

# iDwaRF-Net User Manual

This document is intended to be a guide to using the iDwaRF-Net firmware and to adapting the firmware to own application requirements.

## 1 Introduction

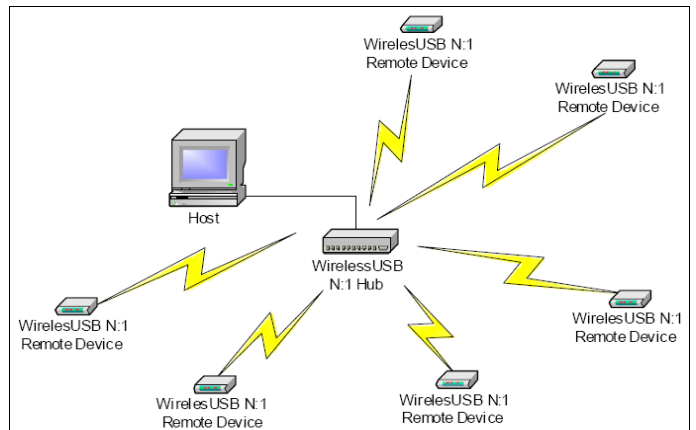
The primary purpose of the iDwaRF-Net firmware is to provide a user-friendly software basis for implementing wireless multipoint-to-point (N:1) applications. In combination with the iDwaRF-168 programmable radio module, an easy-to-use wireless application platform is available. The user can concentrate on the actual application development without the need to immerse oneself in wireless protocol implementation details or to acquire fundamental HF design skills.

The iDwaRF-Net firmware is capable of servicing low data rate higher density node applications far beyond simple point-to-point (1:1) wireless connectivity. The protocol is designed for reliable 2-way communication between a wireless Hub and target Sensor or Actuator applications in N:1 networks. The iDwaRF-Net firmware encapsulates the complete wireless network protocol in convenient easy-to-use C-API functions.

The iDwaRF-Net firmware is based on Cypress' WirelessUSB N:1 DVK (CY3635) software and was ported from Cypress PSoC architecture to the Atmel AVR ATmega168 microcontroller.

### 1.1 iDwaRF Hub

The iDwaRF Hub forms the center of a star network that can contain a multitude of Sensors and Actuators. To facilitate the usage of extremely power-sensitive Sensors, the Hub is typically supposed to be powered by a constant external power supply. The iDwaRF Hub can interface to a local host (e.g. PC or higher level microcontroller) using a standard serial port (UART). A simple plain text (ASCII) serial protocol is used for the communication between the host and the Hub. For simple applications, the Hub might not require any host connection - the entire host application could be implemented in the iDwaRF Hub. For these purposes, the iDwaRF-Net firmware provides easy-to-use high level C-API functions.



### 1.2 iDwaRF Sensor (Remote Device)

Many iDwaRF Sensors can bind to a single iDwaRF Hub and form the N:1 wireless network. The iDwaRF-Net protocol is optimized for low-power Sensors, which predominantly stay in a power-down state and temporarily wake-up at a programmable time interval and exchange data with the hub if required. The iDwaRF Sensor can be adapted to own sensors and actuators and own tasks can be implemented by using convenient high level C-API functions.

### 1.3 iDwaRF-Net Protocol

The iDwaRF-Net protocol utilizes the unlicensed 2.4 GHz Industrial, Scientific and Medical (ISM) frequency band for wireless connectivity. The band is split into 79 distinct 1 MHz channels starting at 2.402 GHz. Each iDwaRF network uses a channel subset spread across the 2.4 GHz band such that either the probability for interference with other iDwaRF networks is minimized and the number of channels each Sensor must search to find the current channel used by the Hub. Robust DSSS communication with Pseudo Noise codes (PN codes) with minimal cross-correlation properties is less susceptible to interference caused by overlapping transmissions on the same channel. The number of frequency / code pairs is large enough for hundreds of iDwaRF modules in the same space. Communication requires the same PN code and channel to be used by all devices in a network. See "WirelessUSB LS Theory of Operation" for details. Several error correction methods are implemented in the iDwaRF-Net protocol, like chip error correction, bit error corrections, and Cyclic Redundancy Check (CRC). An ACK/NAK scheme guarantees retransmission of data packets. The complete wireless protocol is encapsulated in the iDwaRF-Net C-API functions.



For a more detailed description of the underlying protocol see "CY3635 Technical Reference Manual" and "AN033B - WirelessUSB LS Theory of Operation" by Cypress.

### 1.3 iDwaRF-Net Network Parameters

A set of parameters is related to the iDwaRF-Net protocol:

- Radio Manufacturing ID (MID): Each iDwaRF module contains a unique 4-byte MID, which is used for device identification during the bind procedure.
- Network Channel: During bind mode the Hub informs the Sensor of the current channel for data mode. Subsequently the Sensor can calculate other channels contained in the corresponding channel subset.
- Network PN Code: All packets transmitted between bound devices use a single PN code determined by the Hub.
- Device ID: During the bind procedure the Hub assigns each Sensor an 8-bit or 16-bit Device ID (depending on the firmware implementation, currently 8-bit IDs are used), which is used to identify a Sensor.
- Network Checksum Seed and CRC Seed: The parameters are used to calculate checksum values and to reduce the possibility of packets from neighboring systems being accidentally received as valid packets.
- Bind Parameters: During bind mode a fixed set of network parameters is used: CRC Seed = 0x00, Checksum Seed = 0x00, PN Code ID = 0x00, base channel = 0x00.

