WiBo - The Wireless Bootloader

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
	2010/07 - 2012/11		D

Contents

Abstract

The following article describes the usage of the uracoli wireless bootloader. WiBo works like a regular bootloader, except that it uses the radio transceiver instead of a UART. WiBo modifies the flash directly, therefore no special backup flash memory, like in other over the air upgrade (OTA) solutions, is required on the transceiver board.

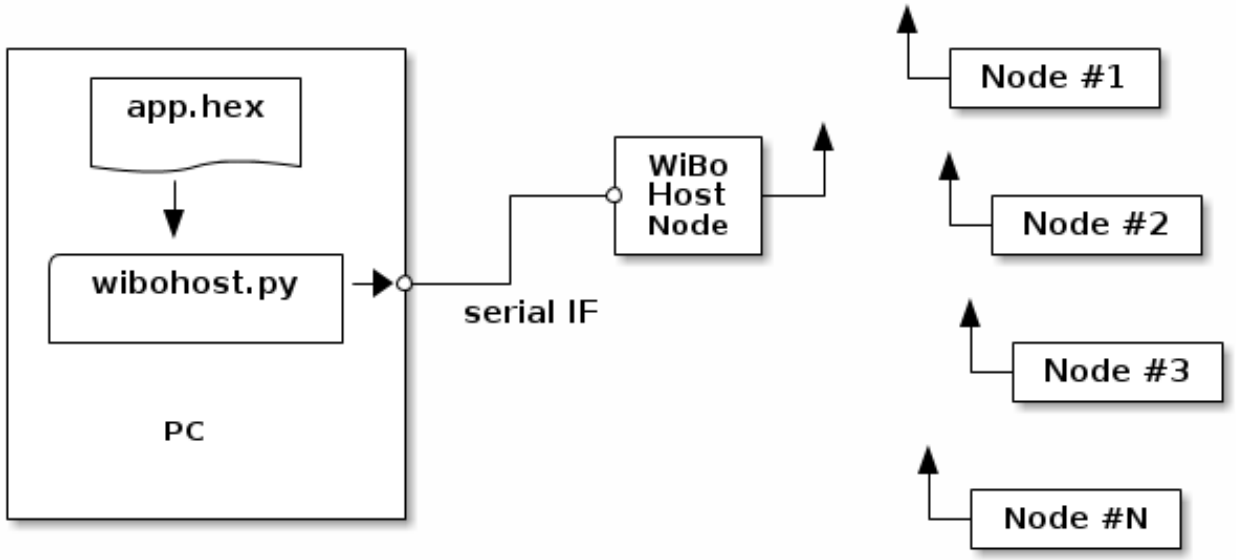
1 Concept

The WiBo framework provides a method to program the MCU that controls the transceiver. Three components are involved,

- a PC with USB or serial interface, running the script wibohost.py,
- a host node, running with the wibohost firmware and
- the network nodes that have the bootloader installed.

The python script `wibohost.py` sends a hexfile `app.hex` in slices to the host node. It transmits these data slices as unicast or broadcast frames to the network nodes. The network nodes collect the data slices and programm them with SPM (self programming mode) commands in the flash application section of the microcontroller.

In order to verfify if a node was programmed correctly, the host and network nodes calculate a CRC16 during the transmission and reception of the data slices. When programming is completed, the wibohost can query the CRC from the nodes and compare with its own checksum.



The bootloading can be done either in **unicast** mode or in **broadcast** mode. In unicast mode one node is programmed with a program image at one time. In broadcast mode, multiple nodes receive the image in parallel. This is useful for programming multiple instances of identical hardware with a firmware image.

The host site consists of a Python script and the host node that runs the wibohost firmware. The script `wibohost.py` uses the module pyserial for serial communication with the host node. It reads and parses the given hex file and transfers it in slices to the host node. The host node firmware sends this slices in frames to one (unicast) or all (broadcast) network nodes.

