



AIR BoosterStack

Programmer's Guide

Revised 1/18/2012



Anaren Integrated Radio

This page intentionally left blank.

Disclaimer

AIR BoosterStack (ABS) is provided as is, for free and the user has rights to change the code to fit the application needs and to use the code in commercial applications as long as:

- ABS is only used with Anaren AIR radio modules.
- ABS is operating in accordance with local radio frequency usage rules.
- ABS is only using the provided radio configuration settings.
- ABS is only being used within the duty cycles allowed by local codes.

By using this guide and developing applications using the AIR BoosterStack software, you agree not to hold Anaren and its subsidiaries liable and responsible for any damages it might cause.

Note For a full description on the ABS disclaimer, see the application source code.

This page intentionally left blank.

Contents

Disclaimer.....	2
1 Introduction.....	10
1.1 Overview	10
1.2 Purpose of this document	11
1.3 Intended Audience.....	11
1.4 Basic Operation	11
1.5 Physical Layer and Logical Channels.....	12
1.5.1 Logical Radio (R)	13
1.5.2 Radio configuration settings (S).....	13
1.5.3 Frequency, channel within configured band (C).....	13
1.5.4 Transmit output power (P).....	13
1.6 Demo Application Overview	13
2 Software Modules	14
2.1 Development platform	14
2.2 Compiler Differences	14
2.3 Loading template project.....	15
2.3.1 Code Composer Studio.....	15
2.3.2 IAR Embedded Workbench.....	16
3 Application Layer	16
3.1 NodeInfo	16
3.2 HubWirelessOperation	18
3.2.1 Prototype	18
3.2.2 Description	18
3.2.3 Parameters.....	18
3.2.4 Return value	18
3.2.5 Example usage.....	19
3.3 SensorWirelessOperation	19
3.3.1 Prototype	19
3.3.2 Description	19

3.3.3	Parameters.....	19
3.3.4	Return value	19
3.3.5	Example usage.....	20
3.4	SetMyNodeID	20
3.4.1	Prototype	20
3.4.2	Description	20
3.4.3	Parameters.....	20
3.4.4	Return value	20
3.4.5	Example usage.....	21
3.5	SetPairingMask	21
3.5.1	Prototype	21
3.5.2	Description	21
3.5.3	Parameters.....	21
3.5.4	Return value	21
3.5.5	Example usage.....	21
3.6	RemoveNodeFromNetwork.....	21
3.6.1	Prototype	21
3.6.2	Description	21
3.6.3	Parameters.....	22
3.6.4	Return value	22
3.6.5	Example usage.....	22
4	Protocol Layer.....	22
4.1	InitProtocol.....	22
4.1.1	Prototype	22
4.1.2	Description	22
4.1.3	Parameters.....	22
4.1.4	Return value	22
4.1.5	Example usage.....	22
4.2	DataExchange.....	23
4.2.1	Prototype	23

4.2.2	Description	23
4.2.3	Parameters.....	23
4.2.4	Return value	23
4.2.5	Example usage.....	23
4.3	Listen	24
4.3.1	Prototype	24
4.3.2	Description	24
4.3.3	Parameters.....	24
4.3.4	Return value	24
4.3.5	Example usage.....	24
4.4	ChangeConfiguration.....	25
4.4.1	Prototype	25
4.4.2	Description	25
4.4.3	Parameters.....	25
4.4.4	Return value	25
4.4.5	Example usage.....	26
4.5	RSSI2dBm	26
4.5.1	Prototype	26
4.5.2	Description	26
4.5.3	Parameters.....	26
4.5.4	Return value	26
4.5.5	Example usage.....	26
5	Graphical User Interface	26
5.1	Command Set	26
5.2	info	27
5.2.1	Description	27
5.2.2	Parameters.....	27
5.2.3	Return value	27
5.2.4	Example usage.....	27
5.3	resp	27

5.3.1	Description	27
5.3.2	Parameters.....	27
5.3.3	Return value	27
5.3.4	Example usage.....	28
5.4	cfgs.....	28
5.4.1	Description	28
5.4.2	Parameters.....	28
5.4.3	Return value	28
5.5	pget.....	28
5.5.1	Description	28
5.5.2	Parameters.....	28
5.5.3	Return value	28
5.6	pset.....	29
5.6.1	Description	29
5.6.2	Parameters.....	29
5.6.3	Return value	29
5.6.4	Example usage.....	29
5.7	states	29
5.7.1	Description	29
5.7.2	Parameters.....	29
5.7.3	Return value	29
5.8	state.....	30
5.8.1	Description	30
5.8.2	Parameters.....	30
5.8.3	Return value	30
5.9	nodes.....	30
5.9.1	Description	30
5.9.2	Parameters.....	30
5.9.3	Return value	30
5.10	idset.....	30