For a more detailed description of the network parameters see "CY3635 Technical Reference Manual" by Cypress.

### 1.4 iDwaRF-Net Packet Structures

The iDwaRF-Net firmware uses a simple packet structure for data packets, which is compatible to Cypress' original WirelessUSB protocol implementation.

The general packet structure consists of:

- 1-byte Packet Header with packet type information and status bits
- 1- or 2-byte Device ID
- N-byte packet type specific Payload Data
- 2-byte CRC Checksum
- 1-byte XOR Checksum

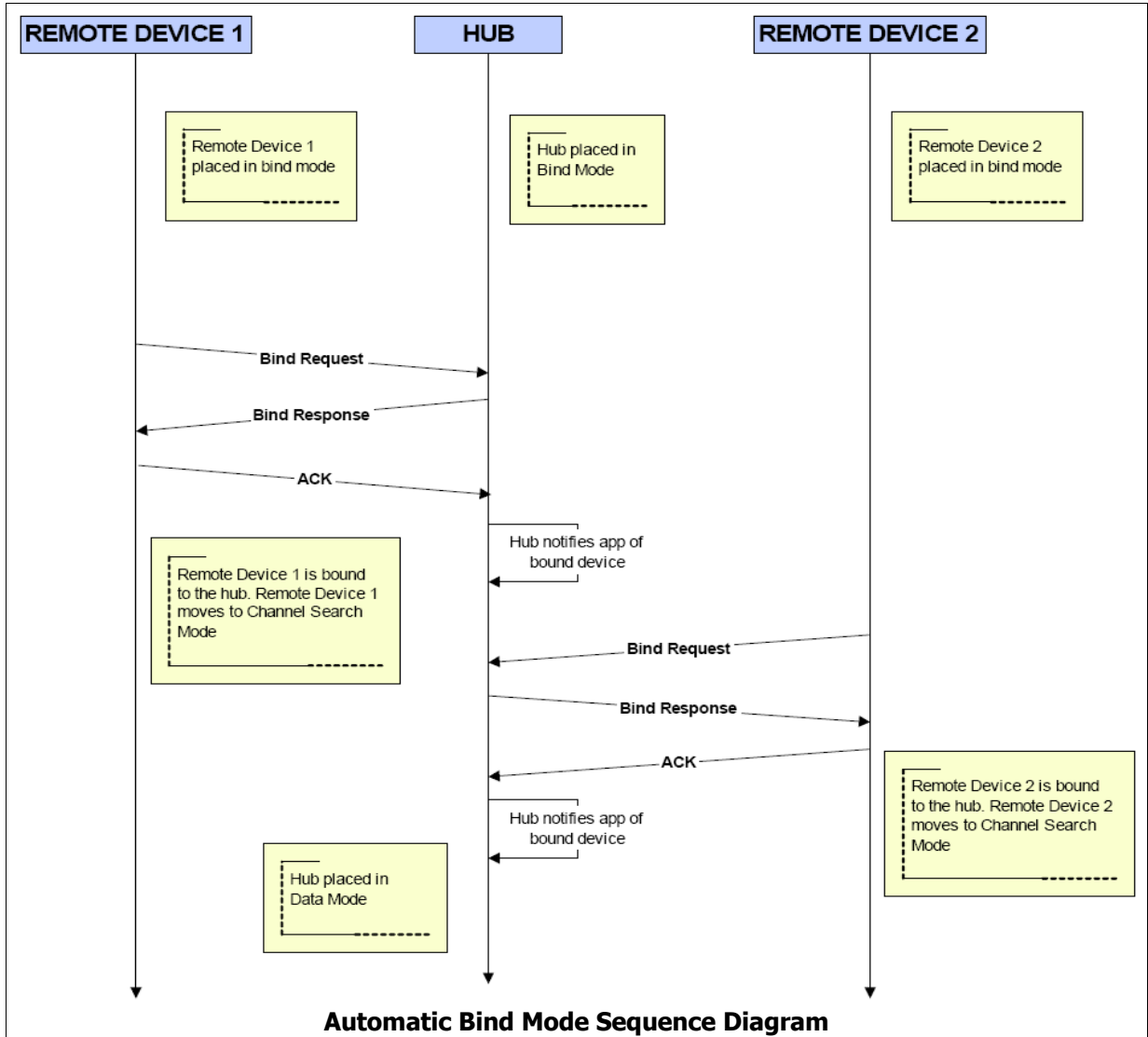
Due to the convenient encapsulation of the lower protocol layers, the user generally must not worry about packet details, but can access the Payload Data with comfortable C-API functions.

For detailed packet information see the "CY3635 Technical Reference Manual".