The wireless bootloader resides in bootloader section of the network nodes and occupy about 1K words of programm memory. Additionally to the bootloader code, configuration record at the end of the bootloader section. This record ensures that the node address and channel information is available, even if the EEPROM was accidentely erased.

The network nodes are passive, that means they never send anything if not queried by the host node.

2 Radio Configuration

Before the Atmel radio transceivers can be used in a wireless network szenario, they need to be configured with some key parameters.

The most important parameter is the radio channel, e.g. the frequency on which the radios communicate. Depending of the radio type, the cannel can be 11 - 26 for the 2.4GHz radios and 0 - 10 for the 868/900 MHz radios. Which channel to choose depends on the area where the network is operated (for 868/900 MHz) and which other interferers are present (e.g. Bluetooth and WLAN for 2.4GHz). Since this topic can't be covered here complete, consult other information sources for selecting a radio channel to operate on.

Other important parameters are the node address and the network address. The IEEE-802.15.4 radios support a 16 bit (SHORT-ADDRESS) and a 64 bit (IEEE-ADDRESS) node address as well as a 16 bit network identifier (PAN-ID).

Since for bootloaders with the limited amount of programm memory no complex search and asociation procedures can be implemented, the channel and addressing parameters needs to be stored persistently in the Flash or EEPROM memory of the network node.

Since the EEPROM in AVR controllers can be easily erased or modified by the application, a safe place to store the radio configuration is the flash memory (the programm memory itself). Since the parameters should be accessible from the bootloader and from the final application, a so called configuration record is stored at the end of the flash memory.

In order to create the config record at the flash end, a Intel Hex file needs to be generated, that is flashed with the hardware programmer and .e.g. with the tool avrdude.

For generating config records for multiple nodes, the pythonscript `wibo/nodeaddr.py` is used. Basically the tool can be used directly on the command line.

```
$python wibo/nodeaddr.py -B rdk230 -c 17 -a 1 -A 0x1234567890ABCDEF -p 0xcafe -o foo.hex
Use Parameters:
 board:      rdk230
 addr_key:   1
 group_key:  None
 short_addr: 0x0001
 pan_id:     0xcafe
 ieee_addr:  0x1234567890abcdef
 channel:    17
 offset:     0x0001fff0
 infile:     None
 outfile:    foo.hex
```

More detailed information about the tools options are displayed with the command

```
$python wibo/nodeaddr.py -h
```

As you see in the above example a command line that covers all relevant parameters is very long and therefore error prone too. The script `nodeaddr.py` can read its parameters also from a configuration file.

A initial nnotated config file for a network setup is created with the command:

```
$python wibo/nodeaddr.py -G mynetwork.cfg
generated file mynetwork.cfg
$cat mynetwork.cfg
# In the "groups" section several nodes can be configured to be in one group,
# e.g. 1,3,4,5 are ravenboards.
#[group]
#ravengang=1,3:5
....
```

Assuming that your network consists just of rdk230 boards, the file `mynetwork.cfg` for the above command line would look so:

```
[board]
default = rdk230

[channel]
default=17
```

```
[pan_id]
default=42

[ieee_addr]
0=0x1234567890abcdef

[firmware]
outfile = node_<saddr>.hex
```

The HEX files for the nodes are generated with the next command:

```
$python wibo/nodeaddr.py -C mynetwork.cfg -a 0
$python wibo/nodeaddr.py -C mynetwork.cfg -a 1
```

or in very freaky pipeline way:

```
$python wibo/nodeaddr.py -Cmynetwork.cfg -a0 -o- | avrdude <OPTIONS> -U fl:w:-:i
```

3 The Host Application

Compiling and Flashing Here is an example for the Raven USB Stick. Applying the configuration record to the hex-file is done in the same manner as for the bootloader application.

```
cd wibo
make -C ../src rzusb
make -f wibohost.mk rzusb
python nodeaddr.py -a 0 -p 1 -c 11 -f ../bin/wibohost_rzusb.hex -B rzusb -o h.hex
avrdude -P usb -c jtag2 -p at90usb1287 -U h.hex
```

Note: The option "-p 1" set the IEEE PAN_ID to "1" and must be identical for the bootloader application and the wibohost application.

