VOIP Project: System Design

Sarfraz Nawaz
Mark Niebur
Scott Schuff
Athar Shiraz Siddiqui

Overview

The purpose of this document is to describe in detail the design of our CSEE 4840 semester project: a voice-over-IP soft-phone. The document is divided into 3 main parts. Part I covers the design of the industry standard network protocols and application logic that we will use to implement VOIP. Part II covers the design of the system software for the phone, and finally part III covers the hardware design.

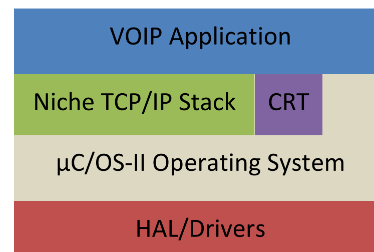
Before we dive into the detailed design of each of these sections, we should first talk about the high level system requirements. The requirements for a soft-phone device are the following:

- A way to "pick up" the phone, dial a number, and see who you are dialing
- A way for the phone to ring
- A way to speak into the phone
- A way to transmit/receive voice data
- A way to hear the person on the other end
- A way to hang up

In our system, these requirements will be correspondingly implemented by:

- A PS2 keyboard and the 2-line LCD display
- A speaker connected to the DE2 audio out, and the LCD
- A microphone connected to the DE2 audio in
- An Ethernet connection
- The same speaker used for ring
- The PS2 keyboard

To implement the system, we will use a NIOS II processor on the Altera FPGA, with a software "stack" that looks roughly as shown in the figure below. In the top layer, we have the VOIP application, including the network protocols and application logic. This will be built on the system software layer, which provides a C environment, a BSD sockets API for the network communication (via the TCP/IP stack), a small operating system for threading and synchronization, and drivers that abstract away the hardware details. At the lowest level, we have hardware peripherals, implemented on the FPGA, to drive the various human interface and communications components listed above.



With the high level picture now in place, we dive into the details of each of the 3 main system layers.

Part I: VOIP Application Layer

The VOIP Module

The VoIP module in the project would be implemented using a SIP/RTP based VoIP soft phone. The SIP protocol would be used to establish, modify, and terminate voice sessions. It encapsulates the SIP connection packets in TCP packets (using the socket API supported on the Niche Stack, provided by Altera via the BSP builder) and connects to the receiver FPGA soft phone. The actual data, comprising of voice packets would be communicated by transferring and receiving RTP based voice packets encapsulated in UDP packets (using the UDP Runtime Library), using the other half of the application module.

Session Initiation Protocol (SIP)

SIP is an application-layer control (signaling) protocol for creating, modifying, and terminating sessions with one or more participants. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences. SIP invitations used to create sessions carry session descriptions that allow participants to agree on a set of compatible media types. SIP makes use of elements called proxy servers to help route requests to the user's current location, authenticate and authorize users for services, implement provider call-routing policies, and provide features to users.