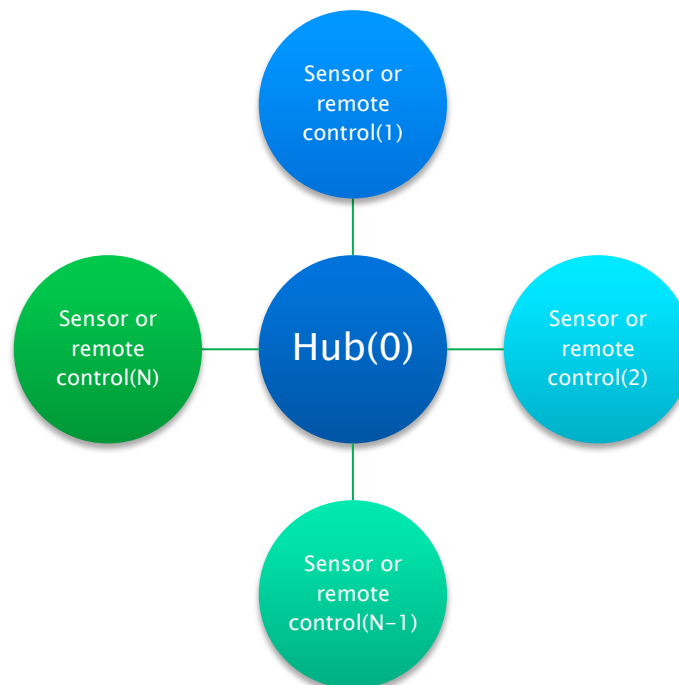
5.10.1	Description	30
5.10.2	Parameters.....	30
5.10.3	Return value	31
5.11	pmask	31
5.11.1	Description	31
5.11.2	Parameters.....	31
5.11.3	Return value	31
5.12	ndel.....	31
5.12.1	Description	31
5.12.2	Parameters.....	31
5.12.3	Return value	31
5.13	time	31
5.13.1	Description	31
5.13.2	Parameters.....	31
5.13.3	Return value	32
5.14	dset.....	32
5.14.1	Description	32
5.14.2	Parameters.....	32
5.14.3	Return value	32
5.15	ddef	32
5.15.1	Description	32
5.15.2	Parameters.....	32
5.15.3	Return value	32
5.16	Responses	32
5.17	Terminal Window	33
5.17.1	Terminal commands	33
6	History	34

This page intentionally left blank.

1 Introduction

1.1 Overview

The AIR (Anaren Integrated Radio) BoosterStack (ABS) is a small memory model wireless stack. The protocol is suitable for Point-to-Point (P2P) or Point to Multipoint (star network) wireless communication. It is specifically designed for sensor and remote control applications. ABS was developed for Texas Instruments MSP430 value line processors, but can easily be adapted to other MSP430 processors.



ABS operates on sensor node actions. This type of operation allows for low current consumption of sensor nodes, but puts an “always listening” requirement on the hub node.

The node that plays the role of a hub, which is the center of a star network or one side of P2P network, draws approximately 20mA continuous current while in receive. During transmission, one will see brief spikes of up to 40mA. Other factors of power consumption include the number of sensors in the network, how frequent they operate, and the data rate. The sensor node’s current draw depends on settings (data rate and node update rate) and **can be less than 1mA average** for infrequent updates and high speed data rates. During on times for the sensor the current draw spikes to approximately 40mA (transmit) and 20mA (receive).

ABS and the AIR Traffic Control (ATC) software provide an example of wireless communication for demonstration and evaluation purposes only. ABS is not intended to solve a specific application requirement. It was created to demonstrate wireless networks. As such, ABS contains source code dedicated purely to interfacing with the ATC software using a defined set of commands.

1.2 Purpose of this document

The purpose of this document is to provide a deeper meaning and understanding to the end user about functions that make up the AIR BoosterStack protocol. It also provides details on how to interact with the protocol for development of custom applications.

1.3 Intended Audience

This document is intended for wireless application developers and programmers who want an understanding of how the AIR BoosterStack protocol works. It will provide additional information about BoosterStack protocol operation.

1.4 Basic Operation

BoosterStack data exchanges are controlled by the sensor nodes. All data exchanges start with a sensor node sending data updates to the hub. The hub will verify if the sensor is a part of its network. If the sensor node is part of the network, the hub sends an acknowledgement to the sensor node that sent the data. If any outgoing data is available, it will be included in the acknowledgement. The hub cannot send data to the sensor node without receiving data from the sensor. Therefore, the sensor node must poll for any updates coming from the hub node.

For sensor applications and some remote control applications a slow updating rate is typically acceptable. If a faster update rate is required, it can be configured by updating the frequency of sensor data exchanges.

The stack includes methods to change logical channel and output power. This can be done during a normal data exchange between a sensor and a hub. However, if the hub wants to initiate a data exchange, it will have to wait for the sensor to send data and request the change as part of the acknowledgement. The logical channel change keeps the sensor and hub in active communication until they both successfully change or the operation times out.

Note that ABS does not include a “listen before talk” (LBT) method. It also does not include adaptive frequency agility (AFA). ETSI requires LBT and AFA for unrestricted duty cycling. ABS uses under ETSI regulations must observe duty cycling restrictions.

