AN024
Frequency Hopping Protocol for CC1010

By Ø. Grotmol

Keywords

- *Frequency Hopping*
- *Acquisition*
- *Synchronisation*
- *FCC 15.247*

Introduction

The demand for spread spectrum techniques is growing as the ISM bands become crowded. Benefits such as increased noise resistance, data security and allowed output power are attractive.
AN014 [1] describes the various implementation techniques available. This document gives a detailed account of the specific methods that are actually employed in the Frequency Hopping Protocol (FHP) implemented in the CC1010. The document also describes the functions available and their typical usage.

Advantages

In the U.S., perhaps the most important reason for using frequency hopping is that an output power of up to 1 Watt is allowed under the FCC regulations section 15.247. The main requirements are that at least 50 frequencies in the 902-928 MHz band are used, that on average each channel is used equally and on average that each channel is used less than 0.4 seconds every 20 second period. All the requirements can be satisfied by the CC1010 using the FHP. Thus, the large output power allows a wide radio range.

Other important advantages are reliability and data security. If some of the channels are jammed because of noise, jamming or multi-path reflections, the system will still be able to operate on the other frequencies. Furthermore, eavesdropping is more difficult since the hostile equipment will have to find the frequency hopping sequence and change channels fast.

Protocol

FHP is packet based and allows one-to-one and one-to-many communication between a master and any number of slaves. Communication between the slaves is indirectly supported through the master. All nodes carry an ID.

Beacons and communication

The master periodically transmits a beacon, each time at a new frequency. The master steps through a pseudo-random sequence of any number of frequencies (typically 50). The slaves know the sequence. The beacon is a byte that is unique for each system so that the slaves do not tune in to the wrong master. Together with the beacon is a control byte that specifies what will happen in the remaining part of the period. There are three main possibilities:

Master transmits
If the master wants to send a packet, the control signal indicates this as well as who shall receive the packet. The destination may be a specific slave or all slaves in the case of a broadcast. Then the addressee(s) listen, and the master transmits the packet. Optionally the addressee will return an acknowledgement. Broadcasted packets are never acknowledged.

Slave transmits
If the master knows that a slave wants to transmit, the control signal specifies the ID of that slave. Then the master listens, and the slave transmits the packet. Optionally the master will return an acknowledgement.

Requests to transmit
If neither of the above, any slave may request a transmission by sending its own ID to the master. To avoid interference in the case that multiple slaves want to transmit, the slaves have to wait a random amount of time before transmitting its request. The master will receive the first request and let the corresponding slave transmit in the next period that it will not send anything itself. If two or more slaves send their requests at the same time and before the others, the master may receive the strongest signal if one is dominant, or otherwise all the slaves will have to try again in the next period.

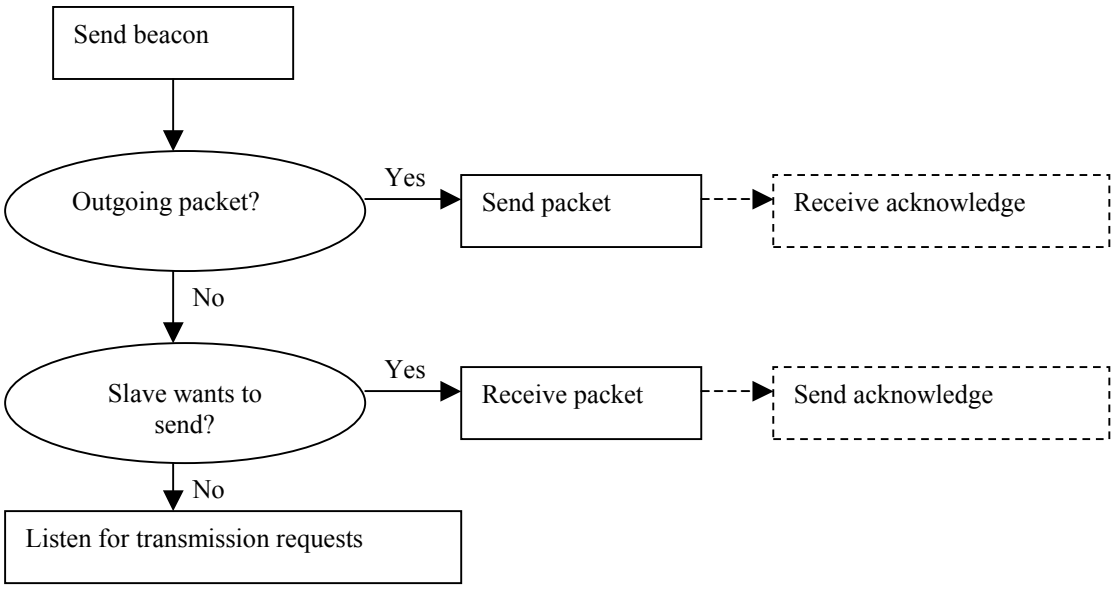

Figure 1 The actions the master performs every period.

