Wireless Firewall

Architecture and Implementation

Matthias Schäfer

November 17, 2010

Contents

1 Introduction

In the context of my studies as a computer scientist, I became involved in a project with the goal to create a magic device which destroys packets of a wireless transmission on the fly. At the first moment I thought "Ough, radio waves travel very fast and packets are very short. This never works.", but nevertheless, I told myself to try this. So three guys and me started to work on this topic. Since every idea has a reason, let me first spend some words on the "why".

Wired networks have the big advantage that data has to take a controllable route to its receiver. That means, one can easily control which data from which sender is allowed to enter a part of a network. So things like malicious injection of packets and eavesdropping isn't that easy.

Since wireless networks receive a very high popularity these days, it is desirable to have the same level of controllability there. Of course – as one can image – the broadcasting nature of wireless networks makes these aims hard to achieve. Using directional antennas and faraday cages may be solutions but in most cases they are not really applicable. While cryptography aims to provide things like confidentiality, authenticity and integrity, there is not much dealing with the physical access control in wireless networks. So the question came up, if it's possible to *detect and destroy non-authorized packets.* And this is where our work starts.

Well, the detection of malicious packets can be done in many ways. Since we want to implement a firewall, the detection has to take account of several things which enable the user to define rules for legitimate packets. Such things are on one hand physical properties like frequency, signal strength and direction. On the other hand the firewall has to check the packets' content to get informations about receiver, protocol properties and so on. The destruction of malicious packets is comparatively easy. We can simply jam the packets by sending a strong signal which drowns the original one.

This was the concept we had in mind but we didn't want to develop the complete firewall. This would have gone beyond the scope of our time frame. So we focused just on the proof of concept with a simple prototype. With respect to our available resources, namely Ettus Research's USRP2 software defined radios [4] and Crossbow's 802.15.4 compliant MICAz sensor motes [1], we restricted our work on a firewall which detects 802.15.4 modulated packets on a specific frequency and checks whether security is enabled. According to the setup, we wanted to be able to destroy selectively packets whose security was disabled.

My part of this project is the development of an architecture, the implementation of the reaction side on the USRP2 and the integration of the detection submodule into that system. And this is what the following is about.

2 Universal Software Radio Peripheral 2

To understand why the wireless firewall is build up like it is, it's necessary to know where we started from. So let's first have a closer look on the platform where the firewall runs on.

2.1 Basics

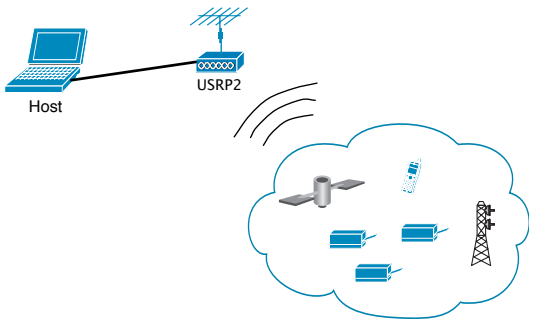


Figure 2.1: Standard USRP2 utilization

As figure 2.1 shows, the actual usage of a USRP2 is to send and receive radio signals. Therefore it is connected to a host computer via a gigabit ethernet link. Over this link, it forwards according to the application the received radio signal to the host or – vice versa – the generated signal from the host to the radio channel. The host can then store, process and visualize this signal data with GNU Radio [6].

2.2 Architecture/Implementation

The USRP2 consists out of three general components. The first component is the host, which controls the USRP2 and generates or processes the signal data. Due to the irrelevance of this component to this work, I just refer to the documentation of GNU Radio at this point.

The second component – and the most important for us – is the firmware. The firmware is a C program, which becomes executed on the USRP2 during runtime. The firmware controls several hardware components and manages the communication with the host. The configuration is done by sending appropriate data over the internal busses (WISHBONE bus, I2C, SPI). Mostly, the configuration data itself is precomputed by the host and the firmware just forwards it to the hardware.

The last component I want to mention here is the FPGA. The FPGA acts as a central unit which implements several functions. For example, it runs a soft processor which executes the firmware. The system running on the FPGA is written in verilog.

To explain which components play which role in the system, I spend a few words on the boot process of the USRP2. The USRP2 has a SD card reader where a SD card with the compiled firmware and the FPGA configuration should be inserted. After powering

on the USRP2, the CPLD automatically reads the first megabyte from the SD card and writes it to the FPGA configuration pins. Once the FPGA is configured, it copies the firmware for the aeMB (soft processor) from the SD card to the RAM. Then the CPLD passes the control to the FPGA which then starts to execute the firmware. After the firmware set up the ethernet interface and initialized the hardware, the USRP2 is ready to receive instructions from the host [5].

Let me now go into more detail about those things which are important for this work, starting from the FPGA.