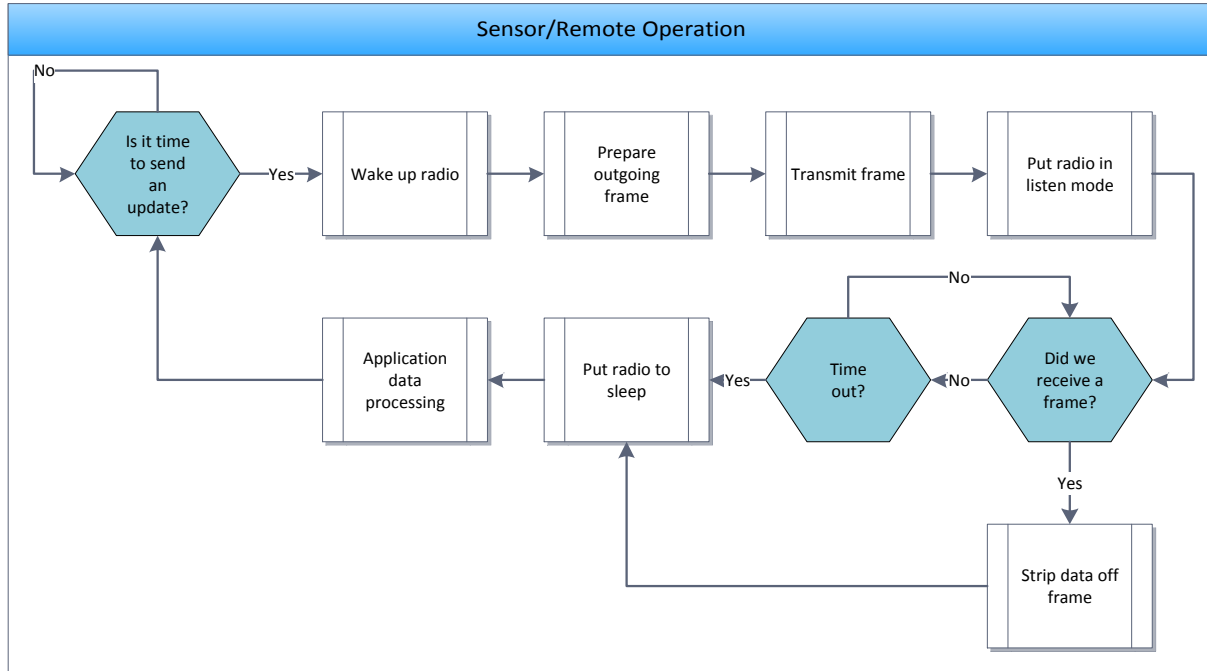


Figure 1 Sensor/remote node behavior

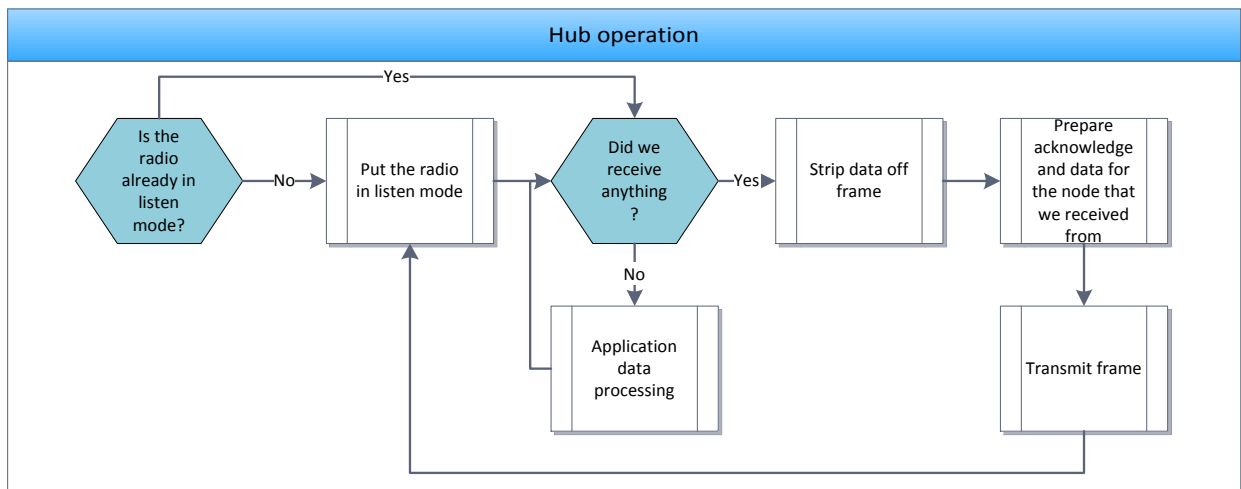


Figure 2 Hub node behavior

1.5 Physical Layer and Logical Channels

ABS uses four parameters to define the physical communication channel, which are: Logical Radio (R), Radio Configuration Settings (S), Frequency (C), and Transmit Output Power (P).

Each RSC combination is referred to as a logical channel. Different power settings can create variations of a logical channel.

A combination of the four parameters is referred to as RSCP. This defines a physical channel. Certain sets of combinations define pre-certified FCC or ETSI physical channels. To comply with ETSI there are also specific duty cycle restrictions that must be considered.

1.5.1 Logical Radio (R)

The logical radio can be either a physical radio, as in an A1101R09A, or it can be a subdivision of a physical radio, as is the case for the A110L09A. The A110L09A spans both the ETSI 868MHz band and the FCC 915MHz band. Multiple logical and physical radios can be defined (if memory allows), but only one can be operated at any given time.

1.5.2 Radio configuration settings (S)

The register settings that are loaded into the radio define a configuration. It contains information about the radio data rate, modulation form, and specific LNA, VCO and PA operations. These register configuration settings must be used in their entirety and may not be modified in order to maintain proper operation under the appropriate radio emission regulations.

1.5.3 Frequency, channel within configured band (C)

The frequency of the transmitter/receiver is often referred to as a channel. To abide by ETSI regulations, channels may not be changed. To comply with FCC, the frequencies may be changed, but may not exceed the lower and upper limits provided in the respective radio modules user guide. We recommend contacting AIR@anaren.com if more channels are needed.

1.5.4 Transmit output power (P)

Transmit power is the actual power level that the module produces at its output. The receiver is not affected directly by this setting. It is however, indirectly affected because transmission power affects the incoming signal strength. Reducing the power is useful when multiple nodes are in close proximity to avoid receiver saturation. Transmission power can also be reduced to lower current consumption during transmission.

1.6 Demo Application Overview

The intention with the included demo application is to illustrate the following:

- Pairing of nodes
- Basic data transfer of sensor data
- Basic remote control
- Wireless link signal strength

The demo application is visualized through a graphical interface application and is meant as an evaluation tool only.

The demonstration application can be setup to comply both with radio regulation rules for the European Union (ETSI) and North America (FCC/IC). It is the responsibility of the user to select only the settings that apply for their region.

2 Software Modules

The software is made up of three major parts:

1. The AIR Framework, which includes Board Support Packages (BSP), Drivers, and a Hardware Abstraction Layer (HAL). The AIR Framework allows the same code to be compiled for different hardware platforms, such as different microprocessors, radios, or application PCBs. The AIR Framework is described in more detail in the document "AIRFrameworkOverview.pdf".
2. The wireless protocol, which handles all radio operations and basic data exchanges. The protocol functions are described below in this document. The protocol interfaces through the AIR Framework to the physical hardware and with the application to take and deliver data etc.
3. The demo application running on top of the protocol.

