [13:0A] 2.16v
Hub Echo 02 from node 02 #8888:6702 [12:10] 1.13v
Hub Echo 03 from node 02 #8888:6702 [12:10] 1.13v
Hub Echo 02 from node 01 #8888:6701 [13:14] 2.16v
Hub Echo 04 from node 02 #8888:6702 [1C:16] 1.13v
Hub Echo 05 from node 02 #8888:6702 [4A:09] 1.13v
```

### 8.5.7 Overwriting a Node Record

If a new node is added to the hub at a position already occupied by another node, the original node will be orphaned. This is a property of the demo firmware, it is left as an exercise for the reader to modify this logic if desired using the licensed source code.

Place Unit D into search mode as node 02, and associate that node with the hub:
<ESC>
User cancelled operation

+2
Searching for node 02
Writing hub file to flash

Found node #8888:6703
Writing hub file to flash

+ 

Display the hub's file and verify that the new node occupies slot 2:

+d
Hub file contents:
wMagicNumber: 0x9387
wFeatures: 0x007F
byRxAddress: 0x40
byChannel: 0x02
byNodeRecordCount: 0x04

Node Record 0x01: Unit #8888:6701 at address 0x41
Node Record 0x02: Unit #8888:6703 at address 0x42
Node Record 0x03: Unused
Node Record 0x04: Unused

+ 

At this point Unit C is orphaned, and the hub will no longer answer its messages, but will respond to Unit B and Unit D (don't forget to place the hub back into active mode):

+a
Listening for nodes as hub
Hub Echo 00 from node 02 #8888:6703 [62:11] 2.50v
Hub Echo 01 from node 02 #8888:6703 [61:09] 2.50v
Hub Echo 03 from node 01 #8888:6701 [12:11] 2.16v
Hub Echo 04 from node 01 #8888:6701 [12:0F] 2.16v
Hub Echo 02 from node 02 #8888:6703 [1B:09] 2.50v
Hub Echo 05 from node 01 #8888:6701 [0F:14] 2.16v

Ping by foreign node #8888:6702
8888:6702 [1C:10] 1.13v

... more retries deleted

Ping by foreign node #8888:6702
Hub Echo 03 from node 02 #8888:6703 [1F:0F] 2.50v
Hub Echo 06 from node 01 #8888:6701 [11:08] 2.16v
Hub Echo 04 from node 02 #8888:6703 [1D:16] 2.50v
failed: 000C (0000): filtered out

Ping by foreign node #8888:6702
8888:6702 [32:0E] 1.13v
Pinged by foreign node #8888:6702
8888:6702 [28:0C] 1.13v
... more stuff

8.5.8 Effort Indicator

As you attempt connections with Unit C, note that D1 illuminates during the effort. D1 is useful when conducting experiments with units in the field, as it gives an indicator in handheld mode of the effort required for the handheld nodes to communicate with the hub.

Unit C and Unit D are already in handheld mode, so move Unit D farther away from the hub. Note that the harder a node has to work to get through to the hub, the longer D1 stays on.

8.5.9 Three Nodes

Next, erase Unit C’s node file by pressing and holding SW3, then command it to search for node slot 04 by successive presses of SW3.

Then, bring the hub back to search mode and search for node 04:

<ESC>
User cancelled operation

+4
Searching for node 04

Found node #8888:6702
Writing hub file to flash

+ 

As before, display the flash file, and then re-enter active mode:

+d
Hub file contents:
wMagicNumber: 0x9387
wFeatures: 0x007F
byRxAddress: 0x40
byChannel: 0x02
byNodeRecordCount: 0x04

Node Record 0x01: Unit #8888:6701 at address 0x41
Node Record 0x02: Unit #8888:6703 at address 0x42
Node Record 0x03: Unused
Node Record 0x04: Unit #8888:6702 at address 0x44

+a
Listening for nodes as hub
Experiment now with all three units:

- Hub Echo 00 from node 04 #8888:6702 [5A:10] 1.13v
- Hub Echo 00 from node 02 #8888:6703 [60:0D] 2.50v
- Hub Echo 01 from node 04 #8888:6702 [3C:0B] 1.13v
- Hub Echo 01 from node 02 #8888:6703 [56:10] 2.50v
- Hub Echo 07 from node 01 #8888:6701 [12:0F] 2.16v
- Hub Echo 08 from node 01 #8888:6701 [0F:0C] 2.16v
- Hub Echo 09 from node 01 #8888:6701 [12:0D] 2.16v
- Hub Echo 02 from node 02 #8888:6703 [31:0C] 2.50v
- Hub Echo 03 from node 02 #8888:6703 [1B:09] 2.50v
- Hub Echo 04 from node 02 #8888:6703 [22:0E] 2.50v
- Hub Echo 05 from node 02 #8888:6703 [FD:0C] 2.50v
- Hub Echo 06 from node 02 #8888:6703 [0B:10] 2.50v
- Hub Echo 07 from node 02 #8888:6703 [13:14] 2.50v
- Hub Echo 08 from node 02 #8888:6703 [12:09] 2.50v
- Hub Echo 09 from node 02 #8888:6703 [02:0C] 2.50v
- Hub Echo 0A from node 02 #8888:6703 [19:0C] 2.50v
- Hub Echo 02 from node 04 #8888:6702 [14:0B] 1.13v
- Hub Echo 03 from node 04 #8888:6702 [10:0D] 1.13v
- Hub Echo 04 from node 04 #8888:6702 [12:0F] 1.13v
- Hub Echo 05 from node 04 #8888:6702 [00:08] 1.13v
- Hub Echo 06 from node 04 #8888:6702 [01:10] 1.13v
- Hub Echo 0B from node 02 #8888:6703 [13:0A] 2.50v

It is left as an exercise for the reader to add Unit E to the network as Node 03.

8.5.10 Handheld Experiments

Unit C, Unit D and Unit E are already in handheld mode.

Unit A and Unit B can also become handheld simply by ensuring both are configured with the Power Options header set to Power On and the Comm Options header set to Stand-Alone mode. There is no concern about an accidental power failure while doing this as each unit will now boot to active mode.

Finally, experiment with walking around with all units and repeating some of the disturbance experiments from previous chapters. As before, note that the harder a node has to work to get through to the hub, the longer that node’s D1 stays on.

Observe that since the hub toggles an LED on each received packet it is easy to see from a distance whether your switch presses are getting through.
9. DCO Tracking Module

The previous chapter described the Hub and Node functionality available as a star network topology in the demo firmware. This chapter provides more detail about the DCO tracking algorithms. The next chapter will describe the functionality of the InfoFlashDrive module.

The CalibrateDCO.s43 assembly language module allows use of the Digitally Controlled Oscillator instead of requiring an external high-frequency crystal on XT2. This provides several advantages for a production application:

- Reduced system cost
- Reduced assembly cost
- Reduced board size
- Fewer assembly errors to find in production testing
- Faster wakeup from sleep rather than waiting for crystal stabilization
- Reduced power consumption in sleep mode
- Fine control of oscillator options

9.1 Theory of Operation

The DCOR feature is enabled via R3, a 100k 1% 0603 resistor located on the back of the module. This resistor was deliberately chosen to be the larger 0603 rather than an 0402 to make it easier for the user to change this value, if desired.

A preliminary step of calibration is required at design time when the external R3 resistor is chosen. This design-time calibration has already been performed for the default 100k 1% resistor shipping with the MRF1611CC1100.

Once the design-time calibration is performed, a closed-loop control system implemented in this module can monitor the DCO frequency to lock it to any desired frequency within the range of the DCOR value.

After a lock has been achieved, the closed-loop tracking algorithm goes into an idle state, consuming zero cycles. Due to the nature of the DCO, changes in temperature or a long interval of time may result in the DCO frequency drifting from the setpoint. In these circumstances, the application may update the tracking.
9.2  Calibration Mode

The system is already calibrated for startup with the 100k 1% R3 external DCO resistor shipped with the unit. If this value is used, no separate calibration step is required as the closed-loop nature of the system handles any remaining error from this *ballpark* estimate.

If other values are used, licensed users of the source code should follow the instructions in this section.

9.2.1  Calibration Build

The first step is to uncomment the `#define` near the top of CalibrateDCO.s43:

```c
// Enable this define to allow calibration code to build
#define ENABLE_CALIBRATION_CODE
```

This statement allows the calibration code to be inserted into the build.

Rebuild the source project.

9.2.2  Execute the Calibration Code

Load the project in the debugger with the Debug button.

Place the cursor on the `nop` placed in the `CalibrateDCO_Initialize` function:

```c
// Set a breakpoint on the code below to examine the calibration
// and copy them to the constant calibration tables
nop
```

Run to the cursor with the Run To Cursor button:

```
// and copy them to the constant calibration tables
// Re-enable TimerX2.0 interrupts
```

*Figure 9-1 Run to Calibrate NOP*

9.2.3  Extract Calibration Values

The calibration code just populated sixteen WORD values which we will use for our tracking algorithm. These sixteen WORDs represent the measured low and high ends of each RSEL range, in terms of SMCLK ticks per ACLK = XT1/8.
These values have been written into the two conditionally compiled arrays:

```c
#ifdef ENABLE_CALIBRATION_CODE
// Table of low-end calibration measurements for the RSEL ranges
// As SMCLK ticks per ACLK (=XT1/8)
sCalibrateDCO_awLoRSEL DS16 8

// Table of high-end calibration measurements for the RSEL ranges
// As SMCLK ticks per ACLK (=XT1/8)
sCalibrateDCO_awHiRSEL DS16 8
#endif
```

These arrays are located near the top of CalibrateDCO.s43. Cut and paste them into a Watch View to obtain the addresses of these arrays:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sCalibrateDCO_awLoRSEL</td>
<td>56</td>
<td>Memory 0x1108</td>
<td>int</td>
</tr>
<tr>
<td>sCalibrateDCO_awHiRSEL</td>
<td>112</td>
<td>Memory 0x1118</td>
<td>int</td>
</tr>
</tbody>
</table>

Figure 9-2 Calibration Array Addresses

As shown, the address of the low end array is 0x1108, and the address of the high end array is 0x1118. These values of course may vary with your specific build.

Now open a Memory View, set it to view Little Endian 2x Units, and view the first address, 0x1108, as shown:

Figure 9-3 Memory View of Calibration Values

The first eight values are the eight RSEL range low end values, and the next eight are the RSEL range high end values.

### 9.2.4 Update Calibration Defines

Read these values from memory directly into the following `#defines`, found near the top of `CalibrateDCO.s43`:
After updating these calibration constants, the tracking algorithm will be able to use
derivatives of these numbers.

As background, each of these values represent the number of SMCLKs per ACLK,
where each ACLK is XT1 / 8. Since these values were collected using an XT1 of 32.768
KHz, the RSEL7 range can be seen to vary from:

\[
\text{RSEL7LO} = 412h \times 32768d / 8d = 4.268 \text{ MHz}
\]
to

\[
\text{RSEL7HI} = 8A3h \times 32768d / 8d = 9.056 \text{ MHz}
\]

When the values have been copied, stop the debugger with the Stop Debugging button.

### 9.2.5 Remove Calibration Code

The last step is to re-comment the #define near the top of CalibrateDCO.s43:

```c
// Enable this define to allow calibration code to build
//#define ENABLE_CALIBRATION_CODE
```

Now the source will no longer be burdened with the calibration code.

Rebuild the source project.
9.3 Normal Operation

The CalibrateDCO module, from the perspective of operating usage, is one of the simplest modules in the licensed source code, and has very few requirements during operation.

Refer to Section 11.1, CalibrateDCO.s43, for a detailed description of the functions exported by this module. This section provides operating tips beyond those presented in the functional description.

9.3.1 Initialization

At system startup, it is necessary to initialize the module by calling protected function CalibrateDCO_Initialize.

This function is hidden from the exported interface, and is called from within System.s43 during execution of the System_InitilizeHW, followed by the CalibrateDCO_SetFrequency function:

```c
// Initialize the DCO calibration module
// This function does not return until the calibration ...
// Call this before enabling general interrupts or any ...
call #CalibrateDCO_Initialize
mov #((SMCLK_RATE+(ACLK_RATE/2))/ACLK_RATE),R12
call #CalibrateDCO_SetFrequency
```

When System_InitializeHW completes, the DCO will have started a tracking sequence, as described in the next section.

9.3.2 Tracking Updates

Periodically, depending on the amount of environmental changes, or in the event of detected timing errors such as bit errors on a UART, etc., it is necessary to update the DCO’s estimate of the desired output frequency on SMCLK. This closed-loop process is referred to here as a tracking update.

A tracking update is initiated by calling the CalibrateDCO_SetFrequency function with the parameter, known as the setpoint, set to the number of desired SMCLKs per ACLK, where ACLK is set to XT1 / 8.

Internally, CalibrateDCO_SetFrequency operates as shown in the following psuedocode:

```c
void CalibrateDCO_SetFrequency( WORD wCountsACLK )
{
    Store setpoint
    Find RSEL range
    Interpolate within that range
    Set interpolated DCOx:MODx
}
```
The `CalibrateDCO_UpdateTracking` function initiates a series of TACCR2 interrupt-based calculations performed at each ACLK tick until the tracking loop locks onto the desired setpoint. Once a lock is attained, the tracking interrupt disables itself, reducing idle cycles to zero.

During normal operation, only a few tracking loops will need to be performed, each of which uses about six ACLK ticks, with about 80 processor cycles per ticks, on average, depending on how large the tracking error is determined to be and whether it is necessary to jump RSEL ranges.

If it is desired to track to the setpoint previously supplied the code can instead directly call `CalibrateDCO_UpdateTracking`. This approach is useful when a portion of code needs to initiate a tracking update, but it is undesirable to distribute the setpoint among various modules in the project.

Should the processor need to pause the tracking algorithm to save cycles for a time-critical process, or to enter a period of disabled SMCLK, the application should call `CalibrateDCO_StopTracking`. This function stops the tracking algorithm, leaving the DCO in a consistent state near the setpoint.

After the time-critical process, or after a prolonged sleep, the application simply calls `CalibrateDCO_UpdateTracking` to restart the tracking process and attain a new lock.

### 9.3.3 Resource Usage

The `CalibrateDCO` module uses the following resources:

- TimerA3 CCR2 interrupt, but only while actively tracking.
- Approximately 2000 total cycles, spread over about 10 milliseconds, or 6% at 6 MHz.
- Zero processor cycles when the tracking algorithm is idle.
- 88 bytes of constant space, and 482 bytes of code, for a total of 570 bytes of flash.
- 8 bytes of RAM.
- 12 bytes of stack (maximum), including the ISR overhead.

As an interrupt-based process, the module does NOT use the hardware multiplier, as to do so would introduce consistency risks with the main application.
9.4 Conversion of Demo to XT2

Some users that require an external high-frequency crystal may choose to convert the demo software to use that frequency source instead of the calibrated DCO. To do so, there are two basic approaches.

The first approach is to only switch SMCLK to XT2, leaving MCLK running from the DCO in an uncalibrated fashion. The second approach would be to switch both SMCLK and MCLK to XT2. The following sections discuss both of these approaches.

While reading through the following sections, keep in mind that the only portion of the demo software which requires high-frequency precision is the 115.2 kbaud UART, and that only +/- 2% or so.

A recommend, although not required, step in this conversion is to switch the system tick to ACLK or a divisor thereof. In the calibrated DCO system, a fraction of ACLK is used to generate tracking ticks. Without a calibrated DCO, there is no longer a need to use ACLK for this purpose, freeing it to drive the system ticks directly. The following steps assume this conversion.

9.4.1 XT2 to SMCLK, DCO to MCLK

Switching the SMCLK to XT2 and leaving MCLK driven by the DCO provides the advantage of fast starts for interrupts for the MCLK, while getting the precision of a high-frequency crystal for SMCLK.

A disadvantage of switching SMCLK to XT2 is the loss of fast startup of SMCLK upon leaving some low power modes. A further disadvantage of this approach is the loss of the ability to analyze MCLK bandwidth, as there is introduced a certain amount of uncertainty for using an uncalibrated version of the DCO. Yet another disadvantage is the dependency of the system to crystal failure.

Remove References to CalibrateDCO

Remove the header declaration in System.s43:

```
#include "CalibrateDCO.h"
```

Remove the following lines from System_InitializeHW:

```
call #CalibrateDCO_Initialize
mov #((SMCLK_RATE+(ACLK_RATE/2))/ACLK_RATE),R12
call #CalibrateDCO_SetFrequency
```

Remove the ISR branch for the TACCR2:

```
System_TimerA_TACCR2_ISR
```
Remove the CalibrateDCO.s43 module from the workspace.

**Change the SMCLK source to XT2 and MCLK source to the DCO**

Replace the following line from `System_InititalizeHW`:

```c
// call #SetupSMCLK_DCOR
call #SetupSMCLK_XT2
call #SetupMCLK_DCO
```

Add the `SetupSMCLK_XT2` function:

```c
PUBLIC SetupSMCLK_XT2
SetupSMCLK_XT2
// Switch SMCLK to XT2
// Switch SMCLK to XT2, without affecting other bits
bis.b #SELS,&BCSCTL2
ret
```

**Hardcode the DCO Frequency in SetupMCLK_DCO**

If using the internal DCO resistor add:

```c
PUBLIC SetupMCLK_DCO
SetupMCLK_DCO
// Set DCO bits to maximum
mov.b #(DCO2|DCO1|DCO0),&DCOCTL
// Set RSEL bits to maximum, all others zero
mov.b #(RSEL2|RSEL1|RSEL0),&BCSCTL1
// Set MCLK from DCO with no divisor, and internal resistor
bic.b #(SELM1|SELM0|DIVM1|DIVM0|DCOR),&BCSCTL1
ret
```

If using the 100k 1% REXT for DCO add instead:

```c
PUBLIC SetupMCLK_DCO
SetupMCLK_DCO
// Set DCO bits to near the maximum, about 6 MHz
mov.b #(DCO2|DCO1),&DCOCTL
// Set RSEL bits to midrange, all others zero
mov.b #(RSEL2),&BCSCTL1
// Set MCLK from DCO with no divisor
bic.b #(SELM1|SELM0|DIVM1|DIVM0),&BCSCTL1
// and external resistor
bis.b #(DCOR),&BCSCTL1
ret
```
Change the System Tick

Change the clocking defines in *System.s43*:

```c
// define SMCLK_RATE 6000000
#define ACLK_RATE 32768
#define SYSTEM_TICK_DIVISOR (ACLK_RATE/SYSTEM_TICK_RATE)
```

Change the TACCR1 Source

In *SetupTimerA3*, change the TimerA3 clock source as follows:

```asm
mov #((TASSEL_1 | ID_0 | MC_2 | TACLR), &TACTL
```

9.4.2 XT2 to SMCLK and MCLK

Switching both SMCLK and MCLK to XT2 would give higher bandwidth for MCLK versus the internal DCO, or, if running from an external DCO resistor, the ability to have a higher bandwidth for MCLK with no risk of outrunning the MSP430 specification for MCLK due to temperature or other variations.

A disadvantage of switching SMCLK to XT2 is the loss of fast startup of SMCLK upon leaving some low power modes. A further disadvantage of this approach is a corresponding loss of fast startup of MCLK as well. Yet another disadvantage is the dependency of the system to crystal failure.

Remove References to CalibrateDCO

Remove the header declaration in *System.s43*:

```c
#include "CalibrateDCO.h"
```

Remove the following lines from *System_InitializeHW*:

```asm
call #CalibrateDCO_Initialize
mov #((SMCLK_RATE+(ACLK_RATE/2))/ACLK_RATE), R12
```
```asm
call #CalibrateDCO_SetFrequency
```

Remove the ISR branch for the TACCR2:

```asm
System_TimerA_TACCR2_ISR
reti
```

Remove the CalibrateDCO.s43 module from the workspace.
Change the SMCLK and MCLK sources to XT2

Uncomment the following line from `System_InitializeHW`:

```c
    call #SetupMCLK_XT2
```

and comment the following line out:

```c
    // call #SetupSMCLK_DCOR
```

Change the System Tick

Change the clocking defines in `System.s43`:

```c
    //#define SMCLK_RATE 6000000
    #define ACLK_RATE 32768
    #define SYSTEM_TICK_DIVISOR ((ACLK_RATE/SYSTEM_TICK_RATE))
```

Change the TACCR1 Source

In `SetupTimerA3`, change the TimerA3 clock source as follows:

```c
    mov #(TASSEL_1|ID_0|MC_2|TACLR),&TACTL
```
10. Info Flash Drive

The previous chapter provided more detail about the DCO tracking algorithms. This chapter describes the functionality of the InfoFlashDrive module. The next chapter provides a reference for the modules, functions, and structures contained within the licensed modules.

10.1 Theory of Operation

The info flash drive is implemented by the licensed module InfoFlashDrive.s43, and is available in source form to licensed users. The functions exported by this module are presented in Section 11.7, InfoFlashDrive.s43. This chapter discusses the background behind this functionality.

10.1.1 Flash File Criteria

It is assumed that the files in use by a typical MSP430 application are rarely dynamically sized, and consist mainly of configuration information that also changes rarely. And, for those applications that need dynamic storage of information, an external serial flash is probably more appropriate, as the MSP430 is likely to contain insufficient room for such applications in the first place.

In addition, some configuration information must survive power cycles with some applications. Applications that can count on uninterrupted power do not need these techniques. But, many applications that are designed with backup power in mind often find that the ingenuity of field staff can defeat even the most determined backup plans! As a result, when possible we choose to design systems without backup power in mind.

The Hub and Node files used in the demo software fit the above criteria. As such, the information presented in this chapter is useful for understanding the techniques for storing configuration information on the MSP430 with a minimum of risk to related data.

The flash drive presented here consumes 2 bytes of RAM, and 726 bytes of code and constants required for operation, plus of course the 256 byte of information flash. Further, it is fail-safe, meaning that should power fail during operation of a file write, the system will have at least the previous file intact.

10.1.2 MSP430 Flash Primer

Flash on the MSP430, while capable of being written as individual bytes, must be erased in complete segments. And, as shown in SLAS368, each flash segment has a life-span of at least 10,000 cycles, and up to 100,000 cycles typically. For most applications, this is plenty.

Per the User's Guide, it is also necessary to not execute instructions from flash while performing flash writes or erases. While it is conceivable to execute instructions from
RAM during these times, the application framework presented here does not support such operation. This decision was made for the practical reason that to do so would so severely cripple the application and interrupt-driven services that it would be best to simply take the performance hit by disabling interrupts and blocking execution until the flash operation has completed.

Finally, as shown in SLAS368, flash memory on the MSP430F1611 must be programmed at a VCC from between 2.7v and 3.6v. Applications using this technique must be guaranteed of having this voltage available during flash operations.

### 10.1.3 MSP430 Info Flash and File Slots

Information flash on the MSP430F1611 is organized into two 128-byte segments, as shown below:

![Figure 10-1 MSP430 Information Flash](image)

We arbitrarily divide each segment further into two 64-byte slots, as shown below:

![Figure 10-2 Info Flash Slots](image)

Each of these slots can contain a file, and are roughly analogous to sectors in a hard drive, with the exception that a file in our model must be contained completely within a single slot.

As mentioned before, the choice of slot size is purely arbitrary. Ideally, the slot size should be a power of two to fit an integer number of them neatly within a flash segment, and must be at least 8 bytes.

In addition, the slot size should be as small as practical to fit the files implemented by the application in order to minimize the amount of wear on the flash. As will be seen shortly, each time a file is written, a new slot is consumed. And, when a segment has been
exhausted, that segment is erased. So, the more slots that fit into a segment, the fewer times a segment will be erased, contributing to increased life of the flash.

We mentioned earlier that the absolute minimum number of cycles guaranteed by Texas Instruments is 10,000, but 100,000 typical. Using the techniques in this chapter can result in a lifetime multiplier of $S$ from what the hardware is capable of by itself, where $S$ is the number of slots designed into the system, where $S = 4$ in the demo application. So, with our demo application, we can count on at least 40,000 file writes, with 400,000 being more typical.

It is also for this reason that we have not taken a traditional file system approach to using linked lists of sectors, as this could contribute to excessive flash wear on the segments containing file tables. And, although the use of leveling algorithms combined with dynamic table reassignment could help mitigate this wear, the algorithms themselves can begin to consume more code than the value they provide to the application over the approach presented here.

### 10.1.4 File Slot Organization

Each file slot is organized internally as follows:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x002</td>
<td>wMagicNumber</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x003C</td>
<td>awFileData[]</td>
</tr>
<tr>
<td>0x003E</td>
<td>0x002</td>
<td>wCRC</td>
</tr>
</tbody>
</table>

**WORD wMagicNumber**: This value is a unique number which provides the flash drive algorithms the ability to determine whether the slot contains a valid file. Each such slot containing this number is a candidate for acceptance as a file, with precedence given to the first such slot found.

**WORD awFileData[]**: This is an array of the file payload data, and is fixed in size as the slot size - four bytes, although the client application may choose to store less data than this, but never more. The flash file system ensures that if a file is delivered that it is valid as stored. The validity and usefulness of the contents, however, is the responsibility of the client application.

**WORD wCRC**: This CRC value is used by the file system to ensure the integrity of the file's data, and is calculated over all previous WORDs in the slot, $wMagicNumber$ included. The first slot containing both a valid $wMagicNumber$ and $wCRC$ is declared to be the active file.

In the next section, we see how *Hub* files serve as examples of using the info flash drive.
10.2 Hub File Example

The file used by the hub is defined by a couple of dedicated structures, and has the following organization:

```c
typedef struct StructHubFile {
    // This magic number denotes a hub file in flash
    WORD wMagicNumber;

    // Denotes the features in use
    // Note: In this version these features are currently unused
    WORD wFeatures;

    // The Rx address is the active mode address of this host for ...
    // All search mode messages are sent to DEFAULT_RX_ADDRESS
    BYTE byRxAddress;

    // The channel is the active mode channel.
    // All search mode messages are on channel 0.
    BYTE byChannel;

    // The node record count is the number of node records to follow,
    // NOT the count of active nodes.
    BYTE byNodeRecordCount;

    // Dummy for WORD alignment
    BYTE byDummy;

    // Array of node data
    SNodeRecord atNodeRecords[MAX_NODE_COUNT];
} SHubFile;
```

where `SNodeRecord` is defined as:

```c
typedef struct StructNodeRecord {
    // The node ID is assigned by the host
    BYTE byNodeID;

    // Flags for this node
    // Bit0 determines whether the node record is valid
    BYTE byFlags;

    // The Tx address is the address of the node for outgoing ...
    BYTE byTxAddress;

    // Dummy BYTE for WORD alignment.
    BYTE byDummy;

    // The serial number is determined by the node's firmware
} SNodeRecord;
```
DWORD dwSerialNumber;
}

SNodeRecord;

Each SNodeRecord is eight bytes long, and the SHubFile contains four node records plus an additional eight bytes of hub information, for a total file size of 40 bytes. The node file is significantly smaller, so the hub file is the largest file in our demo application. As such, the slot size was then chosen to be 64 bytes, which is the smallest power of two equal to or larger than 40+4 (magic number + CRC).

Note that the first member of the SHubFile is another wMagicNumber. This number is distinct from the one preceding in the slot file description, and allows the application to determine whether a hub file is stored or some other file instead (such as a node file). From this perspective, this value can be thought of as the file’s name.

A hub’s actual file data is contained in a RAM image, defined in HubMode.c:

static SHubFile m_tHubFile;

This persistent RAM image is used as an edit buffer for the file contents, and provides a means for communicating back and forth between the flash drive. An alert reader may wish to use the heap or the stack for a temporary working buffer, but we prefer to use dedicated RAM for this purpose as we are not short of RAM in this demo and dedicated RAM leads to simpler and presumably less buggy code!

When the file is to be loaded from the flash drive, the following code can be used:

WORD wResult;

// Load the file and see whether it is valid
wResult = InfoFlashDrive_ReadFile( (LPBYTE)&m_tHubFile,
                                   sizeof(m_tHubFile) );
if( INFO_FLASH_DRIVE_SUCCESS != wResult )
{
    HubMode_PrintFileError( wResult );
    return FALSE;
}

Once the file has been read, it can be validated as a hub file:

// A file was loaded, see if it is a hub file
if( HUB_FILE_MAGIC_NUMBER != m_tHubFile.wMagicNumber )
{
    RxTxHost_QueueTxString( "wrong type, erasing\n" );
    InfoFlashDrive_EraseFile();
    return FALSE;
}

In this example, an invalid file is explicitly erased when discovered. While not absolutely necessary, this erasure returns the flash drive to a known, initialized condition.
The file contents can be edited via the buffer:

```c
m_tHubFile.wMagicNumber = HUB_FILE_MAGIC_NUMBER;
m_tHubFile.wFeatures = m_wFeatures;
... Other edits
m_tHubFile.byDummy = 0xFF;
```

and finally written back to the drive:

```c
RxTxHost_QueueTxString( "Writing hub file to flash\n" );
wResult = InfoFlashDrive_WriteFile( (LPBYTE)&m_tHubFile, sizeof(m_tHubFile) );
if( INFO_FLASH_DRIVE_SUCCESS != wResult )
{   HubMode_PrintFileError( wResult );
    return FALSE;
}
```

In case the file had not been explicitly read from the drive from startup, it is a safe practice to always initialize the magic number when editing the file:

```c
m_tHubFile.wMagicNumber = HUB_FILE_MAGIC_NUMBER;
```

And, as before, if it is desired to erase a file, the following can be used:

```c
InfoFlashDrive_EraseFile();
```

`NodeMode.c` uses the erase function when the user commands an erase by pressing and holding SW3. The absence of a valid node file is a signal to the firmware that the application should proceed directly to search mode, while a valid node file sends the node directly into active mode.
10.3 Relocating To Main Flash

The information flash segments were chosen for this demo app purely arbitrarily. The discussions above could apply equally well with placement of the flash drive into two or more main segments. Where appropriate, we will discuss the changes required for a licensed user to relocate the flash drive into main memory.

10.3.1 Main Flash Segments

Main flash in the MSP430F1611 consists of 512-byte segments, organized as follows:

<table>
<thead>
<tr>
<th>Segment ID</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment 0</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>Segment 1</td>
<td>0xFE00</td>
</tr>
<tr>
<td></td>
<td>0xFDFF</td>
</tr>
<tr>
<td>Segment 2</td>
<td>0xFC00</td>
</tr>
<tr>
<td></td>
<td>0xFBFF</td>
</tr>
<tr>
<td></td>
<td>0xFA00</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Segment 93</td>
<td>0x45FF</td>
</tr>
<tr>
<td>Segment 94</td>
<td>0x4400</td>
</tr>
<tr>
<td></td>
<td>0x43FF</td>
</tr>
<tr>
<td>Segment 95</td>
<td>0x4200</td>
</tr>
<tr>
<td></td>
<td>0x41FF</td>
</tr>
<tr>
<td></td>
<td>0x4000</td>
</tr>
<tr>
<td></td>
<td>0x3FFF</td>
</tr>
</tbody>
</table>

**Figure 10-3 Main Flash Organization**

The flash drive algorithms exploit the use of at least two segments to allow the safe erasure of an upcoming slot, while not disturbing the contents of a previous file. This approach is not only a key component of fail-safe operation, but also leads to smaller code by avoiding buffering the previous file contents during an erasure. It is presumed here that at least two segments continue to be used in main flash, although more would be fine.

Dividing each segment into two slots would allow file sizes of up to 252+4 bytes, or alternatively, sixteen 60+4 byte files for two main segments. In the latter case, the flash would last for 160,000 minimum, 1,600,000 typical file writes!
In this example, we take the middle road and convert the info flash drive into a main flash drive having four 124+4 byte slots per segment, and consume Segments 94 and 95, for a total of eight file slots. Note that while we have used two contiguous segments for simplicity, it is not necessary that the segments used be contiguous.

10.3.2 Updating the Linker Command File

The first step in converting the flash drive over to main flash is to remove the target segments from use for other code or data storage. This is accomplished by modifying the DemoMRF1611CC1100.xcl file as follows:

```plaintext
//  FLASHFILE   4000-43FF       Main flash file
//  CSTART      4400-FFD7       cstartup program code
//  CODE        4400-FFD7       Program code
//  DATA16_C    4400-FFD7       Constant "const" variables AND ...
//  DATA16_ID    4400-FFD7      Initializers for DATA16_I
//  DIFUNCT     4400-FFD7       Dynamic initialization vector used ...
//  CHECKSUM   4400-FFD7       The linker places the checksum ...

and

// Code

- Z(CODE) CSTART=4400-FFD7
- Z(CODE) CODE=4400-FFD7

// Constant data

- Z(CONST) DATA16_C, DATA16_ID, DIFUNCT, CHECKSUM=4400-FFD7

// Main flash file
- Z(CONST) FLASHFILE=4000-43FF

While the FLASHFILE segment is never explicitly referenced in the code, the above reserves memory for this purpose, and documents its usage.