FPGA

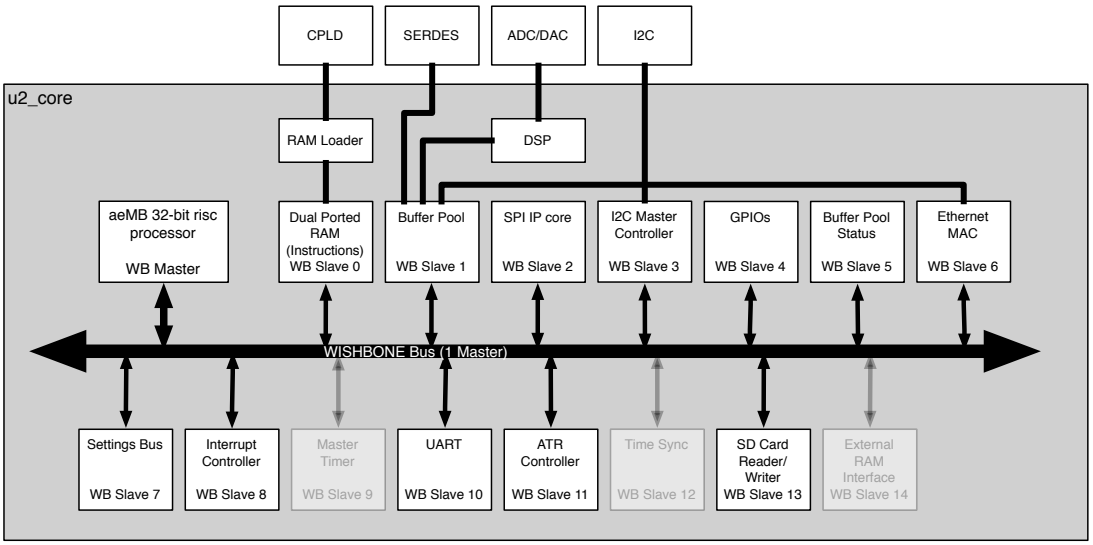


Figure 2.2: Rough FPGA architecture overview

By reading the FPGA verilog code I extracted the rough architecture overview of the FPGA shown in figure 2.2. The core component of the system is the aeMB 32-bit soft processor. It is a clean room reimplementation of Xilinx' Microblaze processor and is connected to the other FPGA components over OpenCore's WISHBONE bus. From the software's point of view, the WISHBONE bus realizes the memory mapped I/O ($\rightsquigarrow$ chip selection). Therefore, each component is assigned to a 16-bit wide ID and a bitmask which are used by the bus master to select the component the firmware wants to communicate with ($\rightsquigarrow$ table 2.1)

The buffer pool (slave 1) contains 8 buffers, whereas each buffer has a size of 2 kB (512 words/lines of size 32 bits). The buffers are dual-ported RAMs, which means they have two ports A and B. Port A is the more relevant for us because it is connected to the wishbone bus and so accessible from firmware. Port B consists out of 8 (4 rd, 4 wr) FIFO-like streaming interfaces and 6 out of those 8 interfaces are dynamically[1]

[1] "dynamically" in terms of each ports can be connected to any buffer

Slave #	Component	Address	Bitmask
0	Dual Ported RAM	0x0000	0x8000
1	Buffer Pool	0x8000	0xC000
2	SPI IP Core	0xC000	0xFC00
3	I2C Master	0xC400	0xFC00
4	GPIO	0xC800	0xFC00
5	Buffer Pool Status	0xCC00	0xFC00
6	Ethernet MAC	0xD000	0xFC00
7	Settings Bus	0xD400	0xFC00
8	Interrupt Controller	0xD800	0xFC00
9	*unused*	0xDC00	0xFC00
10	UART	0xE000	0xFC00
11	ATR Controller	0xE400	0xFC00
12	*unused*	0xE800	0xFC00
13	SD Card	0xEC00	0xFC00
14	*unused*	0xF000	0xFC00
15	*unused*	0xF400	0xFC00

Table 2.1: WISHBONE Bus addresses/masks of the USRP2 FPGA components (Source: `u2_core_udp.v`)

connected to the SERDES component, the DSP pipeline and the ethernet interface, each 1 wr and 1 rd. See also figure 2.3 which illustrates the buffer pool schematically.

The DSP consists out of a RX- and a TX-part. Both parts implement the VITA Radio Transport protocol and do some signal processing in advance. They are directly connected to the DAC/ADC's on one hand and on the other hand they are connected to the buffer pool (compare figure 2.3). In the receiving mode, the DSP RX component stores the received data into the preselected buffer (selection is done by firmware). Conversely the TX-part receives its data from the buffer pool (also controlled by the firmware), processes it and forwards it to the DAC.

Slave 5, the buffer pool status component, provides several registers which contain information about the buffer pool and its transactions. The most important registers

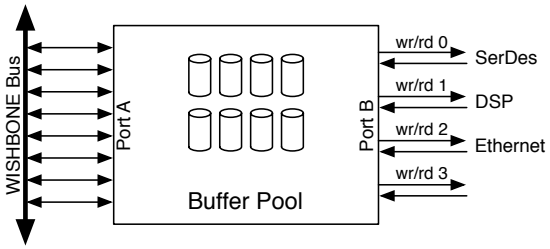


Figure 2.3: Buffer pool component of the USRP2

are the `last_line` registers (1 per buffer) and the `status` register. The former contain the amount of data transfered in the last buffer pool transaction (`send_from_buf` or `receive_to_buf`) for each buffer. The latter, the `status` register, is more important. It is a 32-bit wide register which contains flags indicating buffer pool events for each of the 8 buffers. There are 4 kinds of events. `DONE`-events (bitmask 0x000000ff) signal the completion of a transaction. Errors are indicated by an `ERROR`-event (bitmask 0x0000ff00). When a buffer neither has a `DONE` nor an `ERROR` event or is not processing any transaction, the buffer idles what is announced by the `IDLE`-flags (bitmask 0x00ff0000). In this state, it is safe to start a new transaction. The last 8 bits (0xff000000) are reserved for so called `SLOWPATH`-events. This means that the buffer has a "slow path" packet in it. A packet is identified as such by the FPGA ethernet rx protocol, if this packet requires firmware intervention. This last event is not yet used by the firmware.