Beacon	Preamble	Length	ID	CTRL	CRC
Data packet	Preamble	Length	Data bytes		CRC
Acknowledge	Preamble	Length	ID	ACK	CRC

Figure 2 The packet formats

Acquisition

When powered up the slaves have to perform an acquisition to find out where in the sequence of frequencies the master resides. A new acquisition may also have to be performed if synchronization is lost, for example due to noise or the nodes temporarily being moved out of radio range. In the last case the slave has an idea of which frequency the master is transmitting at and this information is exploited to achieve faster acquisition.

When performing acquisition, the slave must always listen for one full period at each frequency that is being examined to be sure of picking up the beacon. It starts at the frequency that it believes to be most likely (a random one if the chip is being powered up) and then moves on to the most nearby frequencies in the pseudo-random sequence.

Giving the first frequency guess the number 0, the closest frequencies are 0, 1, -1, 2, -2, 3, -3, … However, since the master is also stepping through the frequencies, this sequence must be shifted by the sequence 0, 1, 2, 3, 4, 5, 6, … The actual frequencies examined are thus 0, 2, 1, 5, 2, 8, 3, …

Time	0	1	2	3	4	5	6
Guessed master	0	1	2	3	4	5	6
Slave	0	2	1	5	2	8	3

Table 1 Acquisition sequence

If all frequencies are examined without result, the acquisition must start over again. Assuming the failure is due to the slave being out of radio range, the application may choose to have a delay between each acquisition to reduce power consumption.

Power modes and synchronization

There are three power modes: active, passive, and off.

Active
The standard mode supporting two-way communication between master and slave is the active one. The slave maintains synchronization by listening for every beacon. The timing is performed using interrupts. Since the slave receives all the control signals, the master may send a packet to the slave at any time.

Passive
When only slave-master communication is needed, the slave may enter the passive mode. In passive mode the slave does not regularly listen for the beacons but instead maintains synchronization using a timer. When a packet is ready to be transmitted the slave temporarily follows the beacons until the packet is sent.

If the clocks have drifted so much that the first beacon is missed, the slave enters acquisition. However, since the slave knows approximately where in the sequence of frequencies the master is currently transmitting, the acquisition is very rapid.

The master cannot send packets to the slave at arbitrary moments since the slave is not listening. If master-slave communication is needed, one may let the slave enter active mode now and then and send a message to inform the master that it is listening. The transition from the passive mode to the active mode is smooth as the synchronization is maintained.

In the passive mode even the main crystal oscillator may be shut down, letting the chip run at the 32KHz clock. Thereby, the power consumption is reduced to 1.3mA. One may even enter the Idle mode of the chip, reducing the power consumption to 29uA. The processing starts at every 24^{th} second to process the timer overflow, but this contributes almost nothing to the overall power consumption.

Off
In the off mode even the timer is shut down, allowing the Power-down mode of the chip to be entered. A transition to the passive or the active mode from the off mode requires a full acquisition. The chips are in off mode on power up.

Also the master may enter the off mode, meaning that nothing is transmitted in the whole system. When entering the active mode, the master simply starts transmitting beacons from the last frequency it used. The master does not have a passive mode.

Implementation

Definition File

The settings are specified in the file Definitions.c, which has to be compiled into each project. It should be copied into the project directory and modified there, so that each project can have specific settings. It can be copied from the \LIB\Chipcon\Cul\fhp directory or one of the FHP examples. The settings should not be changed after initializing the FHP.

Frequency settings

The RF settings to be used are specified in the struct *RF_SETTINGS* and the struct array *rxtxpair_settings*. The RF_RXTXPAIR_SETTINGS named *RF_SETTINGS* is used as a

template from which the modem, current and power settings among others are retrieved. This structure is generated with SmartRF Studio. Any settings may be used.

