Universidade do Minho

Escola de Engenharia

Departamento de Electrónica Industrial

Criação de Hardware de comunicação Ethernet, para recepção de som

Sound – Ether

Cristiano Diogo Pereira Santos

Dissertação Submetida à Universidade do Minho para obtenção do grau de Mestre em Electrónica Industrial e Computadores

Outubro de 2008

Dissertação realizada sob a orientação do Professor Doutor António Fernando Macedo Ribeiro, Professor associado do Departamento de Electrónica Industrial da Universidade do Minho

Só sabemos com exactidão quando sabemos
pouco; à medida que vamos adquirindo
conhecimentos, instala-se a dúvida.

Autor: Johann Goethe

Agradecimentos

Tal como o título desta página cita, quero deixar aqui o meu profundo agradecimento a várias pessoas, que durante todo o processo de elaboração deste trabalho de alguma forma contribuíram, participaram, apoiaram, incentivaram e ajudaram para que este trabalho se tornasse realidade.

Em primeiro lugar queria agradecer aos meus colegas de laboratório, Bruno Matos, Sílvia Reis, Paulo Carvalho, João Guimarães, Luís Pacheco e André Oliveira por todo o companheirismo, ajuda e companhia durante o dia a dia na elaboração deste projecto. Aos colegas de curso Luís Monteiro, Luís Carvalho e Sérgio Silva pela ajuda que me deram, não estando no mesmo laboratório também foram umas pessoas que me apoiaram.

Queria agradecer ao meu orientador, Professor Doutor Fernando Ribeiro pela ajuda, apoio, oportunidade e confiança que me deu na elaboração deste projecto.

Queria agradecer ao Carlos Torres técnico das oficinas do departamento de electrónica industrial da Universidade do Minho pela ajuda e apoio na elaboração e construção física do Hardware, nomeadamente nos desenhos e construção dos PCB.

Um agradecimento muito especial à Marina Machado, por todo o tempo que não lhe pude dedicar, pela força, pelas palavras de incentivo e pelo tempo que lhe ocupei para me ajudar na elaboração desta tese.

Ao meu irmão Júlio Santos, por toda a exigência, incentivo e apoio que me deu durante a elaboração deste trabalho.

Por fim mas não menos importantes, queria agradecer aos meus pais, Domingos Santos e Maria Isaura Pereira pelo esforço que fazem, por todo o incentivo, por toda a compreensão e por todo o amor que me dedicaram durante todo este tempo da minha formação.

A todos o meu muito obrigado!

Resumo

A realização do módulo de Hardware, denominado de Sound-Ether, para a reprodução de som via Ethernet, tem como base, um microcontrolador da Dallas Semiconductor's compatível com a tecnologia 8051, um módulo Ethernet da Wiznet o NM7010B+, e um descodificador de *MPEG -1 Áudio Layer-3* (MP3).

Este Hardware é um cliente de uma rede *Transmission Control Protocol/Internet Protocol* (TCP/IP), onde vai receber os dados do ficheiro MP3 enviados pelo servidor através de sockets.

A programação do microcontrolador foi realizada em linguagem C utilizando a ferramenta *Keil µVision 2* V2.04.

A construção do Sound-Ether pode ser dividida em duas partes, a escolha dos componentes que compõem este Hardware e as configurações por software necessárias ao seu funcionamento. Esta construção física requer a realização de alguns passos intermédios, como a configuração do descodificador de MP3, a configuração do módulo Ethernet dependente da rede onde é inserido, e o controlo do fluxo de dados. Configurações necessárias tendo em vista os componentes físicos inseridos na construção do Hardware final, o Sound-Ether.

Inicialmente é configurado o descodificador de MP3, para estabelecer o modo de funcionamento e interacção com o conversor digital analógico (DAC) e com a fonte de dados, através do protocolo *Inter-Integrate Circuit* (I2C). Todo o processo de configuração, controlo e gestão de informação é feito através do microcontrolador, tornando-o assim no "cérebro" do Sound-Ether.

Após a configuração e inclusão na rede do Sound-Ether este necessita de estabelecer uma comunicação com um servidor disposto na rede local. Para isso o Sound-Ether fica infinitamente a enviar pedidos de estabelecimento de conexão ao servidor.

Estabelecida a conexão, pode-se proceder à transmissão de som, esta que é iniciada pelo servidor, enviando o número de dados que compõem o som que se pretende reproduzir, isto é, o tamanho do ficheiro MP3. Seguidamente o cliente irá fazer uma sincronização dos dados a receber, enviando um pedido, à medida que necessite de mais dados para reprodução, permitindo assim ao servidor enviar uma quantidade de dados do ficheiro MP3, até este ter chegado ao fim. O processo de transmissão de dados é repetido sempre que seja desejável reproduzir um som.

Abstract

The Sound-Ether hardware module permits the replay of sound via Ethernet, and has one Dallas Semiconductor's microcontroller based on 8051 technology, one NM7010B+ Ethernet module from Wiznet, and one *MPEG -1 Áudio Layer-3* (MP3) decoder.

This Hardware is a client from a *Transmission Control Protocol/Internet Protocol* (TCP/IP) network, which receives data from an MP3 file, sent by the server through sockets.

The microcontroller programming was carried out in C language using the *Keil µVision 2* V2.04 programming tool.

The Sound-Ether build up can be split in two parts. The components choice for the hardware to build and its software configuration required. This physical build up requires some intermediate steps, like the MP3 decoder configuration, the Ethernet module configuration depending on the network in use, and the data flux control. Also, the required configuration, having in mind the final Hardware physical components, the Sound-Ether.

Initially, the MP3 decoder is taken into account to establish the working mode and interaction with the digital to analog converter (DAC) and the data source, through the *Inter-Integrate Circuit* (I2C) protocol. The whole process of configuration, control and information management is carried out through a microcontroller, acting as the Sound-Ether brain.

After configuring and including it on the Sound-Ether network, it requires establishing a communication with the local network server. The Sound-Ether keeps infinitely sending connection requests to connect to the server.

Having this connection established, the sound transmission can start, started by the server, sending the data which makes up the sound to replay, i.e., the MP3 file size. Then, the client will synchronize the received data, sending a request as it requires more data to replay, allowing the server to send a quantity of data from the MP3 file, until it reaches the end. The data transmission process is repeated for each sound.

Índice Geral

Lista de Figuras

Lista de Tabelas

Abreviaturas e siglas

CD – *Compact Disc*

PCM – *Pulse Code Modulation*

MPEG – *Moving Picture Experts Gro*up

MP3 – *MPEG- 1 Áudio Layer 3*

AC3 – *Áudio Coding 3*

WMA – *Windows Media Áudio*

DVD – *Digital Video Disc ou Digital Versatile Disc*

LAN – *Local Area Network*

IP – *Internet Protocol*

UTP – *Unshielded Twisted Pair*

TCP/IP – *Transmission Control Protocol/ Internet Protocol*

UDP – *User Datagram Protocol*

ICMP – *Internet Control Message Protocol*

IPv4 – *Internet Protocol version* 4

ARP – *Address Resolution Protoc*ol

PPPoE – *Point-to-Point Protocol over Ethernet*

MAC – *Media Access Control*

I2C – *Inter-Integrate Circuit*

SDA – *Serial Data Line*

SCL – *Serial Clock Line*

MSB – *Most Significant Bit*

I^2S – *Inter-Ic Sound Bus*

SRAM – *Static Random Access Memory*

MTK – *Microcontroller Tool Kit*

TTL – *Transistor-Transistor Logic*

CI – Circuito Integrado

LE – *Latch enable*

MCU – *Microcontroller Unit*

MII – *Media Independent Interface*

PLL – *Phase Lock Loop*

CRC – *Cyclic Redundancy Check*

DAC – *Digital Analog Converter*

PCB – *Printed Circuit Board*

API – *Application Program Interface*

Introdução

A música é uma forma de arte, a origem da palavra provém do grego *"musike techne* – e significa *a arte das musas"* [1]. A música é uma forma de transmitir sensações e emoções, uma forma de tornar o espaço mais completo e alegre. E isto pode-se confirmar pelo provérbio, *"Quem canta seus males espanta"*, usado por todos nós, do qual já ouvimos dos nossos antepassados e certamente continuaremos a passar para gerações futuras.

Actualmente existem várias formas de a música chegar até nós; pode ser através do som ao vivo, das gravações de áudio ou áudiovisuais.

Através do som ao vivo, a música não sofre nenhuma transformação, e chega aos nossos ouvidos no mesmo instante em que os músicos a criam. Nas gravações de áudio, a música pode ser gravada de forma analógica e/ou digital, tornando assim possível ao ouvinte escolher o momento, o local e o tipo de música que quer ouvir. Facilitando assim as demonstrações e a divulgação da música, dos mais variados géneros e origens. Claro que este desenvolvimento aparece associado ao negócio da indústria musical, do qual se desencadeia a indústria de gravação e a indústria tecnológica.

Este desenvolvimento associado à geração multimédia e de redes existentes actualmente faz com que seja necessário criar formas de incorporar a música ou outras formas de som neste meio. Mas estes meios ainda têm as suas restrições, pois estão ainda muito limitados pela sua largura de banda, o que torna necessário criar constantemente novas formas de comprimir ou codificar o áudio para que não se comprometa a qualidade da música a reproduzir.

Presentemente, este desenvolvimento permite o aparecimento de várias formas de usar a rede ou a internet para divulgação e comunicação de áudio. Com este trabalho também se pretende contribuir para este desenvolvimento, que consistirá no desenvolvimento de um Hardware para integração numa rede local para reprodução de música ou outro tipo de som no formato MP3.

Ao realizar este trabalho pretende-se que o dispositivo criado possa ser aplicado numa rede local e que permita reproduzir áudio e seleccionar a saída de áudio pretendida para reprodução.

Face aos objectivos propostos, esta dissertação foi divida em VII capítulos. Estes estão estruturados de uma forma ordenada, abordando aspectos teóricos e técnicos até à obtenção da solução final.

No capítulo I será abordado o tema da teoria da música, o seu desenvolvimento e a tecnologia envolvente, os tipos e formatos de gravação de áudio. Neste capítulo também será feita uma pequena abordagem às aplicações possíveis do projecto desenvolvido bem como algumas informações sobre o formato de áudio utilizado.

No capítulo II será feita uma pequena abordagem ao estado da arte, será feita uma análise e verificação de sistemas que também permitam reproduzir áudio enviado pela Ethernet.

No capítulo III é feita uma abordagem teórica aos tipos e protocolos de comunicação usados na elaboração deste projecto.

O capítulo IV descreve a composição do Hardware construído, demonstrando o funcionamento de cada componente e a sua função no Hardware.

O capítulo V fala-nos sobre o funcionamento do hardware através do algorítmo implementado na função *main* desde a transferência, a reprodução e o armazenamento dos dados.

No capítulo VI são apresentados alguns testes e resultados efectuados com o Hardware criado.

No último capítulo é feita uma conclusão da solução criada face aos objectivos pretendidos, e uma análise ao trabalho que ainda pode ser realizado num eventual desenvolvimento futuro e melhoramento deste projecto.

Capítulo I

1 O áudio e o desenvolvimento da tecnologia

No mundo actual, as formas de armazenar áudio têm estado em constante evolução, para além de se terem aperfeiçoado, têm aumentado em número. Os formatos de armazenamento podem ser analógicos, que actualmente são muito pouco utilizados, e/ou digitais, que estão em constante crescimento e progresso.

Nas formas analógicas existem os discos de Vinil e as Cassetes. Nos discos de Vinil a gravação é feita mecanicamente através da criação de sulcos microscópios que provocam vibrações na agulha dos gira-discos. Nas cassetes a gravação é feita magneticamente através de uma cabeça de gravação, onde a fita é composta por um material magnetizável, guardando assim a informação analógica do som.

Fig. 1 Disco de Vinil e Cassete

Na gravação digital, o sinal analógico de áudio é convertido em sinais digitais, onde o formato *Compact Disc* (CD)[1] foi pioneiro. É um formato sem compressão de áudio, a uma frequência de amostragem de 44.1 ou 48 kiloHertz (kHz), usando *Pulse Code Modulation* (PCM) com uma resolução por amostra de 16 bits, o que resulta respectivamente para as amostras numa taxa de 705.6/768 kbits por segundo (kbps) para um só canal (mono) e uma taxa de 1.41/1.54 Mbits por segundo (Mbps) para dois canais

[1] Criado pela Philips e pela Sony

(estéreo). Embora com tamanhos altos, este formato teve bastante sucesso como primeira geração de áudio digital, através dos CD.

Mas com o desenvolvimento das tecnologias e o surgir de uma nova geração de aplicações multimédia e de redes que estão frequentemente sujeitos às larguras de banda, tornaram este formato incompatível. Passou a ser necessário criar formatos com taxas menores mas que não comprometessem a qualidade de reprodução [2].

A Codificação de áudio ou compressão de áudio são algoritmos utilizados para obter representações compactas de sinais de áudio de alta-fidelidade. O principal objectivo da codificação é obter o número mínimo de bits sem perder a qualidade de reprodução. Alguns dos formatos de compressão de áudio mais utilizados hoje em dia são, o *MPEG -1 Áudio Layer-3* (MP3)[2], o *Áudio Coding 3* (AC3)[3], o *Windows Media Áudio* (WMA)[4].

Com a evolução destes formatos, e de forma a tornar ainda mais flexível a sua portabilidade, foram aparecendo tecnologias de desenvolvimento e reprodução com a implementação de algoritmos de descodificação para poderem tornar o áudio codificado num sinal analógico audível pelo ser humano. Através destas tecnologias foram aparecendo descodificadores em Hardware e Software para facilitar a integração destes formatos em aparelhos electrónicos acessíveis ao ser Humano, como por exemplo, os leitores de MP3 portáteis, as aparelhagens de som e leitores de DVD que também reproduzem ficheiros MP3, WMA, etc., telemóveis e auto-rádios com leitores de MP3, os computadores que recorrem a codificadores implementados em Software, etc.

Na elaboração deste projecto, pretende-se desenvolver um Hardware capaz de reproduzir som enviado via Ethernet, e que possa ser inserido numa rede local (LAN). O formato de áudio escolhido para reprodução foi o MP3, devido a ser um formato compacto e um dos mais utilizados nos dias de hoje.

1.1 Aplicações possíveis do Sound – Ether

O sound-ether pode ser aplicado em diversas situações, apesar de ter sido inicialmente pensado para ser integrado num sistema de uma casa automatizada.

[2] Formato de compressão de áudio criado pelo Moving Picture Experts Group (MPEG)
[3] AC3 é também conhecido como Dolby Digital criado pelos Laboratórios Dolby
[4] Formato de compressão de áudio criado pela Microsoft

Pretendendo-se criar uma forma de enviar áudio pela Ethernet e de seleccionar uma determinada saída de áudio. O Sound-Ether iria funcionar como o reprodutor e selector da saída de áudio para cada compartimento de uma casa. A partir de um determinado ponto ou computador poder-se-á escolher a saída pretendida para reprodução do som.