While the buffer pool status component acts as an output interface for the buffer pool engine, the settings bus provides control registers for the buffer pool (and other components). The buffer pool is controlled by a 32-bits wide control register (`buffer_pool_ctrl`). This register can be used to start transactions or clear the buffer's status registers. For more informations on that and its usage, have a look at the source files `memory_map.h` and `buffer_pool.c`.

In order to provide the possibility to inform the firmware about events from components, they are connected to the programmable interrupt controller (PIC, slave 8). The interrupt flags need to be polled by the firmware as often as possible to react accordingly and timely to events. The 32-bit IRQ port of this controller is currently connected in this way:

```
assign irq= {{8'b0},
        {8'b0},
        {3'b0, periodic_int, clk_status, serdes_link_up, uart_tx_int, uart_rx_int},
        {pps_int, overrun, underrun, PHY_INTn, i2c_int, spi_int, onetime_int, buffer_int}};
```

The PIC's WISHBONE bus interface is defined as follows. Let the x be the bits 4 to 2 of the address, then the 32-bit output is the following (multiplexing function):

$$\text{output}(x) = \begin{cases} \text{edge enable register} & \text{if } x = 000 \\ \text{polarity register} & \text{if } x = 001 \\ \text{mask register} & \text{if } x = 010 \\ \text{pending interrupts} & \text{if } x = 011 \end{cases}$$

So if the firmware wants to poll e.g. the overrun interrupt from the buffer pool, it just needs to perform something like (see also table 2.1)

```
uint32_t pending = *((volatile uint32_t *) 0xD80C);

// overrun is the 6th irq (coming from the lsb)
if ((pending & (1<<6)) != 0) {
        //...
}
```

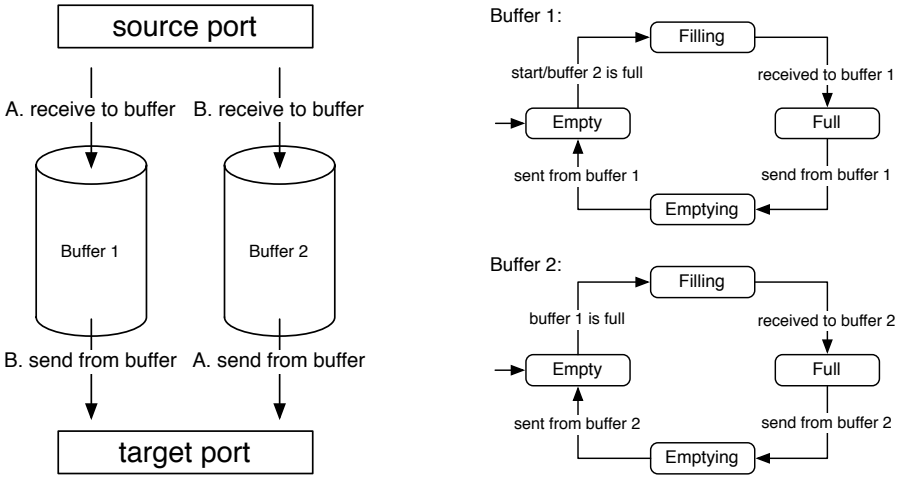
The automatic transmit/receive (ATR, slave 11) controller is responsible for the tx/rx switching of the daughter board. At the moment it supports 4 different states: idle, tx, rx and full duplex. To control the daughter board, it is indirectly connected to the control pins of the daughter board over the GPIO interface.

This was an introduction to the FPGA components which are important for our work. Let us now take a look on a more abstract level of the USRP2, the firmware.

Firmware

As mentioned above, the firmware is a C program which controls the FPGA components during execution on the aeMB processor. It addresses the components over the 16-bit wide addresses listed in table 2.1. On startup, it first initializes the hardware. The completion of this initialization becomes signaled by flashing the LEDs and leaving the D-LED on. It then registers some callback methods to handle events like for example the reception of an ethernet frame.

After doing so, the firmware starts 2 state machines which handle the data flow of the USRP2. These 2 state machines implement the core functionality of the USRP2. They are responsible for the double buffering mechanism of the buffer pool. One forwards the data from the DSP to the ethernet component and the other forwards data from the ethernet buffer to the DSP or the control message handler in the firmware. Let me go into more detail about that.



(a) Schematic (b) State machine

Figure 2.4: Double buffering mechanism of the USRP2 software defined radio. 2.4a illustrates which operations are done simultaniously (A/B). 2.4b shows the corresponding finite state machines of the two buffers. (Source: `dbsm.c`)

Double buffering: The dbsm (double buffering state machine) of the USRP2 acts as follows. At the beginning each dbsm becomes initiated with 2 buffers, a source port and a target port of the buffer pool's port B (see figure 2.3). It then tells the buffer pool to start receiving data from the source port (e.g. the reception of an ethernet frame from the ethernet port 2) into the first buffer and waits for the completion of this task. On completion it calls a pre-registered inspector which analyses the received data. According to the inspector's decision, the dbsm discards the data or – if it is e.g. signal data – it forwards the data to the target port (e.g. the DSP port 1). In both cases, the dbsm instructs the buffer pool to receive again data from the source port into the second buffer while it handles the data of the first one. This enables the USRP2 to receive simultaneously new data from the source port while sending already received data to the target port (compare figure 2.4).