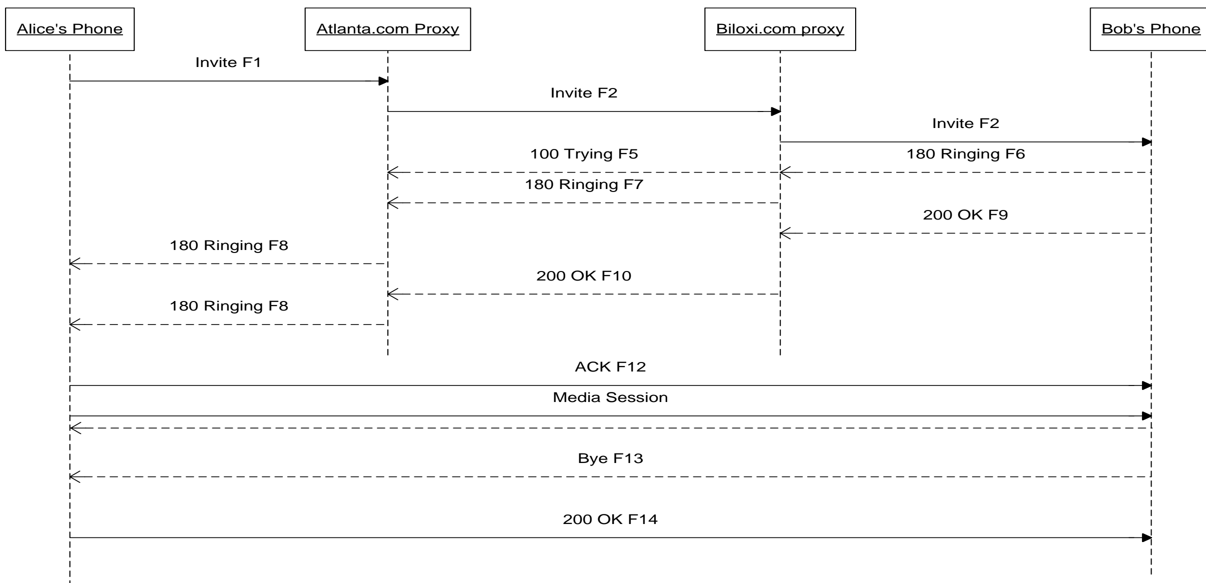
SIP is not a vertically integrated communications system. SIP is rather a component that can be used with other IETF protocols to build a complete multimedia architecture. We plan to use Real-Time Transport Protocol (RTP) for transporting voice data (additional would use RTP Control Protocol RTCP for providing QoS). SIP should be used in conjunction with other protocols in order to provide complete services to the users. However, the basic functionality and operation of SIP does not depend on any of these protocols.

Figure below shows an example of a SIP message exchange between two users, Alice and Bob. In this example, Alice uses a soft phone to call Bob on his SIP phone over the Internet. Also shown are two SIP proxy servers that act on behalf of Alice and Bob to facilitate the session establishment.

SIP is based on an HTTP-like request/response transaction model. Each transaction consists of a request that invokes a particular method, or function, on the server and at least one response. In this example, the transaction begins with Alice's soft phone sending an INVITE request addressed to Bob's SIP URI. Since the soft phone does not know the location of Bob or the SIP server in the biloxi.com domain, the soft phone sends the INVITE to the SIP server that serves Alice's domain, atlanta.com. In this example, the proxy server receives the INVITE request and sends a 100 (Trying) response back to Alice's soft phone. The 100 (Trying) response indicates that the INVITE has been received and that the proxy is working on her behalf to route the INVITE to the destination. The atlanta.com proxy server locates the proxy server at biloxi.com, obtains the IP address of the biloxi.com proxy server and forwards the request. The biloxi.com proxy server receives the INVITE and responds with a 100 (Trying) response back to the atlanta.com proxy server to indicate that it has received the INVITE and is processing the request. Bob's SIP phone receives the INVITE and alerts Bob to the incoming call from Alice so that Bob can

decide whether to answer the call, that is, Bob's phone rings. Bob's SIP phone indicates this in a 180 (Ringing) response, which is routed back through the two proxies in the reverse direction. When Alice's soft phone receives the 180 (Ringing) response, it passes this information to Alice, perhaps using an audio ring back tone or by displaying a message on Alice's screen. In this example, Bob decides to answer the call. When he picks up the handset, his SIP phone sends a 200 (OK) response to indicate that the call has been answered. The 200 (OK) contains a message body, in our case the RTP packets. If Bob did not wish to answer the call or was busy on another call, an error response would have been sent instead of the 200 (OK), which would have resulted in no media session being established.

We plan to either establish a peer to peer connection on a local network between the sender and the receiver, by running a local sip server. If we could connect to the internet then we get URIs from http://iptel.org to identify the sender and the receiver.