4 Python Host Application

Using wibohost.py To test the wireless bootloader environment, the xmpl_wibo application will be used. It blinks a LED with a certain frequency and is able to jump in the bootloader when the special "jump_to_bootloader" frame is received.

At first create some firmware versions, e.g. one slow and one fast blinking. The network nodes shall be rdk230 nodes.

```
make -f xmpl_wibo.mk BLINK=0x7fffUL TARGET=slow rdk230
make -f xmpl_wibo.mk BLINK=0xffffUL TARGET=fast rdk230
```

Next assume that you have 4 network nodes with addresses [1,2,3,4]. In order to check the presence of the nodes, run the scan command.

```
python wibohost.py -P COM1 -S
```

Note that the default address range of wibohost.py is 1 ... 8. This can be modified with the -A option. In the example above, only the nodes 1 to 4 are present, therefore no response from the nodes 5 ... 8 is received.

At first we update all nodes with the slow blinking firmware. Therefore we use the broadcast mode (-U), that means that the image is transfered only once over the air. The address range (-A) is needed to ping the nodes before programming and afterwards to verify their CRC.

```
python wibhost.py -P COM1 -A1:4 -U slow.hex
```

In the next step we selectively flash node 1 and node 3 with the file fast.hex. Since we use unicast programming (-u), the image is transfered for each node over the air seperately.

```
python wibhost.py -P COM1 -A1,3 -u fast.hex
```

5 The Bootloader Application

Build and Flash Build the bootloader for your board with the command

```
make -C ../src <board>
make -f wibo.mk <board>
```

With the command `make -f wibo.mk list` the available <board>s are displayed.

The bootloader expects an address record at the end of the flash memory section. This record can be generated with the script nodeaddr.py. Here is an example for the rdk230 board.

```
# generate a hex file with the configuration record for node #1
python nodeaddr.py -a 1 -p 1 -c 11 \
       -f ../bin/wibo_rdk230.hex -B rdk230 -o a1.hex

# flash node #1 (SHORT_ADDR=1)
avrdude -P usb -p m1281 -c jtag2 -U a1.hex

# Fuses for initial jump to bootloader
avrdude -P usb -p m1281 -c jtag2 -U lf:w:0xe2:m \
        -U hf:w:0x98:m -U ef:w:0xff:m
```

To verify the correct AVR fuse settings refer to http://www.engbedded.com/fusecalc.

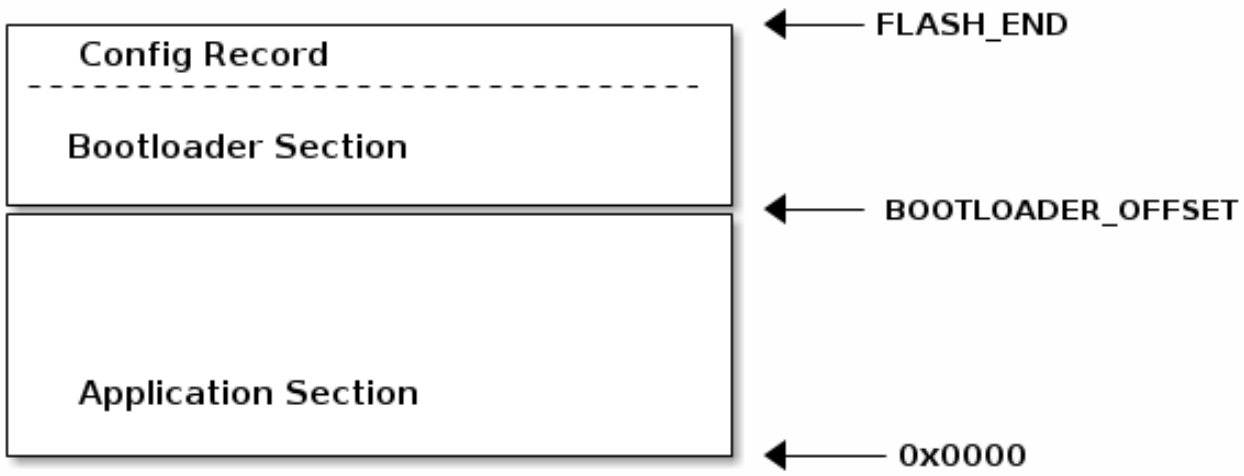
To flash multiple nodes more efficiently, the nodeaddr.py can pipe its output directly into avrdude. This slightly more complex command line can be stored in script `flashwibo.sh` (under Windows replace $1 by %1 for `flashwibo.bat`):

```
python nodeaddr.py -a $1 -p 1 -c 11 -f ../bin/wibo_rdk230.hex -B rdk230 |\
       avrdude -P usb -p m1281 -c jtag2 \
                -U fl:w:-:i -U lf:w:0xe2:m -U hf:w:0x98:m -U ef:w:0xff:m
```