10.3.3 Rename Files

Delete InfoFlashFile.s43 from the workspace.

Then, rename InfoFlashFile.s43, InfoFlashFile.h, and InfoFlashFile_Results.h to MainFlashFile.s43, MainFlashFile.h, and MainFlashFile_Results.h respectively.

Add MainFlashFile.s43 to the workspace, and update .h references in the other project files accordingly.
10.3.4 Rename Declarations and References

In all modules and files, rename all declarations and references to `InfoFlashDrive_` to `MainFlashDrive_`, and all declarations and references to `INFO_FLASH_DRIVE_` to `MAIN_FLASH_DRIVE_`.

10.3.5 Change the File Size

In the renamed `MainFlashDrive_Results.h`, change:

```
#define MAIN_FLASH_DRIVE_MAX_FILE_SIZE 124
```

10.3.6 Update the Documentation

At the top of `MainFlashDrive.s43`, change the documentation:

```
// In this implementation, for a slot size of 128 bytes, the two main...
// segments are divided into a total of eight 128-byte file slots.
// Each slot contains 64 WORDs, as follows:
// 0: MAGIC_VALUE
// 1: FILE WORD 0
// 2: FILE WORD 1
// 3-61: ...
// 62: FILE WORD 61
// 63: 16-bit CRC over MAGIC_VALUE to FILE WORD 61
```

10.3.7 Update the Flash Drive Parameters

Many flash drive parameters are defined near the top of `MainFlashDrive.s43`. Change these accordingly:

```
#define MAIN_FLASH_DRIVE_MAGIC_VALUE 0x5944
#define MAIN_FLASH_DRIVE_SLOT_SIZE 128
#define MAIN_FLASH_SLOT_COUNT 8
#define MAIN_94_ERASE_ADDRESS 0x4200
#define MAIN_95_ERASE_ADDRESS 0x4000
#define INVALID_SLOT_ENTRY_ADDRESS 0x0200
#define MAIN_BASE_ADDRESS 0x4000
#define MAIN_BEYOND_ADDRESS 0x4400
```

and rename the references throughout `MainFlashDrive.s43` accordingly. Take care to make sure that references to `INFO_A_` are changed to `MAIN_94_`, and `INFO_B_` to `MAIN_95_`, respectively.
10.3.8 Update the Slot Table

The flash drive is a table-based application, and makes use of the table defined near the top of `MainFlashDrive.s43`.

The table, which defines a circular linked list of file slots and actions to be taken for each such slot, uses the following entries, defined below:

**Table 10-2 Slot Table Entry Organization**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x0002</td>
<td>pawSlotAddress</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x0002</td>
<td>pfnEraseAction</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x0002</td>
<td>pfnForeignEraseAction</td>
</tr>
<tr>
<td>0x0006</td>
<td>0x0001</td>
<td>byNextSlotIndex</td>
</tr>
<tr>
<td>0x0007</td>
<td>0x0001</td>
<td>byNextForeignSlotIndex</td>
</tr>
</tbody>
</table>

These entries are defined as follows:

LPWORD pawSlotAddress: Address of the first WORD in the slot.

LPFN pfnEraseAction: Pointer to an erase handler to be executed after this slot becomes valid.

LPFN pfnForeignEraseAction: Pointer to an erase handler to be executed if a foreign segment needs to be erased.

BYTE byNextSlotIndex: Index of the slot entry which follows this slot.

BYTE byNextForeignIndex: Index of the first slot entry in the next foreign segment.

A *foreign segment* is defined as any flash drive segment other than that to which the current slot belongs.

Change and add to this table, as shown below:

```assembly
MainFlashDrive_SlotTable
  // Slot 0
MainFlashDrive_SlotTable0
  DW 0x4000          // Address of this slot
  DW MainFlashDrive_EraseMain94  // Erase action to be taken ...
  DW MainFlashDrive_EraseMain94  // Erase action to be taken ...
  DB 0x01           // Index of next slot
  DB 0x04           // Index of next slot after ...
  // Slot 1
MainFlashDrive_SlotTable1
  DW 0x4080          // Address of this slot
```
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain94 // Erase action to be taken ...
DB 0x02 // Index of next slot
DB 0x04 // Index of next slot after ...
// Slot 2
MainFlashDrive_SlotTable2
DW 0x4100 // Address of this slot
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain94 // Erase action to be taken ...
DB 0x03 // Index of next slot
DB 0x04 // Index of next slot after ...
// Slot 3
MainFlashDrive_SlotTable3
DW 0x4180 // Address of this slot
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain94 // Erase action to be taken ...
DB 0x04 // Index of next slot
DB 0x04 // Index of next slot after ...
// Slot 4
MainFlashDrive_SlotTable4
DW 0x4200 // Address of this slot
DW MainFlashDrive_EraseMain95 // Erase action to be taken ...
DW MainFlashDrive_EraseMain95 // Erase action to be taken ...
DB 0x05 // Index of next slot
DB 0x00 // Index of next slot after ...
// Slot 5
MainFlashDrive_SlotTable5
DW 0x4280 // Address of this slot
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain95 // Erase action to be taken ...
DB 0x06 // Index of next slot
DB 0x00 // Index of next slot after ...
// Slot 6
MainFlashDrive_SlotTable6
DW 0x4300 // Address of this slot
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain95 // Erase action to be taken ...
DB 0x07 // Index of next slot
DB 0x00 // Index of next slot after ...
// Slot 7
MainFlashDrive_SlotTable7
DW 0x4380 // Address of this slot
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain95 // Erase action to be taken ...
DB 0x00 // Index of next slot
DB 0x00 // Index of next slot after ...
// Slot 8 (INVALID, but safe)
MainFlashDrive_SlotTable_Invalid
DW INVALID_SLOT_ENTRY_ADDRESS // Address of this slot
DW MainFlashDrive_EraseNone // Erase action to be taken ...
DW MainFlashDrive_EraseMain95 // Erase action to be taken ...
DB 0x00 // Index of next slot
DB 0x00 // Index of next slot after ...
10.3.9 Check the Erase Handlers

Make sure the erase handlers are now written as:

```
MainFlashDrive_EraseMain94
// Erase the Main94 sector
    mov #MAIN_94_ERASE_ADDRESS,R12
    jmp MainFlashDrive_EraseMainSector

MainFlashDrive_EraseMain95
// Erase the Main95 sector
    mov #MAIN_95_ERASE_ADDRESS,R12
    jmp MainFlashDrive_EraseMainSector
```

Also change all references to `InfoFlashDrive_EraseInfoA` to `MainFlashDrive_EraseMain94` and references to `InfoFlashDrive_EraseInfoB` to `MainFlashDrive_EraseMain95`, respectively.

And, update the `MainFlashDrive_EraseMainSector` function name:

```
MainFlashDrive_EraseMainSector
// Called with sector address to be erased in R12
// Assumes SMCLK at about 6 MHz
```

10.3.10 Build and Test

At this point, the main flash drive should be completed. Perform a build, and clean up any errors that may have lingered. To test, execute the hub application, and follow along in the debugger.

Be sure to erase files many times and make sure that after eight tries the current slot rotates back to slot 0, as expected. Also, to simulate a power failure, kill the debugger during the execution of a file write. Note how the algorithms find the previously valid file, as well as recovering from the corruption of a partially written new file.
11. Licensed Module Reference

The previous chapter described the functionality of the InfoFlashDrive module. This final chapter provides a reference to the functions and structures provided by each licensed module.

11.1 CalibrateDCO.s43

This assembly language module allows use of the Digitally Controlled Oscillator instead of requiring an external high-frequency crystal on XT2 through the use of either the internal DCO resistor or an external resistor provided at R3. A later chapter, DCO Tracking Module, details the use of design-time calibration with other resistor values. That chapter also explains the hardware resources consumed by the tracking algorithm.

The high level of integration in this module means that all the functionality is condensed into only three functions, listed in this section and exported in CalibrateDCO.h. No structures are exported by this module.

11.1.1 Function CalibrateDCO_SetFrequency

This function establishes the frequency setpoint to be used by the closed loop tracker, and starts a tracking sequence.

Prototype

void CalibrateDCO_SetFrequency( WORD wCountsACLK );

Parameters

WORD wCountsACLK: Setpoint for the desired frequency, expressed in terms of desired DCO cycles per ACLK.

Return Value

None

Remarks

This module is implemented with the assumption that the system ACLK is set to XT1/8, where XT1 is equal to 32.768 kHz.
11.1.2 Function CalibrateDCO_UpdateTracking

Updates a frequency tracking sequence using the current setpoint.

Prototype

void CalibrateDCO_UpdateTracking( void );

Parameters

None

Return Value

None

Remarks

Use this function to update the frequency tracking, perhaps after a long period of sleep or a change in system operation, or detection of serial port bit errors.

11.1.3 Function CalibrateDCO_StopTracking

Stop a frequency tracking sequence in progress, if any.

Prototype

void CalibrateDCO_StopTracking( void );

Parameters

None

Return Value

None

Remarks

Use of this function is helpful when the cycles consumed by the tracking algorithm may be better used for a higher-priority short-term task, or to enter a long-term sleep.
11.2 CC1100.s43

This assembly language module provides the low-level hardware driver features to access the CC1100 IC over SPI on UART1. No structures are exported, the register and bit definitions are defined in \textit{CC1100\_Registers.h}, and the following functions are exported in \textit{CC1100.h}.

11.2.1 Function \texttt{CC1100\_GlobalResetSequence}

This function performs a global reset to initialize the CC1100 to a known reset state. It is called internally by \textit{CC1100\_InitializeApp}, which should in turn be called from main.c as shown in the demo.

 Prototype

\begin{verbatim}
void CC1100\_GlobalResetSequence( void );
\end{verbatim}

 Parameters

None

 Return Value

None

 Remarks

This function is listed here for completeness only. After the \textit{CC1100\_InitializeApp} call, it is not normally called by user application code.

11.2.2 Function \texttt{CC1100\_SetConfigurationRegister}

This function sets a CC11000 configuration register with the value indicated.

 Prototype

\begin{verbatim}
BYTE CC1100\_SetConfigurationRegister( BYTE byAddress, BYTE byValue );
\end{verbatim}

 Parameters

BYTE byAddress: Configuration register to be set.
BYTE byValue: Value to be written to byAddress.

Return Value
BYTE representing the CC1100 status BYTE.

Remarks
See CC1100_Registers.h for a list of the configuration register mnemonics and bit values to be used with this function.

11.2.3 Function CC1100_GetConfigurationRegister
Reads the indicated CC1000 configuration register.

Prototype
BYTE CC1100_GetConfigurationRegister( BYTE byAddress );

Parameters
BYTE byAddress: Configuration register to be read.

Return Value
BYTE representing the indicated CC1100 configuration register value.

Remarks
See CC1100_Registers.h for a list of the configuration register mnemonics and bit values to be used with this function.

11.2.4 Function CC1100_GetStatusRegister
Reads the indicated CC1000 status register.

Prototype
BYTE CC1100_GetStatusRegister( BYTE byAddress );
Parameters

BYTE byAddress: Status register to be read.

Return Value

BYTE representing the indicated CC1100 status register value.

Remarks

See CC1100_Registers.h for a list of the status register mnemonics and bit values to be used with this function. For clarity, use the status query functions CC1100_QueryXXX instead of this function.

11.2.5 Function CC1100_WriteBurstRegisters

Writes a sequence of registers with the indicated values.

Prototype

void CC1100_WriteBurstRegisters( BYTE byAddress,
                                   LPBYTE abyValues,
                                   BYTE byCount );

Parameters

BYTE byAddress: Starting address of burst registers to be written.

LPBYTE abyValues: Address of a BYTE buffer containing the register values to be written.

BYTE byCount: Count of values to be written to the burst register from abyValues.

Return Value

None

Remarks

User application code will not typically use this function. It is used internally by the DigitalRadio.s43 module to handle transmission bursts.
11.2.6 Function CC1100_ReadBurstRegisters

Reads a sequence of register values into a user-supplied BYTE array.

Prototype

```c
void CC1100_ReadBurstRegisters( BYTE byAddress,
                                 LPBYTE abyValues,
                                 BYTE byCount );
```

Parameters

- BYTE byAddress: Starting address of burst registers to be read.
- LPBYTE abyValues: Address of a BYTE buffer into which the register values are read.
- BYTE byCount: Count of values to be written to `abyValues` from the burst registers.

Return Value

None

Remarks

User application code will not typically use this function. It is used internally by the `DigitalRadio.s43` module to handle reception bursts.

11.2.7 Function CC1100_ReadRxFIFO

Reads a single value from the receive FIFO.

Prototype

```c
BYTE CC1100_ReadRxFIFO( void );
```

Parameters

None

Return Value

BYTE representing the next value from the receive FIFO.
Remarks

Use application code will not typically use this function. It is used internally by the DigitalRadio.s43 module to handle receive operations.

11.2.8 Function CC1100_WriteTxFIFO

Writes a single value to the transmit FIFO.

Prototype

BYTE CC1100_WriteTxFIFO( BYTE byValue );

Parameters

BYTE byValue: Value to be written to the transmit FIFO.

Return Value

BYTE representing the chip status value.

Remarks

Use application code will not typically use this function. It is used internally by the DigitalRadio.s43 module to handle transmit operations.

11.2.9 Function CC1100_StrobeRES

Wrapper function to wait out the chip ready signal and execute the SRES command strobe.

Prototype

BYTE CC1100_StrobeRES( void );

Parameters

None

Return Value

BYTE representing the CC1100 status BYTE.
Remarks
Use instead of synthesizing with other calls.

11.2.10 Function CC1100_StrobeFSTXON

Wrapper function to wait out the chip ready signal and execute the SFSTXON command strobe.

Prototype
BYTE CC1100_StrobeFSTXON( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.

Remarks
Use instead of synthesizing with other calls.

11.2.11 Function CC1100_StrobeXOFF

Wrapper function to wait out the chip ready signal and execute the SXOFF command strobe.

Prototype
BYTE CC1100_StrobeXOFF( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.
Remarks

Use instead of synthesizing with other calls.

11.2.12 Function CC1100_StrobeCAL

Wrapper function to wait out the chip ready signal and execute the SCAL command strobe.