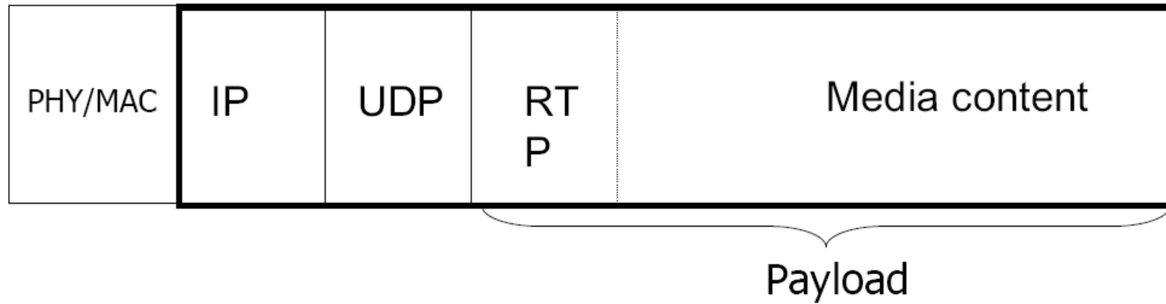


Real Time Transport Protocol (RTP)

RTP is a real-time end-to-end transport protocol. However, considering RTP as a transport protocol may be misleading because it is mostly used upon UDP, which is also considered as a transport protocol. On the other hand, RTP is very closely coupled to the application it carries. So, RTP is best viewed as a framework that applications can use to implement a new single protocol. RTP doesn't guarantee timely delivery of packets, nor does it keep the packets in sequence. RTP gives the responsibility for recovering lost segments and re-sequencing of the packets for the application layer. RTP provides the features -

Payload type identification, Source identification, Sequence numbering and Time stamping, which are required by most multimedia applications. The accompanying RTP Control Protocol (RTCP) provides feedback of the quality of the data delivery and information about session participants. A RTP session usually is composed of a RTP port number (UDP port), a RTCP port number (consecutive UDP port) and the participant's IP address.

Following illustrates the placement of this protocol in the header:



RTP header illustrated below provides information for:

- media content type
- talk spurts
- sender identification
- synchronization
- loss detection
- segmentation and reassembly
- security (encryption)

The RTP packet format (Table 1) is in detail reviewed in the table below:

<table>
<tr><td>V</td><td>P</td><td>X</td><td>CC</td><td>M</td><td>PT</td><td>Sequence Number</td></tr>
<tr><td colspan="7">Timestamp</td></tr>
<tr><td colspan="7">Synchronization source (SSRC) identifier</td></tr>
<tr><td colspan="7">Contributing source (SSRC_1) identifier</td></tr>
<tr><td colspan="7">…</td></tr>
<tr><td colspan="7">Contributing source (SSRC_1) identifier</td></tr>
<tr><td colspan="7">P A Y L O A D</td></tr>
</table>

The first 32 bits of the header consists of several control bits. The version number (V) is currently 2. The padding bit (P) indicates if there is padding octets inserted at the end of this packet. Padding may be required by some applications with fixed length packet sizes. The extension (X) bit indicates if there is an experimental extension after the fixed header. The count field (CC) tells the number of contributing

source identifiers (CSRC) following the fixed header. The marker bit (M) may be used as general marker. The payload type (PT) field identifies the payload format. The sequence number is an incrementing counter which is started by a source from a random number. The timestamp corresponds to the generation instant of the first octet in the payload. The synchronization source identifier (SSRC) is a randomly generated value that uniquely identifies the source within a session. Even if it is very unlikely that two sources generate the same SSRC number, every RTP implementation should have a mechanism to cope with this chance. Following the fixed header there are one or more contributing source identifiers which are supplied by the mixer and the payload.