2.1 Development platform

This software was developed using Code Composer Studio IDE (CCS) and IAR Embedded Workbench IDE (IAR). It supports the MSP430 Value Line microprocessors. This software can be extended for use with other compatible microprocessors.

2.2 Compiler Differences

CCS and IAR compile and optimize software differently. Due to this, the BoosterStack has memory constraints that differ based on the compiler used. These memory constraints affect how many nodes a network could potentially have. The BoosterStack is also limited by the platform that it is running on. If the microcontroller supports more than 16KB of memory, or the user does not desire to use the *CommandInterface* module to interact with the Anaren Traffic Control software, then more nodes may be added to the network. The number of nodes is solely based on memory limitations. The included projects, IAR and CCS, are using **high** optimization that favors **size** over *speed*. The following table describes the out-of-box experience provided by the AIR BoosterStack. It describes the amount of nodes that can be in one network.

	Max # of Hub Nodes	Max # of Sensor Nodes	Max # of Nodes
IAR Compiler	1	4	5
CCS Compiler	1	2	3

2.3 Loading template project

2.3.1 Code Composer Studio

To open the ABS project in CCS, open CCS and browse to the Firmware directory (provided by the CD) for your workspace location. The CCS project is located in Firmware\CCS. Next, select “Project” in the main menu bar and choose “Import Existing CCS/CCE Eclipse Project”. At this point, you should see the following screen:

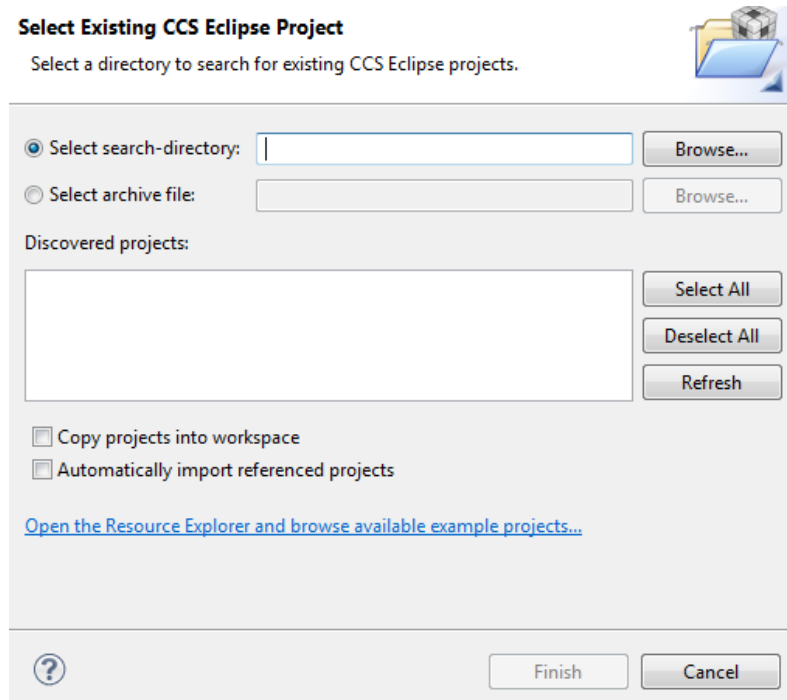


Figure 3 Load Existing BoosterStack Project

In the “Select search–directory:” field, select browse and choose the Firmware directory. This will allow CCS to search in that directory for any CCS projects. Once you’ve selected the Firmware directory, CCS should appear in the “Discovered projects:” window. Make sure CCS is selected, and click “Finish”.

If done correctly, you should now see the CCS AIR BoosterStack project in the “Project Explorer” of Eclipse. If you cannot see the “Project Explorer” in your immediate window, you can display it by going to “Window” > “Show View” > “Project Explorer”.

Now that the project is loaded, you should be able to choose your build configuration and debug the application.

2.3.2 IAR Embedded Workbench

To open the BoosterStack project in IAR, simply navigate to the Firmware\IAR directory and open the workspace file (.eww).

When the workspace is loaded and the project is open in the IAR tool, it will look similar to the figure below:

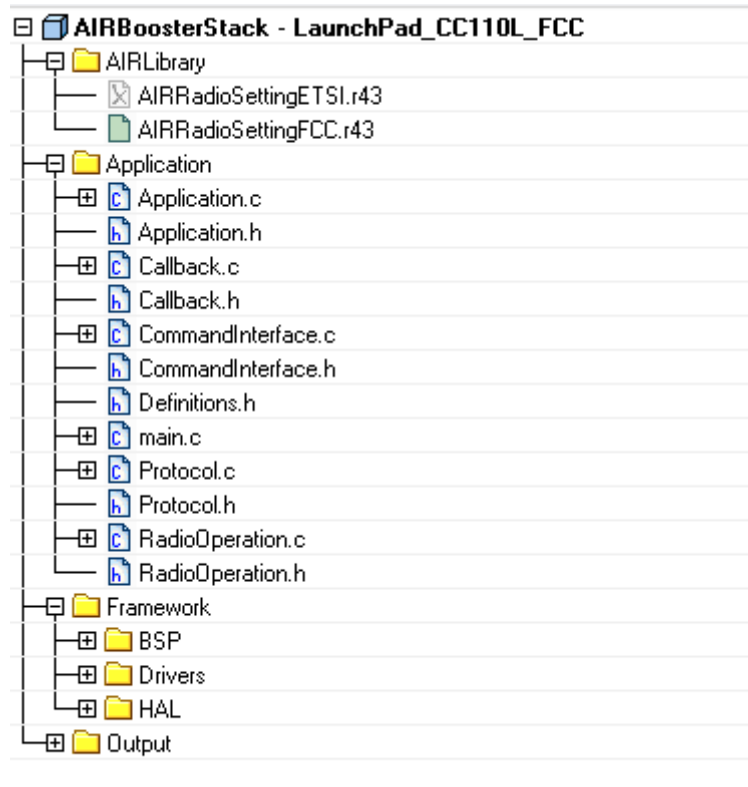


Figure 4 IAR project structure

Now that the project is loaded, you should be able to choose your build configuration and debug the application.