Prototype

BYTE CC1100_StrobeCAL( void );

Parameters

None

Return Value

BYTE representing the CC1100 status BYTE.

Remarks

Use instead of synthesizing with other calls.

11.2.13 Function CC1100_StrobeRX

Wrapper function to wait out the chip ready signal and execute the SRX command strobe.

Prototype

BYTE CC1100_StrobeRX( void );

Parameters

None

Return Value

BYTE representing the CC1100 status BYTE.
Remarks
Use instead of synthesizing with other calls.

11.2.14  Function CC1100_StrobeTX
Wrapper function to wait out the chip ready signal and execute the STX command strobe.

Prototype
BYTE CC1100_StrobeTX( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.

Remarks
Use instead of synthesizing with other calls.

11.2.15  Function CC1100_StrobeIDLE
Wrapper function to wait out the chip ready signal and execute the SIDLE command strobe.

Prototype
BYTE CC1100_StrobeIDLE( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.
Remarks
Use instead of synthesizing with other calls.

11.2.16  Function CC1100_StrobeWOR
Wrapper function to wait out the chip ready signal and execute the SWOR command strobe.

Prototype
BYTE CC1100_StrobeWOR( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.

Remarks
Use instead of synthesizing with other calls.

11.2.17  Function CC1100_StrobePWD
Wrapper function to wait out the chip ready signal and execute the SPWD command strobe.

Prototype
BYTE CC1100_StrobePWD( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.
Remarks
Use instead of synthesizing with other calls.

11.2.18 Function CC1100_StrobeFRX
Wrapper function to wait out the chip ready signal and execute the SFRX command strobe.

Prototype
BYTE CC1100_StrobeFRX( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.

Remarks
Use instead of synthesizing with other calls.

11.2.19 Function CC1100_StrobeFTX
Wrapper function to wait out the chip ready signal and execute the SFTX command strobe.

Prototype
BYTE CC1100_StrobeFTX( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.
Remarks
Use instead of synthesizing with other calls.

11.2.20 Function CC1100_StrobeWORRST
Wrapper function to wait out the chip ready signal and execute the SWORRST command strobe.

Prototype
BYTE CC1100_StrobeWORRST( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.

Remarks
Use instead of synthesizing with other calls.

11.2.21 Function CC1100_StrobeNOP
Wrapper function to wait out the chip ready signal and execute the SNOP command strobe.

Prototype
BYTE CC1100_StrobeNOP( void );

Parameters
None

Return Value
BYTE representing the CC1100 status BYTE.
Remarks
Provided for completeness, but not normally used except as a way to wait out the chip ready signal.

11.2.22 Function CC1100_QueryChipStatus
Waits out the chip ready signal and returns the CC1100 status BYTE.

Prototype
BYTE CC1100_QueryChipStatus( void );

Parameters
None

Return Value
BYTE representing the chip status value. See the CC1100 data sheet for more details concerning this value.

Remarks
Use to determine the current CC1100 operating state.

11.2.23 Function CC1100_TweakIdle
Tweaks the CS signal to start a transition to idle from power-down.

Prototype
void CC1100_TweakIdle( void );

Parameters
None

Return Value
None
Remarks
Used only for power save modes.

11.2.24 Function CC1100_QueryPARTNUM
Wrapper function to execute the PARTNUM status query.

Prototype
BYTE CC1100_QueryPARTNUM( void );

Parameters
None

Return Value
BYTE representing the PARTNUM register value.

Remarks
Use instead of executing the raw CC1100_GetStatusRegister call.

11.2.25 Function CC1100_QueryVERSION
Wrapper function to execute the VERSION status query.

Prototype
BYTE CC1100_QueryVERSION( void );

Parameters
None

Return Value
BYTE representing the VERSION register value.
Remarks
Use instead of executing the raw CC1100_GetStatusRegister call.

11.2.26  Function CC1100_QueryFREQEST
Wrapper function to execute the FREQEST status query.

Prototype
BYTE CC1100_QueryFREQEST( void );

Parameters
None

Return Value
BYTE representing the FREQEST register value.

Remarks
Use instead of executing the raw CC1100_GetStatusRegister call.

11.2.27  Function CC1100_QueryLQI
Wrapper function to execute the LQI status query.

Prototype
BYTE CC1100_QueryLQI( void );

Parameters
None

Return Value
BYTE representing the LQI register value.
Remarks
Use instead of executing the raw `CC1100_GetStatusRegister` call.

11.2.28 Function CC1100_QueryRSSI
Wrapper function to execute the RSSI status query.

Prototype
BYTE CC1100_QueryRSSI( void );

Parameters
None

Return Value
BYTE representing the RSSI register value.

Remarks
Use instead of executing the raw `CC1100_GetStatusRegister` call.

11.2.29 Function CC1100_QueryMARCSTATE
Wrapper function to execute the MARCSTATE status query.

Prototype
BYTE CC1100_QueryMARCSTATE( void );

Parameters
None

Return Value
BYTE representing the MARCSTATE register value.
Remarks
Use instead of executing the raw `CC1100_GetStatusRegister` call.

11.2.30 Function CC1100_QueryWORTIM

Wrapper function to execute the WORTIM status query.

Prototype

```
WORD CC1100_QueryWORTIM( void );
```

Parameters

None

Return Value

WORD representing the WORTIM1:WORTIM0 register values.

Remarks
Use instead of executing the raw `CC1100_GetStatusRegister` calls.

11.2.31 Function CC1100_QueryPKTSTATUS

Wrapper function to execute the PKTSTATUS status query.

Prototype

```
BYTE CC1100_QueryPKTSTATUS( void );
```

Parameters

None

Return Value

BYTE representing the PKTSTATUS register value.
Remarks
Use instead of executing the raw `CC1100_GetStatusRegister` call.

11.2.32 Function CC1100_QueryVCO_VC_DAC
Wrapper function to execute the VCO_VC_DAC status query.

Prototype

```
BYTE CC1100_QueryVCO_VC_DAC( void );
```

Parameters

None

Return Value

BYTE representing the VCO_VC_DAC register value.

Remarks
Use instead of executing the raw `CC1100_GetStatusRegister` call.

11.2.33 Function CC1100_QueryTXBYTES
Wrapper function to execute the TXBYTES status query to get the number of BYTES currently waiting in the transmit buffer.

Prototype

```
BYTE CC1100_QueryTXBYTES( void );
```

Parameters

None

Return Value

BYTE representing the TXBYTES register value.
Remarks

User applications should not call this function directly. DigitalRadio.s43 uses this function instead of executing the raw CC1100_GetStatusRegister call.

11.2.34 Function CC1100_QueryRXBYTES

Wrapper function to execute the RXBYTES status query to determine the number of bytes waiting in the receive buffer.

Prototype

BYTE CC1100_QueryRXBYTES( void );

Parameters

None

Return Value

BYTE representing the RXBYTES register value.

Remarks

User applications should not call this function directly. DigitalRadio.s43 uses this function instead of executing the raw CC1100_GetStatusRegister call.
11.3 ChannelParameters.c

This C language module provides access to the channel parameters used by the DigitalRadio.s43 module and the PacketRadio.c module, which are in turn used by all the packet demos provided in this software.

The feature flags wrapped by this module are defined in DigitalRadio_Structure.h, but use of the functions in this module reduce the amount of brainspace required to define these features.

Each of the features supported by this module are shadow features, which are not actually updated to the DigitalRadio module until the features are read from this module, and written to the DigitalRadio module, as shown below:

\[
wFeatures = \text{ChannelParameters\_GetFeatures}();
\]
\[
\text{DigitalRadio\_SetFeatures}( wFeatures );
\]

In turn, the DigitalRadio module only updates the CC1100 driver if the feature list actually changed.

The functions exported by this module are defined in ChannelParameters.h. This module defines no structures.

11.3.1 Function ChannelParameters\_GetRxAbortTicks

Returns the number of Rx abort ticks to be used by the packet handling modules. This value is defined as the number of system ticks to wait prior to declaring an Rx failure.

Prototype

\[
\text{WORD ChannelParameters\_GetRxAbortTicks}( \text{void} );
\]

Parameters

None

Return Value

WORD representing the number of Rx abort ticks.

Remarks

The default value DEFAULT_RX_ABORT_TICKS is set in ChannelParameters.c to 40.
11.3.2 Function ChannelParameters_GetFeatures

Returns the synthesized feature WORD used by the packet handling modules.

 Prototype

 WORD ChannelParameters_GetFeatures( void );

 Parameters
 None

 Return Value
 WORD representing the feature selections made via this module.

 Remarks
 The bits in this value are defined in DigitalRadio.h, and represent the selections made via the other functions in this module.

11.3.3 Function ChannelParameters_ReportFeatures

Sends the terminal host the current feature settings in human-readable format.

 Prototype

 void ChannelParameters_ReportFeatures( void );

 Parameters
 None

 Return Value
 None

 Remarks
 This function presents terminal output similar to the following:

 Current channel parameters:
Addresses disabled
CRC enabled
RSSI/LQI enabled
Whitening enabled
Sequence numbers enabled
Using channel 0x00 for variable length packets

This display can be accessed using the “D” option from the main menu.

11.3.4 Function ChannelParameters_UpdateAddressFromTerminal

Uses host terminal input to change the current channel address by hooking the current state stream.

Prototype

```c
void ChannelParameters_UpdateAddressFromTerminal( LPFN_STATE_VOID_VOID pfnNextState );
```

Parameters

LPFN_STATE_VOID_VOID pfnNextState: Address of the state function to be executed when the address collection is complete.

Return Value

None

Remarks

A typical address operation provided by this function and launched from the main menu via the “K” command is shown below:

```plaintext
-K
Change address (0x33): 0x21
Address set to 0x21
ok
-
```

This value is only used for the modes in RxTxModes.c, as the Hub and Node modes use predefined addresses.

11.3.5 Function ChannelParameters_EnableAddress

Enables the address filtering feature of the CC1100.
Prototype

void ChannelParameters_EnableAddress( void );

Parameters

None

Return Value

None

Remarks

Commands the packet handling modules to set the address control bits of PKTCTRL1 to ADD_CHK_BCST_00 prior to the next packet operation.

The Hub and Node modes always use addresses and disregard this feature setting.

11.3.6 Function ChannelParameters_DisableAddress

Disables the address filtering feature of the CC1100.

Prototype

void ChannelParameters_DisableAddress( void );

Parameters

None

Return Value

None

Remarks

Commands the packet handling modules to set the address control bits of PKTCTRL1 to ADD_CHK_NONE prior to the next packet operation.

The Hub and Node modes always use addresses and disregard this feature setting.
11.3.7 Function ChannelParameters_ToggleAddress

Toggles the address filtering feature of the CC1100.

**Prototype**

```c
void ChannelParameters_ToggleAddress( void );
```

**Parameters**

None

**Return Value**

None

**Remarks**

Commands the packet handling modules to set the address control bits of PKTCTRL1 as appropriate for whether the address has been enabled or disabled.

The Hub and Node modes always use addresses and disregard this feature setting.

11.3.8 Function ChannelParameters_SetAddress

Programmatically changes the current address.

**Prototype**

```c
void ChannelParameters_SetAddress( BYTE byAddress );
```

**Parameters**

BYTE byAddress: Address value to be used for the next packet operation, if enabled.

**Return Value**

None

**Remarks**
Only used for the modes in \textit{RxTxModes.c}, as the Hub and Node modes use predefined addresses.

\subsection*{11.3.9 Function ChannelParameters\_GetAddress}

Retrieves the currently selected packet address.

\textbf{Prototype}

\begin{verbatim}
BYTE ChannelParameters\_GetAddress( void );
\end{verbatim}

\textbf{Parameters}

None

\textbf{Return Value}

BYTE representing the address value to be used for the next packet operation, if enabled

\textbf{Remarks}

Only used for the modes in \textit{RxTxModes.c}, as the Hub and Node modes use predefined addresses.

\subsection*{11.3.10 Function ChannelParameters\_EnableCRC}

Enables the CRC filtering feature of the CC1100.

\textbf{Prototype}

\begin{verbatim}
void ChannelParameters\_EnableCRC( void );
\end{verbatim}

\textbf{Parameters}

None

\textbf{Return Value}

None
Remarks

Commands the packet handling modules to embed CRC values into transmit packets, and filter received packets by CRC, prior to the next packet operation.

The Hub and Node modes always use the CRC and disregard this feature setting.

11.3.11 Function ChannelParameters_DisableCRC

Disables the CRC filtering feature of the CC1100.

Prototype

void ChannelParameters_DisableCRC( void );

Parameters

None

Return Value

None

Remarks

Commands the packet handling modules to not embed CRC values into transmit packets, and not filter received packets by CRC, prior to the next packet operation.

The Hub and Node modes always use the CRC and disregard this feature setting.

11.3.12 Function ChannelParameters_ToggleCRC

Toggles the CRC filtering feature of the CC1100.

Prototype

void ChannelParameters_ToggleCRC( void );

Parameters

None
Return Value
None

Remarks
Commands the packet handling modules to toggle the CRC feature, prior to the next packet operation.
The Hub and Node modes always use the CRC and disregard this feature setting.

11.3.13 Function ChannelParameters_EnableRSSI_LQI
Enables the RSSI_LQI feature of the CC1100 for received packets.

Prototype
void ChannelParameters_EnableRSSI_LQI( void );

Parameters
None

Return Value
None

Remarks
Commands the packet handling modules to append the RSSI_LQI values to received packet data in the FIFO, prior to the next packet operation.
The Hub and Node modes always use RSSI/LQI and disregard this feature setting.

11.3.14 Function ChannelParameters_DisableRSSI_LQI
Disables the RSSI_LQI feature of the CC1100 for received packets.

Prototype
void ChannelParameters_DisableRSSI_LQI( void );
Parameters
None

Return Value
None

Remarks
Commands the packet handling modules to not append the RSSI_LQI values to received packet data in the FIFO, prior to the next packet operation.

The Hub and Node modes always use RSSI/LQI and disregard this feature setting.

11.3.15 Function ChannelParameters_ToggleRSSI_LQI
Toggles the RSSI_LQI feature of the CC1100 for received packets.

Prototype
void ChannelParameters_ToggleRSSI_LQI( void );

Parameters
None

Return Value
None

Remarks
Commands the packet handling modules to either append the RSSI_LQI values to received packet data in the FIFO, or not, prior to the next packet operation.

The Hub and Node modes always use RSSI/LQI and disregard this feature setting.

11.3.16 Function ChannelParameters_EnableWhiten
Enables the WHITE_DATA feature of the CC1100.
Prototype

```c
void ChannelParameters_EnableWhiten( void );
```

Parameters

None

Return Value

None

Remarks

Commands the packet handling modules to whiten outgoing packets, and dewhiten incoming packets, prior to the next packet operation.

The Hub and Node modes always use data whitening.

11.3.17 Function ChannelParameters_DisableWhiten

Disables the WHITE_DATA feature of the CC1100.

Prototype

```c
void ChannelParameters_DisableWhiten( void );
```

Parameters

None

Return Value

None

Remarks

Commands the packet handling modules to not whiten or dewhiten, prior to the next packet operation.

The Hub and Node modes always use data whitening.
11.3.18 Function ChannelParameters_ToggleWhiten

Toggles the WHITE_DATA feature of the CC1100.

Prototype

```c
void ChannelParameters_ToggleWhiten( void );
```

Parameters

None

Return Value

None

Remarks

Commands the packet handling modules to either whiten/dewhiten, or not, prior to the next packet operation.

The Hub and Node modes always use data whitening.

11.3.19 Function ChannelParameters_EnableSequenceNumber

Enables the sequence number feature of the DigitalRadio.s43 module.