The standard firmware (`txrx_uhd.c`) holds two such state machines. One dbsm (`dsp_tx_sm`) forwards the data from the ethernet port to the DSP. The inspector of this dbsm tests the received udp packets on its content. In the case of signal data, it becomes forwarded to the DSP and in the case of control data, the firmware forwards it to the respective component. The other dbsm (`dsp_rx_sm`) handles the oncoming traffic, i.e. if the USRP2 acts as a radio receiver, this dbsm forwards the received data from the DSP to the ethernet port and thus to the host. Since there arrives only signal data from the DSP, there is no inspection necessary.

Well, to provide a better understanding, let's have a look what happens if an ethernet frame with control data in it arrives. Since one of the two buffers was filling, he receives a `BPS_DONE`-event from the buffer pool which indicates that the buffer keeps data ready to be read. The TX dbsm, which handles this buffer, invokes the registered inspector `eth_pkt_inspector`. This method analyses the data, detects a control packet, calls the control packet handler `handle_udp_ctrl_packet` and returns true, what tells the dbsm that the software (firmware) handled the data. The control packet handler looks into the udp payload and reacts accordingly. For example, if the control packets says "poke 0x800 into the register 0xE400" or in other words "send 0x800 to the first register of the ATR controller" (see table 2.1), the packet handler does this. Other actions could be an I2C read- or write-operation or a SPI transaction with other hardware like the daughterboard.

Back to the firmware after starting the dbsm, the firmware enters the main-loop which polls the buffer pool status registers (slave 5) and the interrupt controller (slave 8) for events. If something happens with the buffer pool (e.g. a `BPS_DONE`-event occures after a completed buffer transaction like `bp_send_from_buf` or `bp_receive_to_buf`), the firmware delegates this event to the respective dbsm which then handles this. Currently, the only two irqs which become noticed by the firmware are buffer overruns and underruns. They simply become cleared and the respective dbsm restarts the current running transaction.

That's roughly how the firmware works. Let's have a look at the communication side of the whole system.

2.3 Communication

As mentioned above, the communication between firmware and fpga components is done via memory mapped I/O or rather the WISHBONE bus. Concerning the communication between fpga components and the environment, I refer to the `u2_core_udp.v` file. This is the topmost verilog module which runs on the FPGA and connects all other modules.

Well, the only thing that is left over here is the communication between host and USRP2. This can be done in two different ways. The most communication is done over the gigabit ethernet link by sending UDP packets. The other possibility to communicate with the firmware from host is the serial UART interface on the USRP2. Since I just used this port to print out some debug messages, I will regard this interface as a one-way interface for USRP2-to-host communication and will consider the gigabit link as the only real communication link.

Ethernet

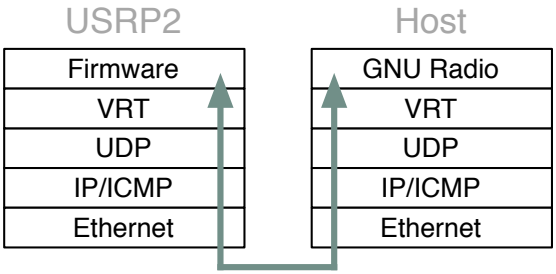


Figure 2.5: The USRP2 communication system layers

Like I already mentioned, the host and the USRP2 communicate over the gigabit ethernet link. This link provides enough bandwidth for full duplex transmissions (sending and receiving with one USRP2 simultaneously). The gaps between samples will be automatically filled up by the DSP's interpolation mechanism. With IP and ICMP on the link layer, the USRP2 uses well-known protocols which have rich support with a lot existing tools and libraries. The USRP2 has the IP 192.168.10.2 as standard. It supports things like the ICMP ping messages and ARP.

Since transmissions has to be done fast and thus one cannot afford things like connection establishment and complex transport protocols, UDP is used as the transport layer protocol. To distinguish in a fast way between control- and data-packets, the inspector uses different UDP ports:

```
#define USRP2_UDP_CTRL_PORT 49152
#define USRP2_UDP_DATA_PORT 49153
```

Every received and processed control-packet becomes acknowledged to the host by a corresponding response. For further details I refer to the source files

- `host/lib/usrp/usrp2/fw_common.h`
- `firmware/microblaze/usrp2/ethernet.c`

in the UHD git repository [3]. Figure 2.5 provides you again an overview of the USRP2 protocol stack.