## 2 Operating the iDwaRF-Net

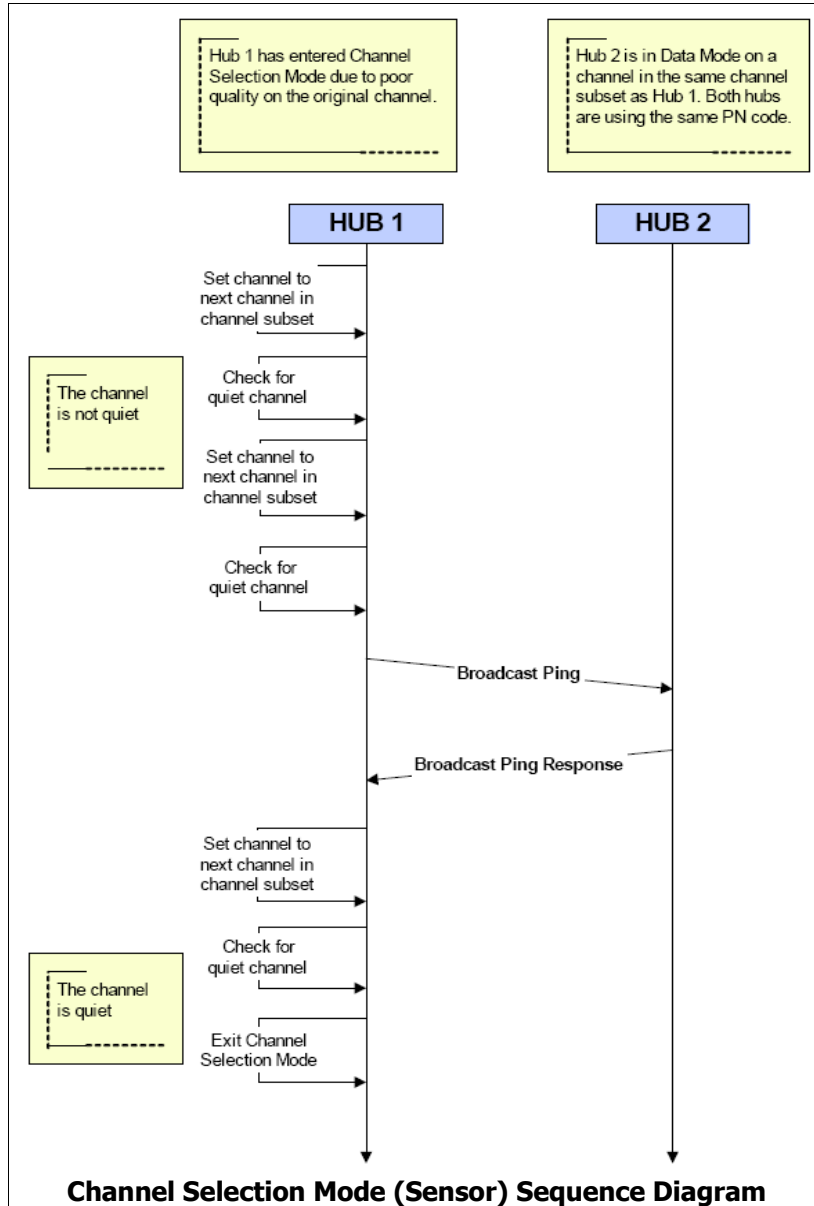
### 2.1 Automatic Bind Mode

During automatic Bind Mode the Sensors retrieve the network parameters from the Hub. The Sensor alternately transmits Bind Requests and waits for a Bind Response containing the actual set of network parameters from the Hub. If no Bind Response is received after a defined number of Bind Requests, the Sensor moves to the next channel. If a Bind Request is received, the network parameters are stored and the Sensor enters Channel Search Mode. If a defined time elapsed without receiving any Bind Response, the Sensor enters Sleep Mode.



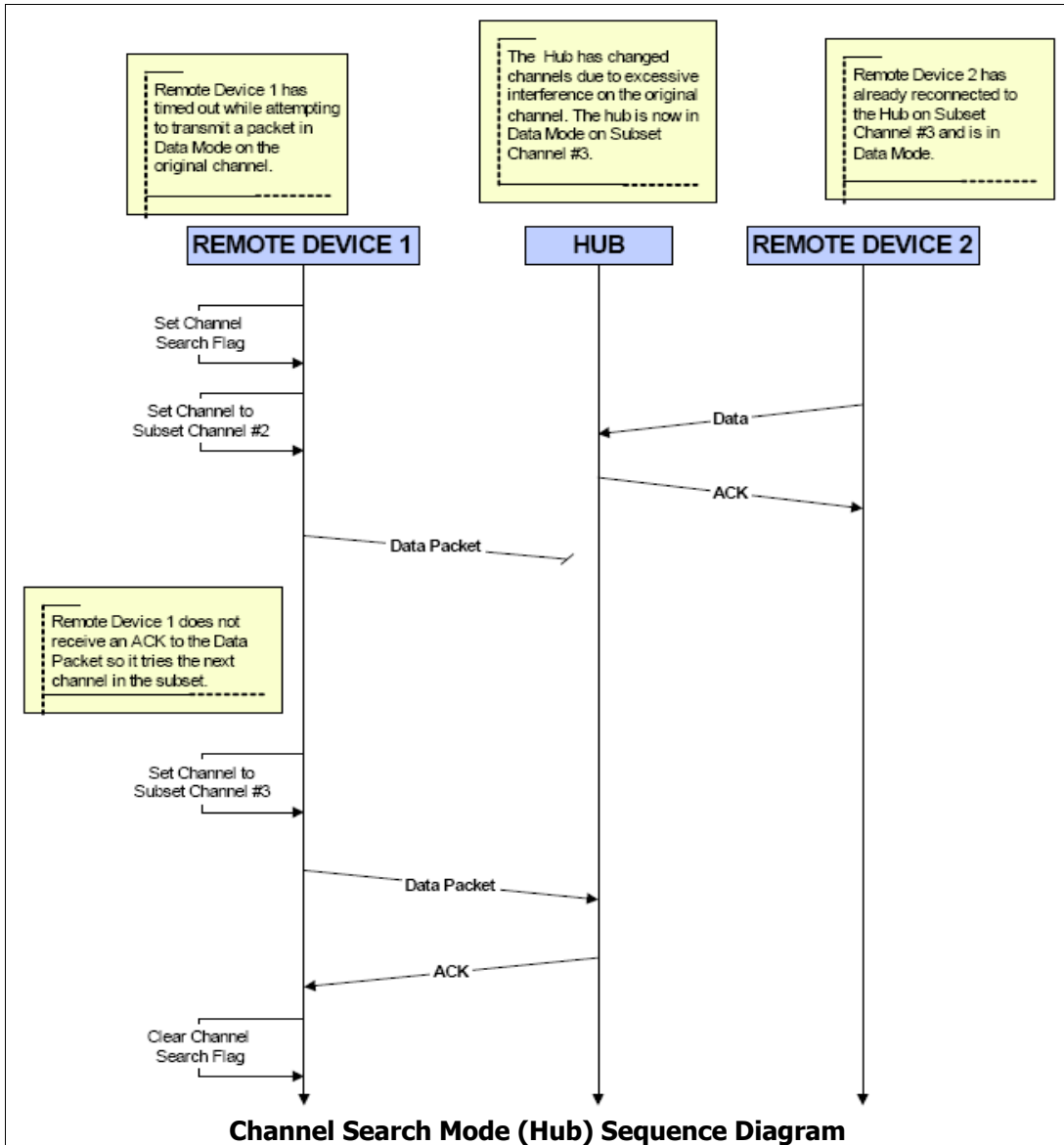
## 2.2 Channel Selection Mode (Hub Only)