O Hardware em questão foi construído com o intuito de receber áudio, e poder ser inserido numa rede local e que permitisse seleccionar a saída pretendida para reprodução. Tendo em conta que numa rede local os dispositivos são identificados pelo seu endereço *Internet Protocol* (IP), como o Sound-Ether foi idealizado para ser inserido numa rede local, decidiu-se que a forma de selecção da saída de áudio ou do local a reproduzir será feita através do seu endereço IP.

A casa necessitará de ter um dispositivo de reprodução (Sound-Ether) nos compartimentos em que se pretende reproduzir o som

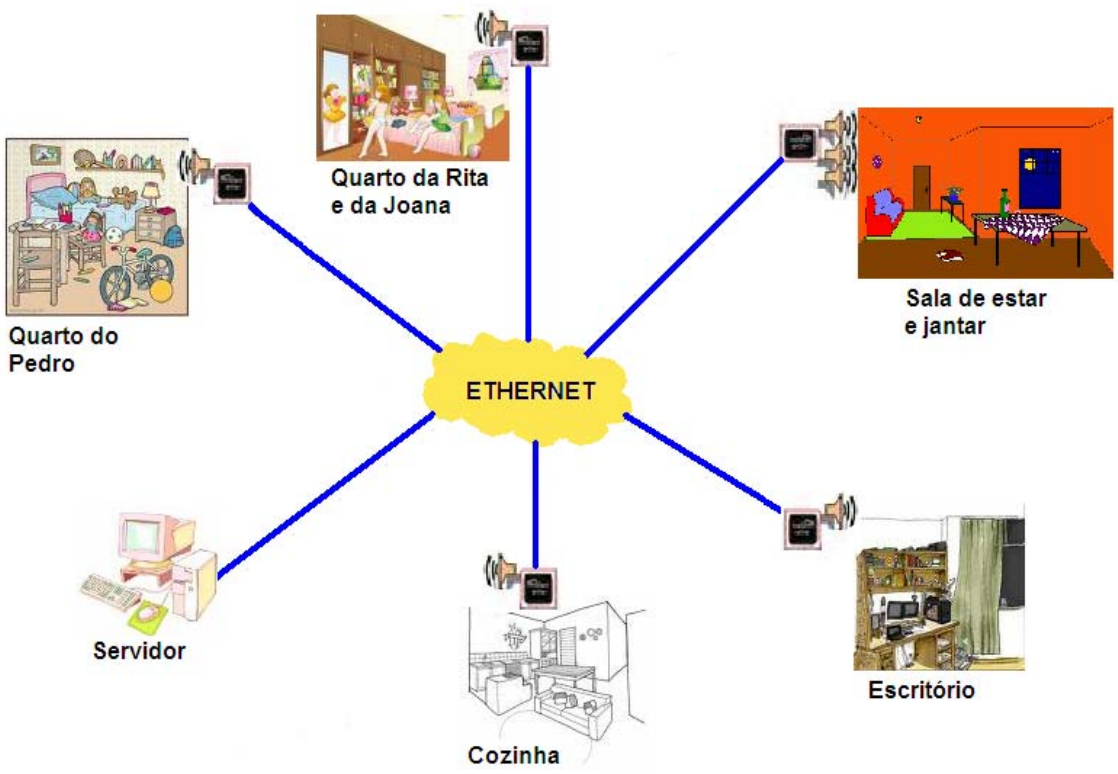


Fig. 2 - Exemplo de uma rede com o Sound-Ether

Esta forma de funcionamento faz com que este dispositivo seja portátil, isto é, possa ser usado noutras aplicações, pois poderá ser inserido numa rede local comum ligada a um servidor.

O dispositivo pode ser aplicado em diversas situações nomeadamente em meio hospitalar. Actualmente toda a gestão hospitalar é informatizada possuindo assim uma rede Ethernet em qualquer ponto do edifício, desta forma seria apenas necessário

conectar o hardware à rede e criar um ponto de ligação com o servidor, o dispositivo poderia reproduzir informações importantes enviadas de determinado local, ou mesmo reproduzir som em determinados locais, funcionando assim como uma forma de terapia.

Outra aplicação possível seria numa estação de comboios, pois permitiria que todas as informações relevantes fossem reproduzidas por este sistema, e sê-lo-iam no local relativo ao terminal em questão, permitindo a circulação de informação por toda a rede de comunicação. Com o mesmo intuito este sistema poderia ainda ser aplicado nos terminais de um aeroporto.

.

1.2 O MP3

O MP3 provém do nome MPEG Layer-3, e foi criado pela *Moving Picture Experts Group* (MPEG), que viu este formato ser standardizado em 1991. O MP3 emergiu como ferramenta principal do áudio pela internet.

O MP3 é visto como uma forma de compressão de áudio com perda de dados, todavia como essa perda é praticamente imperceptível ao ouvido humano não consiste num problema, pois a mesma recorre ao conhecimento das limitações do ouvido humano para o fazer. O MP3 é flexível de forma a ser aplicável em diferentes cenários, a sua compressão funciona em diversas frequências de amostragem, com taxas de conversão que podem ir desde os 8kbit/s até ao 320kbit/s, dispostos em, um canal, dois canais, Stereo ou *joint Stereo*[5]. O MP3 é obtido segundo um algoritmo de compressão baseado no seguinte diagrama de blocos [3].

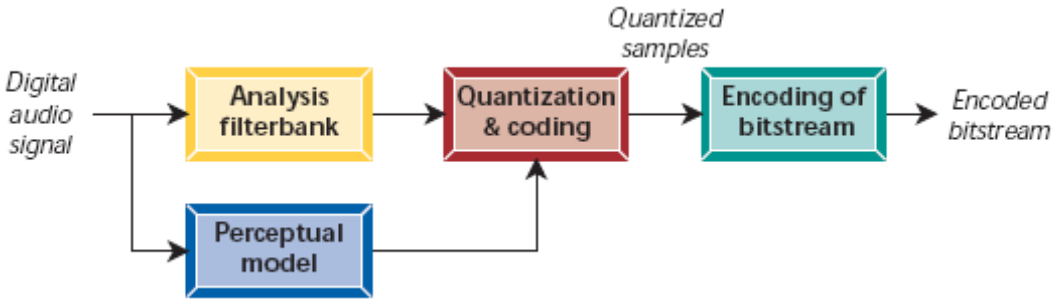


Fig. 3 - Diagrama de blocos típico do codificador de MP3 (retirada de [3])

[5] Técnica usada para fazer o acoplamento de canais, que melhora a eficiência de compressão aproveitando as similaridades entre os dois canais.

- Banco de filtros – usado para decompôr o sinal de entrada numa pequena amostra espectral no domínio tempo/frequência.
- Modelo perceptual – usa o domínio do tempo do sinal de entrada e/ou a saída do banco de filtros, recorrendo a técnicas usadas por computador do modelo psico-acústico para estimar uma máscara actual dependente do tempo e frequência.
- Quantificação e codificação – os valores espectrais são codificados e quantificados com o objectivo de manter o ruído, que é introduzido por quantificação, abaixo da máscara.
- Codificação do fluxo de bits – responsável pelo agrupar do fluxo de bits, que consistem nos coeficientes espectrais codificados e quantificados.[4]

1.2.1 A estrutura do ficheiro MP3

O ficheiro MP3 pode ser visto segundo o seguinte esquema:

{ [TAG v2] [Frame 1] [Frame 2] [Frame 3] [Frame 4]….[Frame N] [TAG v1] }

Em que o TAG é o nome para espaço de dados ou caracteres num ficheiro MP3, estes são opcionais e possuem alguma informação de texto relacionada com o artista, o título da música, nome do álbum, etc. Existem dois tipos de TAGs, o TAG v1, tem sempre o mesmo tamanho (128 bytes) e é colocado sempre no final do ficheiro, e tem o seguinte formato:

Tabela representativa da TAG v1		
Tamanho (bytes)	Posição (bytes)	Descrição
3	(0-2)	A TAG identificação. Deve conter o identificador 'TAG'.
30	(3-32)	Título
30	(33-62)	Artista
30	(63-92)	Álbum
4	(93-96)	Ano
30	(97-126)	Comentário
1	(127)	Género

Tabela 1 - Tabela TAG v1 do ficheiro MP3 (adaptada de [5])

E o TAG v2 é mais complexo, é sempre colocado no início do ficheiro e com um tamanho variável, possui um cabeçalho e algumas frames de dados.

O cabeçalho possui 10 bytes, com o seguinte formato:

Cabeçalho do TAG v2		
Tamanho (bytes)	Posição (bytes)	Descrição
3	(0-2)	TAG de identificação Deve conter o identificador 'ID3'
2	(3-4)	Versão da TAG
1	(5)	Flag[6]
4	(6-9)	Tamanho do TAG v2[7]

Tabela 2 - Tabela do cabeçalho do TAG v2 do ficheiro MP3 (adaptada de [5])

As frames de dados, não são iguais às frames de áudio e a soma de todas as frames de dados têm o tamanho dado pelo cabeçalho da TAG. Cada Frame de dados tem o seguinte cabeçalho:

Cabeçalho das Frames de dados		
Tamanho (bytes)	Posição (bytes)	Descrição
4	(0-3)	TAG de identificação
3	(4-7)	Tamanho da Frame
2	(8-9)	Flags

Tabela 3 - Tabela do cabeçalho das frames de dados do TAG v2 do ficheiro MP3 (adaptada de [5])

Para concluir temos as Frames de Áudio, que são as mais importantes, pois comportam os dados referentes ao áudio. Cada Frame contem informação de áudio correspondente a 26ms, assim o número de frames depende do tamanho da música, e cada frame tem um tamanho de dados variável dependendo da qualidade do som (numero de bit/s denominado BitRate) e da frequência da amostragem. O tamanho da frame pode então ser calculado pela expressão

$$Tamanho\ \ da\ \ Frame = \frac{144 \times BitRate}{Frequência\ \ Amostragem} + Padding$$

[6] Este byte é uma Flag com a estrutura abc00000, onde os três bits correspondem à dessincronização.

[7] Nos bytes correspondentes ao tamanho da TAG é ignorado o bit mais significativo, para não criar uma combinação mal sucedida com o cabeçalho da frame de áudio. E o tamanho deste cabeçalho não está incluído. Por exemplo, o tamanho de 257 é codificado da seguinte forma em Hexadecimal 00 00 02 01, e o tamanho total da TAG é de 257 + 10 bytes de cabeçalho.

Estes dados são encontrados no cabeçalho de cada Frame de áudio, que contém 4 bytes com o seguinte formato:

Byte 1	Byte 2	Byte 3	Byte 4
AAAAAAAA	**AAABBCCD**	**EEEEFFGH**	**IIJJKLMM**

Na tabela seguinte podemos verificar o que significa cada uma destas letras, (que correspondem a bits), para melhor entendermos o formato das frames de áudio do ficheiro MP3.

<table>
<tr><th>Letra</th><th>Numero de Bits</th><th>Posição</th><th>Descrição</th></tr>
<tr><td>A</td><td>11</td><td>(31-21)</td><td>Sincronização da Frame
Todos os bits devem estar activos</td></tr>
<tr><td>B</td><td>2</td><td>(20,19)</td><td>Versão do MPEG ID
00 - Versão MPEG 2.5
01 - Reservado
10 - Versão MPEG 2
11 - Versão MPEG 1</td></tr>
<tr><td>C</td><td>2</td><td>(18,17)</td><td>Descrição das Camadas (Layers)
00- Reservado
01- Camada III
10- Camada II
11- Camada I</td></tr>
<tr><td>D</td><td>1</td><td>(16)</td><td>Protecção do bit para a verificação da redundância cíclica
0 – Protegido
1 – Não protegido</td></tr>
<tr><td>E</td><td>4</td><td>(15,12)</td><td>Indicador da taxa de transferência de bits (kbit/s)
<table>
<tr><td>Bits</td><td>ID=1</td><td>ID=0</td></tr>
<tr><td>0000</td><td>Livre</td><td>Livre</td></tr>
<tr><td>0001</td><td>32</td><td>8</td></tr>
<tr><td>0010</td><td>40</td><td>16</td></tr>
<tr><td>0011</td><td>48</td><td>24</td></tr>
<tr><td>0100</td><td>56</td><td>32</td></tr>
<tr><td>0101</td><td>64</td><td>40</td></tr>
<tr><td>0110</td><td>80</td><td>48</td></tr>
<tr><td>0111</td><td>96</td><td>56</td></tr>
<tr><td>1000</td><td>112</td><td>64</td></tr>
<tr><td>1001</td><td>128</td><td>80</td></tr>
<tr><td>1010</td><td>160</td><td>96</td></tr>
<tr><td>1011</td><td>192</td><td>112</td></tr>
<tr><td>1100</td><td>224</td><td>128</td></tr>
<tr><td>1101</td><td>256</td><td>144</td></tr>
<tr><td>1110</td><td>320</td><td>160</td></tr>
<tr><td>1111</td><td>Proibido</td><td>Proibido</td></tr>
</table></td></tr>
<tr><td>F</td><td>2</td><td>(11,10)</td><td>Indicador das amostras da taxa de frequência
<table>
<tr><td>bits</td><td>MPEG1</td><td>MPEG2</td><td>MPEG2.5</td></tr>
<tr><td>00</td><td>44100</td><td>22050</td><td>11025</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td></td><td></td><td><table>
<tr><td>01</td><td>48000</td><td>24000</td><td>12000</td></tr>
<tr><td>10</td><td>32000</td><td>16000</td><td>8000</td></tr>
<tr><td>11</td><td>reserv.</td><td>reserv.</td><td>reserv.</td></tr>
</table></td></tr>
<tr><td>G</td><td>1</td><td>(9)</td><td>Enchimento
0 - Frame não permite enchimento
1 - Frame permite enchimento
Analisando o cálculo do tamanho da frame, verificamos que dependendo da taxa de bits o tamanho da frame pode não ser um número inteiro, a activação deste bit faz com que exista uma correcção do tamanho da frame.</td></tr>
<tr><td>H</td><td>1</td><td>(8)</td><td>Bit Privado. Pode ser livremente usado para necessidades específicas de uma aplicação.</td></tr>
<tr><td>I</td><td>2</td><td>(7,6)</td><td>Modos dos Canais
00 - Stereo
01 - Joint stereo
10 - Dois canais
11 - Mono</td></tr>
<tr><td>J</td><td>2</td><td>(5,4)</td><td>Modo de extensão (Apenas para joint stereo)
Este modo apenas é usado para o joint stereo, indicando que tipo de método é usado na codificação do joint stereo.<table>
<tr><td>Valor</td><td>Intensidade do stereo</td><td>MS stereo</td></tr>
<tr><td>00</td><td>Off</td><td>Off</td></tr>
<tr><td>01</td><td>On</td><td>Off</td></tr>
<tr><td>10</td><td>Off</td><td>On</td></tr>
<tr><td>11</td><td>On</td><td>On</td></tr>
</table></td></tr>
<tr><td>K</td><td>1</td><td>(3)</td><td>Patente
0- Áudio não tem direitos de autor
1- Áudio tem direitos de autor</td></tr>
<tr><td>L</td><td>1</td><td>(2)</td><td>Originalidade
0- Cópia de original áudio
1- Original áudio</td></tr>
<tr><td>M</td><td>2</td><td>(1,0)</td><td>Ênfase
00-nenhuma
01-50/15 ms
10- Reservado
11- CCIT J.17</td></tr>
</table>

Tabela 4 - Cabeçalho de uma frame de áudio do ficheiro MP3(adaptada de [5])

Capítulo II

2 Estado da arte

Neste tópico serão analisados alguns dispositivos que de alguma forma estejam relacionados com o objectivo deste trabalho, isto é, que de alguma forma reproduzam áudio enviado pela Ethernet. Através da pesquisa na internet, encontraram-se dois dispositivos que mais se aproximavam com o objectivo deste trabalho, o EtherSound – 100 Speaker e o Barix Exstreamer.