3 Application Layer

BoosterStack consists of a few basic functions that allow for data exchanges between a hub and sensor nodes. The *application layer* consists of the C module “Application”. The following sections describe the functions that make up this layer.

3.1 NodeInfo

For the application demonstration one structure contains all data that is pertinent to the network:

```
typedef struct NodeInfo
{
    unsigned long ID;
    unsigned int CurrentCycleTime;
    unsigned int InData[DATA_CHANNELS];
    unsigned int OutData[DATA_CHANNELS];
    unsigned char LastRSSI;
} NodeInfo;

NodeInfo Nodes[MAX_NODES] = {0};
```

The maximum number of nodes in a network is dictated by platform memory constraints. The BoosterStack network is composed of a single hub and one-to-many sensor nodes. For the sensor/remote only two locations are used (0 itself and 1 the paired hub). If more nodes are added then attention must be paid to system duty cycling for applications deployed where ETSI radio emission rules apply.

Each node in the network has a unique ID, which is comprised of a 32-bit value (## ## ## ##)_h. The range of values for this ID are (00 00 00 01)_h to (FF FF FF FE)_h.

Note An ID value of (00 00 00 00)_h and (FF FF FF FF)_h are **reserved** and should not be used.

The `CurrentCycleTime` contains the currently used data update rate, i.e. how frequently each sensor sends/receives data from the hub. This variable should not be modified directly, but through associated functions only. The time is in milliseconds.

`InData[]` contains the data that is sent from a remote node to this node, regardless of sensor or hub status. `InData` is always data received from the wireless link from the node with this ID.

`OutData[]` is the equivalent for any data scheduled to be transmitted. At the time of transmission this data is sent as a payload and thus whatever was in it at the time will be transmitted to the node ID for which this is stored.

`LastRSSI` holds the signal strength of the remote node. The RSSI value is updated after receive and transmit cycle, i.e. this is always 1 transmission behind. The exception is that for each received data `Nodes[0].LastRSSI` is always updated with the signal strength that this node (0) saw.

In and out data is organized as an array of 16 bit integers and can be used in any way the application desires. For the demo these holds temperature and bit values for controlling/displaying LED's

After each successful reception of data the `Nodes[]` array is updated, thus if the values have long term meaning to the application they must be copied elsewhere.

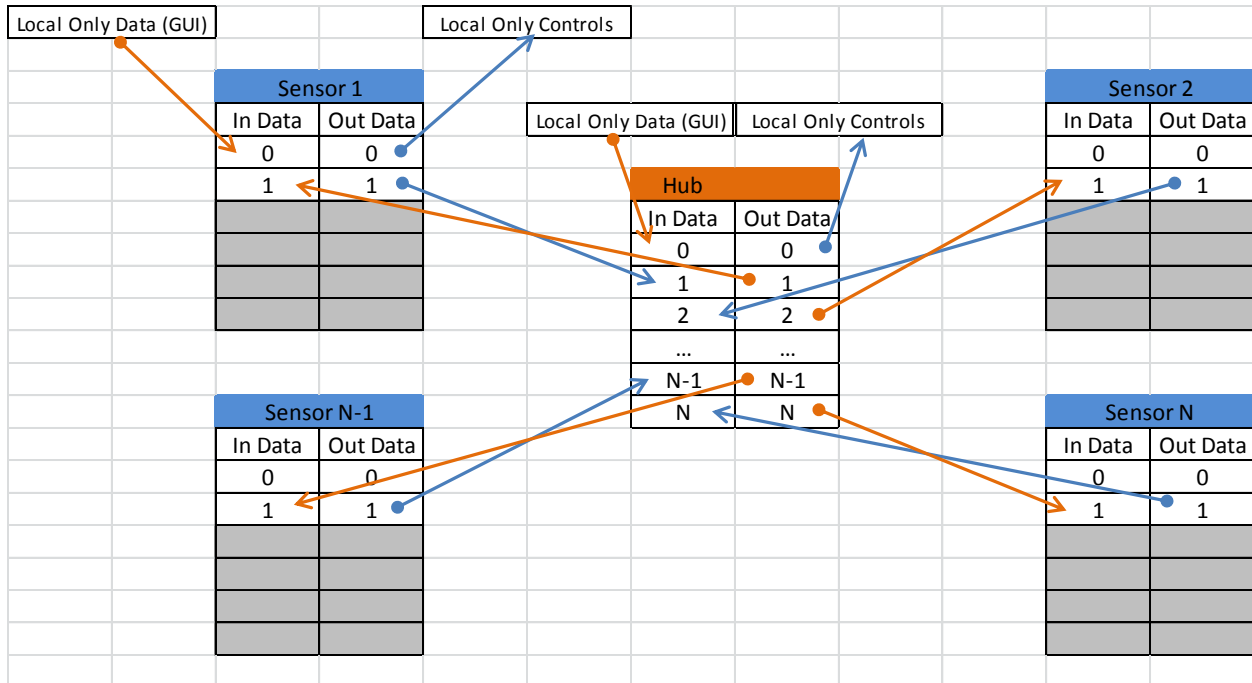


Figure 5 Exchanged data illustration

3.2 HubWirelessOperation

3.2.1 Prototype

```
uint8_t HubWirelessOperation(void);
```

3.2.2 Description

This wireless operation performs a continuous listen operation on a *hub network node*. On any received data, this operation will prepare an acknowledge message for the associated sensor node. This acknowledgement may contain available outgoing data meant for the paired node.

3.2.3 Parameters

There are no parameters.