Channels are unavailable if another network uses the channel with the same PN code or if there is excessive noise on the channel. A Hub checks the Receive Signal Strength Indicator (RSSI) on the radio to determine if the channel is already used by another wireless system. If not, the Hub transmits Broadcast Ping Packets on this channel and waits for Broadcast Response Packets for a defined period of time. If such a packet is received, the Hub selects another channel and repeats the procedure. Another channel is also selected, if RSSI is high, indicating an already used channel. If no Broadcast Response Packet is received and RSSI is low, the Hub assumes the channel is available and moves to Data Mode.



### 2.3 Channel Search Mode (Sensor Only)

During Channel Search Mode the Sensor tries to determine the current channel used by the Hub. The Sensor alternately transmits Data Packets with its Device ID and listens for an ACK Packet from the Hub. If no ACK Packet is received, the Sensor selects the next channel and repeats. If an ACK Packet is received the Sensor moves to Data Mode. If no ACK Packet is received on any channel, the Sensor may either continue Channel Search Mode or go to sleep and reenter Channel Search Mode later again.

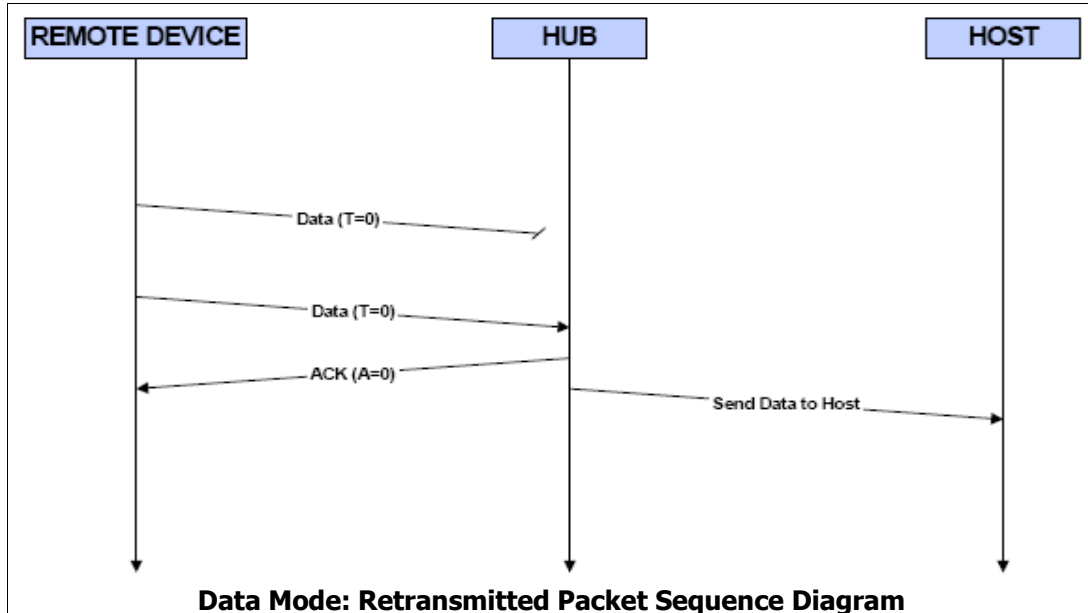
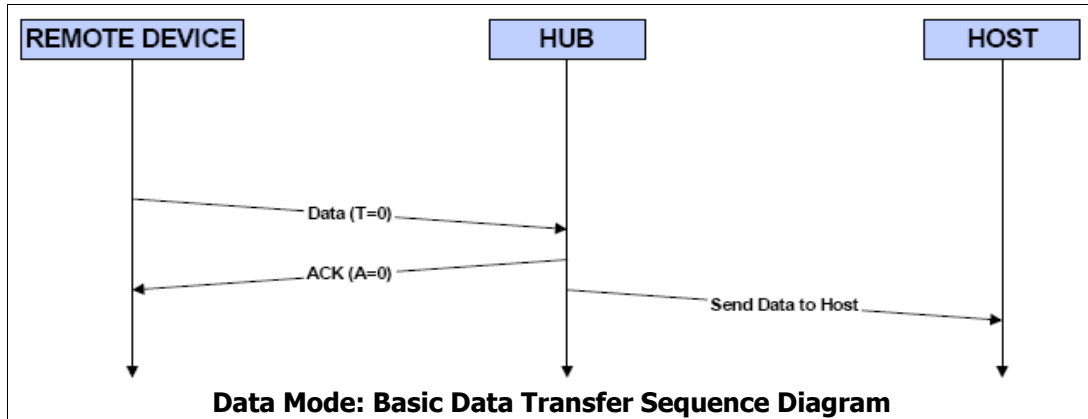