UART

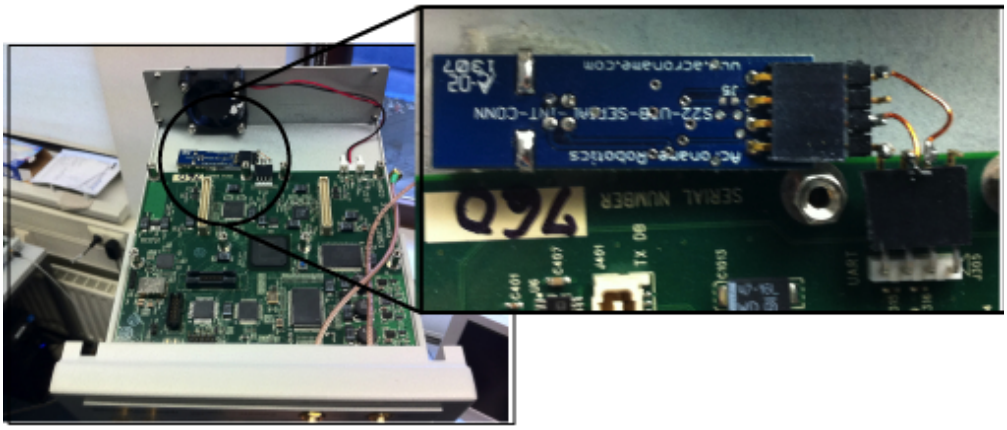


Figure 2.6: Connected UART interface of the USRP2's motherboard

To simplify debugging for the development on the USRP2 platform, it is recommendable to use the USRP2's serial interface. The J305 pin on the USRP2 motherboard provides an UART interface on TTL level with a baud rate of 230400 symbols per second. We used FTDI's FT232R USB 2.0 to serial UART IC to connect the USRP2 via USB to the host computer (see figure 2.6). This allows you to read the messages printed by the firmware with e.g. `printf` or `putstr`. After successfully connecting the UART interface to the host, the baud rate has to be set with a tool like `stty` (linux/Unix):

```
stty -F /dev/ttyUSB0 230400
```

After setting the baud rate, you can easily read the messages with e.g.

```
cat /dev/ttyUSB0
```

The firmware prints informations like the MAC address of its NIC, its IP address, compatibility numbers and so on. For common usage this is maybe kind of needless but it is very helpful in developing firmware.

So far, I gave an introduction to the aspects of the USRP2 architecture which are relevant for our firewall. I simply skipped the parts which are not touched by the wireless firewall. In the following chapter, I will present you the implementation and integration of the wireless firewall (WiFire) into this system.

3 Wireless Firewall (WiFire)

The over-all goal of this project is to create a device which protects an area (e.g. a room) from unauthorized wireless transmissions or – more generally – destroys unwanted traffic. This reduces or even eliminates the risk of attacks like message insertion or simply prohibits the usage of insecure protocols ($\leadsto$ rollback attacks). To cope with this task we use the already presented USRP2 and embed the WiFire functionality into the existing architecture. The USRP2 should then be able to do the following things

1. Listen on a specific frequency
2. Detect and demodulate 802.15.4 packets
3. Check if the packet breaches predefined rules
4. Destroy unauthorized packets by jamming

As one can imagine, this is a very time critical behaviour. Since a IEEE 802.15.4 packet has a transmission time less than 1 ms starting from the start of frame delimiter (SFD) until the end of the packet, the WiFire has to react as soon as possible after detecting the packet. There is of course not enough time for sending the signal data to the host computer, which analyses it and responds with the appropriate reaction. This leads to the necessity to implement the most of the the above mentioned points direct in firmware or hardware ($\leadsto$ FPGA). With these facts in mind, we decided to locate separate parts of the WiFire in the following parts of the USRP2 architecture.

The host is responsible for the setup and the tuning of the WiFire. There are several reasons for this decision. First, this has to be done once only on startup and this is not a time crititcal task. Tuning means the setup of the hardware in terms of setting the frequency, gain and the right antennas. Since there exists a bulk of code from GNU Radio doing exactly these things, there is no need to reimplement this on a much more complex platform. The setup of the software, i.e. the firmware comprises other settings like mode of operation (see below), jamming rules, jamming length and so on.

After setup is done, the firmware controls the WiFire without the need of a host. It handles the host-to-hardware- and hardware-to-hardware-communication by controlling the WISHBONE Bus. It further puts the buffer pool and the DSP components into the right state dependent on the mode of operation. Besides these tasks, it is responsible for the "usual" things mentioned in section 2.2.

The detection is done directly on the FPGA. Therefore, the WiFire FPGA module reads the samples, captured by the daughterboard and processed by the DSP, and searchs for 802.15.4 modulated signals.

So far so good. Let me now have a look at the concrete implementation and integration of the WiFire.

3.1 Architecture/Implementation

The main functionality of the WiFire is implemented on the FPGA and in firmware. So I skip the host-side stuff at this point and refer to section 3.3 for more details.