The settings that are specific for each frequency, FREQ_A, FREQ_B, PLLR and PLLT, are stored in the array *rxtxpair_settings*. The settings are separated in this way to reduce memory consumption.

If the requirements of FCC Rules section 15.247 are to be satisfied, it is necessary to specify at least 50 frequencies in the 902-928 MHz band in *rxtxpair_settings* in random order.

Parameters

There are several constants defined in the struct *fhpSettings* that influence the behaviour of the protocol. These are used to fine-tune the system to the needs of a particular application. Many of the parameters have standard settings that work well in most applications.

MAX_PACKET_LENGTH
specifies the maximum number of bytes that may be transmitted as a single packet. The maximum length supported is 251 bytes. The only advantage of reducing this parameter is that less space is set aside in the external memory as buffer for incoming messages. (This parameter is given as a define-statement, not in *fhpSettings*.)

period
specifies the time in milliseconds between the beacons. The period must be large enough to allow transmission of one packet and control signals according to the packet format. Increasing the parameter further will reduce power consumption, but the throughput falls and the time to perform acquisition increases.

With a high data rate and short packets a *period* of 30 ms can be achieved. The upper limit on *period* is 16711298/*clkFreq*, which amounts to 1133 [ms] for the evaluation board frequency of 14746 kHz.

listenTime
specifies the timeout in tenths of milliseconds when listening for beacons, packets and acknowledgements.

overhead
specifies the approximate time in milliseconds that passes from the chip is interrupted to the beacon is received. The reason this parameter is necessary is that the slave's timer starts after the beacon is received. However, the slave has to start listening again before the next time the beacon is transmitted. Therefore, it cannot wait for one whole period before being interrupted again. Instead, the waiting time is set to *period–overhead*.

To find optimal settings for *listenTime* and *overhead*, some experimenting is required. First, let *listenTime* be large (~20) and use binary search to find the smallest *overhead* that allows the master and slaves to maintain synchronization. When *useLEDs* is TRUE, the slaves should alternate between the yellow and the green LED. Then, using this value of *overhead*, find the smallest value of *listenTime* that works in the same way.

timeslots and slotSize
slotSize specifies the time in milliseconds a slave uses to transmit a request to send a packet. *timeslots* specifies the number of such periods the master will listen for requests. When using only one slave, *timeslots* should be one. If there are many slaves and they transmit often, *timeslots* should be larger to prevent the slaves from jamming each other when requesting transmissions.

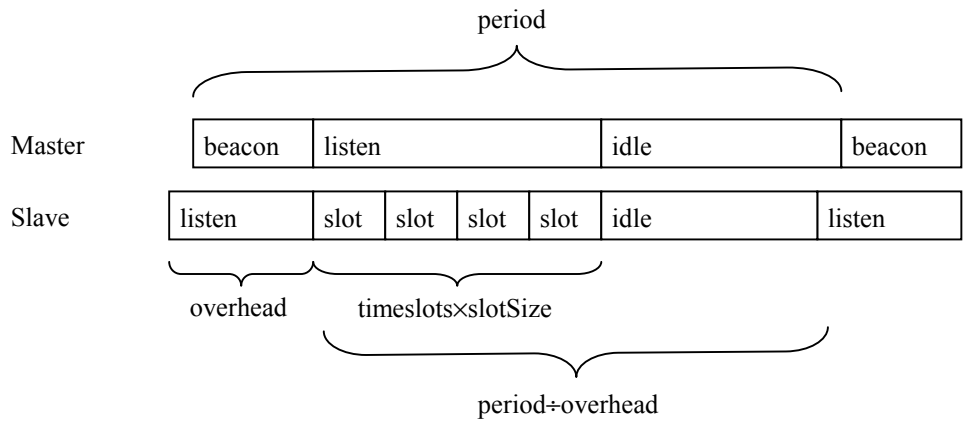

Figure 3 Timing of transmission requests

useLEDs
If this parameter is TRUE, status information is displayed on the LEDs on the evaluation board. This feature is good for debugging.

useAck
If this parameter is TRUE, all non-broadcasted packets are acknowledged. This prevents packet loss, but the period has to be somewhat longer. The packet is resent until acknowledgement is received or the sending is cancelled by the application through the *cancelSending* function. If the acknowledgement is lost, a packet will be received multiple times. A user application may therefore have to include a packet ID in the packets to prevent acting twice on the same packet.

clkFreq
specifies the clock frequency of the main crystal oscillator of the chip in kilohertz.