Prototype

```c
void ChannelParameters_EnableSequenceNumber( void );
```

Parameters

None

Return Value

None

Remarks
Commands the *DigitalRadio.s43* module to embed sequence number BYTE values to transmit packets, and extract them from received packets, prior to the next packet operation.

The Hub and Node modes always use sequence numbers and disregard this feature setting.

### 11.3.20 Function ChannelParameters_DisableSequenceNumber

Disables the sequence number feature of the *DigitalRadio.s43* module.

**Prototype**

```c
void ChannelParameters_DisableSequenceNumber( void );
```

**Parameters**

None

**Return Value**

None

**Remarks**

Commands the *DigitalRadio.s43* module to not embed sequence number BYTE values to transmit packets, and not extract them from received packets, prior to the next packet operation.

The Hub and Node modes always use sequence numbers and disregard this feature setting.

### 11.3.21 Function ChannelParameters_ToggleSequenceNumber

Toggles the sequence number feature of the *DigitalRadio.s43* module.

**Prototype**

```c
void ChannelParameters_ToggleSequenceNumber( void );
```

**Parameters**

None
Return Value
None

Remarks
The new command state takes effect prior to the next packet operation.
The Hub and Node modes always use sequence numbers and disregard this feature setting.

11.3.22 Function ChannelParameters_SetChannel
Programmatically changes the current CC1100 channel.

Prototype

void ChannelParameters_SetChannel( BYTE byChannel );

Parameters
BYTE byChannel: Channel value to be used for the next packet operation.

Return Value
None

Remarks
The Hub and Node modes use predefined channels for network formation.

11.3.23 Function ChannelParameters_GetChannel
Retrieves the currently selected packet channel.

Prototype

BYTE ChannelParameters_GetChannel( void );

Parameters
Return Value

BYTE representing the channel to be used for the next packet operation.

Remarks

The Hub and Node modes use predefined channels for network formation.

11.3.24 Function ChannelParameters_SetLengthFixed

Sets the packet format to fixed-length, with no Forward Error Correction (FEC), for future packet operations.

Prototype

void ChannelParameters_SetLengthFixed( void );

Parameters

None

Return value

None

Remarks

For fixed length mode, the packet length is defined in the PKTLEN register.

The Hub and Node modes always use fixed length with FEC.

11.3.25 Function ChannelParameters_SetLengthFixedFEC

Sets the packet format to fixed-length, with Forward Error Correction (FEC), for future packet operations.

Prototype

void ChannelParameters_SetLengthFixedFEC( void );
Parameters

None

Return value

None

Remarks

For fixed length mode, the packet length is defined in the PKTLEN register.
The Hub and Node modes always use fixed length with FEC.

11.3.26 Function ChannelParameters_SetLengthVariable

Sets the packet format to variable-length, with no Forward Error Correction (FEC), for future packet operations.

Prototype

void ChannelParameters_SetLengthVariable( void );

Parameters

None

Return value

None

Remarks

The packet length is defined by data embedded in the transmit stream by DigitalRadio.s43. It is not possible to enable FEC with variable length packets.
The Hub and Node modes always use fixed length with FEC.
11.4 DigitalRadio.s43

This assembly language module provides a low-level functional wrapper around the CC1100.s43 driver, and adds addressing, sequence numbers, signal quality indicators, and other filtering operations to implement Tx and Rx operations.

These higher level Tx and Rx operations are implemented by taking over the state stream for a task. To ensure consistency, a task must first lock this resource using the functions provided.

The following type definition is used in this module:

```c
typedef BOOL ( * LPFN_RADIO_HANDLER_BOOL_VOID )( void );
```

and is typically implemented as:

```c
BOOL MyRadioHandler( void )
{
    return TRUE;
}
```

These handlers are called by the digital radio module during handler events, such as idle, Tx packet complete, Rx packet ready, and on errors. The interpretation of the return value is dependent upon the context of the handler.

The functions exported by this module are defined in DigitalRadio.h, and the callbacks and structures exported by this module are defined in DigitalRadio_Structure.h.

11.4.1 Function DigitalRadio_ResetHW

Resets the digital radio module, and in turn the CC1000 module, to the application defaults.

**Prototype**

```c
void DigitalRadio_ResetHW( LPBYTE pabyPATABL E );
```

**Parameters**

LPBYTE pabyPATABL E: An array of eight PATABL E bytes. See the CC1100 datasheet for a description of these bytes. Set to NULL to specify default values.

**Return Value**

None
Remarks

Internally, this function calls `DigitalRadio_SetPATABLE` after resetting application and CC1100 parameters.

11.4.2 Function DigitalRadio_SetPATABLE

Sets the PATABLE bytes in the CC1100.

Prototype

```c
void DigitalRadio_SetPATABLE( LPBYTE pabyPATABLE );
```

Parameters

LPBYTE pabyPATABLE: An array of eight PATABLE bytes. See the CC1100 datasheet for a description of these bytes. Set to NULL to specify default values.

Return Value

None

Remarks

The PATABLE is updated immediately and will be used on the next packet radio operation.

11.4.3 Function DigitalRadio_Lock

Attempts to lock the digital radio resource for the current task.

Prototype

```c
BOOL DigitalRadio_Lock( void );
```

Parameters

None

Return Value
BYTE representing the lock status. Returns TRUE if the radio was locked, FALSE if not.

Remarks
The digital radio resource must be locked by the current task prior to a Tx or Rx operation.

11.4.4 Function DigitalRadio_Unlock
Unlocks the digital radio, but only if the current lock is held by the current task.

Prototype

    void DigitalRadio_Unlock( void );

Parameters
None

Return Value
None

Remarks
Does not check for state consistency. Lock owner is responsible for maintaining state consistency. This is not an issue if the digital radio state stream is allowed to run to completion.

11.4.5 Function DigitalRadio_IsLocked
Reports the locked status of the digital radio resource.

Prototype

    BOOL DigitalRadio_IsLocked( void );

Parameters
None
Return Value

BYTE set to TRUE if the radio is locked, FALSE if not.

Remarks

Use prior to attempting a lock if multiple tasks can access the radio resource.

11.4.6 Function DigitalRadio_SetChannel

Sets the channel to be used by the digital radio resource for Tx and Rx operations.

Prototype

BOOL DigitalRadio_SetChannel( BYTE byChannel );

Parameters

BYTE byChannel: Desired channel for Tx and Rx operations

Return Value

BYTE set to TRUE if the channel was set, FALSE if the channel could not be set.

Remarks

Sets the channel in byChannel via the shadow mechanism, meaning that the CC1100 is not actually updated if the new channel is the same as the existing channel setting, and if different, the CC1100 is not actually updated until a Tx or Rx operation starts.

The shadow is not updated if the radio is not locked by the current task.

11.4.7 Function DigitalRadio_GetChannel

Reports the current shadowed channel setting of the digital radio resource.

Prototype

BYTE DigitalRadio_GetChannel( void );

Parameters
None

**Return Value**

BYTE representing the currently shadowed channel.

**Remarks**

This function only returns the shadowed value last passed to `DigitalRadio_SetChannel`, it does NOT report the CC1100 channel value.

### 11.4.8 Function `DigitalRadio_SetFeatures`

Enables the features defined in `DigitalRadio_Structure.h`.

**Prototype**

```c
BOOL DigitalRadio_SetFeatures( WORD wFeatures );
```

**Parameters**

WORD wFeatures: Bitmask of features to be used for the next Tx or Rx operation.

**Returns**

BYTE set to TRUE if the features were set, FALSE if the features could not be set.

**Remarks**

This function sets the features in `wFeatures` via the shadow mechanism, meaning the CC1100 values are not actually updated until the next Tx or Rx operation, and then only if the new settings are different from the old.

The set of features include:

- `DIGITAL_RADIO_FEATURE_FIXED_LENGTH`: Use fixed-length mode for both Tx and Rx. Must be used if `DIGITAL_RADIO_FEATURE_FEC` is enabled. If this bit is clear, then variable-length mode is used.

- `DIGITAL_RADIO_FEATURE_ADDRESS`: Embed address information in the packet framing for Tx packets, or filter Rx packets for the proper address.

- `DIGITAL_RADIO_FEATURE_SEQUENCE`: Embed sequence numbers in the packet framing for Tx packets, or parse a sequence number from Rx packets.
DIGITAL_RADIO_FEATURE_FEC: Enable Forward Error Correction (FEC) for fixed-length packets. Not available for variable length packets.

DIGITAL_RADIO_FEATURE_CRC: Embed CRC information in Tx packets, or filter Rx packets for the proper CRC.

DIGITAL_RADIO_FEATURE_RSSI_LQI: For Rx packets, report RSSI and LQI.

DIGITAL_RADIO_FEATURE_WHITEN: Applies data whitening for the transmitted and received packets.

11.4.9 Function DigitalRadio_GetFeatures

Returns the shadowed feature settings.

Prototype

```c
WORD DigitalRadio_GetFeatures( void );
```

Parameters

None

Return Value

WORD indicating the currently shadowed features.

Remarks

This only returns the shadowed value last passed to DigitalRadio_SetFeatures, not the settings on the CC1100 hardware.

11.4.10 Function DigitalRadio_StartTx

Starts a Tx operation by taking over the state stream of the calling task.

Prototype

```c
BOOL DigitalRadio_StartTx( const SDigitalRadioTx* ptTx,
LPFN_STATE_VOID_VOID pfnNextState );
```
Parameters

const SDigitalRadioTx* ptTx: Address of a ROM-able structure containing Tx parameters and handlers.

LPFN_STATE_VOID_VOID pfnNextState: Address of the default client state to be executed after completion of the Tx operation. This can be overridden in the OnError or OnPacketSent handlers.

Return Value

Returns TRUE if the Tx operation was started, FALSE if not. If FALSE, the OnError handler will be called with the failure code.

Remarks

See the SDigitalRadioTx structure for more details.

11.4.11 Function DigitalRadio_StartRx

Starts an Rx operation by taking over the state stream of the calling task.

Prototype

BOOL DigitalRadio_StartRx( const SDigitalRadioTx* ptTx,
                          LPFN_STATE_VOID_VOID pfnNextState );

Parameters

const SDigitalRadioRx* ptRx: Address of a ROM-able structure containing Rx parameters and handlers.

LPFN_STATE_VOID_VOID pfnNextState: Address of the default client state to be executed after completion of the Tx operation. This can be overridden in the OnError or OnPacketReady handlers.

Return Value

Returns TRUE if the Rx operation was started, FALSE if not. If FALSE, the OnError handler will be called with the failure code.

Remarks

See the SDigitalRadioRx structure for more details.
11.4.12 Structure SDigitalRadioOperation

Defines parameters used for all digital radio operations to report the operation status and communicate user parameters. This structure should be placed into RAM.

Members

WORD wResult: Variable to receive operation result. The operation results are spaced by two, to allow the efficient switch statement supported by IAR.

WORD wResultParam: Variable to receive operation result parameter, if any.

WORD wUserParam0: User parameter 0.

WORD wUserParam1: User parameter 1.

Remarks

This is a system-level structure. Application developers should not make changes to this structure without carefully modifying DigitalRadio.s43 as well.

The user parameters are helpful if multiple radio operations execute with the same handler, but different operations must be handled uniquely by client code. These two parameters can be used by the application developer to distinguish among different radio operations.

The following results can be reported by DigitalRadio.s43. PacketRadio.c can add additional results unique to the packet radio layer.

DIGITAL_RADIO_SUCCESS: The radio operation succeeded.

DIGITAL_RADIO_CANCEL: The radio operation was cancelled during an idle handler by client code. The wResultParam value will be set to 0x0000.

DIGITAL_RADIO_BAD_PARAMS: Invalid parameters were passed to the previous function call. The wResultParam value will be set to 0x0000.

DIGITAL_RADIO_RESOURCE_LOCKED: The digital radio resource was locked by another task. The wResultParam value will be set to 0x0000.

DIGITAL_RADIO_FAIL_TIMEOUT: The digital radio operation failed due to a timeout while waiting for the CC1100 SYNC signal. If the timeout occurred while waiting for SYNC to be asserted, the wResultParam value will be set to 0x0001. If the timeout occurred while waiting for SYNC to be deasserted, the wResultParam value will be set to 0x0000.
DIGITAL_RADIO_FAIL_CRC: The receive packet failed CRC filtering. The \texttt{wResultParam} value will be set to 0x0000.

DIGITAL_RADIO_FILTERED_OUT: The receive packet was filtered out due to an address mismatch. The \texttt{wResultParam} value will be set to 0x0000.

DIGITAL_RADIO_INVALID_SIZE: An error occurred due to a lack of sufficient buffer size. The \texttt{wResultParam} value contains the concatenated BYTES \texttt{byRxPayloadCount}:\texttt{byRxPayloadCapacity}, where \texttt{byRxPayloadCount} is the count of bytes to receive extracted from the channel, while \texttt{byRxPayloadCapacity} is the supplied buffer capacity.

DIGITAL_RADIO_CONTENT_MISMATCH: The received packet contained in the FIFO resolved to a different size than the expected packet size. The \texttt{wResultParam} value contains the concatenated BYTES \texttt{byRxCountFIFO}:\texttt{byPacketLength}, where \texttt{byRxCountFIFO} is the count of bytes to receive extracted from the channel, while \texttt{byPacketLength} is the expected packet size.

DIGITAL_RADIO_RX_BUFFER_OVERFLOW: The received packet overflowed the Rx FIFO. The \texttt{wResultParam} value will be set to 0x0000.

DIGITAL_RADIO_TX_BUFFER_UNDERFLOW: The transmitted packet overflowed the Tx FIFO. The \texttt{wResultParam} value will be set to 0x0000.

11.4.13 Structure SDigitalRadioRxParams

Defines parameters used for digital radio receive operations. This structure should be placed into RAM.

Members

\texttt{LPBYTE pabyRxData}: Address of buffer to be populated with received data.

\texttt{BYTE byRxDataSize}: Size of bytes to be received into \texttt{pabyRxData}.

\texttt{BYTE byRxCountRead}: Count of bytes read upon completion.

\texttt{BYTE byRxSequenceNumber}: Sequence number received, if enabled.

\texttt{BYTE byRxAddress}: Address for receive packet filtering, if applicable.

\texttt{WORD wRxTimeoutTicks}: Ticks allowed prior to declaring a receive timeout. Set this to \texttt{WAIT_RX_INFINITE} to wait until cancelled.

\texttt{WORD wRxAbortTicks}: Ticks allowed prior to aborting a receive packet in progress after sync. See \texttt{Remarks} below for more information about this member.

\texttt{BYTE byRSSI}: RSSI value to be populated upon completion, if enabled.
BYTE byLQI: LQI value to be populated upon completion, if enabled.

Remarks

This is a system-level structure. Application developers should not make changes to this structure without carefully modifying DigitalRadio.s43 as well.

Certain CC1100 input conditions can result in a state hang during an Rx sync. Set wRxAbortTicks to greater than the maximum number of ticks required for a given packet size, as determined by the following formula:

\[
\text{wRxAbortTicks} = \frac{N \times dwTickRate \times (8 + byRxPacketSize + byExtraBytes)}{dwBitRate}
\]

where

\( N = 12 \) for non-FEC, or 20 for FEC,
\( byRxPacketSize \) is the expected packet size to be transmitted to this unit,
\( dwTickRate \) is the system tick rate,
\( dwBitRate \) is the channel bit rate,
and \( byExtraBytes \) is the count of extra-payload bytes such as sequence numbers, length, and address.

Non-FEC Example:

\( byRxPacketSize = 32 \text{ bytes} \)
\( dwTickRateHz = 10000 \text{ ticks/sec} \)
\( dwBitRate = 250000 \text{ bits/sec} \)
\( byExtraBytes = 2 \) for sequence number and address

\[
\text{wRxAbortTicks} = \frac{12 \times (8 + 32 + 2) \times 10000}{250000} = 20.16 \Rightarrow 21
\]

Same example for FEC:

\[
\text{wRxAbortTicks} = \frac{20 \times (8 + 32 + 2) \times 10000}{250000} = 33.6 \Rightarrow 34
\]

11.4.14 Structure SDigitalRadioRx

Defines a receive operation, including client application parameters and handlers.