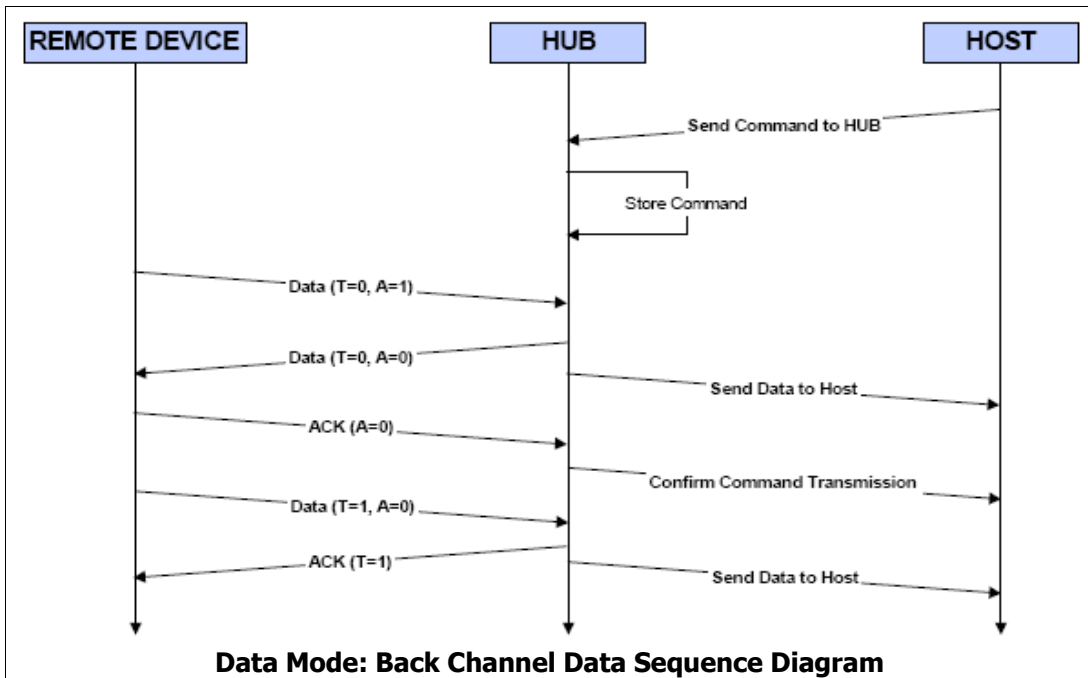
## 2.4 Data Mode

Data Mode is the usual mode during normal operation of the wireless iDwaRF-Net network.

Sensors can send data packets to the Hub and listen for a response, either ACK or data packet. The Sensor can also send an empty data packet in order to poll the Hub for data. If the Sensor receives a data packet, it may respond with an ACK or wait until the next data packet to acknowledge the transmission. If no response is received, the Sensor retransmits the packet. If no response is received after a defined number of retransmissions, the Sensor assumes the channel has become unavailable (e.g. excessive interference) and moves to Channel Search Mode.

The Hub continuously listens for data packets from the Sensors. Valid data packets are responded with either ACK or a data packet, if there is application data to be sent back to the Sensor. The quality of the current network channel is observed by periodically monitoring the RSSI and the frequency of corrupted packets. If the channel quality becomes too low, the Hub moves to Channel Selection Mode to find a better channel.





For a more detailed description of the operation modes see "CY3635 Technical Reference Manual" by Cypress.

### 2.5 Hub / Host Communication

The host (PC) communicates with the iDwaRF-Hub through a serial port (or USB, see 3.3 iDwaRF-HubAdapter below). The commands are simple three character ASCII words followed by also ASCII parameters. The communication can be tested most simple with a terminal program (e.g. Windows HyperTerm) and the commands can be entered with the keyboard. The following commands are currently understood by the iDwaRF-Net Hub firmware.

#### Send Data

```
snd [deviceID] [beaconTime] [payload]
```

Send the beacon time and payload data to the iDwaRF-Sensor with the specified Device ID. The first parameter deviceID can be a valid Device ID or the letter x, i.e. the data will be sent to all active Sensors. The second parameter specifies the Beacon Time as a multiple of 125msec or can be -1. In case of -1, no Beacon Time is transmitted and the Payload Data can be two bytes larger than the regular length. The third parameter payload is an ASCII string, which is sent as Payload Data to the Sensor(s).