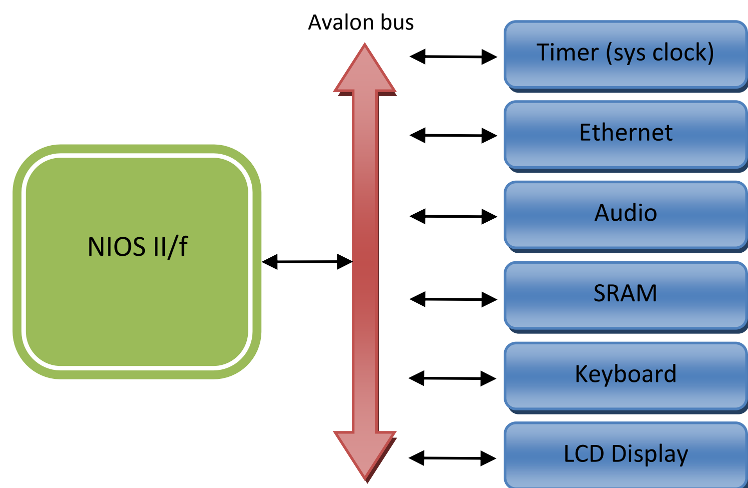
RTP Control Protocol

The RTP data transport is augmented by a control protocol (RTCP), which provides the RTP session participants feedback on the quality of the data distribution. The underlying protocol must provide multiplexing of the data and control packets, with UDP this is usually implemented using separate port numbers. The format of the RTCP packets is fairly similar to RTP packets, f.g. the type indication is at the same location. The main functions of the RTCP are - QoS monitoring and congestion control, Identification, Session size estimation and scaling. The RTCP packets contain direct information for quality-of-service monitoring.

Part II: System Software Design

Overview

It is the primary goal of the system software (and hardware) to provide a comfortable abstraction to the VOIP application layer, so that it may focus on the application logic and communications protocols. This section describes the design of the system layer and its various elements, and the considerations that went into the design process. For reference, a high level block diagram of the system looks as shown at right.



CPU

We will use the fastest available NIOS II processor, with no MMU or MPU, and a small data cache. The OS we plan to use, µC/OS-II, does not support paging anyway, so an MMU would be a waste. This CPU is expected to fit easily on the Cyclone II FPGA along with additional hardware peripherals and controllers.

Memory

We plan to use the 512k SRAM chip on the DE2 board for code and data. Note that some peripherals will have additional buffer memory, and the NIOS II on the FPGA will include cache memory, and may be outfitted with a small amount of TCM (tightly coupled memory) for performance sensitive code if necessary.

OS

We plan to use the µC/OS-II operating system provided by the Altera NIOS board support package (BSP) builder. This provides multi-threading to the TCP/IP stack, system software, as well as the VOIP application. Threads are needed for the network stack, and allow the system software to keep ISRs very short (offloading their main processing to system threads). This provides good (read: small) interrupt latency, which keeps everything running smoothly.

TCP/IP Stack

We plan to use the Niche stack, also provided by Altera via the BSP builder. Note that this package requires the µC/OS-II operating system. This stack provides a standard sockets (sometimes called BSD sockets) abstraction to the application for network communication.

CRT

The C runtime library is provided by the Altera BSP builder via their newlib library, and provides the C environment for the system software and application layer software.

Peripheral Drivers

Peripheral drivers for most peripherals are trivial or already provided. A keyboard driver was given in lab2. The SRAM needs a simple controller (provided by Quartus SOPC builder), but no driver software. The timer is very simple (just set the rate, hook the interrupt, and increment an integer). The audio driver should be fairly simple, but not trivial. Some memory mapped registers will be provided to modify sampling rate, start/stop sampling, etc. Buffered audio will be read and written to/from the peripheral. When the peripheral buffer is half full, the CPU will be interrupted. The interrupt handler will signal a semaphore that wakes a driver thread that reads the data out of the peripheral.

The one potential hiccup here is the Ethernet driver. It appears Altera does not provide a full driver for the DM9000A Ethernet MAC chip on the board, although some sample code is given in an Ethernet “demo” (this was the code used during lab 2). This may take some considerable finessing to make it suitable for the VOIP application, which is mildly timing sensitive.

Design Considerations (or, what the hell makes us think this could work?)

I) Memory Footprint

The µC/OS-II is a very bare bones operating system, making it perfect for our application. It only supports basic multi-threading and a small set of synchronization primitives and not much else. There is no virtual memory or paging support, and thus no process abstraction, just simple threads. This makes the OS very tiny, but the tradeoff is some of the robustness that the ‘big’ operating systems provide (if a process crashes it doesn’t take down the system or even other processes). The OS was originally designed for small microcontrollers, and indeed runs on 8-bit micros with 4kb of memory. So the memory footprint of the operating system is expected to be tiny (likely less than 16k on the 32 bit NIOS). The TCP/IP stack we have chosen, the NicheStack from interniche technologies (www.iniche.com), is a light TCP/IP stack designed for embedded systems. Interniche claims that a full-featured

configuration of the stack including the sockets API requires about 30k on a 32 bit processor[1]. On top of this the stack must use some amount of memory for a packet pool, etc, but at our modest data rates (roughly 16-20 KB/sec for uncompressed audio), a pool size of 32 packets (48k) should be sufficient. The pool in this case represents several seconds of data, and the NIOS will have no trouble consuming it on that time scale.