Members

SDigitalRadioOperation* ptOperation: Address of an operation structure to receive the operation result, as well as user parameters that can be tracked throughout the operation of the packet functions.

SDigitalRadioRxParams* ptRxParams: Pointer to a receive parameters structure.
LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnIdle: Address of an application-defined handler called by the digital radio module during idle. Returning FALSE cancels operation, causing radio shutdown and transition to the OnError state.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnPacketReady: Address of an application-defined handler called after the Rx operation completes.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnError: Address of an application-defined handler called if the Rx operation fails.

Remarks

This is a system-level structure. Application developers should not make changes to this structure without carefully modifying DigitalRadio.s43 as well.

This structure is defined independently of the SDigitalRadioOperation structure to allow the operation parameters to be shared between the Tx and Rx modes. This shared usage saves RAM.

This structure is defined independently of the SDigitalRadioRxParams structure to allow this structure to be ROM-able, which also saves RAM, since the members can all have fixed addresses, even though the ptOperation (in RAM) and ptRxParams (also in RAM) contents can and will change during operation.

11.4.15 Structure SDigitalRadioTxParams

Defines parameters used for digital radio transmit operations. This structure should be placed into RAM.

Members

LPBYTE pabyTxData: Address of buffer populated with transmit data.

BYTE byTxDataSize: Size of transmit buffer contents.

BYTE byTxCountSent: Count of bytes transmitted upon completion.

BYTE byTxSequenceNumber: Sequence number to be transmitted, if enabled, with the packet data.

BYTE byTxAddress: Address value to transmit in packet preamble, if enabled

WORD wTxTimeoutTicks: Ticks allowed prior to declaring a transmit (CCA) timeout.

Remarks
This is a system-level structure. Application developers should not make changes to this structure without carefully modifying DigitalRadio.s43 as well.

### 11.4.16 Structure SDigitalRadioTx

Defines a transmit operation, including client application parameters and handlers.

**Members**

SDigitalRadioOperation* ptOperation: Address of an operation structure to receive the operation result, as well as user parameters that can be tracked throughout the operation of the packet functions.

SDigitalRadioTxParams* ptTxParams: Pointer to a transmit parameters structure.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnIdle: Address of an application-defined handler called by the digital radio module during idle. Returning FALSE cancels operation, causing radio shutdown and transition to the *OnError* state.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnPacketSent: Address of an application-defined handler called after the Tx operation completes.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandlerOnError: Address of an application-defined handler called if the Tx operation fails.

**Remarks**

This is a system-level structure. Application developers should not make changes to this structure without carefully modifying DigitalRadio.s43 as well.

This structure is defined independently of the *SDigitalRadioOperation* structure to allow the operation parameters to be shared between the Tx and Rx modes. This shared usage saves RAM.

This structure is defined independently of the *SDigitalRadioTxParams* structure to allow this structure to be ROM-able, which also saves RAM, since the members can all have fixed addresses, even though the *ptOperation* (in RAM) and *ptTxParams* (also in RAM) contents can and will change during operation.
11.5 Display.c

This C language module provides an example of managing LEDs on the MRF1611CC1100 through the use of an array of custom event handlers, and a sub-system tick to implement cycling of LEDs.

The example used in HubMode.c and NodeMode.c are alternatives to defining an LED manager as a separate task, but only manages one LED (D1). While less than exciting as a way to manage one LED, this example does illustrate the concept of decoupling the display from other events running simultaneously.

This module provides the following type definition:

```c
typedef BOOL ( * LPFN_DISPLAY_EVENT_HANDLER_BOOL_VOID )( void );
```

and is typically implemented as:

```c
BOOL MyDisplayEventHandler( void )
{
    return TRUE;
}
```

The functions exported by this module are defined in Display.h. This module exports no structure definitions.

11.5.1 Function Display_Open

This function initializes any resources needed by this module.

**Prototype**

```c
BOOL Display_Open( void );
```

**Parameters**

None

**Return Value**

TRUE if the module was successfully opened, FALSE if not.

**Remarks**

This function is equivalent to object-initialization in C++ (constructor), and must be balanced by a later Display_Close call.
11.5.2 Function Display_Close

This function closes any resources needed by this module.

Prototype

BOOL Display_Close( void );

Parameters

None

Return Value

None

Remarks

This function is equivalent to object-closure in C++ (deconstructor), and must be called after an earlier Display_Open call.

11.5.3 Function Display_Tick

This function ticks the display state machine.

Prototype

void Display_Tick( void );

Parameters

None

Return Value

None

Remarks
This function supplies execution cycles to the display module’s internal state machines. It is the responsibility of the client to supply sufficient cycles by calling this function to allow the display cycling algorithms opportunity to work.

The client can withhold cycles during critical tasks without risk, but be aware that withholding cycles may cause some disturbance in the appearance of the display, although the effect on the display will be much less noticeable than a similar lack of cycles with the switches.

11.5.4 Function Display_RegisterEventHandlers

Registers a set of client-defined handlers for display events.

Prototype

```c
void Display_RegisterEventHandlers(
    const LPFN_DISPLAY_EVENT_HANDLER_BOOL_VOID * pahDisplayEvents );
```

Parameters

- `const LPFN_DISPLAY_EVENT_HANDLER_BOOL_VOID * pahDisplayEvents`: Address of a client-application-defined array of handlers for display events, described below under Remarks.

Return Value

None

Remarks

The set of handlers is implemented as an array of handler function pointers. To unregister a set of handlers, call this function with a NULL parameter.

The handlers are organized as follows:

```c
pahDisplayEvents[0] = OnDisplayCycleStart;
pahDisplayEvents[1] = OnDisplayCycleEnd;
```

`OnDisplayCycleStart` is fired when a display cycle is about to start, while `OnDisplayCycleEnd` is fired when a display cycle is about to end.

Registered handlers are unregistered when `Display_Close` is called. Returning FALSE from a display handler forces an immediate idle, and clears the display.

By monitoring these events, a client can determine when to cancel or change modes without introducing artifacts to the user.
11.5.5 Function Display_Off

This function forces an immediate display idle, and extinguishes the display.

Prototype

```c
void Display_Off( void );
```

Parameters

None

Return Value

None

Remarks

This function is also called internally from `Display_Close`.

11.5.6 Function Display_Flash

This function demonstrates a 0.1s / 0.9s flash sequence, optionally repeating with a given idle time between flashes.

Prototype

```c
void Display_Flash( BYTE byFlashCount,
                    BOOL bRepeat,
                    BYTE byIdleSeconds );
```

Parameters

BYTE byFlashCount: Count of flashes to display at 0.1 seconds on, 0.9 seconds off.

BOOL bRepeat: If TRUE, the sequence repeats until cancelled. If FALSE, the sequence executes once and stops.

BYTE byIdleSeconds: The number of seconds to dwell between a sequence of flashes.
Return Value
None

Remarks
If the cycle is set to repeat, then OnDisplayCycleStart and OnDisplayCycleEnd continue firing throughout the repeated cycles.

11.5.7 Function Display_IsIdle
Call to determine whether a display sequence has completed.

Prototype

BOOL Display_IsIdle( void );

Parameters
None

Return Value
TRUE if the display is in an idle state. If FALSE, the display is currently running a display pattern.

Remarks
This function should be used after returning FALSE to a display handler to ensure that the display has stopped. The current implementation will always return TRUE immediately after Display_Off has been called, or FALSE has been returned to a display handler, but the most general case would be to assume there may be some latency.
11.6 HubMode.c

This C language module implements a sample hub for a star network. The interface to this mode is simple, consists of only three functions defined in HubMode.h. This mode takes over the state stream, and also presents its own menu. The menu is described in the chapter concerning the operation of this sample.

11.6.1 Function HubMode_Open

Opens the hub mode module and initializes any resources required. Equivalent to object initialization in C++ (constructor).

Prototype

```c
BOOL HubMode_Open( void );
```

Parameters

None

Return Value

TRUE if the module was successfully opened, FALSE if not.

Remarks

Attempts to lock the digital radio resource for the calling task.

11.6.2 Function HubMode_Close

Closes the previously opened hub mode module, and releases any allocated resources. Equivalent to object de-initialization in C++ (destructor).

Prototype

```c
void HubMode_Close( void );
```

Parameters

None
Return Value

None

Remarks

Releases the digital radio resource lock, if previously locked for the calling task.

11.6.3 Function HubMode_Start

Starts hub mode operation by taking over the state stream using the indicated feature list.

Prototype

```c
void HubMode_Start( WORD wFeatures,
                   LPFN_STATE_VOID_VOID pfnNextState );
```

Parameters

WORD wFeatures: Bitmap of digital radio features to be used for this operation.

LPFN_STATE_VOID_VOID pfnNextState: Address of a client application state to be executed after completion of hub mode.

Return Value

None

Remarks

Requires that the HubMode_Open function was previously called.
11.7 InfoFlashDrive.s43

This assembly language module implements a flash drive system capable of managing a single file with a 60 byte capacity. For wear leveling, the flash file system sequences through 64-byte portions of both info flash segments. An alert licensed developer could easily adapt this module to use two or more main flash segments, with greater capacity.

While this module ensures that a valid file exists or not, it is the responsibility of the client application to ensure that the file data is valid for the applications. Sample applications in the demo do this by ensuring that the first WORD in a given file matches a “magic number” unique to each application.

The following values may be returned by functions in this module, and are defined in InfoFlashDrive_Results.h. These result codes are defined as consecutive even values to exploit use of the __even_in_range() intrinsic function supported by IAR Embedded Workbench.

INFO_FLASH_DRIVE_SUCCESS: The flash file operation succeeded.
INFO_FLASH_DRIVE_FILE_TOO_LARGE: The requested file size exceeds the available file data.
INFO_FLASH_DRIVE_FILE_NOT_FOUND: There is no file currently in flash.
INFO_FLASH_DRIVE_ERASE_FAILED: Unable to erase the flash in order to write.
INFO_FLASH_DRIVE_WRITE_FAILED: Unable to write the flash.

The functions exported by this module are defined in InfoFlashDrive.h. No public structures are defined by this module.

11.7.1 Function InfoFlashDrive_WriteFile

Performs a info drive file write operation.

Prototype

WORD InfoFlashDrive_WriteFile( LPBYTE pabyBuffer, WORD wCount );

Parameters

LPBYTE pabyBuffer: Address of a buffer containing BYTEs to be written to the info flash drive.

WORD wCount: Count of BYTEs in pabyBuffer to be written to the info flash drive.
Return Value

WORD containing one of the values from InfoFlashDrive_Results.h.

Remarks

This function can potentially delay for up to 20 milliseconds. The WDT may be disabled during this operation, and then re-enabled, if used. Due to the flash erase restrictions with the MSP430, it is not possible to interleave execute other functionality with flash erase or writes, so this call necessarily blocks until completion.

11.7.2 Function InfoFlashDrive_ReadFile

Performs a flash read operation from the info drive.

Prototype

WORD InfoFlashDrive_ReadFile( LPBYTE pabyBuffer, WORD wCount );

Parameters

LPBYTE pabyBuffer: Address of a buffer into which BYTEs are to be read from the info flash drive.

WORD wCount: Count of BYTEs to be read from the info flash drive into pabyBuffer.

Return Value

WORD containing one of the values from InfoFlashDrive_Results.h.

Remarks

This function imposes no inordinate delay as it merely reads flash, rather than writing.

11.7.3 Function InfoFlashDrive_EraseFile

This function unconditionally erases the info flash file, if it exists.

 Prototype

void InfoFlashDrive_EraseFile( void );
Parameters

None

Return Value

None

Remarks

After invoking this function, subsequent InfoFlashDrive_ReadFile operations will fail with the error INFO_FLASH_DRIVE_FILE_NOT_FOUND.

This function can potentially delay for up to 20 milliseconds. The WDT may be disabled during this operation, and then re-enabled, if used. Due to the flash erase restrictions with the MSP430, it is not possible to interleave execute other functionality with flash erase or writes, so this call necessarily blocks until completion.
11.8 Menu.c

This C language module manages the main application menu that is displayed on the terminal. Its sole export is declared in Menu.h. No structures are defined for this module.

11.8.1 Function Menu_GetInitialState

Prototype

LPFN_STATE_VOID_VOID Menu_GetInitialState( void );

Parameters

None

Return Value

Address of the initial state of the menu system. The main module uses this initial state to launch the menu system after some initial preparatory states.

Remarks

Once the menu state chain is launched, it never exits. Applications that are launched from the menu run to completion, and then exit back to the menu.
11.9 NodeMode.c

This C language module implements a sample node for a star network. The interface to this mode is simple, consists of only three functions defined in NodeMode.h. This mode takes over the state stream, and also presents its own menu. The menu is described in the chapter concerning the operation of this sample.

11.9.1 Function NodeMode_Open

Opens the node mode module and initializes any resources required. Equivalent to object initialization in C++ (constructor).

Prototype

BOOL NodeMode_Open( void );

Parameters

None

Return Value

TRUE if the module was successfully opened, FALSE if not.

Remarks

Attempts to lock the digital radio resource for the calling task.

11.9.2 Function NodeMode_Close

Closes the previously opened node mode module, and releases any allocated resources. Equivalent to object de-initialization in C++ (destructor).

Prototype

void NodeMode_Close( void );

Parameters

None
Return Value
None

Remarks
Releases the digital radio resource lock, if previously locked for the calling task.

11.9.3 Function NodeMode_Start

Starts node mode operation by taking over the state stream using the indicated feature list.

Prototype

```c
void NodeMode_Start( WORD wFeatures,
                     LPFN_STATE_VOID_VOID pfnNextState );
```

Parameters

WORD wFeatures: Bitmap of digital radio features to be used for this operation.

LPFN_STATE_VOID_VOID pfnNextState: Address of a client application state to be executed after completion of node mode.

Return Value
None

Remarks
Requires that the `NodeMode_Open` function was previously called.
11.10 PacketRadio.c

This C language module defines paired Tx+RxAck (Ping) or Rx+TxAck (Echo) operations. By ignoring the RxAck portion of a Ping operation, a simple Tx-only operation can be performed. Similarly, by ignoring the TxAck portion of an Echo operation, a simple Rx-only operation can be performed.

The packet format supported in this module is as follows:

```
PPPPPSSLAsDD...DDKK
P: Preamble bytes
S: Sync bytes
L: Length byte of A:s:Dn, inclusive
A: Address byte
s: Sequence number
D: Payload data bytes
K: CRC bytes
```

Three additional handlers (callbacks) are added to those required for the DigitalRadio.s43 module:

For packet Rx (Echo) mode:

```c
BOOL MyRxPreAckHandler( void );
```

This handler is called by the packet radio module during packet Rx mode prior to sending an Ack to a received packet. Returning FALSE aborts the Ack, causing radio shutdown and transition to the `OnPacketRxComplete` state.

For packet Tx (Ping) mode:

```c
BOOL MyTxPreAckHandler( void );
```

This handler is called by the packet radio module during packet Tx mode prior to waiting for an Ack to a transmitted packet. Returning FALSE aborts the wait for Ack, causing radio shutdown and transition to the `OnPacketTxComplete` state.

and

```c
BOOL MyTxRetryHandler( void );
```

This handler is called by the packet radio module during packet Tx mode prior to sending a retry of a transmitted packet upon failure to ack or upon a failure to transmit based on CCA, for example. Returning FALSE aborts the retry, causing radio shutdown and transition to the `OnPacketTxError` state for the last encountered error. This handler is only called if `SPacketRadioParams.wRetryCount` is greater than zero.