2.1 O EtherSound – 100 Speaker

O EtherSound é uma tecnologia criada e patenteada pela Digigram, fácil de implementar, síncrona, baixa latência, multi-canal, transporte bidireccional de áudio no formato PCM através de uma rede Ethernet. Esta tecnologia é compatível com o protocolo de comunicação IEEE 802.3x e opera no modo full duplex de uma rede Fast Ethernet (100Mbps).

As Frames do EtherSound são transportadas via uma Rede Local (LAN), e requer uma largura de banda de 100Mbps no modo full duplex, que pode ser aplicada numa topologia em estrela, em barramento, numa combinação entre estrela e barramento ou em anel.

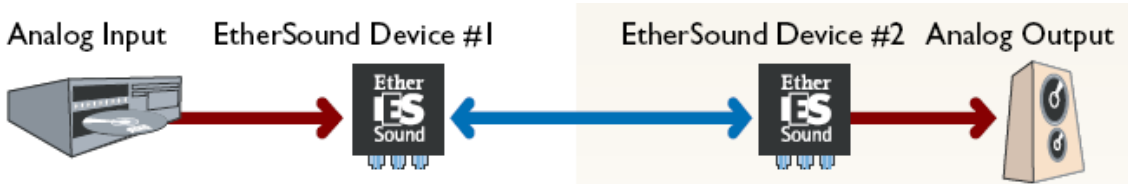


Fig. 4 - Funcionamento do EtherSound (retirada de [25])

Esta tecnologia para funcionar necessita da existência de um dispositivo *master* e um dispositivo *slave*. Em que o áudio entra no dispositivo *master* e pode ser reproduzido em qualquer um dos dispositivos da rede, controlados por um software de controlo instalado num computador ligado à rede. Como podemos observar na figura

seguinte uma aplicação típica através da implementação dos vários dispositivos desta tecnologia

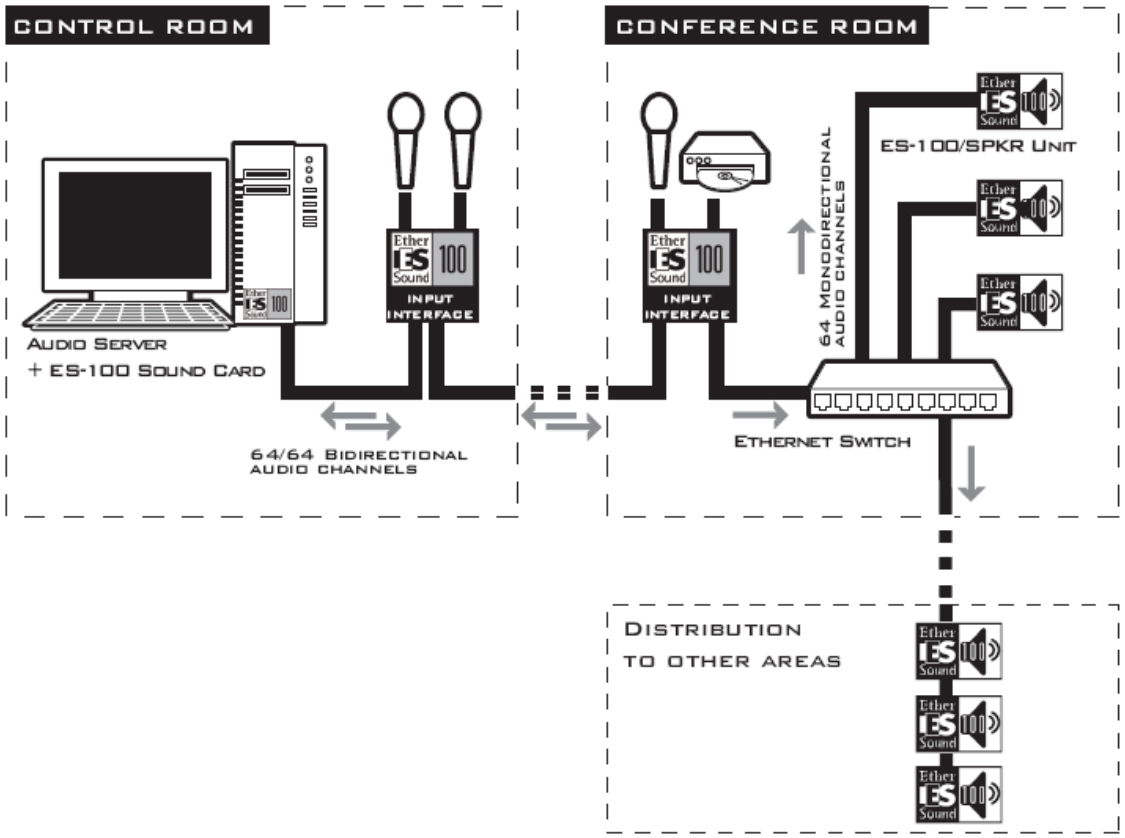


Fig. 5 - Aplicação EtherSound (reterida de [25])

Nesta aplicação podemos observar três tipos de dispositivos construídos pela Ethersound, o ES-100 Sound Card, o ES-100 e o ES-100/SPKR.

O ES-100/SPKR (Ethersound – 100 Speaker) é o reprodutor de som criado pela Digigram, um dispositivo variante da tecnologia EtherSound capaz de reproduzir áudio, enviado através de uma rede local com largura de banda de 100Mbps. Este dispositivo é unidireccional, apenas recebe e reproduz o áudio, que provém de um dispositivo que permita a recepção de áudio em sinal analógico, convertendo-o para um sinal digital para ser enviado pela Ethernet através do protocolo EtherSound desenvolvido por esta tecnologia, no formato PCM de 24 bits. Estes dispositivos são controlados por um computador ligado à mesma rede local.[25]

2.2 O Barix Extreamer

Esta tecnologia Extreamer foi criada pela Barix, denominada pelo Barix Extreamer, que é composta por quatro dispostivos, o Barix Extreamer 100, Barix Extreamer 110, Barix Extreamer 200 e o Barix Extreamer Digital.

Fig. 6 - Barix a) Extreamer 100, b) Extreamer 110, c) Extreamer 200, d) Extreamer Digital (retirada de [26])

A base destes dispositivos é a mesma, ambos reproduzem áudio enviado pela rede através de streams de áudio em formato MP3, permitem uma ligação Ethernet de 10/100 Mbps. Esta base está assente no Extreamer 100.

O Extreamer 110 é um melhoramento do Extreamer 100, sendo o principal melhoramento a inclusão de reprodução de mais formatos de áudio, o formato WMA e o formato sem compressão o PCM, e fisicamente foi acrescentado um display LCD de 16x2.

O Extreamer 200, é equivalente ao Extreamer 100, apenas incorpora um amplificar de áudio interno.

E por fim o Extreamer Digital é equivalente ao Extreamer 100, apenas incorpora uma saída digital de áudio para outro tipo de aplicações disponíveis ao utilizador.

Qualquer um destes dispositivos suporta MP3 até 320kbps. [26].

Capítulo III

3 As comunicações e protocolos

3.1 A Ethernet

A Ethernet foi criada nos anos 70 pela Xerox e consiste num protocolo de rede para redes locais *(LAN)*, sendo um dos tipos de rede mais utilizado, cuja função é baseada na ideia de criar pontos de rede enviando mensagens entre si. Estes pontos de rede estão assentes em topologias de rede que podem ser de vários tipos:

- Ponto a ponto – entre dois pontos distintos.
- Anel – os vários dispositivos estão dispostos numa rede em forma de anel.
- Estrela – os dispositivos tem de estar ligados obrigatoriamente a um posto central.
- Barramento – todos os dispositivos estão ligados a um barramento.
- Árvore – os dispositivos estão ligados de uma forma física que se assemelha a uma árvore.

As ligações podem ser feitas em três tipos de cabos, o cabo coaxial, o cabo *Unshielded Twisted Pair* (UTP) e por fibra óptica, dependendo dos tipos, podem adquirir as seguintes velocidades de comunicação:

- 10Mbit/s – 10Mbit Ethernet
- 100Mbit/s – Fast Ethernet
- 1000Mbit/s – Gigabit Ethernet
- 10Gbit/s – 10Gigabit Ethernet

A Ethernet define também o formato da informação e o método de acesso à rede. Onde os bits não são enviados de uma forma aleatória para a rede, mas sim numa trama com vários campos, como se pode ver na figura seguinte:

Preâmbulo	End. Destino	End. Origem	Tipo	Dados	FCS

Fig. 7 - Formato da trama Ethernet

O método de acesso à rede tem como característica enviar e monitorizar a rede, cujo objectivos são verificar se o meio está livre antes de enviar informação, caso o meio esteja ocupado continua a monitorizar até estar livre para poder enviar a informação, após isto e ao mesmo tempo que a estação envia a informação, monitoriza o meio para confirmar se o que aí circula é o que foi enviado. Estando assim a verificar se existiram colisões devido a transmissões simultâneas de informação e caso seja detectada uma colisão é lançado no meio um sinal de erro e todas as estações param de transmitir, e posteriormente uma nova transmissão irá ocorrer [8].

3.1.1 A comunicação TCP/IP

A comunicação é feita através do protocolo *Transmission Control Protocol/Internet Protocol* (TCP/IP), este que é talvez o protocolo de rede mais usado, pois a internet é basicamente uma mega-rede que usa o TCP/IP. Este protocolo é baseado no sistema de comunicação Cliente/Servidor. É neste conceito que se baseiam todas as aplicações de rede em que o *"cliente é uma aplicação que solicita um serviço e o servidor é a aplicação que o fornece"*[8].

O IP é um protocolo do nível de rede, não fiável e não orientado à conexão, cuja funcionalidade é fazer o encaminhamento da informação desde a origem ao destino inerente ao endereço usado. O IP em conjunto com protocolos de nível superior forma uma infra-estrutura da internet.

O TCP é um protocolo do nível de transporte, fiável e orientado à conexão, isto é, para haver troca de informação os sistemas têm de estabelecer uma ligação entre si e o receptor tem sempre de fazer o *acknowledgment* da informação recebida. A ligação é feita em três fases, o estabelecimento de conexão, a troca de informação e o fecho de conexão.

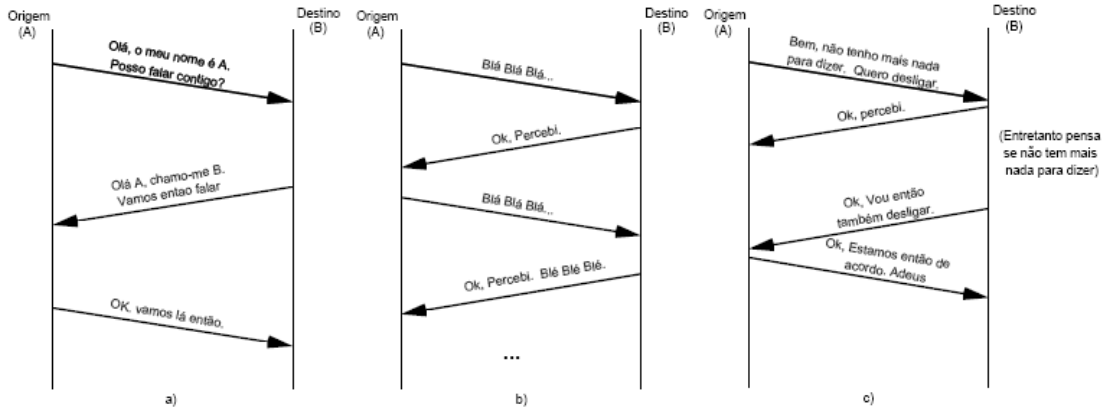


Fig. 8 - a) Estabelecimento da ligação; b) troca de informação; c) fecho de conexão (retirada de [8])

3.2 O Barramento I2C

O *Inter-Integrate Circuit* (I2C) é um protocolo de comunicação série criado pela PHILIPS, onde a comunicação é feita por um barramento, constituído por duas linhas, uma linha de dados denominada de serial data *(SDA)* e uma de relógio denominada de serial clock *(SCL)*, que permitem o endereçamento de vários dispositivos. Este sistema é gerido por um dispositivo principal, denominado de *Master,* que emite ordens aos dispositivos secundários, "*pendurados*" no barramento, denominados de *Slaves*.

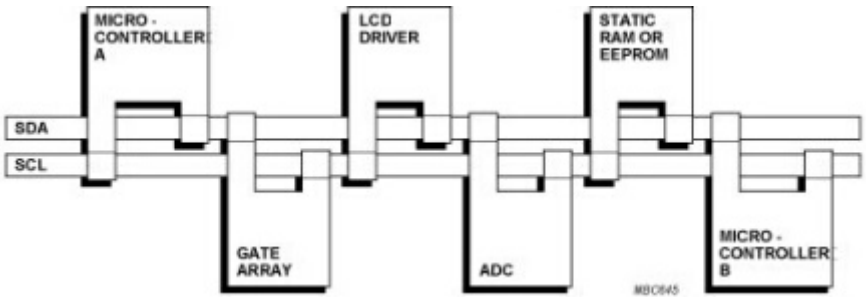


Fig. 9 - Exemplo de um barramento I2C (retirado de [9])

Nas linhas segue a informação em sinais digitais onde, dependendo dos dispositivos e da sua alimentação, os níveis de tensão podem variar. No I2C, a validação do dado da linha SDA ocorre quando o sinal na linha SCL está a nível alto, a velocidade de transmissão é sempre definida pela linha de SCL controlada pelo *Master*, é possível adicionar ou remover dispositivos no barramento sem este ser afectado, qualquer dispositivo pode operar como receptor ou transmissor.