Examples:

```
snd 0 40
```

Send beacon time 40 (40 \* 0.125sec = 5sec) to Sensor with Device ID 0.

```
snd x 30 Hello
```

Send beacon time 40 and String "Hello" to all (x) sensors.

```
snd 1 -1 Cheers
```

Send no beacon time (-1) and string "Cheers" to Sensor 1.

#### Reset

```
rst
```

The iDwaRF-Hub performs a software reset. All Sensors have to bind again to the Hub and sensor IDs will be reassigned.

#### Get Protocol Status

```
gps
```

Display an internal status variable from the protocol layer, which indicates bind, idle, etc.

## Start Bind Mode

```
bon
```

When the Hub is set to a certain PN Code, he can no longer pick up messages from Sensors with other PN Codes. To make a bind possible even so, this command enables bind mode. When active, the Hub periodically scans the bind channel for unbound Sensors. If a Sensors is found, the actual PN Code is transmitted and the Sensor becomes part of the Hub's network.

## Stop Bind Mode

```
bof
```

The boff command is the counterpart to bon and disables bind mode again.

## Enumerate

```
enu
```

This command displays (enumerates) all currently bound Sensors. Also orphaned Sensor (i.e. Sensors which are no longer existing) entries will be displayed.

## Clean Sensor List

```
cln
```

Due to performance reasons, it is not possible to avoid double entries of certain Sensors in the Hub's Sensor list. When a Sensor is being reset for some reason, it performs the bind again and a new Device ID will be assigned. Additionally, the old Device ID is still stored in the list. The cleaning up algorithm is time consuming and can be initiated at any appropriate time with the cln command.

Note: This command is currently being updated, since the algorithm does not use a timestamp for each Sensor entry, hence the cleaning up of certain Sensors might fail.

## Configure Network

```
cnf [pnValue] [chValue]
```

The iDwaRF-Net wireless network will be reconfigured. The Hub uses the new PN Code pnValue and starts with channel chValue, hence already bound Sensors will be lost. The specified channel chValue is just a start value - it is possible that the Hub moves to another channel, if either there is excessive noise on that channel or if the channel is already being used by another Hub.

## Delete Sensor

```
del [deviceID]
```

The Sensor with the specified Device ID will be deleted from the sensor list. If this Sensor was still active at that moment, it will perform a new Bind Request.

## 3 Hardware Building Blocks

The iDwaRF-Net firmware is tailored to the iDwaRF-168 module, which is a particular suitable basis for iDwaRF-Net wireless network applications. To simplify the first steps with the iDwaRF-Net technology, a set of complementing hardware building blocks is available.

### 3.1 iDwaRF-168 Programmable Radio Module

The iDwaRF-168 is the first programmable radio module combining Cypress' 2.4GHz DSSS radio technology with Atmel's AVR microcontroller architecture on a tiny module. The ATmega168 microcontroller provides 16kbytes of flash memory, which is shared between the iDwaRF-Net firmware (12k Hub, 8k Sensor) and the user application. For more information on the iDwaRF-168 module see [www.chip45.com/iDwaRF-168](http://www.chip45.com/iDwaRF-168).



### 3.2 iDwaRF-SensorBox

The iDwaRF-SensorBox is an exemplary wireless sensor application with iDwaRF-168 module, power supply (batteries or external) and several sensor-like components like LED, push button, potentiometer, battery monitoring and temperature sensor integrated in a matching plastic enclosure. Operating the SensorBox is simplified by a downloadable precompiled version of the iDwaRF-Net Sensor firmware. Instead of the onboard sensor components own sensor or actuator extensions can be connected to an expansion header. See [www.chip45.com/iDwaRF-SensorBox](http://www.chip45.com/iDwaRF-SensorBox) for details.



### 3.3 iDwaRF-HubAdapter

To simplify the iDwaRF-Hub connection to a host PC, the iDwaRF-HubAdapter integrates an iDwaRF-Hub module, a USB to UART interface module (littleUSB) and power supply into a convenient plastic enclosure. A standard USB cable can be used for PC connection and a USB driver (virtual COM port) simplifies access to the Hub from the PC application. For more information on the iDwaRF-HubAdapter see [www.chip45.com/iDwaRF-HubAdapter](http://www.chip45.com/iDwaRF-HubAdapter).



### 3.4 iDwaRF-StarterKits

Currently three StarterKits are available, which are differently composed out of the above mentioned hardware building blocks:

iDwaRF-StarterKit-S: 3x iDwaRF-168, 1x iDwaRF-HubAdapter, 2x iDwaRF-SensorBox, USB cable

iDwaRF-StarterKit-M: 5x iDwaRF-168, 1x iDwaRF-HubAdapter, 4x iDwaRF-SensorBox, USB cable

iDwaRF-StarterKit-L: 3x iDwaRF-168, 1x iDwaRF-HubAdapter, 9x iDwaRF-SensorBox, USB cable

## 4 Customizing the iDwaRF Modules

The iDwaRF-Net firmware can be easily adapted to own sensors and actuators.

### 4.1 iDwaRF-Sensor User Functions

The following functions are called by the iDwaRF-Net firmware at certain program states. Custom specific code can be inserted in these functions as a means of customizing the iDwaRF modules.