The functions exported by this module are defined in `PacketRadio.h`. The structures exported by this module are defined in `PacketRadio_Structure.h`. 
11.10.1 Function PacketRadio_StartTx

This function takes over the state stream for the current task and starts a Tx+RxAck operation.

Prototype

```c
BOOL PacketRadio_StartTx( const SPacketRadio* ptPacketTx,
                         LPFN_STATE_VOID_VOID pfnNextState );
```

Parameters

- `const SPacketRadio* ptPacketTx`: Address of a (ROM-able) structure containing parameters for this packet transmit operation.
- `LPFN_STATE_VOID_VOID pfnNextState`): Address of a client-application state to be executed when the Tx operation has been completed.

Return Value

Returns TRUE if the Tx operation was started, FALSE if not

Remarks

The digital radio lock must be owned by the calling task prior to calling this function. The packet radio module neither acquires, nor releases the digital radio lock so that a single digital radio lock may span many packet operations.

`pfnNextState` is the default next state after the operation. This next state can be overridden in the `OnPacketSent` and `OnError` handlers.

11.10.2 Function PacketRadio_StartRx

This function takes over the state stream for the current task and starts an Rx+TxAck operation.

Prototype

```c
BOOL PacketRadio_StartRx( const SPacketRadio* ptPacketRx,
                         LPFN_STATE_VOID_VOID pfnNextState );
```

Parameters
const SPacketRadio* ptPacketRx: Address of a (ROM-able) structure containing parameters for this packet receive operation.

LPFN_STATE_VOID_VOID pfnNextState ): Address of a client-application state to be executed when the Rx operation has been completed.

**Return Value**

Returns TRUE if the Rx operation was started, FALSE if not

**Remarks**

The digital radio lock must be owned by the calling task prior to calling this function. The packet radio module neither acquires, nor releases the digital radio lock so that a single digital radio lock may span many packet operations.

*pfnNextState* is the default next state after the operation. This next state can be overridden in the *OnPacketReady* and *OnError* handlers.

### 11.10.3 Structure SPacketRadioParams

This structure contains RAM-able variables used in both Tx and Rx modes.

**Members**

WORD wRxToAckTicks: Ticks to wait prior to declaring an Ack transmit. This member is only used for incoming packets (Echo mode).

WORD wRetryCount: Count of retry attempts allowed for outgoing packets (Ping mode only). Set to zero to try only once.

WORD wRetryTickCount: Count of state ticks prior to retry (Ping mode only). This value is loaded after the OnRetry handler is called, so there exists an opportunity for the client to vary the retry interval.

**Remarks**

This structure is only used as a member of the *SPacketRadio* structure, defined below.

This is a system-level structure. Application developers should not make changes to this structure without carefully modifying *PacketRadio.c* as well.

### 11.10.4 Structure SPacketRadio

This structure contains ROM-able members used in both Tx and Rx modes.
Members

SDigitalRadioOperation* ptOperation: Address of a digital radio operation structure.


SDigitalRadioRxParams* ptDigRadRxParams: Address of a digital radio Rx parameters structure for the Rx portion.

SDigitalRadioTxParams* ptDigRadTxParams: Address of a digital radio Tx parameters structure for the Tx portion.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnIdle: Application-defined idle handler, typically called while waiting for Rx.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnPreAck: Application-defined pre-ack handler called before the second (ack) phase. Return FALSE to abort the ack phase.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnPreRetry: Application-defined retry handler, called prior to attempting a retry in Ping mode. Ignored in Rx (Echo) mode.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnPacketComplete: Handler called after the packet operation is complete and acked. This handler is also entered if the Ack is aborted by caller.

LPFN_RADIO_HANDLER_BOOL_VOID pfnHandler_OnError: Application-defined handler, called if the packet operation fails.

Remarks

This is a system-level structure. Application developers should not make changes to this structure without carefully modifying PacketRadio.c as well.

This structure is defined independently of the SDigitalRadioOperation structure to allow the operation parameters to be shared between the Tx and Rx modes. This shared usage saves RAM.

This structure is defined independently of the SDigitalRadioTxParams and SDigitalRadioRxParams structures to allow this structure to be ROM-able, which also saves RAM, since the members can all have fixed addresses, even though the ptOperation (in RAM), ptDigRadTxParams (also in RAM), and ptDigRadRxParams (also in RAM) contents can and will change during operation.
11.11 RxTxHost.s43

This assembly language module provides host terminal services for receiving keystrokes and sending text. It uses Tx and Rx buffers to support a 115.2 kbps connection.

This module also manages the UART0 Tx and Rx interrupts internally. For proper queue management, it is critical that the RxTxHost_TickTx function be called from the main idle loop.

The functions exported by this module are defined in RxTxHost.h. This module exports no structures.

11.11.1 Function RxTxHost_QueueErasingBackspace

This function queues an erasing backspace, which is capable of removing a character on the terminal on the current line.

Prototype

void RxTxHost_QueueErasingBackspace( void );

Parameters

None

Return Value

None

Remarks

None

11.11.2 Function RxTxHost_QueueTxChar

This function queues a character for transmission to the host terminal.

Prototype

void RxTxHost_QueueTxChar( BYTE byTx );

Parameters
BYTE byTx: The outgoing ASCII character.

Return Value
None

Remarks
None

11.11.3 Function RxTxHost_QueueTxCRLF
This function queues a carriage return (0x0D) followed by a line feed character (0x0A).

Prototype
void RxTxHost_QueueTxCRLF( void );

Parameters
None

Return Value
None

Remarks
None

11.11.4 Function RxTxHost_QueueTxString
This function queues a null-terminated (C-style) string.

Prototype
void RxTxHost_QueueTxString( PSTR pString );

Parameters
PSTR pString: Address of a null-terminated string.
Return Value

None

Remarks

Ensure the outgoing string ends with 0x00!

11.11.5 Function RxTxHost_QueueTxHexBYTE

This function queues an outgoing BYTE value as hexadecimal.

Prototype

void RxTxHost_QueueTxHexBYTE( BYTE byValue );

Parameters

BYTE byValue: Outgoing BYTE to be converted and displayed as two hexadecimal nybbles, with a leading zero, if required.

Return Value

None

Remarks

If a leading “0x” is required, the client code is responsible for sending that first as a null-terminated string.

11.11.6 Function RxTxHost_QueueTxHexWORD

This function queues an outgoing BYTE value as hexadecimal.

Prototype

void RxTxHost_QueueTxHexWORD( WORD wValue );

Parameters

WORD wValue: Outgoing WORD to be converted and displayed as four hexadecimal nybbles, with leading zeroes, if required.
Return Value

None

Remarks

If a leading "0x" is required, the client code is responsible for sending that first as a null-terminated string.

11.11.7 Function RxTxHost_QueueTxHexDWORD

This function queues a DWORD value as a hexadecimal value.

Prototype

void RxTxHost_QueueTxHexDWORD( DWORD dwValue );

Parameters

DWORD dwValue: Outgoing DWORD to be converted and displayed as eight hexadecimal nybbles, with leading zeroes, if required.

Return Value

None

Remarks

Example: 0x12345678 is displayed as "12345678"

If a leading "0x" is required, the client code is responsible for sending that first as a null-terminated string.

11.11.8 Function RxTxHost_QueueTxHexColonDWORD

This function queues a DWORD value as two hexadecimal WORDs separated by a colon.

Prototype

void RxTxHost_QueueTxHexColonDWORD( DWORD dwValue );

Parameters
DWORD dwValue: Outgoing DWORD to be converted and displayed as eight hexadecimal nybbles, with leading zeroes, if required, and a colon in the middle.

**Return Value**

None

**Remarks**

Example: 0x12345678 is displayed as “1234:5678”

If a leading “0x” is required, the client code is responsible for sending that first as a null-terminated string.

### 11.11.9 Function RxTxHost_TickTx

This function provides idle cycles for driving outgoing characters from the queue out the UART.

**Prototype**

```c
void RxTxHost_TickTx( void );
```

**Parameters**

None

**Return Value**

None

**Remarks**

Typically called by the main.c idle loop, or during any cycle-intensive processes, although the latter should be refactored as a series of smaller states.

### 11.11.10 Function RxTxHost_GetRx

This function checks the receive queue and indicates whether any characters have been received from the host terminal.
Prototype

WORD RxTxHost_GetRx( void );

Parameters

None

Return Value

WORD containing the received character in the low BYTE, or 0xFFFF if no character has been received.

Remarks

Use this function if it is desired to examine the character prior to releasing an echo.

11.11.11 Function RxTxHost_GetRxEcho

This function checks the receive queue and indicates whether any characters have been received from the host terminal, and echoes the current character, if any.

Prototype

WORD RxTxHost_GetRxEcho( void );

Parameters

None

Return Value

WORD containing the received character in the low BYTE, or 0xFFFF if no character has been received.

Remarks

Use this function if it is desired to unconditionally echo the received character.

11.11.12 Function RxTxHost_GetHexBYTE
This function takes over the task state stream and collects a single BYTE value from the user as a hexadecimal value.

Prototype

BOOL RxTxHost_GetHexBYTE( LPFN_STATE_VOID_VOID pfnNextState, LPBYTE pbyValue, LPBOOL pbResult );

Parameters

LPFN_STATE_VOID_VOID pfnNextState: Address of a client-task state to be executed after the BYTE has been collected, or the collection terminated with Escape.

LPBYTE pbyValue: Address of a client BYTE to receive the collected BYTE.

LPBOOL pbResult: Address of a client BOOL to receive the result of the operation. If set to TRUE upon execution of pfnNextState, the value addressed by pbyValue is valid. If FALSE, then the user cancelled the operation.

Return Value

TRUE if the operation could be started, FALSE if not.

Remarks

The current implementation always returns TRUE.

11.11.13 Function RxTxHost_GetCountTxInQueue

This function returns the count of queued outgoing characters.

Prototype

WORD RxTxHost_GetCountTxInQueue( void );

Parameters

None

Return Value

WORD containing the indicated count.
Remarks

To determine the number of characters that can be queued before overflow, call \textit{RxTxHost\_GetCountTxCapacity}.

11.11.14 Function \texttt{RxTxHost\_GetCountRxInQueue}

This function returns the count of queued incoming characters.

Prototype

\begin{verbatim}
WORD RxTxHost_GetCountRxInQueue( void );
\end{verbatim}

Parameters

None

Return Value

WORD containing the indicated count.

Remarks

To determine the number of characters that can be queued from the host before overflow, call \textit{RxTxHost\_GetCountRxCapacity}.

11.11.15 Function \texttt{RxTxHost\_GetCountTxCapacity}

This function returns the number of characters that can be queued to the host before overflowing the Tx buffer.

Prototype

\begin{verbatim}
WORD RxTxHost_GetCountTxCapacity( void );
\end{verbatim}

Parameters

None
Return Value
WORD containing the indicated count.

Remarks
To determine the number of characters that are currently in the Tx queue, call `RxTxHost_GetCountTxInQueue`.

11.11.16 Functions RxTxHost_GetCountRxCapacity
This function returns the number of characters that can be queued from the host before overflowing the Rx buffer.

Prototype

WORD RxTxHost_GetCountRxCapacity( void );

Parameters
None

Return Value
WORD containing the indicated count.

Remarks
To determine the number of characters that are currently in the Rx queue, call `RxTxHost_GetCountRxInQueue`. 
11.12 RxTxModes.c

This C language module implements sample Rx-only, Tx-only, Echo, and Ping nodes for a point-to-point network. The interface to these modes consists of only six functions defined in RxTxModes.h. These modes take over the state stream.

Tx mode transmits a 32-byte packet to an Rx mode unit. Ping mode sends a 32-byte packet in Tx mode, then waits for a 32-byte Rx packet as an acknowledgement. Echo mode waits for a 32-byte packet, then sends a 32-byte Tx packet as an acknowledgement.

11.12.1 Function RxTxModes_Open

Opens the module and initializes any resources required. Equivalent to object initialization in C++ (constructor).

Prototype

BOOL RxTxModes_Open( void );

Parameters

None

Return Value

TRUE if the module was successfully opened, FALSE if not.

Remarks

Attempts to lock the digital radio resource for the calling task.

11.12.2 Function RxTxModes_Close

Closes the previously opened module, and releases any allocated resources. Equivalent to object de-initialization in C++ (destructor).

Prototype

void RxTxModes_Close( void );
Parameters
None

Return Value
None

Remarks
Releases the digital radio resource lock, if previously locked for the calling task.

11.12.3 Function RxTxModes_StartTx
Starts Tx-only operation by taking over the state stream using the indicated feature list.

Prototype
void RxTxModes_StartTx( WORD wFeatures,
                        LPFN_STATE_VOID_VOID pfnNextState );

Parameters
WORD wFeatures: Bitmap of digital radio features to be used for this operation.
LPFN_STATE_VOID_VOID pfnNextState: Address of a client application state to be executed after completion of this mode.

Return Value
None

Remarks
Requires that the RxTxModes_Open function was previously called.

11.12.4 Function RxTxModes_StartRx
Starts Rx-only operation by taking over the state stream using the indicated feature list.
Prototype

void RxTxModes_StartRx( WORD wFeatures,
                        LPFN_STATE_VOID_VOID pfnNextState );

Parameters

WORD wFeatures: Bitmap of digital radio features to be used for this operation.

LPFN_STATE_VOID_VOID pfnNextState: Address of a client application state to be executed after completion of this mode.

Return Value

None

Remarks

Requires that the RxTxModes_Open function was previously called.

11.12.5 Function RxTxModes_StartPing

Starts Ping mode operation by taking over the state stream using the indicated feature list.

Prototype

void RxTxModes_StartPing( WORD wFeatures,
                         LPFN_STATE_VOID_VOID pfnNextState );

Parameters

WORD wFeatures: Bitmap of digital radio features to be used for this operation.

LPFN_STATE_VOID_VOID pfnNextState: Address of a client application state to be executed after completion of this mode.

Return Value

None

Remarks
Requires that the *RxTxModes_Open* function was previously called.

### 11.12.6 Function RxTxModes_StartEcho

Starts Echo mode operation by taking over the state stream using the indicated feature list.

**Prototype**

```c
void RxTxModes_StartEcho( WORD wFeatures,
                          LPFN_STATE_VOID_VOID pfnNextState );
```

**Parameters**

- **WORD wFeatures**: Bitmap of digital radio features to be used for this operation.
- **LPFN_STATE_VOID_VOID pfnNextState**: Address of a client application state to be executed after completion of this mode.

**Return Value**

None

**Remarks**

Requires that the *RxTxModes_Open* function was previously called.
11.13 StarPacket.h

While not strictly a module, this header file defines some structures shared between the sample hub and node modes. Those structures are defined here.

All hub and node packets are 0x20 (32d) BYTES long for filter compatibility with the RxTxModes.c packets. As such, additional room remains at the end of each packet for the user to piggyback additional data for development fun.

The first BYTE in any star packet is a packet type byte. The packet type is selected from the following list:

STAR_PACKET_TYPE_HUB_SEARCH: The hub search packet is used to search for nodes.

STAR_PACKET_TYPE_HUB_SEARCH_ACK: The hub search acknowledge is sent by nodes in response to a hub search.

STAR_PACKET_TYPE_PING: The ping packet is used from nodes in active mode.

STAR_PACKET_TYPE_PING_ACK: The ping packet acknowledge is sent by hubs in response to a node ping.

11.13.1 Structure SStarPacketSearch

The hub search packet is the outgoing packet format for search mode.

Members

BYTE byPacketID: Set to STAR_PACKET_TYPE_HUB_SEARCH.

BYTE byNodeID: The node ID sought by the hub.

BYTE byNodeTxAddress: The active mode Tx address to be used by the node.

