

CS476 Realtime Embedded Systems – Final project

Antoine Albertelli (205323)

Atri Bhattacharyya (269369)

June 12, 2017

1. Project description

For the final project in the “Realtime Embedded Systems” class, we decided to build an audio streaming system. Our system is pretty similar in principle to what would be used in a modern-day radio studio. It takes an analog sound input, converts it to digital values, compresses the audio, then broadcasts over the internet.

2. Hardware design

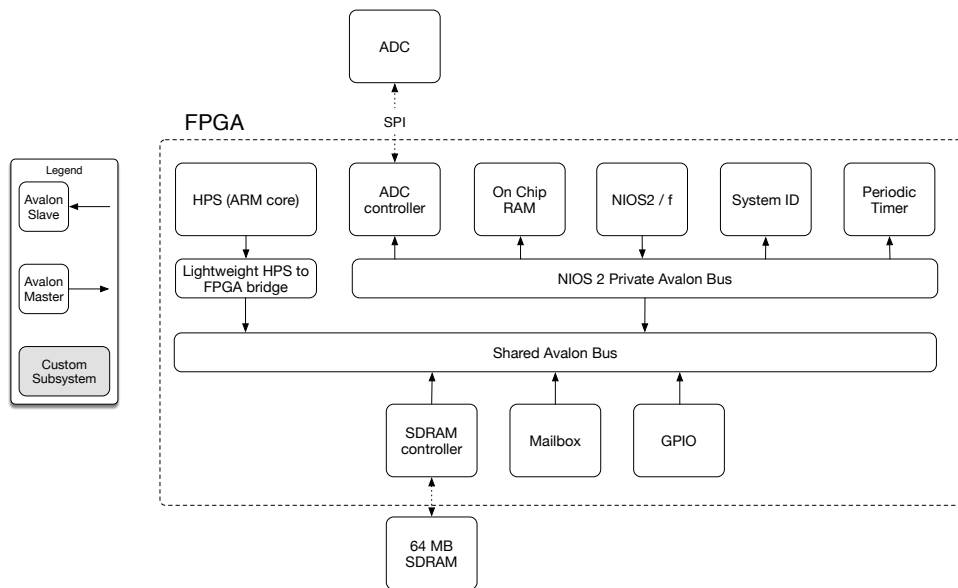


Figure 1: Overview of the test system.

The test system (figure 1) is organized around a common Avalon bus connecting several subsystems:

- A NIOS2 processor reads samples from the Analog-to-Digital Converter (ADC) and write them to the shared memory.

- An On-chip memory holds the code and data for the NIOS processor.
- An ADC controller handles the SPI communication with the on-board ADC, an AD7928 from Analog Device. The microphone is connected to the first input of the ADC.
- A System ID peripheral prevents downloading a version of the software not compatible with the hardware.
- A timer provides a periodic interrupt at 44.1 kHz, to schedule the audio sampling.
- An SDRAM controller provides a shared memory to both processors, used to hold audio samples.
- A mailbox transfers messages from the NIOS to the HPS.
- A GPIO port is connected to the onboard switches, to allow the user to change various parameters at runtime.

3. Software design

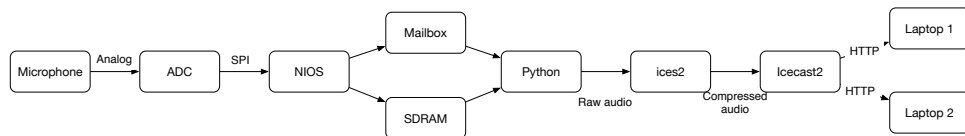


Figure 2: Software architecture of the system

The software for our project is split in two parts. The first part of the system runs on the NIOS processor. It manages the audio sample acquisition and storage in the shared memory. When enough samples (1024) are collected, it notifies the Linux system through a shared mailbox. It then switches to another buffer and starts recording again.

On the Linux side, one program is waiting for a buffer to be ready by polling the shared mailbox. When it happens, it reads the buffer and sends it to an external program for audio compression (IceS). IceS then sends the compressed stream to the broadcast server (IceCast) using TCP. Multiple clients can now connect to the broadcast server to listen to the stream.