FPGA

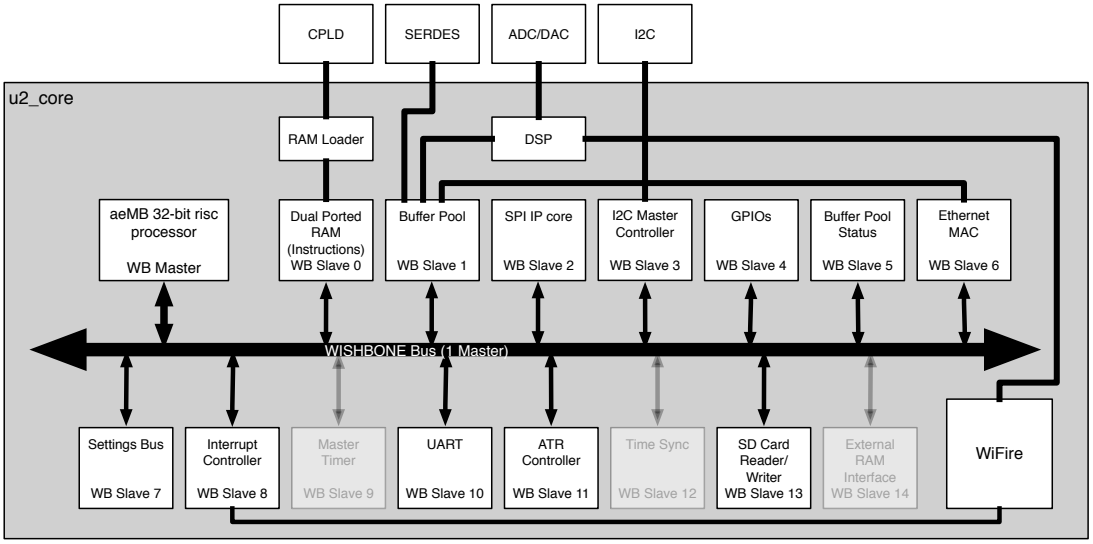


Figure 3.1: FPGA architecture with the WiFire component embedded

Figure 3.1 shows conceptionally the WiFire architecture running on the FPGA (compare figure 2.2). The module implements the 802.15.4 receiver. For that purpose, it is connected directly to the output of the DSP pipeline by simply passing the sample-wire in the module. Since these samples already contain the I- and Q-phase and our 802.15.4 demodulator is a I/Q-demodulator, there is no need to change anything in the existing DSP pipeline. The 15 bit I- and Q-phase of the samples are simply extracted in this way:

```
assign sample_i = sample[31:17];
assign sample_q = sample[15:1];
```

To allow the WiFire module to communicate with the firmware, it is also connected to the IC (slave 8). If the WiFire module detects a 802.15.4 packet, it sends an IRQ via this connection. Compared to the code snippet in section 2.2, the IC's extended IRQ port looks now like this:

```
assign irq= {{8'b0},
        {8'b0},
        {jamming_int_p, jamming_int_r, jamming_int_s,
                        periodic_int, clk_status, serdes_link_up, uart_tx_int, uart_rx_int},
        {pps_int,overrun,underrun,PHY_INTn,i2c_int,spi_int,onetime_int,buffer_int}};
```

There are 3 new IRQs added:

- `jamming_int_p` denotes that there is any signal on the carrier, i.e. there is something on the carrier which exceeds a defined threshold (very experimental, not yet tested).

- `jamming_int_r` becomes enabled if the *rule system* of the WiFire module signals a violation of at least one firewall rule. At the moment, the only rule is that the security enabled-flag in the 802.15.4 header has to be set. This is just for testing purposes and implementing a real dynamic rule system for several rules is future work.

- `jamming_int_s` indicates the detection of a 802.15.4 start of frame delimiter and thus the detection of a data packet.

To simplify the access to these IRQs from software, we added some preprocessor macros to the WiFire firmware:

```
#define IRQ_TO_MASK(x)            (1 << (x))

#define PIC_SOD_DETECTED_INT      IRQ_TO_MASK(13)
#define PIC_RULES_INT             IRQ_TO_MASK(14)
#define PIC_POWER_DETECTED_INT    IRQ_TO_MASK(15)
```

These IRQs can now be queried with this code:

```
uint32_t pending = pic_regs->pending; // pending interrupts

if (pending & (PIC_SOD_DETECTED_INT |
               PIC_RULES_INT |
               PIC_POWER_DETECTED_INT)) {
        //...
}
```

In the end, we didn't touch the original FPGA code but embedded our WiFire module beside. This has the big advantage, that the WiFire FPGA code can also be used for normal TXRX application with GNU Radio. Further reading concerning the detailed implementation of the WiFire module can be found in [2].

Firmware

The WiFire firmware (WiFireFW) isn't a complete reimplementation of the firmware, it is an extension of the original `txrx_uhd.c` firmware. So the initialization phase is basically the same as described in section 2.2. The behaviour of the firmware after this initialization depends on its state. The WiFireFW has 3 different states or *modes of operation* (`wifire.h`):

```
// wifire app states
typedef enum {
        STATE_TXRX,
        STATE_JAMMING,
        STATE_WIFIRE
} wifire_state_t;
```

Initially, the WiFireFW is in the `TXRX`-state. In this state, it behaves exactly like the original firmware (see section 2.2) what enables the use of GNU Radio. The only difference is that UDP packets with destination port 49154 are accepted. This port is reserved by the WiFireFW for WiFire control data (see section 3.2).

The second state is `JAMMING`. This state is actually for testing purposes. In this state, the WiFire simply starts to emit continuously the jamming signal. This cuts almost all traffic on the preset frequency off and can be used e.g. for placing the WiFire in a good jamming position during installation. The double buffering mechanism is not necessary in this state since we have to time critical data flow. The jamming data is written in the jamming buffer (`jam_buf`) once on startup and then continuously sent to the DSP. Also all incoming ethernet frames are handled in just one buffer (`eth_buf`).