3.2.4 Return value

Returns a **uint8_t** value representing receive status of the hub after performing a continuous listen operation.

Return value	Definition
0x00	The hub node has timed out listening for incoming data from a sensor node.
0xN	Data has been received from sensor node _N , where N is the index of the node. This

can only happen if sensor node_n has been previously paired with the hub, or a pairing session is in progress.

3.2.5 Example usage

The following code snippet gives an example of how HubWirelessOperation should be used:

```
void main()
{
    uint8_t nodeIDIndex = 0;
    ...
    // Run main application.
    while (1)
    {
        nodeIDIndex = HubWirelessOperation();
        if (nodeIDIndex > 0)
        {
            // Logic to handle receive data events from paired nodes.
        }
    }
}
```

3.3 SensorWirelessOperation

3.3.1 Prototype

```
uint8_t SensorWirelessOperation(void);
```

3.3.2 Description

This wireless operation performs the primary *sensor/remote control node* function of waking up the radio and sending out a data update. The radio wakes up and goes into active state from its normal sleep state (~200nA). It will then listen for an acknowledgement from a hub node.

3.3.3 Parameters

There are no parameters.

3.3.4 Return value

Returns a **uint8_t** value representing the status of receiving an acknowledgement from the hub node.

Return value	Definition
0x00	The sensor node has timed out without receiving an acknowledgement from the hub node, or has received a message from an unpaired node.
0x01	The acknowledgement was within the specified timeout period and is from the paired hub node. 0x01 represents the node index of the paired hub.

3.3.5 Example usage

The following code snippet gives an example of how SensorWirelessOperation should be used:

```
void main()
{
    uint8_t nodeIDIndex = 0;
    ...
    // Run main application.
    while (1)
    {
        nodeIDIndex = SensorWirelessOperation();
        if (nodeIDIndex == 1)
        {
            // Logic to handle data received from paired hub.
        }
    }
}
```

3.4 SetMyNodeID

3.4.1 Prototype

```
unsigned char SetMyNodeID(unsigned long NodeID);
```

3.4.2 Description

Assigns a user specified network identification number to the associated node. If flash storage is enabled, which it is by default, then the assigned node ID will be stored in flash memory and will persist until it is explicitly changed.

3.4.3 Parameters

Data type	Definition	Range of values
unsigned long	A 32-bit sequence representing a node identification number for the network.	{0x00000001 – 0xFFFFFFFF}

3.4.4 Return value

Returns a **unsigned char** value representing whether the operation successfully assigned a new identification number to the node.

Return value	Definition
0x00	The node was not assigned a new identification number. The previous ID will remain.
0x01	The node's ID was successfully changed.

3.4.5 Example usage

There is no example usage.

3.5 SetPairingMask

3.5.1 Prototype

```
void SetPairingMask(unsigned long mask);
```

3.5.2 Description

Controls the network pairing mechanism. By default, the pairing mask is set to zero, which prevents any node from pairing with this node. Once a mask of non-zero value is applied, devices that fall inside the mask (bit pattern) may pair. Once a node becomes paired, the node IDs are available for the application to query using `Nodes[idIndex].ID`. The `Nodes` array is not sorted.

There is an associated `unsigned long GetPairingMask(void)` function to query for the current pairing mask.

3.5.3 Parameters

Data type	Definition	Range of values
<code>unsigned long</code>	A 32-bit sequence representing a node pairing mask for the network.	{0x00000000 – 0xFFFFFFFF}

3.5.4 Return value

There is no return value.

3.5.5 Example usage

There is no example usage.

3.6 RemoveNodeFromNetwork

3.6.1 Prototype

```
unsigned char RemoveNodeFromNetwork(unsigned char NodeIndex);
```

3.6.2 Description

Removes a paired node from the list of paired nodes. This action is performed when a node is no longer desired in the network.

3.6.3 Parameters

Data type	Definition	Range of values
unsigned char	An index representing a node in the list of paired nodes Nodes[] .	{1 - 255}

3.6.4 Return value

Returns a **unsigned char** value representing whether the operation was successful.

Return value	Definition
0x00	The node did not exist or was not removed from the Nodes array.
0x01	The paired node was removed successfully.

3.6.5 Example usage

There is no example usage.

4 Protocol Layer

The *protocol layer* is comprised of two C modules: "Protocol" and "RadioOperations".

4.1 InitProtocol

4.1.1 Prototype

```
void InitProtocol(void);
```

4.1.2 Description

Sets up the protocol for normal operation. *This function must be called prior to any interactions with the protocol.*

4.1.3 Parameters

There are no parameters.

4.1.4 Return value

There is no return value.

4.1.5 Example usage

There is no example usage.

4.2 DataExchange

4.2.1 Prototype

```
uint8_t DataExchange(uint8_t * pBuffer, uint8_t * count);
```

4.2.2 Description

Takes a user provided buffer and copies the amount of data desired by a count to the transmit frame in the protocol. It then wakes up the radio to transmit the frame. The radio will then move into a receive state to listen for incoming frames. The node will listen until either a frame is received or a timeout period is reached. Once either of these occurs, the radio is put back into a sleep state.

If a frame is received, the user buffer will be overwritten with data from the received frame. The count will be updated to reflect the amount of data received. Therefore, the user buffer must be large enough to handle holding any incoming frame in the network.

Any non-local logical channel change requests are also handled by `DataExchange` as a variation of a normal data exchange.

4.2.3 Parameters

Data type	Definition	Range of values
uint8_t *	Pointer to a user buffer. The buffer is used for transmitting data and receiving data in a node data exchange operation.	Address to a buffer that is large enough to hold the largest receive/transmit frame in the network.
uint8_t *	Pointer to a count representing the amount of data in the buffer.	{0 – (maxFrameSize–header)}

4.2.4 Return value

Returns a `uint8_t` value representing whether the data exchange operation was successful.