O Start e o Stop são feitos através das transições ocorrentes nas linhas do barramento. O Start ocorre quando a transição do nível alto para o nível baixo da linha SDA enquanto a linha de SCL está a nível alto. O Stop ocorre quando existe uma transição de nível baixo para alto de ambas as linhas. Como podemos observar pela figura seguinte:

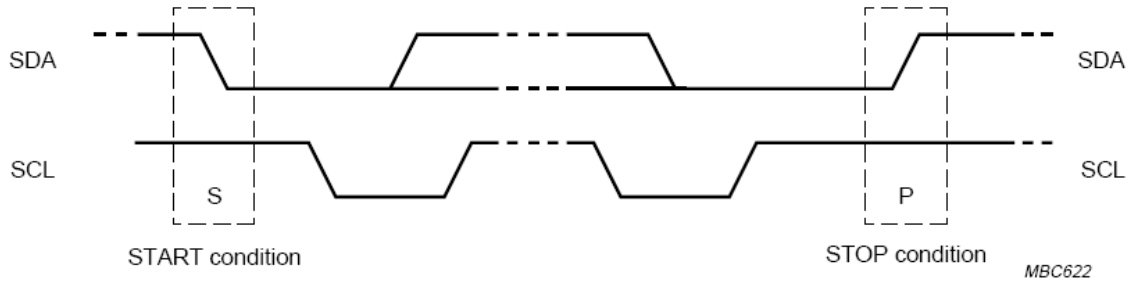


Fig. 10 - Condições de Start e de Stop do I2C (retirada de [10])

Cada transmissão é iniciada pelo *Master* com um sinal de Start, seguidamente são enviados os 7 bits de endereço do dispositivo que queremos comunicar e o bit que informa ao dispositivo da pretensão de leitura ou escrita, seguidamente o *Master* fica à espera de confirmação de *acknowledgment* do *Slave*. Se o *Slave* enviar o sinal de *acknowledgment* confirmando a comunicação ela é continuada, e o envio de dados é iniciado, sempre numa sequência de 8 bits e um *acknowledgment*, o sentido depende do pedido de leitura ou escrita anteriormente pedido pelo *Master*. A comunicação é encerrada sempre que não exista *acknowledgment* ou seja enviado um sinal de Stop.

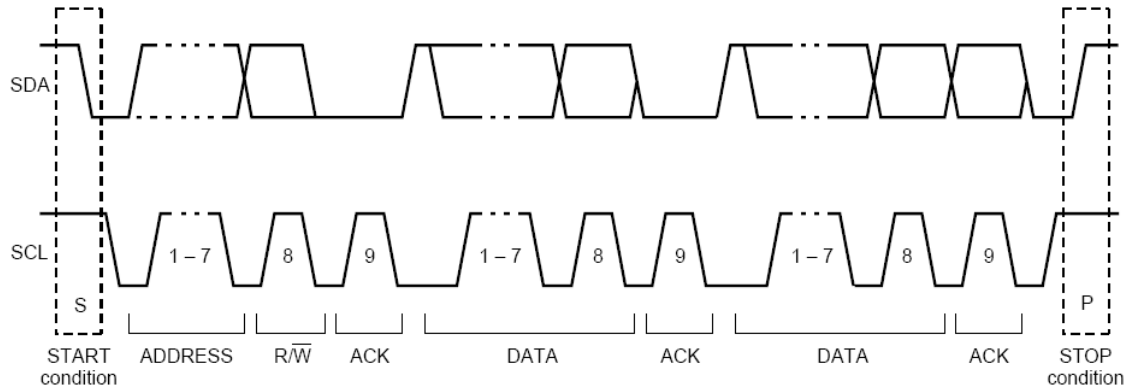


Fig. 11 - Exemplo de Uma transmissão completa (retirada de [10])

3.3 O *Serial Input Interface*

Esta é uma forma de conexão série síncrona bastante simples, em que normalmente só são necessárias duas linhas, uma de dados e uma de relógio. Em que é sempre enviado primeiro o bit mais significativo (MSB), através da linha de dados que se denomina de SDI e tem uma linha de relógio denominada de SCKR, que impõe a velocidade de transmissão. A validação destes sinais ocorre por transição da linha de relógio, que pode ser feita na transição alto para baixo ou de baixo para alto.

3.4 O *Inter-Ic Sound Bus (I^2S)*

O I^2S é um protocolo que consiste num barramento série desenvolvido para a tecnologia e dispositivos de áudio, tais como, o CD, processadores digitais de som, etc. Este barramento é constituído por três linhas, uma de relógio, uma de selecção da palavra e uma de dados que transmite os dois canais. Este protocolo é executado entre dois dispositivos em que um é o transmissor e o outro o receptor. Geralmente o Transmissor é o *Master,* é ele que controla todas as linhas do barramento, embora em sistemas mais complexos com vários transmissores e receptores é mais difícil definir o *Master*, sendo necessário nestes sistemas inserir um controlador para controlar o fluxo de dados de áudio entre os vários dispositivos [11].

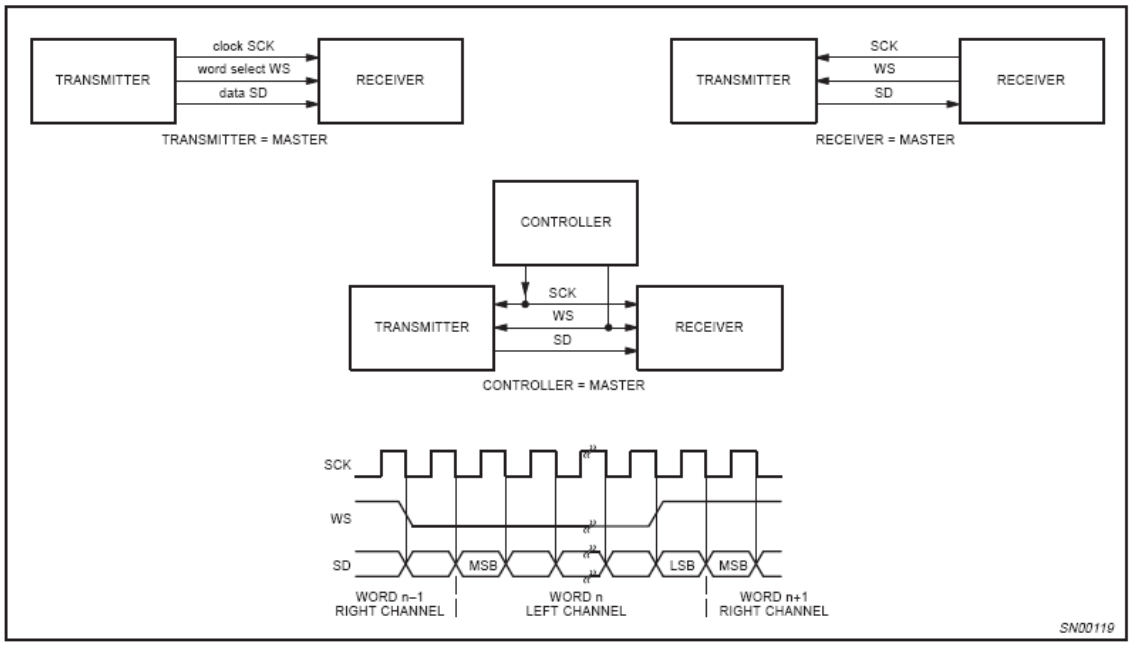


Fig. 12 - Simples sistemas de configuração e diagrama temporal (retirada de [11])

Neste protocolo, o bit mais significativo é sempre o primeiro a ser transmitido, uma vez que o transmissor e o receptor podem ter diferentes tamanhos de palavras. A validação dos bits é feita por transição do sinal de relógio (SCK), que pode ser de alto para baixo ou de baixo para alto. A linha de selecção da palavra (WS), define se transmite o canal esquerdo ou o canal direito e executa a mudança um período antes de ser transmitido novamente o bit mais significativo.

Capítulo IV

4 O Hardware

Na figura seguinte apresenta-se um diagrama do hardware implementado e o modo como os componentes estão interligados entre si.

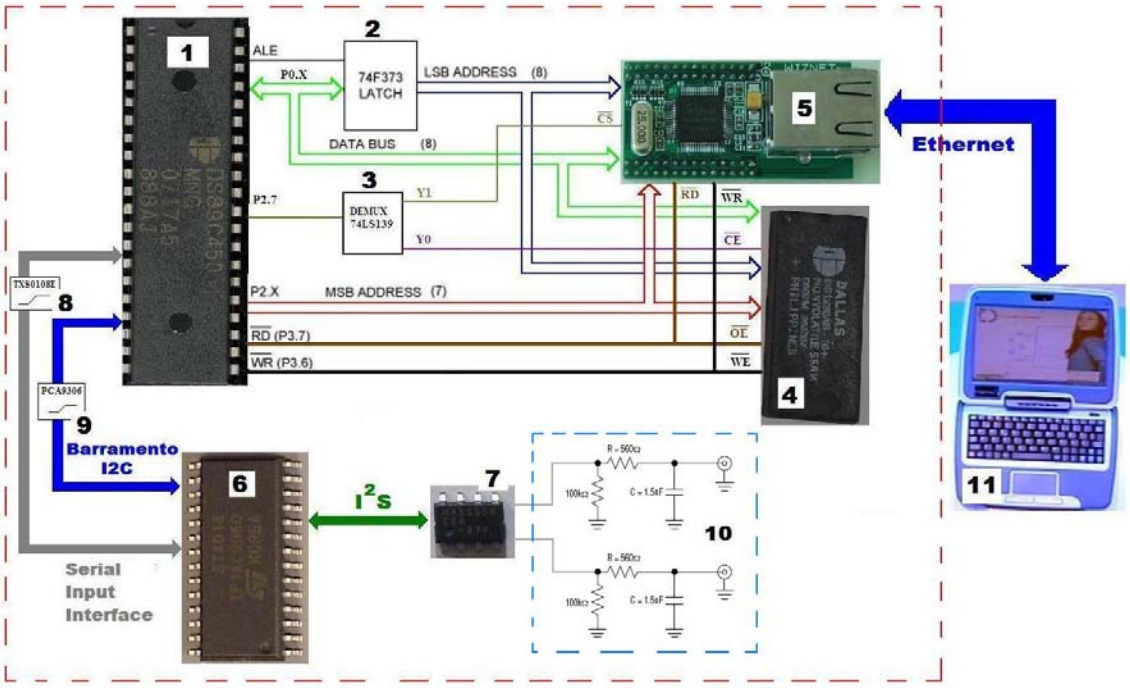


Fig. 13 - Diagrama do Hardware implementado

Neste diagrama observamos onze componentes diferentes e interligados das mais variadas formas. Todo este hardware tem um ponto principal de comando que é o microcontrolador compatível 8051, o *DS89C450* da *Dallas Semiconductor* (1), uma *Latch* do tipo D (2), um desmultiplexer (3), uma memória SRAM da *Dallas Semiconductor* a *DS1230AB* (4), um módulo de rede da *wiznet*, o *NM7010B* (5), descodificador de MP3 *STA013* (6), um conversor digital analógico de áudio (7), um tradutor bidireccional de níveis de tensão de 8 bits (8), um tradutor bidireccional de nível de tensão de barramentos I2C, PCA9606 (9), um filtro analógico passivo (10) e um computador (11).

4.1 O microcontrolador (DS89C450)

Com a necessidade de ter uma unidade de processamento na construção deste Hardware a escolha recaiu sobre um microcontrolador compatível com os 8051, pois alguns conhecimentos desta arquitectura já tinham sido previamente adquiridos durante a formação académica.

O microcontrolador *DS89C450* da *Dallas Semiconductor* é compatível com a tecnologia 8051, é um microcontrolador de 8 bits, com quatro portos de 8 bits de entrada/saída, possui uma memória interna de programa do tipo Flash com 64kbytes.

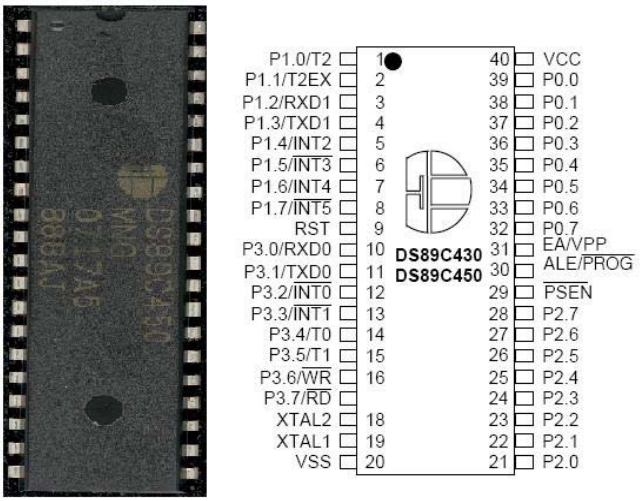


Fig. 14 - Descrição dos pinos do microcontrolador

Este microcontrolador possui treze interrupções das quais seis são externas, cinco níveis de prioridade, duas portas serie, três temporizadores ou contadores de 16 bits, permite acesso a todo o mapa de memória externo (64kbytes) e um conjunto de instruções compatível com o 8051 [16].

4.1.1 A programação do microcontrolador

Os microcontroladores executam instruções de baixo nível, isto é, são programados numa linguagem de baixo nível, denominado assembler. Cada microcontrolador tem o seu próprio conjunto de instruções (*Instruction Set),* e as instruçoes do assembler estão ao nível dos seus registos e endereços. O conjunto de instruções do DS89C450 é compatível com o conjunto de instruções do 8051.

O Assembly é uma linguagem que permite ter um maior controlo sobre o Hardware, permite economizar código, o que leva a economizar memória de código, um programa em Assembly é mais rápido que programas compilados por linguagens de alto nível, no entanto esta linguagem não permite portabilidade para outros microcontroladores diferentes, programas envolvendo várias tecnologias costumam ser muito grandes e quanto maior o programa maior a possibilidade de se introduzir bugs. Então, usando uma linguagem de mais alto nível consegue-se ter uma maior portabilidade, a menor quantidade de linhas de código para executar uma determinada tarefa, o uso de bibliotecas de comandos, a utilização de rotinas e funções, a facilidade de manipulação dos números e cálculos complexos, são algumas das razões pela qual as linguagens de alto nível se assumam perante as de baixo nível.

Neste projecto a linguagem adoptada foi a linguagem C, é uma linguagem de alto nível mas também permite a manipulação de variáveis de baixo nível. Esta linguagem hoje em dia é muito usada para programação orientada a microcontroladores.

No entanto os microcontroladores só executam instruções Assembly, o que para isso seja necessário transformar a linguagem C em linguagem assembly para o microcontrolador, o mecanismo usado chama-se de compilador. O compilador usado para programar este microcontrolador foi o *Keil C51 compiler,* que corre na ferramenta de programação *Keil µVision 2*.