Note: The option "-p 1" set the IEEE PAN_ID to "1" and must be identical for the bootloader application and the wibohost application.

With the `python nodeaddr.py -h` the help screen is displayed.

Flash memory partitioning The AVR flash memory can be divided in an application and a booloader section. The application section is located in the lower address memory. The bootloader section is located in the upper flash memory. In this section the so called self programming opcodes (SPM) can be executed by the AVR core. This SPM opcodes allows erasing and reprogramming the application flash memory.



The start address of the bootloader section is determined by the BOOTLSZ fuse bits. The BOOTRST fuse bit determines, if the AVR core jumps after reset to the application section (address 0x0000) or to the bootloader section (e.g. address 0xf000). The AVR fuse bits and the content of the bootloader section can only be changed either by ISP, JTAG or High Voltage programming.

In order to have enough memory for the application available, the booloader section is choosen to be rather small, e.g. for 8K devices a 1K bootloader section and 7K application section is a reasonable choice. The larger 128k AVR devices are partitioned usually with a 4K bootloader section, leaving 124K flash memory for the application.

The Configuration Record at FLASH_END For WiBo, the last 16 byte of the flash memory are reserved for a configuration record, that holds address and channel parameters, which are needed for operation. The structure of this record is defined in file `board.h` in the type `node_config_t`. It stores

- 2 byte SHORT_ADDRESS
- 2 byte PAN ID
- 8 byte IEEE_ADDRESS
- 1 byte channel hint
- 2 reserved bytes
- 2 byte CRC16

`The configuration record is accessible from the applicatation and from the bootloader sec`

6 The WiBoHost API

The file wibohost.py can also be used as a python module. The following script shows how a broadcast and a unicast flash can be programmed.

```
from wibohost import WIBOHost

wh = WIBOHost()
# open serial connection to wibohost node
wh.close()
wh.setPort("COM19")
wh.setBaudrate(38400)
wh.setTimeout(1)
wh.open()
# check if local echo works.
print WHOST.echo("The quick brown fox jumps")

# scan addresses 1 to 4
addr_lst = wh.scan(range(1,4+1))

# broadcast mode, flash all nodes
for n in addr_lst:
    wh.xmpljbootl(n)
    print "PING  :", n, wh.ping(n)

print "FLASH :", wh.flashhex(0xffff, "foo.hex")
for n in addr_lst:
    print "CRC  :", n, wh.checkcrc(n)
    print "EXIT :", n, wh.exit(n)

# unicast mode, flash node 1
wh.xmpljbootl(1)
print "PING  :#1", wh.ping(1)
print "FLASH :#1", wh.flashhex(1, "bar.hex")
print "CRC   :#1", wh.checkcrc(1)
print "EXIT  :#1", wh.exit(1)
```