The firewall behaviour is realized in the `WIFIRE`-state. During this state, the firmware keeps the hardware permanently in a receiving mode. This causes, that the WiFire FPGA module is continuously fed with signal data from the DSP. On an IRQ from the FPGA component, the firmware reacts accordingly. Such a reaction could be the emission of a short jamming signal by calling the buffer pool to send the jamming data to the DSP. This jamming data becomes stored in the jamming buffer in advance when changing the state to the `WIFIRE` mode of operation. The reaction depends on the firewall policies and settings. These policies are currently stored in the 1 byte wide `wifire_mode` variable. For now, the bits are assigned with the following settings (`wifire.h`):

```
// wifire modes
#define JAM_ALL_PACKETS     (1<<7) // 1 -> destroy plain packets
#define JAM_PLAIN_PACKETS   (1<<6) // 1 -> destroy encrypted packets
// there is room for more (bits 2 to 5)
#define PRINT_IRQ           (1<<1) // print out IRQ counters
#define PAUSE_WIFIRE        (1) // stop destroying packets temporarily
```

The initial mode is `JAM_ALL_PACKETS`. A possible setup could be e.g.

```
wifire_mode = (JAM_PLAIN_PACKETS | PRINT_IRQ);
```

which means, that all packets with the security bit disabled become destroyed and each reception of an IRQ from the WiFire FPGA component becomes counted and this counter will be printed out on the serial port.

3.2 Communication

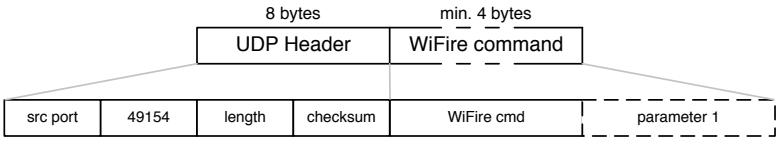


Figure 3.2: Structure of a WiFire UDP ctrl packet

As mentioned above, the interface of the wireless firewall for the user is the ethernet port. Besides the control data and "fast path" data ports used by the USRP2 and GNU Radio, the UDP port 49154 is reserved for control data from the host to the WiFire. To set e.g. the mode of operation or the firewall policies, one has to send a UDP packet to the USRP2 with the destination port 49154 and appropriate payload. Such a UDP control datagram contains a 32-bit wide command, followed by its arguments. Figure 3.2 shows the structure of a control data packet.

At the moment, the following commands are supported (`wifire.h`):

```
#define CMD_START_TO_JAM        1
```

The `START_TO_JAM`-command has no arguments. It stops the two double buffering state machines, changes the WiFireFW control state to `JAMMING`, fills the jamming buffer with signal data and start to emit a continuous jamming signal.

```
#define CMD_RESET               2
```

When receiving this command, the firmware resets all buffers, starts the tx- and rx-dbsm and changes the control state to `TXRX`. This can be used to reset the system on errors or to reuse the system as a transceiver for GNU Radio.

```
#define CMD_SETUP               3
```

This command changes the system state to `WIFIRE`. Therefore, it stops the double buffering mechanism, fills the jamming buffer and brings the hardware into a receiving state. From that point on, the systems acts as the desired firewall.

```
#define CMD_SET_JAM_LENGTH      4
```

The default jamming length is set to 1 buffer length, what corresponds approximately with a duration of 125 μs. This length can be changed by this command. This command takes the length in terms of buffers as parameter.

```
#define CMD_SET_MODE            5
```

This command is used to change the `wifire_mode` veriable mentioned in section 3.1. Its parameter is the new mode. Since the wordlength of the commands is 4 bytes, the byte which represents this new mode is packed into the least significant byte of the second word of the payload.

```
#define CMD_PING                6
```

To check if the system is alive, this command can be used. On reception, the WiFire prints out a "Pong." to the serial port.

```
#define CMD_CLEAR               7
```

This command clears the status flags of all buffers used by the WiFire. Be careful with this command because it cancels all running buffer transactions and so it can lead to a deadlock of the system. The `CLEAR`-command is only for testing purposes.

```
#define CMD_PRINT_COUNTERS      8
#define CMD_RESET_COUNTERS      9
```

As the names suggest, these 2 commands are used to control the IRQ counters. The `PRINT_COUNTERS`-command instructs the WiFire to print out the IRQ counters on the serial port and the `RESET_COUNTERS` resets these counters to zero. These counters are mainly supposed to provide statistical data for experiments.

3.3 Tools

For usability reasons, we developed two commandline tools to control the WiFire from a host computer. Therefore, the host has to be connected to the USRP2 which runs the WiFire FPGA code and firmware. They then communicate with the device as described in sections 2.3 and 3.2 with UDP messages over the gigabit ethernet link.

setup_tool

This tool tunes the WiFire. It is written in C++ and uses the GNU Radio C++ API to set things like center frequency, gain for the RF chain and the rx/tx-antennas.