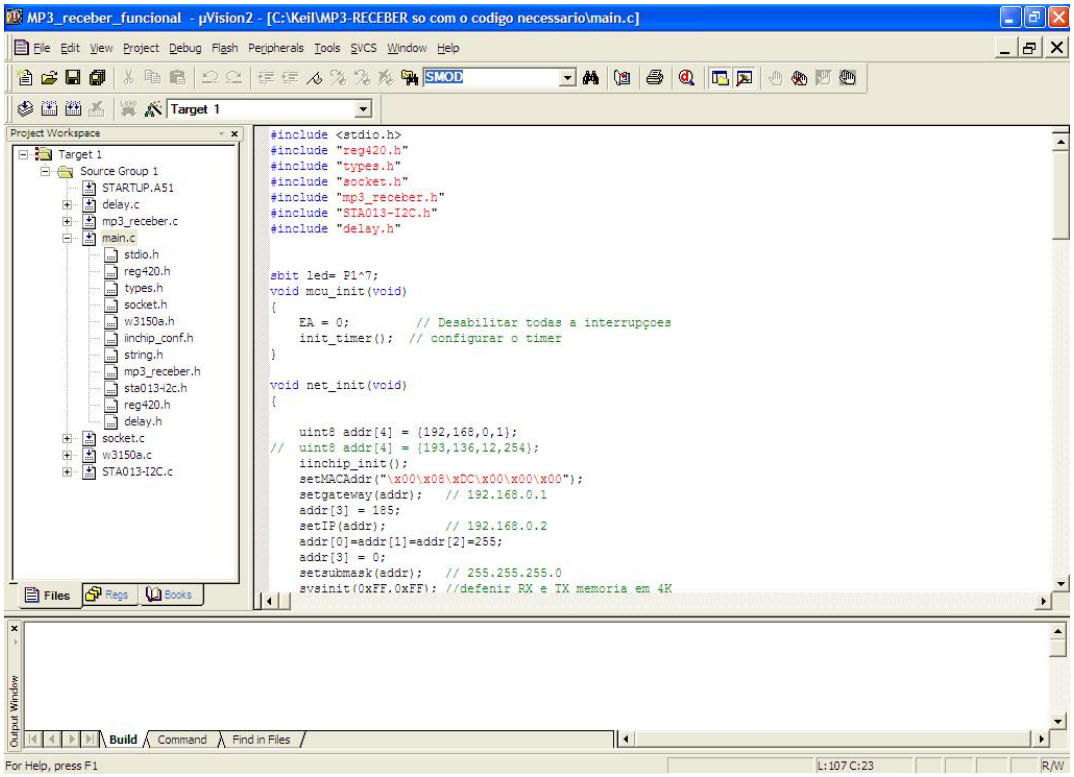


Fig. 15 - Imagem do *Keil µVision 2*

Este compilador compila e cria um ficheiro ".hex", que posteriormente tem de ser carregado para a memória de programa do microcontrolador através de uma aplicação de carregamento do programa.

4.1.2 A aplicação de carregamento de programa e o Loader

O Microcontroller Tool Kit (MTK) é uma aplicação usada para comunicar com a memória de programa da maior parte dos microcontroladores da *Dallas Semiconductor.* Esta aplicação tem como principal actividade configurar, carregar e descarregar código para o dispositivo conectado através de uma porta série de um computador. Mas também se podem distinguir outras funcionalidades, tais como:

- Carregar ficheiros para o dispositivo
- Verificar o conteúdo da memória dos dispositivos
- Apagar a memória dos dispositivos
- Controlar Hardware externo de carregamento via pino DTR da porta série.

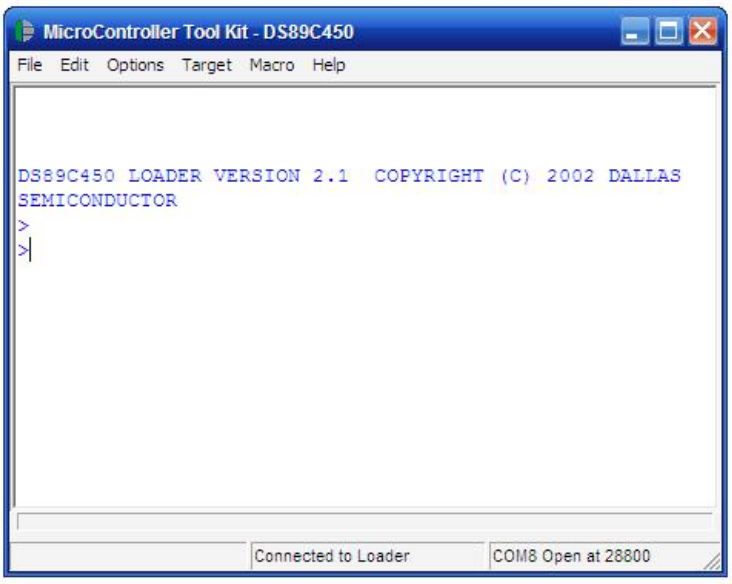


Fig. 16 - Microcontroller Tool Kit interface

Nesta figura pode-se observar o interface do MTK. É através deste software que se procede ao carregamento do ficheiro ".hex" criado pela ferramenta de programação,

Keil µVision 2. Na figura seguinte é possivel verificar o esquemático do Hardware de carregamento (Loader).

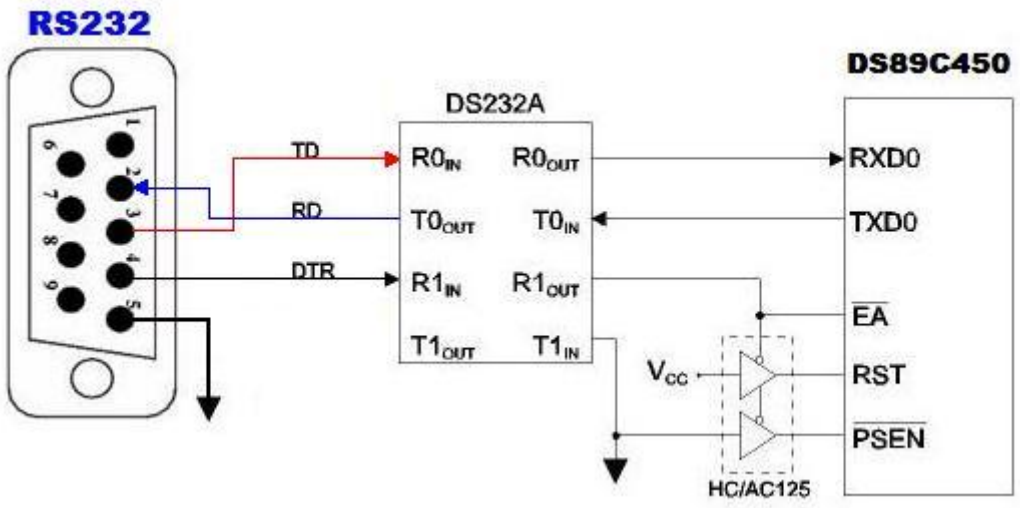


Fig. 17 – Esquema do Hardware de carregamento (Loader)

Este Hardware é composto por um driver, que converte os sinais RS232[8], para sinais *Transístor-Transistor Logic* (TTL[9]) e por um buffer (74HC125) para activar o modo de carregamento nos pinos do microcontrolador. Esta activação é feita pelo pino DTR da porta série, quando colocado a nível lógico baixo.

4.2 A latch (74HCT373)

Uma latch (trinco) é um elemento lógico que acompanha as variações dos dados e transfere as mudanças para a linha de saída. Uma latch serve para guardar ou memorizar um bit lógico. O 74HCT373 é um circuito integrado (CI) constituído por oito flip-flops do tipo D, com saídas de três estados para aplicações em barramentos. A latch do tipo D tem uma única entrada (D), que funciona como entrada de um bit de dados e é caracterizada por manter a saída com o mesmo valor de D, acompanhando qualquer variação da entrada.

[8] Os níveis de tensão do RS232 são entre os -3V (para o nível lógico "1") e os +15V(para o nível lógico"0")
[9] Os níveis de tensão para o TTL são entre os 0V (para o nível lógico "0") e os +5V(para o nível lógico"1")

Tabela da verdade

Inputs			Output
LE	$\overline{OE}$	D_n	O_n
H	L	H	H
H	L	L	L
L	L	X	O_n (no change)
X	H	X	Z

H = HIGH Voltage Level
L = LOW Voltage Level
X = Immaterial
Z = High Impedance State

Diagrama de ligações

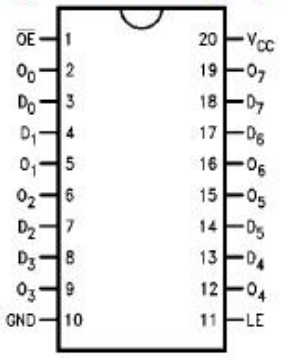


Diagrama Logico

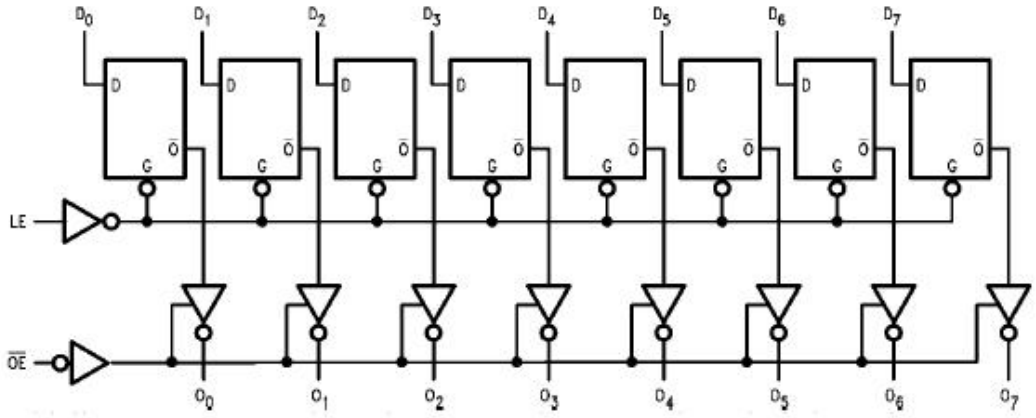


Fig. 18 - Diagramas do 74HCT373 (adaptado de [14])

Este CI é usado para fazer uma desmultiplexação do Porto 0 do microcontrolador, tornando assim disponíveis dois barramentos, um de endereço e um de dados, na saída de um único Porto. A desmultiplexação é accionada pelo pino ALE do microcontrolador que vai actuar no *latch enable* (LE) da latch. Quando o LE está a nível lógico alto a latch é transparente à entrada, isto é, na saída irá aparecer o que estiver na entrada, quando o LE estiver a nível lógico baixo, na saída D latch irá aparecer o valor anterior à transição de alto para baixo do LE. Isto acontece quando o pino de output enable está a nível lógico baixo, quando está a nível lógico alto as saídas da latch ficam em alta impedância.

4.3 O desmultiplexador (74LS139)

O desmultiplexador tem como função seleccionar uma saída mediante os níveis lógicos dispostos à entrada. O 74L139 é constituído por dois desmultiplexadores de 2:4, isto é, de duas entradas para quatro saídas.

Tabela da verdade

Inputs			Outputs			
$\overline{E}$	A_0	A_1	$\overline{O}_0$	$\overline{O}_1$	$\overline{O}_2$	$\overline{O}_3$
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	H	L	H	L	H	H
L	L	H	H	H	L	H
L	H	H	H	H	H	L

H = HIGH Voltage Level
L = LOW Voltage Level
X = Immaterial

Diagramas Lógicos

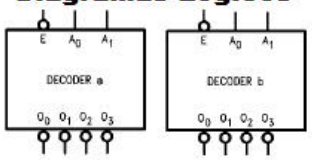


Diagrama Lógico

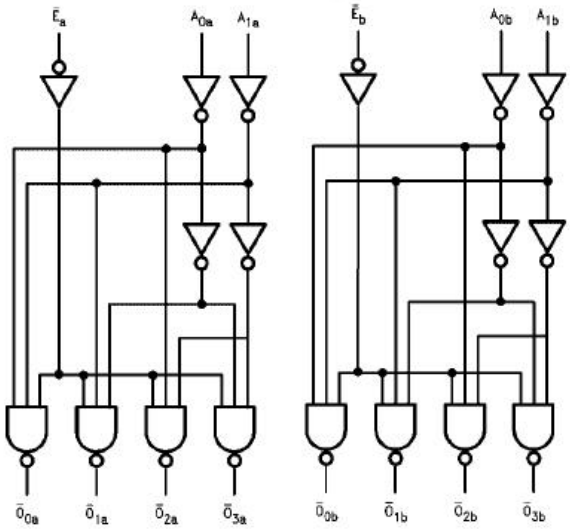


Fig. 19 - Diagramas do 74LS139 (adaptado de [15])

Através deste desmultiplexador faz-se a selecção de uma saída mediante os níveis lógicos dispostos à entrada, como se pretende usar duas memórias externas de 32kbytes, cujas posições vão desde o endereço 0x0000H ao 0x7FFFH, o que faz que apenas 15 linhas de endereço sejam necessárias para as endereçar, no entanto pretende-se endereçar na totalidade 64kbytes de memória, desde o endereço 0x0000H ao 0xFFFFH, tornando necessárias 16 linhas de endereço na totalidade. Então a solução foi colocar este CI a fazer selecção da memória que se pretende ler ou escrever, através da linha de endereço mais significativa, o que ficaria uma memória a ocupar os endereços desde o 0x0000H até ao 0x7FFFH e a outra desde o endereço 0x8000H até ao 0xFFFFH, sendo seleccionadas pela linha de endereço mais significativa do microcontrolador, tornando assim possível endereçar 64kbytes de memória com duas memórias de 32kbytes.

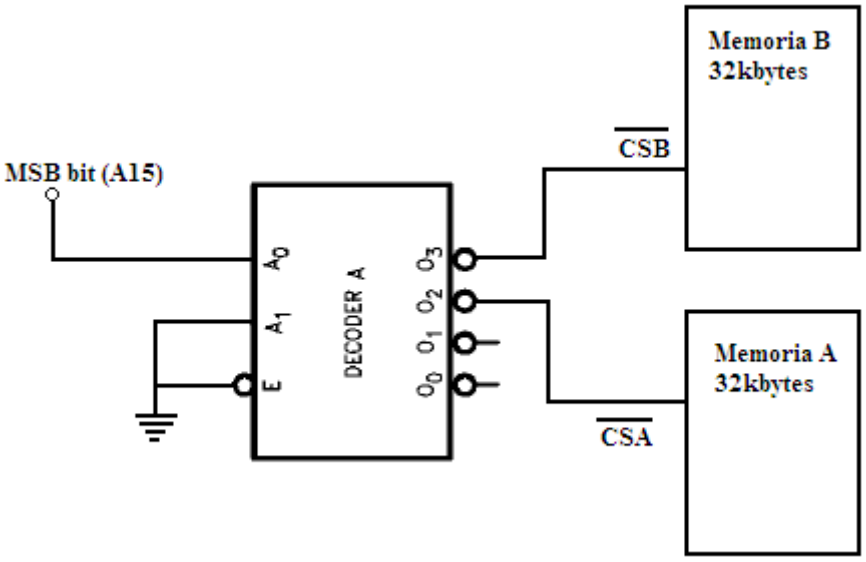


Fig. 20 - Aplicação do 74LS139

4.4 A memória SRAM (DS1230AB)

O DS1230AB é uma memória SRAM não volátil de 32kbytes. O uso desta memória permite um maior armazenamento de dados chegados pela rede, expandindo assim o tamanho do buffer de áudio no Hardware.