System code size in terms of C library, drivers and application code is expected to be fairly modest for our application. We are allowing for a somewhat outrageously conservative estimate of 128k, with an additional 32k for the .const section (read only constant data such as string literals, etc).

Runtime memory consumption of the application should also be quite modest. We expect conservatively 6-8 threads, with varying stack sizes between 4 and 16k. If the average size is a fairly generous 8k, we should use no more than 64k of memory in thread stacks. We plan to use almost no heap memory (at least not dynamically allocated while a VOIP call is in progress). But we will estimate another 64k of heap used.

Memory consumer	Size
Constant data space	32k
Global data space	48k
Stack space	64k
Heap space	64k
Application code	128k
OS & TCP/IP stack code	64k
TCP/IP packet memory	48k
Total	**448k**

The SRAM on the DE2 board is 512k in size. It appears that we should be able to fit within it, with our conservative estimates, with a small amount of room to spare. So that's our plan. On the off chance we blow the limit, we may add a small tightly coupled memory on the FPGA, and offload performance sensitive code and data there (ISRs, any data used by ISRs, etc). Such an approach may be helpful for performance anyway.

II) Memory Bandwidth

Our goal for the memory bandwidth is to be able to move around all the data we need to using the CPU (i.e., without using a DMA). DMA is an extremely effective mechanism for moving data between peripherals and main memory without tying up the CPU, but it does complicate the project in a variety of ways (since we are using a cache), so we are hoping to get away without it. The trick to this is not to busy the processor so much moving data around that we aren't able to do all the processing we need. Since the data rates for the application are modest (about 40KB/sec for the audio data), and we have both instruction and data cache memory on the FPGA keeping the bandwidth demand from the core a bit lower, we think this is reasonable.

III) Interrupt Frequency

We expect interrupts from the timer, ethernet MAC, audio hardware and keyboard. The keyboard interrupt rate depends on how fast people can type, so it is extremely low (< 10 per second). The timer will be used as a "system clock" (Altera's term, not Scott approved :) for the operating system and we plan to set it for a rate of 1 kHz (although only preempting a running task if it has run for 25-50 system clock ticks). The audio peripheral buffers 128 audio samples before interrupting the processor, which

---

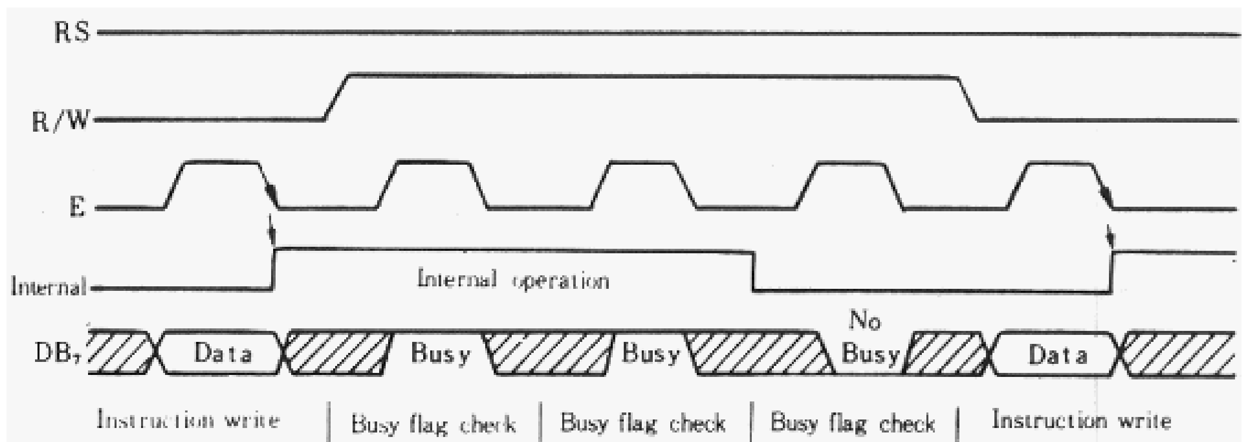
[1] According to the NichStack IPv4 datasheet. Oddly, a slightly different