Adding Linux in our environment allowed us to re-use a lot of existing code for audio compression and networking. However, as Linux is not real time, it cannot be used to guarantee a fixed and precise sample rate.

3.1. Accessing HPS peripherals from Linux userland

The HPS includes a bridge to connect Avalon peripherals to the ARM processor's bus. The Avalon peripherals get remapped with an offset of `0x2ff00000`, e.g. a peripheral whose Avalon address is at `0x00004000` can be accessed by the ARM processor at `0x2ff04000`. However, those are *physical* addresses; whereas processes running with an operating system such as Linux can only access *virtual* addresses. This means one cannot access a peripheral by de-referencing a pointer like it is commonly done on microcontrollers. This is done for security reasons, to make sure one process cannot read or write memory belonging to another process. However, processes running with superuser rights can do this by using the `/dev/mem` special file and the `mmap` system call. Listing 1 shows an example in Python, which writes a pattern to LEDs present at Avalon address `0x1000`:

Listing 1: Sample Python program showing peripheral register access

```

from mmap import mmap
import struct

GPIO_ADDR = 0xff201000

# Opens the file in read write mode (binary)
with open("/dev/mem", "r+b" ) as f:
    # Creates a memory view at the GPIO address
    mem = mmap(f.fileno(), length=0x10, offset=GPIO_ADDR)

# writes 0xaa to the first register (data) of the GPIO
mem[0x0:0x4] = struct.pack('I', 0xaa)

```

This approach has several advantages over writing a kernel module, especially in the development phase. First, the code is running in userland, which means it can be debugged by usual methods (stepping debuggers, console logging, etc.). Then, it can be written in an interpreted language, such as Python, which removes the need for a cross compiler.

However, it also has some drawbacks. First, the code must run as root, which is a security issue. Then, and perhaps most importantly, this approach does not allow IRQ handling. For example, we use our mailbox in polling mode on the Linux side rather than having an interrupt on message arrival. This makes the system less efficient.

4. Results

Audio recording from the NIOS is working. This means we can drive the ADC controller periodically and store those samples in our buffers. We had to pre-process samples, as the microphone had a DC bias. This was done using a first order high pass filter with a cutoff frequency of 20 Hz. Such a low cutoff frequency preserves most of the signal while quickly suppressing the offset. We decided to do this processing on the NIOS directly, since it can be implemented as a streaming process, using no additional buffers.

We also have a working communication with the Linux side. Every time a buffer is full, a message is sent on the shared mailbox. A small Python program (Appendix B) periodically polls the mailbox to see if any new sample arrived. If this is the case, those samples are recorded on the SD card. Using this mechanism, we were able to confirm sound was properly recorded and of reasonable quality (enough for spoken radio).

Unfortunately, the internet streaming solution is not finished yet. We were not able to configure Icecast to properly integrate with our programs. When playing back one of our samples recorded to disk, everything is working properly. However, when trying to connect the two programs to do real time audio (using a pipe), it does not work anymore. We are not quite sure of why this is happening, but our guess would be something related to buffering.

A. C program running on the NIOS

```

/*
 * NIOS Application acquire audio and transmit it to another processor
 * Created on: Jun 9, 2017
 * Authors:
 * Antoine Albertelli
 * Atri Bhattacharyya