4.5 O módulo de Rede (NM7010B)

O NM7018B é um módulo de rede que inclui: um chip que implementa por Hardware a pilha TCP/IP (W3150A), um RTL8201BL que implementa todas as funções da camada física Ethernet e um MAGJACK para cabos RJ45.

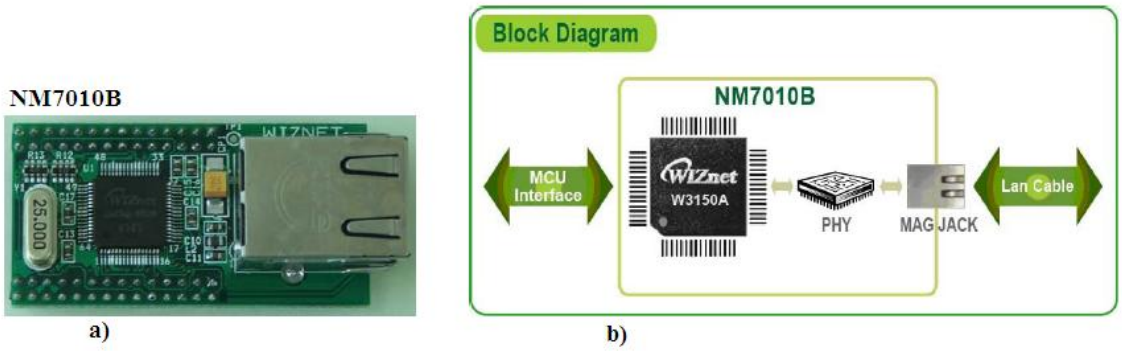


Fig. 21 - a) Foto do NM7010B, b) Diagrama de Blocos do NM7010B (retiradas de [12])

Características:

- Suporta 10/100 Base TX
- Suporta operações em Half/Full duplex
- Opera 3.3V mas tolerante a sinais de I/O de 5V
- IEEE 802.2/802.3u
- Possui LEDS indicadores do estado da rede
- Implementa por Hardware os protocolos de internet: TCP, IP, UDP, ICMP, ARP, PPPoE, IGMP
- Implementa por Hardware os protocolos de Ethernet: DLC, MAC
- Suporta simultaneamente quatro ligações independentes
- Suporta *MCU bus interface*
- Suporta acesso directo ou indirecto do *bus interface*
- Suporta *Socket API*, para aplicações simples de programação

A implementação da pilha TCP/IP é feita por Hardware, através do chip W3150A, tornando-se assim uma solução de baixo custo para ligações de internet de alta velocidade. Como se pode verificar no diagrama de blocos do chip, ele possui como protocolos da pilha *TCP/IP o TCP, UDP, ICMP, IPv4, ARP, PPPoE* e da Ethernet os protocolos, *Data Link Control* e *MAC.* O W3150A possui também um barramento local para interface com vários *MicroController Unit* (MCU) e um padrão especificado do *Media Independent Interface* (MII) para um barramento de dados para dispositivos físicos de Ethernet (tais como o RTL8201BL).

As características do W3150A são:

- Suporta por Hardware os Protocolos da pilha TCP/IP: TCP, UDP, IPv4, ICMP, IGMP, ARP, PPPoE, Ethernet
- Suporta ligações de ADSL (através do protocolo de PPPoE)
- Suporta simultaneamente 4 sockets independentes
- Suporta ligações 10BaseT/100BaseTX
- Modos Full-Duplex
- Possui uma memória interna de 16kbytes para buffers de RX/TX
- Opera com sinais de 3.3V mas tolerante a sinais de 5V

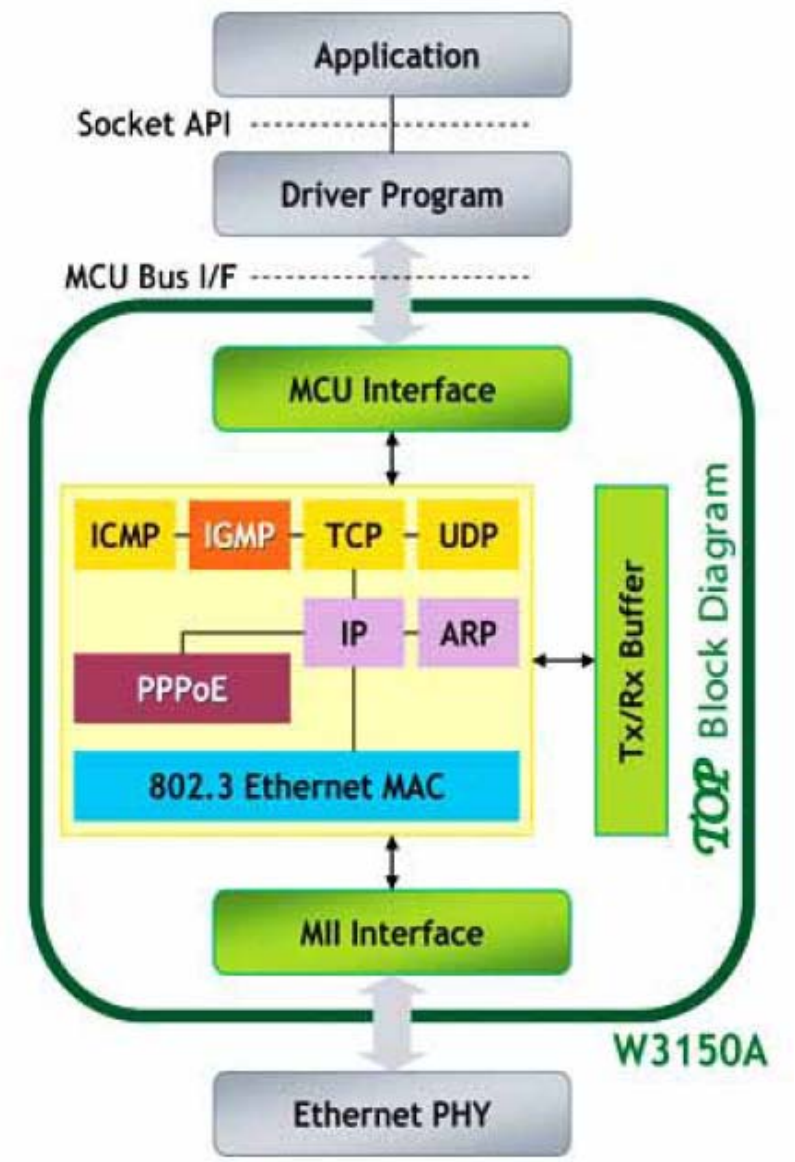


Fig. 22 - Diagrama de Blocos do W3150A (retirada de [13])

O mapa de memória do W3150A é composto por Registos Comuns, Registos dos Sockets, memória TX e memória RX, podemos observar na imagem a seguir como está dividido cada campo.

Além dos datasheet, também se pode encontrar mais informações sobre este módulo e chip no site da wiznet, bem como um Driver, referente ao chip W3150A, que possui o código fonte para interacção com este chip.

Na elaboração deste projecto, foram utilizadas as funções referentes ao chip, utilizando o ficheiro “w3150a.c” e o ”w3150a.h” referentes ao chip W3150A, e funções referentes ao socket usando o ficheiro “socket.c” e o “socket.h”.

Fig. 23 - Mapa de memória do W3150A (adaptada de [13])

4.5.1 A configuração da Rede

A configuração da rede baseia-se na configuração dos registos comuns do chip W3150A módulo de rede NM7010. A configuração dos registos é feita através do barramento local de interface com o microcrocontrolador. Todas as funções para configuração da rede podem-se encontrar no ficheiro “w3150a.c” que se encontra em anexo.

Na seguinte figura podemos observar o mapa de memória dos registos comuns do W3150A.

Address	Register
0x0000	Mode (MR)
	Gateway Address
0x0001	(GAR0)
0x0002	(GAR1)
0x0003	(GAR2)
0x0004	(GAR3)
	Subnet mask Address
0x0005	(SUBR0)
0x0006	(SUBR1)
0x0007	(SUBR2)
0x0008	(SUBR3)
	Source Hardware Address
0x0009	(SHAR0)
0x000A	(SHAR1)
0x000B	(SHAR2)
0x000C	(SHAR3)
0x000D	(SHAR4)
0x000E	(SHAR5)
	Source IP Address
0x000F	(SIPR0)
0x0010	(SIPR1)
0x0011	(SIPR2)
0x0012	(SIPR3)
0x0013	Reserved
0x0014	
0x0015	Interrupt (IR)
0x0016	Interrupt Mask (IMR)
	Retry Time
0x0017	(RTR0)
0x0018	(RTR1)
0x0019	Retry Count (RCR)

Address	Register
0x001A	RX Memory Size (RMSR)
0x001B	TX Memory Size (TMSR)
	Authentication Type in PPPoE
0x001C	(PATR0)
0x001D	(PATR1)
0x001E	
~	Reserved
0x0027	
0x0028	PPP LCP Request Timer (PTIMER)
0x0029	PPP LCP Magic number (PMAGIC)
	Unreachable IP Address
0x002A	(UIPR0)
0x002B	(UIPR1)
0x002C	(UIPR2)
0x002D	(UIPR3)
	Unreachable Port
0x002E	(UPORT0)
0x002F	(UPORT1)
0x0030	
~	Reserved
0x03FF	

Fig. 24 - Mapa dos registos comuns do W3150A(retirada de [13])

Sendo assim necessário:

- Configurar os registos de informação da rede de acordo com a rede onde será inserido, necessitando assim de se configurar:
  - O endereço do *Gateway*[10]
  - O endereço físico da placa
  - O endereço da máscara de rede
  - E o endereço IP da placa

[10] Máquina intermediária cuja funcionalidade é interligar duas ou mais redes que usem protocolos de comunicação internos.

O procedimento de configuração:

A configuração é feita através da escrita de valores nos registos comuns. Escrita esta, que do ponto de vista do microcontrolador, não é mais que a escrita de um valor em determinada posição de uma memória externa. A leitura procede da mesma forma e é apenas activado o modo de leitura em vez do modo de escrita por parte do microcontrolador.

As funções de escrita e leitura no W3150A:

```
//função de escrita
uint8 IINCHIP_WRITE(uint16 addr,uint8 val)
{
       *((vuint8 xdata *)(addr))=val;

       return 1;
}

//função de leitura
uint8 IINCHIP_READ(uint16 addr)
{
       uint8 val;
       val = *((vuint8 xdata*)(addr));
       return val;
}
```

- Informar o Gateway para actualização da tabela ARP (*Adress Resolution Protocol*), e obtenção do endereço físico do Gateway.

O procedimento de actualização:

Este procedimento de actualização foi implementado de forma a obter o endereço MAC do Gateway, fazendo um pedido de conexão TCP com um cliente numa rede diferente.

```
{
START:
    /* set TCP on socket 0 mode register and open socket */
    S0_MR = 0x01;
    S0_CR = OPEN;
    if (S0_SSR != SOCK_INIT) S0_CR = CLOSE; goto START;

    /* request TCP connection to any host in different network */
    S0_DIPR0 = SIPR0 + 1; // for making different network
    S0_CR = CONNECT; // Set CONNECT command
    Wait 10 msec;

    /* getting Gateway Hardware Address */
    If (S0_DHAR0 == 0xFF) goto START; // Can't receive information
    else gGatewayHA = S0_DHAR; // save gateway hardware address (6bytes)
```

Fig. 25 - Algoritmo para informar o Gateway e obter o seu endereço MAC (retirado de [13])

A função implementada:

```
void getGWMAC_processing(void)
{
   uint8 i = 0;
   uint8 j = 0;
       do
       {
          IINCHIP_WRITE(OPT_PROTOCOL(0),SOCK_STREAM);
          IINCHIP_WRITE(COMMAND(0),CSOCKINIT);
          IINCHIP_WRITE(DST_IP_PTR(0),L_IP[0]+1);
          IINCHIP_WRITE(COMMAND(0),CCONNECT);
          wait_10ms(10);
          for (i = 0; i < 6; i++)GW_MAC[i] =
IINCHIP_READ(DST_HA_PTR(0)+i);
          IINCHIP_WRITE(COMMAND(0),CCLOSE);
          IINCHIP_WRITE(DST_IP_PTR(0),0x00);


       } while ((IINCHIP_READ(DST_HA_PTR(0)) == 0xff) && j++ <3);
}
```

- Configurar a memória de cada socket, como neste caso só se irá usar um socket, pode-se assim disponibilizar toda a área de memória do RX/TX, que são de 8kbytes cada.

A configuração da memória do socket:

A configuração é feita através da escrita de um valor correspondente ao registo de configuração da memória.

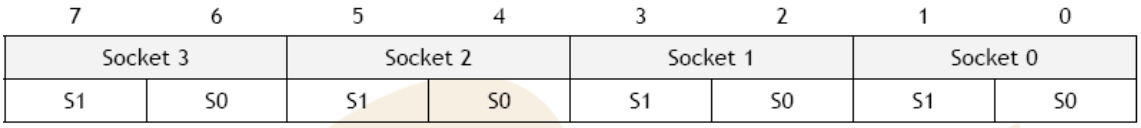


Fig. 26 – Registo de configuração do tamanho de memória TX/RX (retirada de [13])

Neste caso usou-se um só socket, definiu-se toda a área de memória do TX e do RX. A tabela seguinte demonstra como é feita essa configuração.

S1	S0	Tamanho da Memória
0	0	1KB
0	1	2KB
1	0	4KB
1	1	8KB

Tabela 5 - Tabela de configuração da memória dos socket

Então para 8KB, teremos de escrever no registo da memória RX e TX o valor de 255.

4.5.2 A Comunicação

A comunicação é realizada pelo protocolo TCP, entre um cliente e um servidor. Nesta implementação criou-se o modo cliente, baseado na implementação do algoritmo da Fig. 27. Este algoritmo pode ser dividido em três passos distintos:

1) Criação e inicialização do socket
2) Pedido de conexão com o servidor
3) Estabelecimento da conexão

A implementação destes passos requer conhecimento dos registos dos socket, que funcionam como controladores da comunicação e de implementação do tipo de protocolo a usar.