Return value	Definition
0x00	No data is available from the data exchange.
0x01	Data is available from the data exchange. The user buffer has been filled with the received data and the count has been updated with the number of bytes.

4.2.5 Example usage

There is no example usage.

4.3 Listen

4.3.1 Prototype

```
uint8_t Listen(uint8_t * pBuffer, uint8_t * pCount);
```

4.3.2 Description

Checks if a frame has been received. If a frame is received and from a paired node, an acknowledgement is prepared and any data destined for that node is sent out.

Any non-local logical channel change requests are also handled by this function as a variation of its normal operation.

Upon exit the radio is left listening for incoming frames.

4.3.3 Parameters

Data type	Definition	Range of values
uint8_t *	Pointer to a user buffer. The buffer is used for receiving data in a node data exchange operation.	Address to a buffer that is large enough to hold the largest receive frame in the network.
uint8_t *	Pointer to a count representing the amount of data in the buffer.	{0 - (maxFrameSize-header)}

4.3.4 Return value

Returns a **uint8_t** value representing whether the listen operation was successful and if so, the amount of data received that is available to evaluate.

Return value	Definition
0x00	No data is available from the listen operation.
0x01	Data is available from the listen operation. The user buffer has been filled with the received data and the count has been updated with the number of bytes.

4.3.5 Example usage

There is no example usage.

4.4 ChangeConfiguration

4.4.1 Prototype

```
uint8_t ChangeConfiguration(uint8_t remote, uint8_t radioIndex, uint8_t configIndex,
uint8_t channelIndex, uint8_t powerIndex);
```

4.4.2 Description

Initiates a logical channel change. If “remote” is true, the change will be coordinated with a specified remote radio. If “remote” is false, it changes only the local logical channel.

A local change of logical channel takes effect immediately, but a coordinated change with a remote node will not take effect until that node sends a data update.

The following functions can retrieve information about available radios:

- `uint8_t CurrentRadio(void)`, returns the current local logical radio index.
- `uint8_t CurrentBaseConfig(void)`, returns the current local radio register configuration index.
- `uint8_t CurrentChannel(void)`, returns the current local frequency index.
- `uint8_t CurrentPower(void)`, returns the current local power setting index.

4.4.3 Parameters

Data type	Definition	Range of values
uint8_t	A Boolean value set to true if the configuration change is meant to occur on both the local and remote units.	{0 - }
uint8_t	An index to the radio that is being set.	{0 - }
uint8_t	An index to the configuration that is being set.	{0 - }
uint8_t	An index to the channel that is being set.	{0 - }
uint8_t	An index to the power setting that is being set.	{0 - }

4.4.4 Return value

Returns a **uint8_t** value representing whether the configuration change operation was successful.

Return value	Definition
0x00	An invalid parameter was found and the configuration change could not proceed.
0x01	All parameters are valid and the configuration change has been performed successfully.

4.4.5 Example usage

There is no example usage.

4.5 RSSI2dBm

4.5.1 Prototype

```
signed int RSSI2dBm(unsigned char rssi);
```

4.5.2 Description

Converts a radio RSSI value to a signed 16-bit fixed point number in dBm. The signed 16-bit fixed point value is represented by an 8-bit signed integer and an 8-bit decimal fraction.

4.5.3 Parameters

Data type	Definition	Range of values
unsigned char	RSSI value that is read and stored by the local radio.	{0 - }

4.5.4 Return value

Returns an **int16_t** value representing the RSSI value converted to dBm.

Return value	Definition
signed int	A signed 16-bit fixed point number representing dBm from an RSSI value.

4.5.5 Example usage

There is no example usage.

5 Graphical User Interface

The following description of the commands that the GUI uses to control the demo application are provided here for information only and are not intended as a full description of how to use these commands through a terminal program.

5.1 Command Set

The demo application communicates to the graphical user interface through a UART (9600N8) port with the following command set functions.

5.2 info

5.2.1 Description

Calls the function `AppInformation()`.

5.2.2 Parameters

There are no parameters.

5.2.3 Return value

Returns a **string** representing application information and firmware revision.

5.2.4 Example usage

There is no example usage.

5.3 resp

5.3.1 Description

Calls the function `ResponseCtrl()`. Each bit turns on/off a specific output type from the embedded application, the Flag Bits functions as a mask and any set bits in the Flag bits will be changed to the bit value provided in Bit Value. Thus allowing multiple changes or single changes.

5.3.2 Parameters

Flag bits: 0=Debug 1=Status 2=Data 3=Message 4=Network 5=AppCHg 6=Physical 7=XML; (value of same bit) 0=off 1=on". These flags control what information is allowed to be output from the embedded code.

Parameterized format: `"\resp(Flag Bits)##(Bit Values)##\"`.

5.3.3 Return value

Returns **different types** of information depending on the parameters passed.

Debug are the outputs generated by using the `unsigned char DebugNote(char * Desc)` function specifically in the code.

Status is the updates of received signal strengths.

Data is the received/updated data.

Message is not currently implemented.

Network updates occur during pairing, when the pairing mask is changed and when a node is either added or removed from the member list.

AppChg occurs if the functional application state changes.

Physical occurs when a change is made to the logical channel (RSCP).

XML makes all outputs in XML segments, this is useful for interpretation of responses. In non-XML an abbreviated comma separated listing is used, which is more efficient but is non-descriptive.

5.3.4 Example usage

`respFF7E`, turns on all but debug information and XML format and clears debug information and XML format.

`resp8080`, turns on only XML format.

5.4 cfgs

5.4.1 Description

Calls the function `ConfigurationInfo()`.

5.4.2 Parameters

There are no parameters.

5.4.3 Return value

Returns a **string** of data listing all physical channel configuration information. From this, all valid RSCP index configurations can be derived.

5.5 pget

5.5.1 Description

Calls the function `GetCurrentPhysical()`.

5.5.2 Parameters

There are no parameters.