```

```

*/
#include "sys/alt_stdio.h"
#include "altera_avalon_mailbox_simple.h"
#include "altera_up_avalon_adc.h"
#include "altera_up_avalon_adc_regs.h"
#include "altera_avalon_timer.h"
#include "sys/alt_irq.h"
#include "sys/alt_cache.h"
#include "../bsp/system.h"

#define MULT_FACTOR 0

/* Comment to disable filtering. */
#define FILTER

#define CHANNEL 0
#define NUM_BUFS 40

#define TIMER_RESET (0x1 << 0)
#define TIMER_START (0x1 << 1)
#define TIMER_STOP (0x1 << 2)
#define TIMER_INTEN (0x1 << 3)
#define TIMER_INTDIS (0x1 << 4)
#define TIMER_INTACK (0x1 << 5)

/* Filter parameters. */
const float alpha_low = 0.99956, alpha_high = 0.68113;

alt_u32 sample_count=0;
volatile alt_u32 which = 0;
alt_16 x_prev = 0, y = 0, z_prev = 0, z = 0, x;

alt_up_adc_dev *adc;
altera_avalon_mailbox_dev* mbox;
alt_u32 data[2] = {0xdeadbeef, 0};
__attribute__((section(".sdram"))) alt_16 buffer[NUM_BUFS][1024];

void adc_timer_isr(void)
{
    alt_u32 result;

    IOWR(CUSTOM_TIMER_0_BASE, 1, TIMER_INTACK);
    //Faster version of alt_up_adc_read(adc, CHANNEL)
#ifdef FILTER
    x = IORD(ADC_0_BASE, CHANNEL);
    y = alpha_low * y + alpha_low * (x - x_prev);
    x_prev = x;

    //Use z if high pass filtering is desired
    z = z_prev + alpha_high * (y - z_prev);
    z_prev = z;

    buffer[which][sample_count++] = y << MULT_FACTOR;
#else
    buffer[which][sample_count++] = IORD(ADC_0_BASE, CHANNEL) << MULT_FACTOR;
#endif
}

/* If we reached a full buffer, notify the HPS */
if(sample_count == 1024) {

```

```

        alt_dcacheflush_all();
        data[1] = buffer[which];
        result = altera_avalon_mailbox_send(mbox, data, 2, POLL);
        sample_count = 0;
        which = (which == NUM_BUFS - 1)? 0 : which+1;
    }
}
#endif

void setup()
{
    //Setup timer
    IOWR(CUSTOM_TIMER_0_BASE, 1, TIMER_RESET);
    // Interrupt at 44.1 Khz
    IOWR(CUSTOM_TIMER_0_BASE, 2, 1134);

    // Register interrupts
    alt_ic_isr_register(CUSTOM_TIMER_0_IRQ_INTERRUPT_CONTROLLER_ID,
                      CUSTOM_TIMER_0_IRQ,
                      adc_timer_isr, NULL, 0);

    // Enable interrupt
    alt_ic_irq_enable(CUSTOM_TIMER_0_IRQ_INTERRUPT_CONTROLLER_ID,
                     CUSTOM_TIMER_0_IRQ);

    //Setup mailbox
    mbox = altera_avalon_mailbox_open("/dev/mb_nios_to_linux", NULL, NULL);

    //ADC setup
    adc = alt_up_adc_open_dev("/dev/adc_0");
    alt_up_adc_auto_enable(adc);

    //Start timer
    IOWR(CUSTOM_TIMER_0_BASE, 1, TIMER_START | TIMER_INTEN);
}

int main()
{
    setup();

    /* The rest is happening in ISR. */
    while(1);
    return 0;
}

```

B. Python program running on Linux

Listing 2: This program reads audio samples coming from the NIOS and outputs them as raw uncompressed audio (PCM) files.

```

#!/usr/bin/env python2
"""
Reads audio data coming from the NIOS through the shared SDRAM.

Authors:
Antoine Albertelli
Atri Bhattacharyya
"""

```

```

from mmap import mmap
import time, struct
import sys

SDRAM_BASE = 0xff240000
SDRAM_SIZE = 0x40000

PERIPHERAL_BASE = 0xff201000
PERIPHERAL_SIZE = 0x30
LED_OFFSET = 0x0
MBR_OFFSET = 0x20

# Maps SDRAM and peripherals
f = open("/dev/mem", "r+b" ):
per = mmap(f.fileno(), length=PERIPHERAL_SIZE, offset=PERIPHERAL_BASE)
ram = mmap(f.fileno(), length=SDRAM_SIZE, offset=SDRAM_BASE)

def read_reg(mem, offset):
    return struct.unpack('I', mem[offset:offset+4])[0]

def read_shared_memory(address, length):
    return ram[address: address+length]

while True:
    # Waits for a message to arrive on the mailbox
    while (read_reg(per, MBR_OFFSET + 0x08) & 0x01) == 0:
        pass

    # Reads the message, which contains the audio sample address
    msg_addr = read_reg(per, MBR_OFFSET + 0x04)
    command = read_reg(per, MBR_OFFSET + 0x0)

    # Reads the audio samples and outputs it
    data = read_shared_memory(msg_addr, 2048)
    sys.stdout.write(data)

```