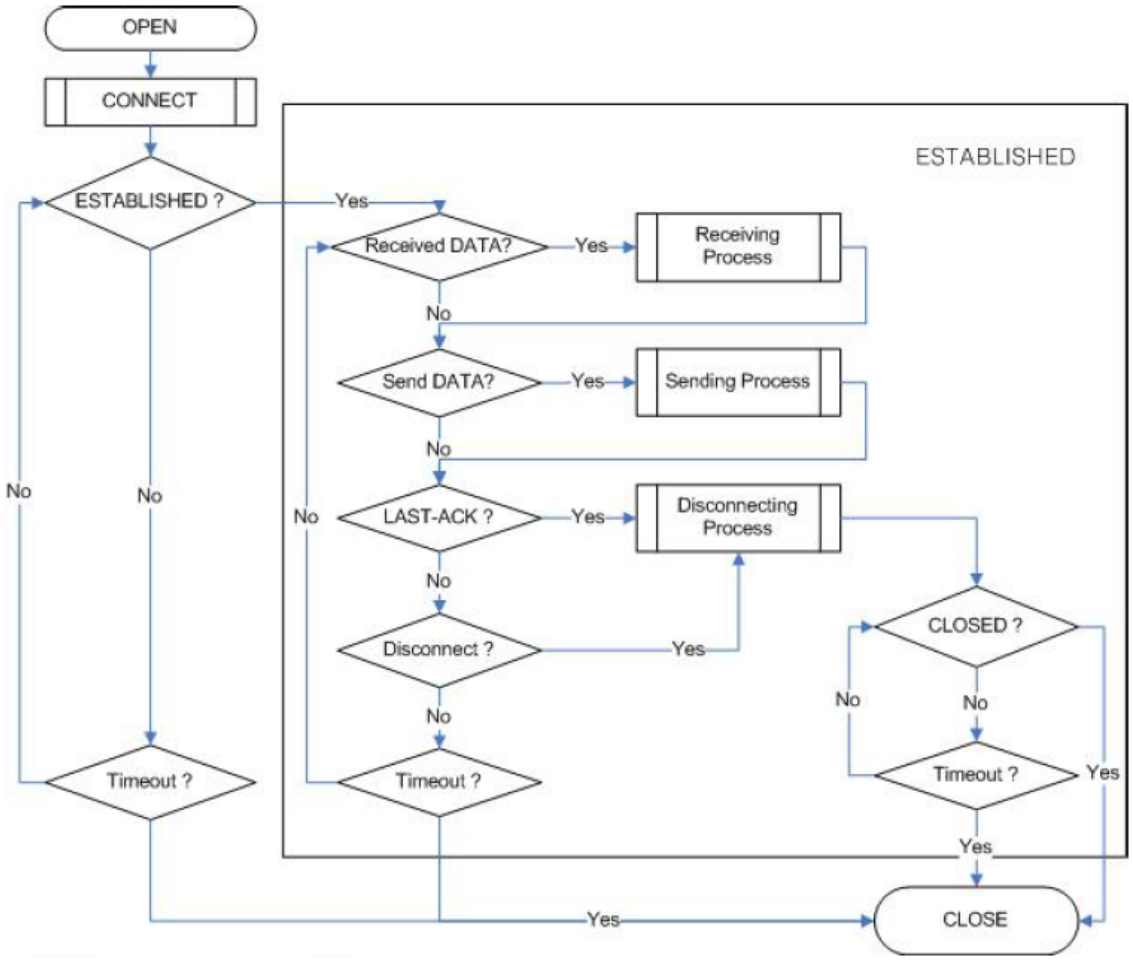


Fig. 27 - Algoritmo do Protocolo TCP implementado (retirado de [13])

Na Fig. 28, pode-se observar o mapa de memória dos registos do socket 0. Neste mapa de memória têm-se os registos necessários ao controlo e estado do socket. Podemos definir o protocolo e as opções do socket através do **Registo Modo**, controlar a comunicação através do **Registo Comando** e verificar o estado da comunicação através do **Registo Estado do socket**.

Seguidamente são apresentadas as tabelas com os valores que cada um destes registos pode tomar e o seu respectivo significado. Estes valores estão mais pormenorizados no Datasheet do chip W3150A, na referência [13].

As funções implementadas relativas à comunicação estão dispostas no ficheiro “socket.c” que se encontra em anexo.

Address	Register
0x0400	Socket 0 Mode (S0_MR)
0x0401	Socket 0 Command (S0_CR)
0x0402	Socket 0 Interrupt (S0_IR)
0x0403	Socket 0 Socket Status (S0_SSR)
0x0404	Socket 0 Source Port (S0_PORT0)
0x0405	(S0_PORT1)
0x0406	Socket 0 Destination Hardware Address (S0_DHAR0)
0x0407	(S0_DHAR1)
0x0408	(S0_DHAR2)
0x0409	(S0_DHAR3)
0x040A	(S0_DHAR4)
0x040B	(S0_DHAR5)
0x040C	Socket 0 Destination IP Address (S0_DIPR0)
0x040D	(S0_DIPR1)
0x040E	(S0_DIPR2)
0x040F	(S0_DIPR3)
0x0410	Socket 0 Destination Port (S0_DPORT0)
0x0411	(S0_DPORT1)
0x0412	Socket 0 Maximum Segment Size (S0_MSSR0)
0x0413	(S0_MSSR1)
0x0414	Socket 0 Protocol in IP Raw mode (S0_PROTO)

Address	Register
0x0415	Socket 0 IP TOS (S0_TOS)
0x0416	Socket 0 IP TTL (S0_TTL)
0x0417 ~ 0x041F	Reserved
0x0420	Socket 0 TX Free Size (S0_TX_FSR0)
0x0421	(S0_TX_FSR1)
0x0422	Socket 0 TX Read Pointer (S0_TX_RD0)
0x0423	(S0_TX_RD1)
0x0424	Socket 0 TX Write Pointer (S0_TX_WR0)
0x0425	(S0_TX_WR1)
0x0426	Socket 0 RX Received Size (S0_RX_RSR0)
0x0427	(S0_RX_RSR0)
0x0428	Socket 0 RX Read Pointer (S0_RX_RR0)
0x0429	(S0_RX_RR1)
0x042A 0x042B	Reserved
0x042C ~ 0x04FF	Reserved

Fig. 28 - Mapa de memória dos registos do Socket 0 (retirada de [13])

Valores para configuração do Registo Modo (Endereço 0x0400h)		
Valor (HEX)	LABEL	Descrição
0x00	SOCK_CLOSEDM	Socket não esta a ser usado
0x01	SOCK_STREAM	TCP
0x02	SOCK_DGRAM	UDP
0x03	SOCK_ICMPM	ICMP
0x03	SOCK_IPL_RAWM	IP LAYER RAW SOCK
0x04	SOCK_MACL_RAWM	MAC LAYER RAW SOCK
0x05	SOCK_PPPOEM	PPPoE
0x10	SOCKOPT_ZEROCHKSUM	Usado no UDP para colocar a zero o checksum
0x20	SOCKOPT_NDACK	
0x80	SOCKOPT_MULTI	Suporta Multicasting

Tabela 6 - Configurações do Registo Modo

Valores para configuração do Registo de Comando (Endereço 0x0401h)		
Valor (HEX)	LABEL	Descrição
0x01	CSOCKINIT	Inicializa ou cria o socket
0x02	CLISTEN	Espera por pedido de conexão (Modo servidor)
0x04	CCONNECT	Envia pedido de conexão (Modo Cliente)
0x08	CDISCONNECT	Envia pedido de desconexão do socket no modo TCP
0x10	CCLOSE	Fecho do socket
0x20	CSEND	Actualização do apontador TXbuf, Envio de dados
0x21	CSENDMAC	Envio de dados com endereço MAC
0x22	CSENDKEEPALIVE	Envio de mensagem para manter conexão activa
0x40	CRECV	Actualização do apontador RXbuf, Recepção de dados

Tabela 7 - Configurações do Registo Comando

Valores para verificação e configuração do Registo de Estado (Endereço 0x0403h)		
Valor (HEX)	LABEL	Descrição do socket
0x00	SOCK_CLOSED	Fechado
0x13	SOCK_INIT	Iniciar
0x14	SOCK_LISTEN	Estado de Escuta
0x15	SOCK_SYNSENT	Estado de conexão
0x16	SOCK_SYNRECV	Estado de conexão
0x17	SOCK_ESTABLISHED	Conexão estabelecida
0x18	SOCK_FIN_WAIT1	Estado de fecho
0x19	SOCK_FIN_WAIT2	Estado de fecho
0x1A	SOCK_CLOSING	Estado de fecho
0x1B	SOCK_TIME_WAIT	Estado de fecho
0x1C	SOCK_CLOSE_WAIT	Estado de fecho
0x1D	SOCK_LAST_ACK	Estado de fecho
0x22	SOCK_UDP	Socket UDP
0x32	SOCK_IPL_RAW	Socket IP RAW
0x42	SOCK_MACL_RAW	Socket MAC RAW
0x5F	SOCK_PPPOE	Socket PPPoE

Tabela 8 - Configurações do Registo do Estado

A configuração do Cliente TCP:

1) Criação e inicialização do socket

Para inicialização do socket, foi necessário definir o protocolo de comunicação, a porta disponível para este socket, e activar o comando de inicialização do socket.

A função utilizada é a `uint8 socket(SOCKET s, uint8 protocol, uint16 port, uint8 flag)`, que implementa o seguinte algoritmo:

```
Definir o tipo de Protocolo a usar
      Protocol = SOCK_STREAM

Definir a Porta da fonte
      SRC_PORT_PTR = Porta

Iniciar o socket
      COMMAND = CSOCKINIT
```

2) Pedido de conexão com o servidor

Criado o socket, é necessário fazer o pedido de conexão com o servidor, definindo assim o endereço do servidor e a sua porta, e é ainda activado o comando de conexão. A função implementada para o fazer é a `uint8 connect(SOCKET s, uint8 * addr, uint16 port)`, que implementa o seguinte algoritmo:

```
Escrever o endereço do Servidor
      DST_IP_PTR = IP do servidor

Escrever a porta do Servidor
      DST_PORT_PTR = Porta do servidor

Activar o pedido de conexão
      COMMAND = CCONNECT
```

3) Estabelecimento da conexão

A verificação do estabelecimento da conexão é feita através da leitura do registo de estado. Este registo pode ter vários estados, como se pode verificar pela Tabela 8.

```
Verificar se já esta estabelecida a conexão
      SOCK_STATUS == SOCK_ESTABLISHED
```

Após o estabelecimento da conexão, o cliente e o servidor ficam num estado de habilidade para transferência de dados entre eles.

4.6 O descodificador de MP3 (STA013)

O STA013 é um completo e flexível integrado descodificador de áudio MPEG Layer III (MP3), capaz de descodificar ficheiros MP3 com fluxos desde 8 a 320kbit/s,

com frequências de amostragem desde os 8 aos 48kHz. Este descodificador tem as seguintes características:

- Descodifica áudio em stereo, dois canais, e mono
- Tem controlo digital do volume
- Controlo digital do Bass e Treble
- Interface série síncrona de bits
- Configuração e controlo de informação por interface I2C
- Interface série de saída de sinais PCM (I^2S e outros formatos)
- *Phase Lock Loop* (PLL) para sinal de relógio interno e para geração de relógio de saída do sinal PCM
- Baixo consumo de energia
- Verificação de redundância cíclica (CRC) e sincronização de erros
- Tecnologia CMOS de baixo consumo

O STA013 recebe os dados referentes ao áudio por um comunicação série síncrona de entrada, o sinal é descodificado podendo ser mono, dois canais ou stereo, sendo enviado directamente para o conversor digital analógico, através de uma comunicação série síncrona de áudio (I^2S), no formato PCM. O STA013 pode operar de dois modos diferentes no modo Multimédia, em que o STA013 activa um pino (o DATA_REQ) sinalizando o pedido de mais dados à fonte de dados e no modo Broadcast, em que a fonte de dados tem de enviar os bits a uma velocidade por si regulada.

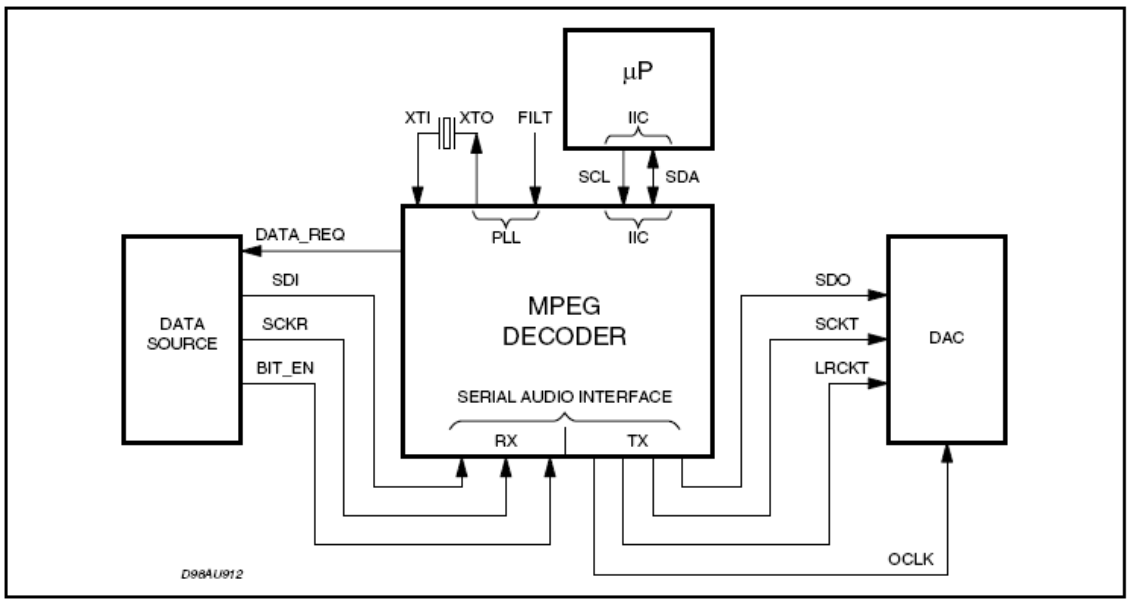


Fig. 29 - Diagrama de interfaces do descodificador (retirada de [19])

4.6.1 A configuração do descodificador STA013

Apesar das ligações existentes entre o descodificar, a fonte de dados e o conversor digital analógico de áudio, é necessário proceder a algumas definições de interface e de operação referentes aos protocolos de comunicação entre eles. É ainda necessário configurar os registos correspondentes ao sistema gerador do sinal de relógio e da *Phase-Locked-Loop* (PLL), estes que dependem do valor do cristal externo ligado ao STA013. Estas definições são possíveis através da configuração de registos disponíveis no STA013 e acessíveis através do protocolo de comunicação I2C. Podemos ver através do seguinte diagrama de blocos a sequência de configuração.