These four functions are the basis for customizing the iDwaRF-Sensor. No deeper immersion into protocol details are required by the user. Nevertheless, with the direct access to payload data, even more sophisticated protocol layers could be implemented by the user to extend the functional range of the base protocol.

#### 4.1.1 Prepare the Sensor for Sleep

```
void cbConfigForSleep (U16 *sampleInterval);
```

This function is called by the firmware before the Sensor goes to sleep. All custom specific sensors, actuators or other tasks should be set to an energy saving mode. The parameter U16 \*sampleInterval points to the actual duration the sensor is going to sleep as a multiple of 125msec. The parameter can be altered to change the sleep behaviour to the application's needs.

iDwaRF-SensorBox example: Before the sensor enters sleep mode, a conversion of the temperature sensor is initiated. Due to this, the temperature conversion takes place during sleep of the sensor and the last temperature value is read and transferred to the hub - this saves active time, since conversion is slow. The LED is turned off and a string "configForSleep" is output through the serial port (UART).

```
void cbConfigForSleep (U16 *sampleInterval)
{
    // start conversion of the temp sensor
    startTempConversion();

    clearSensorBoxLED(); // debug output - sleep means LED off
    // sleep for 1x125ms - do not waste batterie life
    for (U8 n = 1; n > 0 && *sampleInterval > 0; n--) {
        avrSleep(); // sleep for 125ms
        (*sampleInterval)--;
    }
    setSensorBoxLED(); // debug output - active node means LED on

    // read the sampled and converted temperature from the sensor
    lastSampledTemp = readTemperature();

    clearSensorBoxLED(); // debug output - sleep means LED off
    OutStr_P (PSTR("configForSleep "));
    OutDec (*sampleInterval);
    OutStr_P (PSTR("\r"));
    _delay_ms (1); // delay that is needed if a sleep follows directly after
    // it ensures that OutStr_P can transmit all its data
}
```

#### 4.1.2 The Sensor returns from sleep

```
void cbExitFromSleep (void);
```

This function is called right after Sensor wake-up. All custom wake-up code, e.g. reinitialization of sensors, actuators or other tasks, should be placed here.

iDwaRF-SensorBox example: A string "exitFromSleep" is output through UART and the LED is turned on.

```
void cbExitFromSleep (void)
{
    OutStr_P (PSTR("exitFromSleep\r"));
    setSensorBoxLED(); // debug output - active node means LED on
}
```

#### 4.1.3 Process Data Received from the Hub

```
void cbBackchannelProcess (U8 userdata, U8 *buf, U8 length);
```

This function is called when data has been received from the hub. A pointer to the data buffer as well as the length of the buffer is passed as parameters to this function.

iDwaRF-SensorBox example: The received data is output through the serial port (UART).

```
void cbBackchannelProcess (U8 userdata, U8 *buf, U8 length)
{
    OutStr_P (PSTR("backchannelProcess:\r"));
    OutStr_P (PSTR("\tuserdata=")); OutDec (userdata);
    OutStr_P (PSTR(" length=")); OutDec (length);
    OutChar (' ');

    // if there's userdata - display the string
    if (length > 0) {
        OutChar ('<');
        for (U8 n = 0; n < length; n++) {
            OutChar (buf[n]);
        }
        OutChar ('>');
    }
    OutChar ('\r');
}
```

#### 4.1.4 Process Data for Transmission to the Hub

```
U8 cbTxProcess (volatile U8 *pTxData);
```

This function allows the user to set up data which should be sent to the Hub. A pointer to the data buffer is passed as a parameter to the function and the return value is the amount of data bytes stored in the buffer. The user can store data up to the maximum amount of payload data, which is calculated from the maximum packet size (MAX\_PACKET\_SIZE - 6). The maximum packet size is preset to 17, hence the maximum amount of data bytes is 11.

iDwaRF-SensorBox example: The ADC channels, button and temperature sensor are read and the values are stored in the data buffer (\*pTxData++ = value;). The amount of data is calculated and returned.

```
U8 cbTxProcess (volatile U8 *pTxData)
{
    // read all sensor values
    enableADC();
    U8 batt = readSensorBoxBattVal();
    U8 poti = readSensorBoxPotVal();
    U8 btn = readSensorBoxBtn();
    disableADC();

    // remember original ptr value
    U8 *pTxData_old = (U8*)pTxData;

    // store the values inside the data paket
    *pTxData++ = poti;
    *pTxData++ = (lastSampledTemp >> 8); // highbyte
    *pTxData++ = (lastSampledTemp & 0xff); // lowbyte
    *pTxData++ = batt;
    *pTxData++ = btn;

    // Store some dummy data to showcase the payload
    for (U8 n = 5; n < 11; n++)
    {
        *pTxData++ = n;
    }

    // return the amount of bytes stored in the buffer
    return pTxData - pTxData_old;
}
```

## 4.2 iDwaRF-Hub User Functions

Since the Hub is assumed as being always-on, no sleep-related functions are available in the Hub firmware. There exist mainly three functions for handling the data received from the USART or received from the Hub.

Protocol related functions as described in 2.5 can also be accessed through C-functions. See 4.2.2 source code for details.