Example settings
These settings have been found to work well in two example programs, a chat program (Chat.c) optimized for high data rates and a temperature sensor (Temperature.c) optimized for low power consumption and long range:

Parameter	Chat	Temp
Baud rate (kBaud)	*38.4*	*2.4*
MAX_PACKET_LENGTH	100	10
period	60	400
listenTime	1	6
overhead	10	80
timeslots	2	2
slotSize	5	45

Table 2 Example parameter settings

RTC Parameters

The parameters in RealTime.c have been calculated for the evaluation board clock frequency 14746 kHz. If another frequency is used, the parameters must be updated to maintain synchronization while the slave is in the PASSIVE mode. If only the ACTIVE mode is used, this is not strictly necessary.

XOSC_FACTOR must be set to 12000/*clkFreq*. The reason is that the timer is clocked by the main clock source, and so the timer value has to be multiplied by the correct factor to obtain the real time.

If the 32 kHz crystal oscillator will be used as main clock source, TIME_LOST_DURING_POWER_UP and TIME_LOST_DURING_POWER_DOWN have to be adjusted by using the example program RTCcalibrate. The reason is that the timer must be stopped during change of main clock source, and an amount of time is added after each change to correct the clock. However, the time spent during these transitions depends on the frequency of the main crystal oscillator.

Functions

initializeFHP
calibrates all the frequencies and performs other initializations as well. This function must be called before the *setMode*, *newSending* or *synchronize* functions. The calibration results are stored for all the frequencies, so that they can be retrieved later for fast changing between frequencies. This is necessary since the calibration of a single frequency requires 30 ms.

newSending
is used to initiate a transmission.

getSendStatus
gives the status of the current transmission.

cancelSending
There is no internal timeout on transmissions, but the application may cancel the transmission at any time using this function.

synchronize
is used by the slaves to synchronize with the master. There is no need for the user application to call this function unless the *setSynchronizationFailedFunction* is used.

setMode
specifies OFF, PASSIVE or ACTIVE mode. This function is normally called only once on startup, after *initializeFHP*.

setSystemID
specifies the ID of the system. May be used to avoid interference between different systems using FHP.

setOwnID
specifies the ID of the slave (no effect on master). The slave only listens for packets addressed to its own ID or the broadcast ID.

setReceivingFunction
specifies a function that is called each time a packet arrives. The packet is stored in a buffer called *message* and the length of the packet is available as *messageLength*. Note that the function is called from within an interrupt service routine and should return quickly.

setCallbackFunction
specifies a function that is called when a packet is successfully transmitted (acknowledge received if that is required, otherwise just transmitted). The user application might not need to use this functionality.

setSynchronizationFailedFunction
specifies a function that is called when acquisition is unsuccessful. The user application will then have to call the *synchronize* function to try again. If the *setSynchronizationFailedFunction* is never called, FHP will continue performing acquisitions until it succeeds.

useX32 and useXOSC
If the user application saves power by using the 32 kHz crystal oscillator as clock source while in passive mode, these functions need to be used for switching between the sources in order to maintain synchronization.

Emulating SPP

An emulator called Frequency Hopping Simple Packet Protocol (FHSPP) for the Simple Packet Protocol (SPP) using the FHP instead of single frequency operation has been developed. See the header file Fhspp.h and the implementation file Fhspp.c. This is mainly meant as an easy way to test an application that has already been developed using SPP with frequency hopping. If an application is developed from scratch, it is recommended to use the functions provided by the FHP directly instead of FHSPP.

Usage

The usage of FHP is rather simple if one sticks to a set of standard settings. Many of the parameters and functions need not be considered unless one wants to tune the behaviour to obtain the fullest performance. The example programs described in the next section may be used as templates.Following, a short guide on how to get an application up and running on two evaluation boards is given.

- Call *initializeFHP(TRUE)* on one board and *initializeFHP(FALSE)* on the other. This sets up the first as a master and the second as a slave.
- Write a function that processes incoming messages and call *setReceivingFunction* with this function as argument. The function can read *messageLength* bytes from the buffer *message*.
- Call *setMode(ACTIVE)*. This puts the boards in the active mode and allows full duplex communication.
- Now the boards can send packets to each other using the function *newSending*. The slave is given slave ID 1 on start-up, so the master will have to address the packets to address 0 (the broadcast ID) or 1.