translates to an interrupt every 12.5 milliseconds at our sampling rate. The ethernet MAC interrupts for every received packet. At a data rate of 20 kHz for incoming audio, that's actually a very low interrupt rate (< 20 per second if we fill the packets, which we plan to do). So we are talking about approximately 1200-1400 interrupts per second or so. Just as a sanity check, we look at interrupt overhead for the processor and find it is conservatively 200 ticks per interrupt (for NIOS II/f) according to the NIOS II Software developer's handbook (table 8-1). Add in another 400 ticks on average for the ISR body code (this should be a wild exaggeration given that the timer interrupt is only a few ticks of processing usually), and tallying up, we find we are using a CPU load of less than 2% for processing system interrupts. That is a reasonable number (i.e., it could double without any real cause for concern) indicating that this NIOS processor should be quite capable of handling our VOIP system load, with better than 95% of the CPU load available to the application.

Part III: System Hardware Design

The LCD Screen

The on-board LCD screen will be the display device showing call status and connection information. The interface to the LCD screen is either 4 bit or 8 bit parallel with 4 status pins. On the DE2 board, it is set up in the 8 bit configuration. Data must be sent according to the commands below. Please note that the value of RS is 1.



The above picture does one write followed by one read. To do this, data must be presented on the 8 data pins and then the enable line is pulsed. To perform a read, change the R/W line to 1 and wait until the busy signal on DB7 goes to 0, then the data will be presented on the next pulse.

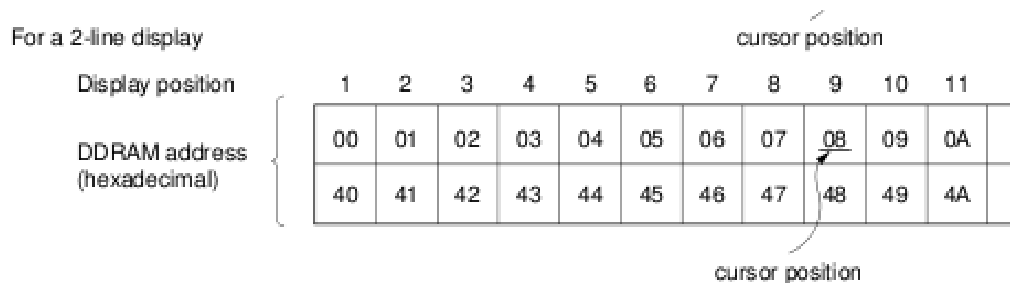
In addition to sending the LCD characters to display, the LCD can also be sent specific commands. This is done by setting the RS line to 0, and then writing or reading on the DB0-7 lines as above. All the possible commands are shown in the table below.

Instruction	Data Bit Value
Function set (8-bit interface, 2 lines, 5*7 Pixels)	0x38
Function set (8-bit interface, 1 line, 5*7 Pixels)	0x30
Function set (4-bit interface, 2 lines, 5*7 Pixels)	0x28
Function set (4-bit interface, 1 line, 5*7 Pixels)	0x20
Entry mode set	See Below
Scroll display one character right (all lines)	0x1E
Scroll display one character left (all lines)	0x18
Home (move cursor to top/left character position)	0x02
Move Cursor one character left	0x10
Move cursor one character right	0x14
Turn on visible underline cursor	0x0E
Turn on visible blinking-block cursor	0x0F
Make cursor invisible	0x0C
Blank the display (without clearing)	0x08
Restore the display (with cursor hidden)	0x0C
Clear screen	0x01
Set cursor position (DDRAM address)	80 + address

For the entry mode, there are four possible configurations.