#### 4.2.1 Serial Data Byte Received from the USART

```
void cbSerialDataReceived (U8 Data);
```

This function is called by the USART subsystem when a byte was received.

iDwaRF Terminal Example: The byte is stored in a global buffer and a flag is set. The time intensive processing of the data is done in the main loop.

```
void cbSerialDataReceived (U8 Data)
{
    // wrap around of the buffer index
    if (rxpos >= RXBUFSIZE) {
        rxpos = 0;
    }

    // store the received char in the buffer
    rxbuf[rxpos++] = Data;

    // flag the new rx data
    rxnewData = 1;
}
```

#### 4.2.2 Process Data Received from the USART

```
void cbProcessRxData (void);
```

This function is called from the main loop when data from the USART has been received.

iDwaRF Terminal Example: This function implements the serial protocol of the Hub. It checks for an end characted '\r' and then parses the received string for commands and parameters.

```
void cbProcessRxData (void)
{
    U8 Data = 0;

    // get last received byte
    if (rxpos>0) Data = rxbuf[rxpos-1];

    // check for the end char - \r triggers the command matching
    if (Data == '\r') {

        if (strncmp_P ((U8*)rxbuf, PSTR("rst\r"), 4) == 0) {
            // RESET
            sendOkPacket();          // acknowledge the rs command
            rfReset();

        } else if (strncmp_P ((U8*)rxbuf, PSTR("gps\r"), 4) == 0) {
            // GETPROTOCOLSTATUS
            U8 pStatus = rfGetProtocolStatus();
            sendOkPacket(); OutDec (pStatus);

        } else if (strncmp_P ((U8*)rxbuf, PSTR("bon\r"), 4) == 0) {
            // start bind / bind on
            sendOkPacket();
            rfStartBind();

        } else if (strncmp_P ((U8*)rxbuf, PSTR("bof\r"), 4) == 0) {
            // stop bind / bind off
            sendOkPacket();
            rfStopBind();

        } else if (strncmp_P ((U8*)rxbuf, PSTR("enu\r"), 4) == 0) {
            // enumerate all sensors
            sendOkPacket();
            EnumerateAllSensors();

        } else if (strncmp_P ((U8*)rxbuf, PSTR("cIn\r"), 4) == 0) {
            // clear up the sensorlist
        }
    }
}
```

```

    sendOkPacket();
    CleanupSensorList();

} else if (strncmp_P ((U8*)rxbuf, PSTR("cnf"), 3) == 0) {
    // configure network parameters
    do_CNF_Command ((U8*)rxbuf, rxpos);

} else if (strncmp_P ((U8*)rxbuf, PSTR("snd"), 3) == 0) {
    // send backchannel data
    do_SND_Command ((U8*)rxbuf, rxpos);

} else if (strncmp_P ((U8*)rxbuf, PSTR("del"), 3) == 0) {
    // remove sensor(s) from the hub
    do_DEL_Command ((U8*)rxbuf, rxpos);

} else { // unknown command - return error code
    //for (U8 n = 0; n < rxpos; n++) OutChar (rxbuf[n]);
    //for (U8 n = 0; n < rxpos; n++) { OutHex (rxbuf[n]); OutChar (' '); }
    sendERRPacket(); // signal a problem
}
rxpos = 0; // reset the buffer to sync to the input
}
}

```

### 4.2.3 Process Data Received from the Sensors

```

void cbSensorPacketReceived (PACKET_TYPES PacketType,
                             U16 DeviceId,
                             U8 UserDataCount,
                             volatile U8 *buf);

```

#### Parameters:

PacketType	The type of the received packet: BIND_REQUEST, VARIABLE_DATA_PACKET or FIXED_DATA_PACKET.
DeviceID	Device ID of the Sensor, which sent the data.
UserDataCount	Amount of user payload data bytes stored in buf.
buf	Pointer to the transmitted data bytes. Only UserDataCount bytes are valid.

This function is called when a Sensor data packet was received in the Hub.

iDwaRF Terminal Example: The example prints some short packet data (PacketType, DeviceID) through the USART and payload data is copied to another buffer for later processing in the main loop. The processing of the data (e.g. °C calculation from the ADC data) is time consuming and therefore done in the main loop.

```

void cbSensorPacketReceived (PACKET_TYPES PacketType,
                             U16 DeviceId,
                             U8 UserDataCount,
                             volatile U8 *buf)
{
    OutStr_P (PSTR("S"));
    OutDec (PacketType);
    OutStr_P (PSTR(" : "));

    if (PacketType == BIND_REQUEST_PACKET) { // = 0x00,
        OutStr_P (PSTR("BIND_REQUEST\r"));
    } else if (PacketType == VARIABLE_DATA_PACKET ||
              PacketType == FIXED_DATA_PACKET) {
        // Data packet of variable length
        // Data packet of fixed length as defined by SIZE_OF_FIXED_DATA_PACKET

        if (PacketType == FIXED_DATA_PACKET)
            OutStr_P (PSTR("FDP ")); // signal the fixedDataPacket special case

        OutStr_P (PSTR("ID "));
        OutDec (DeviceId);
        OutChar (' ');
    }
}

```

```
// copy payload to the buffer for later, time intensive processing
for (U8 n = 0; n < UserDataCount; n ++) {
    sData[n] = *buf++;
}
sSize = UserDataCount;
}
```

**Disclaimer** – Erik Lins makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein. Erik Lins products are not intended for use in medical, life saving or life sustaining applications. Erik Lins retains the right to make changes to these specifications at any time, without notice.

All product names referenced herein are trademarks of their respective companies. chip45.com is a registered trademark of Erik Lins.