Examples

A number of example programs have been developed demonstrating the use of FHP. They are available from the Chipcon website. Note the use of different settings in Definitions.h and Frequencies.h to adapt FHP to the particular application.

Tennis (Tennis.c)

This is a two-player “tennis” game played on the CC1010 development board using four LEDs, four buttons and the ADC. It is the simplest of the examples and demonstrates the basic usage of FHP. Here the FHP_USE_LEDS is not defined because the LEDs are used for the game. Codesize 8 kB.

Temperature Sensor (Temperature.c)

This program demonstrates the use of power saving modes in addition to the basic FHP usage. One development board is connected to a terminal window (i.e. HyperTerminal) on a PC via the serial port and acts as master. One or several boards are placed at different locations and act as slaves. They measure the temperature regularly at their locations and

transmit the information to the master, which prints it to the terminal window. The slaves turn off the main crystal oscillator and go into idle mode between transmissions. Note also the use of 2.4 kBaud data rate and 4 dBm output power for long radio range, specified in Frequencies.h. Codesize 11 kB.

Chat (Chat.c)

This is the most complex example. It was originally written for single frequency operation through the SPP library (that version is also available from the Chipcon website). A simple substitution of function names was sufficient to use the FHSPP instead. Two or more demonstration boards have to be connected to terminal windows on the same or different PCs. One must be set up as server (this one will act as master), the others as clients. Then any user may broadcast messages to everyone or send private, encrypted messages to individual users. All messages are routed through the server, but this is invisible to the users. Codesize 25 kB.

Resources

The following table lists the usage of internal, external and program memory in bytes:

	data	**xdata**	**const**	**code**
Hopp.c	1	206	427	775
Fhp.c	9	116		1502
Realtime.c	11			525
Assorted HAL		5		1304
Basic total	**21**	**327**	**427**	**4106**
Fhspp.c		110		616
Assorted CUL		36	768	3303
Extended total	*0*	*146*	*768*	*3919*
Full total	**21**	**473**	**1195**	**8025**

Table 3 Memory usage

Under normal usage (without FHSPP), only 4K of program memory is required for the FHP. Only timer 1 is available when using FHP. The reason is that the RTC module uses timer 0, FHP uses timer 2 and HAL uses timer 3.

Spurious emissions

Note that to satisfy the FCC regulations, the spurious emissions have to be below –49.2dBm at the band edges 902 and 928 MHz. If high output power and frequencies near the edges are used, precautions must be taken. The recommended solution is to use an external power amplifier and switch. To avoid splatter, the PA must be turned on before the switch is changed when going from RX to TX. If no external switch is used, ramping of the output power should be added to the function fhpRFSetRxTxOff().

References

Cited references

[1] Chipcon Application Note AN014 Frequency Hopping Systems.
www.chipcon.com

General references

[2] None.

This application note is written by the staff at Chipcon to the courtesy of our customers. Chipcon is a world-wide supplier of RFICs. For further information on the products from Chipcon please contact us or visit our web site. An updated list of distributors is also available on our web site.

Address Information

Web site: http://www.chipcon.com

Technical Support E-mail: support@chipcon.com
Technical Support Hotline: +47 22 95 85 45

Headquarters:
Chipcon AS
Gaustadalléen 21
NO-0349 Oslo
NORWAY

Tel: +47 22 95 85 44
Fax: +47 22 95 85 46
E-mail: wireless@chipcon.com

US Office:
Chipcon Inc.
19925 Stevens Creek Blvd.
Cupertino, CA 95014-2358
USA

Tel: +1 408 973 7845
Fax: +1 408 973 7257
Email: USsales@chipcon.com

Sales Office Germany:
Chipcon AS
Riedberghof 3
D-74379 Ingersheim
GERMANY

Tel: +49 7142 9156815
Fax: +49 7142 9156818
Email: Germanysales@chipcon.com

Disclaimer

Chipcon AS believes the furnished information is correct and accurate at the time of this printing. However, Chipcon AS reserves the right to make changes to this application note without notice. Chipcon AS does not assume any responsibility for the use of the described information. Please refer to Chipcon's web site for the latest update.

Trademarks

SmartRF® is a Norwegian registered trademark of Chipcon AS. All other trademarks or registered trademarks are the sole property of their respective owners.

Chipcon AS is an ISO 9001:2000 certified company.