5.5.3 Return value

Returns a **string** representing the current logical channel information (RSCP indexes).

5.6 pset

5.6.1 Description

Calls the function `SetCurrentPhysical()`.

5.6.2 Parameters

ID index – index into node list, for which this should take effect. “00” is local only and “FF” is for all paired nodes (not currently implemented).

Radio index.

Radio configuration settings index.

Frequency index.

Power index.

Parameterized format: `"\pset (ID index)##(radio)##(config)##(ch)##(power)##\";` (ID index) 0=local; FF=all; other=node and local".

5.6.3 Return value

There is no return value.

5.6.4 Example usage

`pset0001030600`, sets local only to radio 1 (915MHz band), configuration 3 (250kb/s), frequency number 6 (912MHz) and max power. This command also stores in flash memory the RSCP info, such that a node will start up in its most recent logical channel if power cycled or reset.

5.7 states

5.7.1 Description

Calls the function `CmdIF_GetAppStates()`.

5.7.2 Parameters

There are no parameters.

5.7.3 Return value

Returns a **string** representing a list of all possible application states.

5.8 state

5.8.1 Description

Calls the function `CmdIF_SetAppState()`. This command changes the active application, either sensor or hub. Note that changes to this eradicates the network member list i.e. any pairing. This command also stores in flash memory the application state, such that a node will start up in its most recent application if power cycled or reset.

5.8.2 Parameters

Application state index.

Parameterized format: `"\state##\"`.

5.8.3 Return value

There is no return value.

5.9 nodes

5.9.1 Description

Calls the function `CmdIF_GetNodeIDs()`.

5.9.2 Parameters

There are no parameters.

5.9.3 Return value

Returns a **string** representing a list of all nodes in network. This list has only those with a non-zero ID, which make up valid nodes. This information is used to derive the ID indexes that other commands use to address specific nodes in the member list.

5.10 idset

5.10.1 Description

Calls the function `CmdIF_SetThisNodeID()`.

5.10.2 Parameters

32-bit ID for the local node.

5.10.3 Return value

There is no return value.

5.11 pmask

5.11.1 Description

Calls the function `CmdIF_SetPairingMask()`.

5.11.2 Parameters

A 32-bit pairing mask. Use this command to allow pairing by setting the mask to a non-zero number (FFFFFFFF to allow pairing with any ID) and use this command again to set the mask to zero to dis-allow pairing

5.11.3 Return value

There is no return value.

5.12 ndel

5.12.1 Description

Calls the function `CmdIF_RemoveNode()`.

5.12.2 Parameters

An ID index of a node in the member list. The specified node will be removed from the list of paired members of the network.

5.12.3 Return value

There is no return value.

5.13 time

5.13.1 Description

Calls the function `CmdIF_SetCycleUpdateRate()`. Sets the interval between sensor updates from the specified node.

5.13.2 Parameters

An ID index of the node for which we wish to set the update rate and the interval time.

Parameterized format: `"\time (ID index)##(time ms)####\"`.

5.13.3 Return value

There is no return value.

5.14 dset

5.14.1 Description

Calls the function `CmdIF_SetDataValue()`. Sets the value for specified (abstract) channel on specified node. Use this command for instance for remote control on a sensor node.

5.14.2 Parameters

An ID index of the node for which to set the specified data, data channel to be used (this must be a valid channel as provided by the `ddef` command, and a value to set the channel to.

Parameterized format: `"\dset (ID index)##(channel)##(value)####\"`.

5.14.3 Return value

There is no return value.

5.15 ddef

5.15.1 Description

Calls the function `CmdIF_DataDefinitions()`.

5.15.2 Parameters

There are no parameters.

5.15.3 Return value

Returns a string representing a list of all (abstract) data channels that can be manipulated. This command defines what exchanged data really means for a node.

5.16 Responses

There are two types of responses that may come from the demo application to the GUI. The first one is a response to a command issued and the second is a spontaneous response caused by a specific event in the demo application (could for instance be arrival of new data). The description here of the responses are for information purposes only and are not intended as a full description for using the demo application with a terminal window.

There are two types of output formats for the responses. The first is in XML segments (note it is not in fully compliant XML, but in “snippets”). Numbers in XML format are decimal. The other format is a comma separated abbreviated response that is more efficient, but hard to interpret. Numbers in abbreviated format are hexadecimal. Below the specific response format will not be described, just a brief description.

- **Command** responses to a command issued begins with repeating the command as interpreted, followed by any data as a result of executing the command. If the response has either “#” or “err” then the command failed.
- **Debug** responses consist of a message number and a text string describing what the event that caused it was. These are caused directly by program execution and is provided to allow debugging or notifications of specific events not covered elsewhere.
- **Status** responses are caused by radio receive activity. The response contains sets of ID index (the source of the event) and RSSI (received signal strength).
- **Data** responses are caused either programmatically or by radio receive activity. The response consists of sets of ID index (the source of the data), data item number (use the “ddef” command for interpretation) and value of the data item.
- **Network** responses are caused by changes to the network member list or pairing status. This response will output first the current pairing mask, then the entire list of node ID’s, including blank spots (ID=0).

5.17 Terminal Window

The following commands allow a user to interface with the demo application through a terminal window (9600N8):

5.17.1 Terminal commands

- `info`, use this command to see if there is “hole through”
- `resp7F7F`, allows all responses and sets the output format to abbreviated
- `state01`, Sets the demo application into “hub” mode

Now you should see periodic data responses, these are temperature updates from the hub itself

- `maskFFFFFFFF`, sets the pairing mask to wide open.

Now on a remote node (in sensor mode and on the same RSCP as the hub) press and hold down the pairing button until you see a Network response. Now you should see periodic status responses and data responses.

6 History

Date	Author	Change Note No./Notes
19 Oct 2011		Initial release
18 Jan 2012		Added information about CCS support. Removed GUI commands that are no longer used.