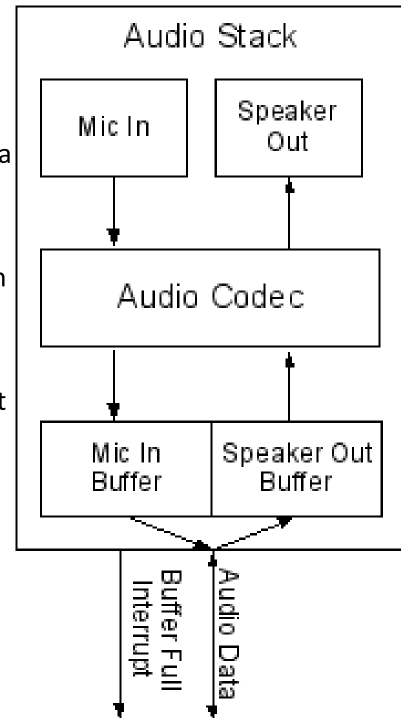
Instruction	Data Bit Value
The cursor decrements and the letters stay in the same place	0x04
The cursor stays in the same place and the letters shift right	0x05
The cursor increments and the letters stay in the same place	0x06
The cursor stays in the same place and the letters shift left	0x07

Finally, a picture of the address of all the cursor positions is shown below. In order to move the cursor to a specific location, add the number shown in the desired space below to 0x80 and send that as a command to the LCD.



The Audio Stack

The audio stack in this project will consist of five discreet components and one wrapper in order to encapsulate these components with a simple interface. The stack is laid out as shown below. The mic in is the output of an ADC that feeds a stream of data to the codec that make up discreet 16 bit samples. The audio codec takes this stream of values and groups them into the 16 bit samples. This is then stored in the audio buffer. When the buffer is half full an interrupt is issued to the processor notifying of the samples. When the processor services the interrupt it will read 256 samples. Any additional samples between when the interrupt is issued and when it is serviced will be put in the other half of the buffer and will be read when the next interrupt is serviced. The speaker out line works exactly opposite to the mic in. 16 bit samples will be taken from the buffer and turned into a stream of single data bits by the audio codec. When the buffer is half empty, an interrupt will be issued to the processor. Since the mic in and the speaker out use the same clock, we can combine both the interrupts into a single interrupt.



The audio stack, consisting of the physical devices, codec, memory, data and interrupt lines.

The wrapper will group all of these into a single entity so that when it is read, the mic data is sent, and when it is written, the speaker out data is given.

Conclusion

We have presented here the layered design of our VOIP system: Application Layer, System Layer, and Hardware Layer. We believe that the design elements that we have chosen for each of these layers will lead to a successful implementation, but there are some potential risks we identify here along with potential 'backup' strategies to mitigate these risks.

Risks

- Development time on Ethernet MAC chip (DM9000A driver)
- Possibly outgrowing the 512k SRAM

Potential Backup Strategies

- Re-arrangement of work to continue SIP/RTP protocol development on stable platform (linux) until MAC driver is functioning well.
- Using on-chip (FPGA) TCM or, worst case, use 8MB SDRAM.

Milestones

1. 25%
    - Initiate a SIP connection from Linux.

- Working OS and Stack on the NIOS II.
- Each of the peripherals working in isolation at a register-level.

2. 50% (Integration Milestone)
    - Working protocol on linux, nominally on NIOS II.
    - System Software, OS, TCP/IP all these functional.

3. 75% (Testing and debugging)
    - Working application with a few bugs and enhancements remaining.

References

1. Nios II Software Developer's Handbook (http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf)
2. NicheStack IPv4 Datasheet (http://www.iniche.com/pdf/nichestackipv4_ds.pdf)
3. µC/OS-II (http://www.micrium.com/products/rtos/kernel/benefits.html)
4. RFC 3261 (SIP: Session Initiation Protocol) (http://www.ietf.org/html/rfc3261#page8)
5. RFC 3261 (SIP: Session Initiation Protocol) (http://www.ietf.org/html/rfc3261#section-18)
6. RFC 1889 (RTP) (http://tools.ietf.org/html/rfc1889#page-3)
7. HD44780U LCD Display Datasheet (http://www.sparkfun.com/datasheets/LCD/HD44780.pdf)
8. LCD interface timing diagram (http://home.iae.nl/users/pouweha/lcd/lcd0.shtml#_8bit-transfer)
9. LCD interface commands (http://www.geocities.com/dinceraydin/lcd/commands.htm)