BYTE byNodeRxAddress: The active mode Rx address to be used by the node.

BYTE byChannel: The active mode channel to be used by the node.

BYTE byDummy: Dummy for WORD alignment

DWORD dwSerialNumber: The DWORD serial number for the hub.

Remarks

The HubMode.c menu drives this packet out on channel 0x00 at address 0xF0.
11.13.2  Structure SStarPacketSearchAck

The hub search acknowledge packet is the incoming packet format for search mode.

Members

BYTE byPacketID: Set to STAR_PACKET_TYPE_HUB_SEARCH_ACK.
BYTE byNodeId: The node ID assigned to the node by the user.
DWORD dwSerialNumber: The DWORD serial number for the node.
BYTE byRSSI: The RSSI value observed by the node for the hub’s search packet.
BYTE byLQI: The LQI value observed by the node for the hub’s search packet.

Remarks

The NodeMode.c module drives this packet out on channel 0x00 at address 0xF0.

11.13.3  Structure SStarPacket

The star packet is the outgoing packet format for active mode.

Members

BYTE byPacketID: Set to STAR_PACKET_TYPE_PING
BYTE byNodeId: The node ID sending the message.
DWORD dwSerialNumber: The DWORD serial number for the node sending the message.
WORD wADC: ADC value for the node.

Remarks

The NodeMode.c module drives this packet out on channel SStarPacketSearch.byChannel at address SStarPacketSearch.byNodeTxAddress.
11.13.4 Structure SStarPacketAck

The star acknowledge packet is the incoming packet format for active mode.

**Members**

BYTE byPacketID: The value STAR_PACKET_TYPE_PING_ACK.

BYTE byNodeID: The node ID addressed by the hub.

DWORD dwSerialNumber: The DWORD serial number for the node.

BYTE byRSSI: The RSSI value observed by the hub for the node’s ping packet.

BYTE byLQI: The LQI value observed by the hub for the node’s ping packet.

WORD wADC: The measured ADC value for the hub.

**Remarks**

The *HubMode.c* module drives this packet out on channel *SStarPacketSearch*.
*byChannel* at address *SStarPacketSearch.byNodeRxAddress*. 
11.14 Switch.c

This C language module provides an example of managing switches on the HMRF1611 through the use of an array of custom event handlers, and a sub-system tick to implement software debouncing.

This example is an alternative to defining a switch manager as a separate task.

11.14.1 Function Switch_Open

This function initializes any resources needed by this module.

Prototype

BOOL Switch_Open( void );

Parameters

None

Return Value

TRUE if the module was successfully opened, FALSE if not.

Remarks

This function is equivalent to object-initialization in C++ (constructor), and must be balanced by a later Switch_Close call.

11.14.2 Function Switch_Close

This function closes any resources needed by this module.

Prototype

BOOL Switch_Close( void );

Parameters

None
11.14.3 Function Switch_Tick

This function ticks the switch state machine.

Prototype

```c
void Switch_Tick( void );
```

Parameters

None

Return Value

None

Remarks

This function supplies execution cycles to the switch module's internal state machines. It is the responsibility of the client to supply sufficient cycles by calling this function to allow the debouncing algorithms opportunity to work. The client can withhold cycles during critical tasks without risk, but be aware that withholding cycles will delay reporting of switch events, or even cause some repeated events to be missed.

11.14.4 Function Switch_RegisterEventHandlers

This function registers a set of event handlers (callbacks) to be executed upon detection of a switch press or release.

Prototype

```c
void Switch_RegisterEventHandlers( const LPFN_SWITCH_EVENT_HANDLER_VOID_VOID * pahSwitchEvents );
```
Parameters

const LPFN_SWITCH_EVENT_HANDLER_VOID_VOID * pahSwitchEvents : Address of an array of eight switch event handlers, as defined below. Call with this parameter set to NULL to unregister the handlers. Declared as const, this array is ROM-able.

Return Value

None

Remarks

The switch event handlers are automatically unregistered when Switch_Close() is called.

This array of switch handlers is organized as follows:

pahSwitchEvents[0] = OnSwitch1_Down; // Switch 1 was pressed
pahSwitchEvents[1] = OnSwitch1_Up;   // Switch 1 was released
pahSwitchEvents[2] = OnSwitch2_Down; // Switch 2 was pressed
pahSwitchEvents[3] = OnSwitch2_Up;   // Switch 2 was released
pahSwitchEvents[4] = OnSwitch3_Down; // Switch 3 was pressed
pahSwitchEvents[5] = OnSwitch3_Up;   // Switch 3 was released
pahSwitchEvents[6] = OnSwitch4_Down; // Switch 4 was pressed
pahSwitchEvents[7] = OnSwitch4_Up;   // Switch 4 was released

where each handler is of the form:

void MySwitchEventHandler( void );

11.14.5 Function Switch_SetRepeatParameters

This function establishes the parameters to be used for synthesizing auto-repeat events for when a switch is pressed and held pressed.

Prototype

void Switch_SetRepeatParameters( WORD wDwellTicks, WORD wRepeatTicks );

Parameters

WORD wDwellTicks: The number of state ticks to delay during a switch press before declaring the first repeated event. To turn off repeats, set this value to 0. The default value is equivalent to 0.5 seconds, or 4096 ticks for the demo application.
WORD wRepeatTicks: The number of state ticks to delay between repeated switch event declarations during a repeat stream. The default value is equivalent to 0.1 seconds, or 819 ticks for the demo application.

Return Value

None

Remarks

These values are restored to the defaults when `Switch_Close()` is called. Even so, each client application opening the switch module for use should explicitly initialize its unique repeat parameters for consistency.
11.15  Task.s43

This assembly language module contains code that manages tasks for the application. The functions exported by this module are defined in Task.h, while the structures exported by this module are defined in Task_Structure.h.

11.15.1  Function Task_TickTask

The Task_TickTask function provides a means to drive a task’s state machine with state ticks. While this function is executing, the indicated task is defined as the current task.

Prototype

void Task_TickTask( STask* ptCurrentTask );

Parameters

STask* ptCurrentTask: Address of a structure for the task to be given state ticks. Application programmers should not access members of this structure directly.

Return Value

None

Remarks

The Task_TickTask function is only used in the main idle loop, and should not be called from elsewhere. Each task defined in the system will have its task ticked individually.

11.15.2  Function Task_InitializeTask

The Task_InitializeTask function is called by the owner of a task to properly initialize the task memory and place it into service.

Prototype

void Task_InitializeTask( STask* ptTask );

Parameters

None
STask* ptTask: Address of a structure for the task to be given state ticks. Application programmers should not access members of this structure directly.

Return Value

None

Remarks

This function should be called for each task in the system at startup.

11.15.3 Function Task_GetCurrentTask

This function retrieves the address of the current task’s structure. The current task is defined as that task for which Task_TickTask is executing.

Prototype

STask* Task_GetCurrentTask( void );

The Task_GetCurrentTask function can be used by the application at any time to get a pointer to the currently active task. This function enables multitasking applications without task dependencies.

Parameters

None

Return Value

Pointer to the current task structure.

Remarks

This function should be used sparingly by application programmers, if at all.

11.15.4 Function Task_SetUserParam0

Sets the wUserParam0 value for the current task.
Prototype

void Task_SetUserParam0( WORD wUserParam );

Parameters

WORD wUserParam: 16-bit value to be stored in the wUserParam0 member of the current task.

Return Value

None

Remarks

Each task includes two WORD values, wUserParam0 and wUserParam1, which are generic storage as deemed fit for use by the application developer (user of the application framework). These four functions provide a safe way to access these parameters for the current task.

These functions can be used to identify task-specific memory that allow application code to access different parameters, depending on which task is currently active.

11.15.5 Function Task_GetUserParam0

Retrieves a previously set value for the wUserParam0 member for the current task.

Prototype

WORD Task_GetUserParam0( void );

Parameters

None

Return Value

WORD representing the previous set value for wUserParam0.

Remarks

Each task includes two WORD values, wUserParam0 and wUserParam1, which are generic storage as deemed fit for use by the application developer (user of the
application framework). These four functions provide a safe way to access these parameters for the current task.

These functions can be used to identify task-specific memory that allow application code to access different parameters, depending on which task is currently active.

11.15.6 Function Task_SetUserParam1

Sets the wUserParam1 value for the current task.

Prototype

void Task_SetUserParam1( WORD wUserParam );

Parameters

WORD wUserParam: 16-bit value to be stored in the wUserParam1 member of the current task.

Return Value

None

Remarks

Each task includes two WORD values, wUserParam0 and wUserParam1, which are generic storage as deemed fit for use by the application developer (user of the application framework). These four functions provide a safe way to access these parameters for the current task.

These functions can be used to identify task-specific memory that allow application code to access different parameters, depending on which task is currently active.

11.15.7 Function Task_GetUserParam1

Retrieves a previously set value for the wUserParam1 member for the current task.

Prototype

WORD Task_GetUserParam1( void );
Parameters
None

Return Value
WORD representing the previous set value for wUserParam1.

Remarks
Each task includes two WORD values, wUserParam0 and wUserParam1, which are generic storage as deemed fit for use by the application developer (user of the application framework). These four functions provide a safe way to access these parameters for the current task.

These functions can be used to identify task-specific memory that allow application code to access different parameters, depending on which task is currently active.

11.15.8 Function Task_SetCriticalErrorHandler

Sets the critical function handler for the current task.

Prototype

```c
void Task_SetCriticalErrorHandler( LPFN_EVENT_HANDLER_VOID_WORD_WORD pfnCriticalErrorHandler );
```

Parameters

LPFN_EVENT_HANDLER_VOID_WORD_WORD pfnCriticalErrorHandler: Address of the critical error handler to be called by the task module.

Return Value
None

Remarks

The Task_SetCriticalErrorHandler function defines the event handler to be called in the event of an (application-defined) critical error. This demo software only uses this function as an example. Live applications which require a critical error handler should call this function in the initial state if critical error notification is desired.
11.15.9  Function Task_DeclareCriticalError

Calls the function previously registered as the critical error handler. If no such handler was previously registered, no action is taken.

Prototype

void Task_DeclareCriticalError( WORD wErrorID, WORD wParam );

Parameters

WORD wErrorID: Application-defined error ID to be passed to the critical error handler.

WORD wParam: Application-defined error parameter to be passed to the handler.

Return Value

None

Remarks

Task_InitializeTask, which initializes the critical error handler to a safe default value, MUST be called prior to using this function.

11.15.10  Function Task_SetEventHandler

Sets the event function handler for the current task.

Prototype

void Task_SetEventHandler( LPFN_EVENT_HANDLER_VOID_WORD_WORD pfnEventHandler );

Parameters

LPFN_EVENT_HANDLER_VOID_WORD_WORD pfnEventHandler: Address of the event handler to be called by the task module.
Return Value

None

Remarks

The Task_SetEventHandler function defines an event handler to receive non-critical events in an application which uses events. This demo application uses callbacks rather than event handlers for most purposes. The “handlers” defined in the packet and digital radio modules are actually implemented as callbacks.

11.15.11 Function Task_FireEvent

Calls the function previously registered as the event handler. If no such handler was previously registered, no action is taken.

Prototype

void Task_FireEvent( WORD wEventID, WORD wParam );

Parameters

WORD wEventID: Application-defined event ID to be passed to the event handler.
WORD wParam: Application-defined event parameter to be passed to the handler.

Return Value

None

Remarks

Task_InitializeTask, which initializes the event handler to a safe default value, MUST be called prior to using this function.

11.15.12 Function Task_UpdateState

Set the state to be executed at the next tick.

Prototype

void Task_UpdateState( LPFN_STATE_VOID_VOID pfnNewState );
Parameters

LPFN_STATE_VOID_VOID pfnNewState: Address of the state to be executed at the next tick.

Return Value

None

Remarks

The Task_UpdateState function causes an unconditional change to a new state. Use of this function is the state equivalent of a branch.

11.15.13 Function Task_PushState

Stores state in the task’s state stack for later reference.

Prototype

void Task_PushState( LPFN_STATE_VOID_VOID pfnPushState );

Parameters

LPFN_STATE_VOID_VOID pfnPushState: Address of the state to be stored.

Return Value

None

Remarks

The Task_PushState function uses the 8-position state stack to store a return state to assist in implementation of state functions. The depth of this stack enables up to eight levels of state chain calls, allowing for remarkably complex state architectures. Use of the Task_PushState function, followed by the Task_UpdateState function is the state equivalent of a call.

11.15.14 Function Task_PopState

Restores a previously pushed state from the task’s state stack.
Prototype

void Task_PopState( void );

Parameters

None

Return Value

None

Remarks

The Task_PopState function activates the next state on the state stack, and is the state equivalent of a return.

11.15.15 Function Task_DelayTicks

Inserts a given number of delay states into a state stream.

Prototype

void Task_DelayTicks( LPFN_STATE_VOID_VOID pfnNextState, WORD wDelayCount );

Parameters

LPFN_STATE_VOID_VOID pfnNextState: Address of the state following the delay.

WORD wDelayCount: Number of state ticks to delay.

Return Value

None

Remarks

The Task_DelayTicks function is a library implementation of the state delay example seen earlier. The actual chronological delay is dependent on the tick granularity (8192 Hz in this demo software), and whether any ticks have been skipped due to overrun.
Most applications can tolerate a certain amount of slop in timing for some purpose. For example, a switch debounce algorithm may not care whether 4.88 milliseconds (40 ticks) or 5.00 milliseconds (41 ticks) have transpired. In these non-critical applications, use of this function can save use, and deconfliction, of a hardware timer resource. Applications needing more precise timing should use a timer channel for this purpose.

Note also that between delay ticks, the processor sleeps per the mode set in the idle loop, thus saving power. This is a superior approach to simply burning cycles in a timing loop.

11.15.16 Structure STask

This structure defines a task, which is capable of supporting a state machine and state stack, user parameters, a critical error handler, and an event handler.

Members

LPFN_STATE_VOID_VOID pfnState: Address of the next state to be executed.

WORD wFlags: Flags used by the Task module to manage the task.

WORD wUserParam0: Application-defined user parameter 0.

WORD wUserParam1: Application-defined user parameter 1.

LPFN_EVENT_HANDLER_VOID_WORD_WORD pfnCriticalErrorHandler: Application-defined critical error handler.

LPFN_EVENT_HANDLER_VOID_WORD_WORD pfnEventHandler: Application-defined event handler.

WORD wCounter: Generic task counter, to only be used by the Task module.

WORD wStateStackIndex: Index to be used only by the Task module to manage the state stack.

LPFN_STATE_VOID_VOID apfnStateStack[STATE_STACK_SIZE]: State stack for managing state transitions.

Remarks

The application developer will only directly access this task structure to pass it to the Task_InitializeTask and Task_TickTask functions. All other accesses should be via functions defined in the Task module.
11.16 Utility.s43

This assembly language module is provided as a collecting point for various otherwise unattached functions that may accumulate during development, especially those which may be of interest to more than one module.

Currently, the only functionality exported by this module is the single function defined in Utility.h. There are no structures defined by this module.

11.16.1 Function Utility_FormatVoltage_2_50v

This function formats an ADC12 measurement (versus the 2.5v internal reference) into a null-terminated string suitable for display to the user as a voltage between “0.00v” to “2.50v”.

Prototype

void Utility_FormatVoltage_2_50v( WORD wADC12, char* szVoltage );

Parameters

WORD wADC12: ADC12 measurement versus the internal 2.5v reference, in the range 0x0000 to 0x0FFF.

char* szVoltage: Address of a character buffer to receive the formatted string.

Return Value

None

Remarks

The szVoltageBuffer must contain room for at least six characters, counting the null-terminator.

If a voltage reference other than 2.5v was used, the value will be scaled between 0.00v (0x0000) and 2.50v (0x0FFF) as if the reference had been 2.5v.