Remark here that the tool explicitly must be terminated by using the

```
std::exit(0);
```

command at the end. This prohibits the invocation of the GNU Radio destructors which reset the USRP2. The compilation requires the same things like the C++ examples in the UHD host repository [3]. For informations about the usage of this tool, use the "--help" argument. An example usage could be

```
# setup the WiFire with a frequency of 2.48 GHz and a rx gain of 61 db
setup_tool --freq 2.48e9 --gain 61
```

wifireadm

After tuning the WiFire with `setup_tool`, one can control things like mode of operation, jamming policies and so on with `wifireadm`. This is a perl script which sends the aforementioned UDP packets with the commands to the WiFire. It doesn't support all commands mentioned in section 3.2, since many of them are just for debugging purposes. The script uses 192.168.10.2 as the WiFire destination IP and 49154 as the destination port for the UDP packets. An example usage is

```
# tell WiFire to start acting as a firewall, jam all
# plain packets and print out statistical
wifireadm --opmode wifire --jam_plain --reset --verbose
```

This tool also has a "--help" argument which provides you more details.

4 Problems

Well, the preceding topics may sound “straight forward” but there was a long way to go. There occured several problems of different kinds which will be mentioned in this chapter.I distinguish between problems caused by limitations (e.g. of the hardware) and problems concerning the development.

4.1 Limitations

By now, it should be obvious that a fast reaction time is very important for the firewall. But as one can imagine that detection and the selection and execution of the appropriate reaction needs time. Since the detection runs on the FPGA, this is done very fast and introduces less delay.

The firmware, which is executed by a 50 MHz clocked soft processor introduces more delay since many instructions have to be worked off until the device emits a jamming signal. Further, the firmware introduces some variance to the reaction time because the position of the PC in the firmware at the point of time when an event occurs is rather random. Of course this could be optimized but the more complex the firewall becomes, the more should be sourced on the FPGA because complex computations in firmware will result in unacceptable delays.

If doing so, one has to consider the limitations of the FPGA. Sadly there are finitely many memory and FPGA resources available and much of these resources are already used by the existing FPGA code. Detailed informations about remaining resources can be found in [2].

4.2 Development

Due to several reasons, the development of this firewall was very time consuming. First of all, there is a huge lack of documentation of the USRP2 firmware and FPGA code. The only source of informations about the stuff mentioned in section 2 were the FPGA and firmware source code and the mailing list. Methods like sniffing network traffic or reading code in different languages and programming paradigms (C, C++, Verilog) is not a very efficient way. The USRP2 is a complex interplay with many components involved.

Also a big issue was the debugging problem. Even though the serial interface helped a lot with developing the firmware, it also introduced new problems like so called *Heisenbugs*. Inserting debugging symbols sometimes solved the problem or produced new errors since printing also needs time and resources. Such sneaky bugs made the development more to a handicraft work. Mostly, the only thing you see when an error occurs is that nothing works. This is not very informative and leads to aggravating trouble-shootings.

But finally, due to its open nature, the USRP2 provided a suitable platform for the firewall.

5 Future Work/Open Topics

This last chapter can be viewed as a to-do or a wish list for the completion of the firewall. As mentioned in section 1, we focused on the development of a simply prototype which just proofs the concept but does not implement the whole firewall. There are several things skipped in this prototype.

An important thing missing is the rule system of the firewall. The prototype reduces the considered properties of the 802.15.4 transmission severely and makes a complex rule system impossible. There is e.g. the possibility to connect the WiFire FPGA module to the WISHBONE bus (slave 15), to provide a broader interface between firmware and FPGA module. This enables the firewall to pass things like complete header fields or payload to the firmware which is then able to apply its rule system on these things. The use of interrupts can then be reduced to the signaling of available data for the rule system.

The rule system could also be realized in a more dynamic way in terms of dynamic rule chains and so on. Therefore a protocol has to be created to manage, transfer and save the rules in a suitable form from the host. This may necessitate the use of the SD card interface to store rules and settings in a persistent manner.

Another thing is to make the WiFire completely stand-alone. Therefore settings (frequency, rules, etc) have to be stored on the SD card in advance. The firmware should then load these settings automatically and tune the device without the need of a host. This would facilitate the deployment of the firewall in larger experiments e.g. with multiple firewalls.

There are a few things running on the FPGA and the firmware which are actually not needed for firewall functionality. These things could be cleared out in case of lack of resources. If e.g. debugging is not necessary any more or the device is stand-alone, the UART and the ethernet FPGA and firmware stuff can be omitted and thus one can make way for more firewall stuff.

In any case, there is room for optimization using more suitable hardware. We used the XCVR2450 daughter board which does not really support full duplex, i.e. switching between reception- and transmission-mode needs time and cannot be done simultaneously. There are full duplex boards for the ISM band around (e.g. RFX2400). The firmware has “just” to be adopted to this board and thus archieve better performance.

References

[1] Crossbow Technology Inc, "MICAz data sheet," accessed on 11/01/2010. [Online]. Available: http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf

[2] S. Dyckmans, "Real-time Classification of Wireless Communication Using Software Defined Radios," Master's thesis, TU Kaiserslautern, Department of Computer Science, 2010.

[3] Ettus Research LLC, "UHD git repository," accessed on 11/08/2010. [Online]. Available: http://code.ettus.com/redmine/ettus/projects/uhd/repository/

[4] ——, "USRP2 data sheet," accessed on 11/01/2010. [Online]. Available: http://www.ettus.com/downloads/ettus_ds_usrp2_v5.pdf

[5] ——, "USRP2 General FAQ," accessed on 11/01/2010. [Online]. Available: http://gnuradio.org/redmine/wiki/gnuradio/USRP2GenFAQ

[6] GNU Radio. [Online]. Available: http://gnuradio.org