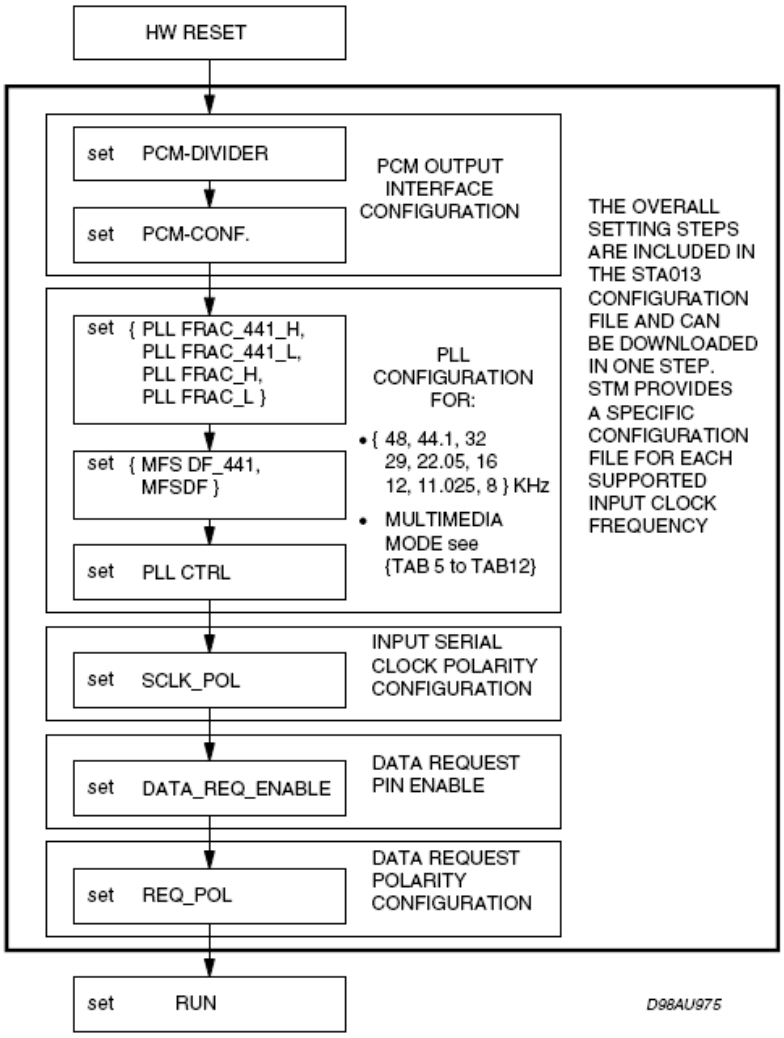


Fig. 30 - Diagrama de configuração (retirada de [19])

O procedimento de configuração:

Para iniciar a configuração é necessário conhecer o valor do cristal que se vai utilizar, para este trabalho o valor escolhido foi de 14.318MHz, pois é um dos cristais

aconselhados pelo fabricante. O DAC possui uma relação entre o Master Clock e o Left/Rigth Clock Input que podem ser de 256, 384 ou 512, esta relação é denominada de sobre-amostragem, e tem de ser configurada no descodificar para envio de dados para o DAC. Tendo estes valores pode-se iniciar a configuração do descodificador de MP3.

Configuração do sinal de saída PCM

A configuração do primeiro bloco relaciona-se com a configuração do sinal de saída PCM entre o descodificador e o DAC. A configuração do PCM-DIVIDER é realizada segundo a seguinte tabela:

MSB LSB

b7	b6	b5	b4	b3	b2	b1	b0	Description	
PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0		
0	0	0	0	0	1	1	1	16 bit mode	512 x Fs
0	0	0	0	0	1	0	1	16 bit mode	384 x Fs
0	0	0	0	0	0	1	1	16 bit mode	256 x Fs
0	0	0	0	0	0	1	1	32 bit mode	512 x Fs
0	0	0	0	0	0	1	0	32 bit mode	384 x Fs
0	0	0	0	0	0	0	1	32 bit mode	256 x Fs

Tabela 9 - Configuração do PCM-DIVIDER (retirada de [19])

Para uma sobre-amostragem de 512 no modo de 32bits:

PCM-DIVIDER = 3

A configuração do PCM-CONF é usada para configuração de saída do sinal de PCM, este registo é configurável dependendo do tipo de DAC que se irá usar. A tabela seguinte tem uma descrição deste registo:

b7	b6	b5	b4	b3	b2	b1	b0	Descrição		MAX5556
X	ORD	DIF	INV	FOR	SCL	PREC	PREC			
X	1							Ordem do PCM	LSB bit transmitido primeiro	
X	0								MSB bit transmitido primeiro	✓
X		0						Palavra é enviada à direita		
X		1						Palavra é enviada à esquerda		✓
X			1					Polaridade LRCKT	Compatível com o I2S	
X			0						Invertida	✓
X				0				Formato I2S		✓
X				1				Formato Diferente		
X					1			Envio dos Dados	Na Transição Baixo para Alto	✓
X					0				Na Transição Alto para Baixo	

X						0	0	Modo	16 bits	
X						0	1		18 bits	
X						1	0		20 bits	
X						1	1		24 bits	✓

Tabela 10 - Registo PCM-CONF (Adaptada de [19][19])

Para obter a compatibilidade com o DAC MAX5556 o valor que se escreveu neste registo foi:

PCM-CONF = 27

Configuração da PLL

A configuração do segundo bloco diz respeito à PLL do descodificador, neste bloco é necessário configurar vários registos, para o cristal de 14.318MHz e para o valor de sobre-amostragem de 512, os registos e os valores de configuração podem-se observar na seguinte tabela:

REGISTER ADDRESS	NAME	VALUE
6	reserved	11
11	reserved	3
97	MFSDF (x)	6
80	MFSDF-441	7
101	PLLFRAC-H	3
82	PLLFRAC-441-H	157
100	PLLFRAC-L	211
81	PLLFRAC-441-L	157
5	PLLCTRL	161

Tabela 11 - Configuração da PLL (Retirada de [19])

SCLK_POL

A configuração da polaridade do sinal de relógio, SCLK_POL, define o ponto de envio de dados para o DAC, isto é, o dado ou é enviado na transição de alto para baixo ou de baixo para alto. A configuração deste registo adquire apenas dois valores:

MSB							LSB	
b7	b6	b5	b4	b3	b2	b1	b0	
X	X	X	X	X	0	0	0	(1)
					1	0	0	(2)

X = don't care

Para o caso (1) os dados são activados na transição de alto para baixo, no caso (2) o envio dos dados é feito na transição de baixo para alto. Segundo o DAC utilizado os dados devem ser enviados na transição de baixo para alto. Então temos que o SCLK_POL = 4.

DATA_REQ_ENABLE

Este bloco define se deve existir um sinal de pedido de dados ou um sinal de relógio para sincronizar com o envio de dados, no pino 28 do descodificador, denominado de DATA_REQ. Este pino vem de acordo com os modos de funcionamento do descodificador, Multimédia ou Broadcast.

MSB							LSB	
b7	b6	b5	b4	b3	b2	b1	b0	Description
X	X	X	X	X	0	X	X	buffered output clock
X	X	X	X	X	1	X	X	request signal

Neste caso ele irá funcionar no modo Multimédia, isto é, activação de sinal de pedido de dados, DATA_REQ_ENABLE = 4.

REQ_POL

O bloco seguinte corresponde à configuração da polaridade do pino de pedido de dados o DATA_REQ, isto é, o pedido de dados pode ser efectuado quando este pino está a nível baixo, ou quando está a nível alto.

A configuração para o nível alto é REQ_POL = 1

e para o nível baixo REQ_POL = 5

Na elaboração deste projecto o DATA_REQ, irá ser activo a nível alto, o que faz com que o registo REQ_POL = 1.

RUN

Por fim, após a configuração do descodificar deve ser activado o modo RUN, que colocará o STA013 em modo de descodificação, e pronto para iniciar a descodificação dos dados MP3.

Todas funções referentes a estes registos de configuração estão presentes no ficheiro que se encontra em Anexo “STA013-I2C.c”

4.7 O conversor Digital Analógico de Áudio (MAX5556)

O MAX5556 é um conversor digital analógico (DAC) de áudio stereo sigma-delta, e confere uma solução simples e completa para aplicações de servidores de media, vídeo jogos, entre outras aplicações gerais domésticas de áudio.

Este DAC recebe os dados por uma ligação flexível de três fios compatível com o I^2S, ajustada à esquerda até 24bits de dados.

A aquisição dos dados é feita através da leitura da linha de dados (SDATA) na transição do nível lógico de baixo para alto da linha de relógio (SCLK). A linha LRCLK é a linha que selecciona a escrita do dado referente ao canal esquerdo e ao canal direito, estando a nível baixo para o canal esquerdo e alto para o canal direito. Na linha SDATA é sempre colocado o bit mais significativo do dado. Como podemos verificar no seguinte diagrama temporal dos dados.

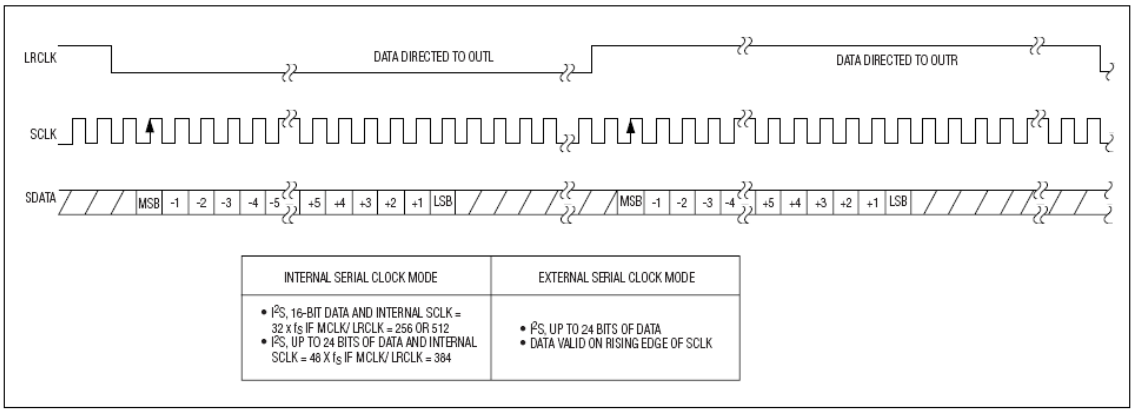


Fig. 31 - Diagrama Temporal dos dados (retirada de [22])

O MAX5556 funciona em um dos dois modos de sinal de relógio, sinal de relógio externo e interno. O sinal de relógio externo é dado pela fonte externa que envia os dados para o DAC, o modo interno funciona sempre que não seja detectado o sinal externo de SCLK. No modo interno o SCLK deriva e é síncrono com o *Master Clock* MSCLK e o LRCLK.

O fornecedor deste DAC aconselha na saída analógica o uso de um filtro analógico passa baixo para ajudar a reduzir harmónicos e ruídos. Podendo ser activos ou passivos dependendo da aplicação [22].

4.8 Os tradutores de níveis de tensão (o PCA9306 e o TXS0108E)

Os tradutores dos níveis de tensão fazem a conversão dos níveis de tensão dispostos na entrada para uns níveis de tensão diferentes na saída. Os tradutores são ideais para transferências de dados entre dois dispositivos cujos níveis de tensão para os sinais lógicos são diferentes.

O PCA9306 é usado para tradutores de níveis de tensão do barramento I2C, permitindo conversões desde 1.2V até 5V, sendo dividido em dois lados, um lado de níveis de tensão baixa e outro de níveis de tensão alta. Podemos verificar na figura seguinte o esquema de ligações deste tradutor. Os valores das resistências colocadas em Pull Up nos barramentos podem ser calculados pela expressão:

$$RPU = \frac{V_{DPU} - 0{,}35V}{0.015A} \text{ [23]}$$

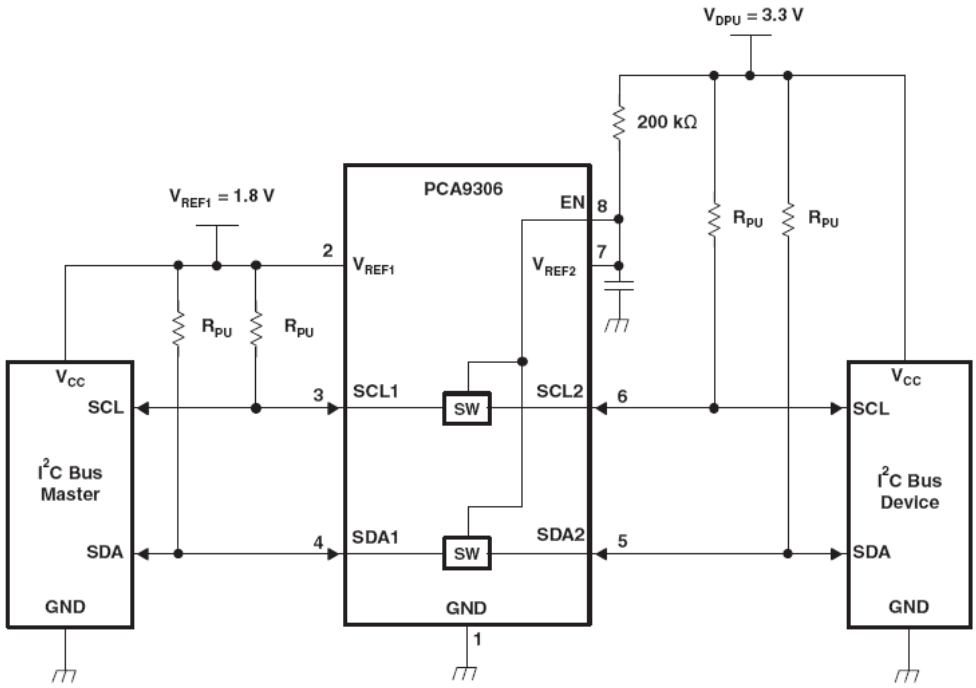


Fig. 32 - Típica aplicação do PCA9306 (retirada de [23])

Neste projecto a aplicação deste tradutor de I2C tornou possível a comunicação entre o microcontrolador com níveis de tensão de 5V e o STA013 com níveis de tensão de 3.3V.

O TXS0108E é um tradutor de níveis de tensão de 8 bits, com um porto A e um porto B. O porto A aceita tensões desde o 1.2V até 3.6V e o porto B desde 1.65V até 5V. Na figura seguinte podemos verificar um esquema de ligações de uma simples aplicação.

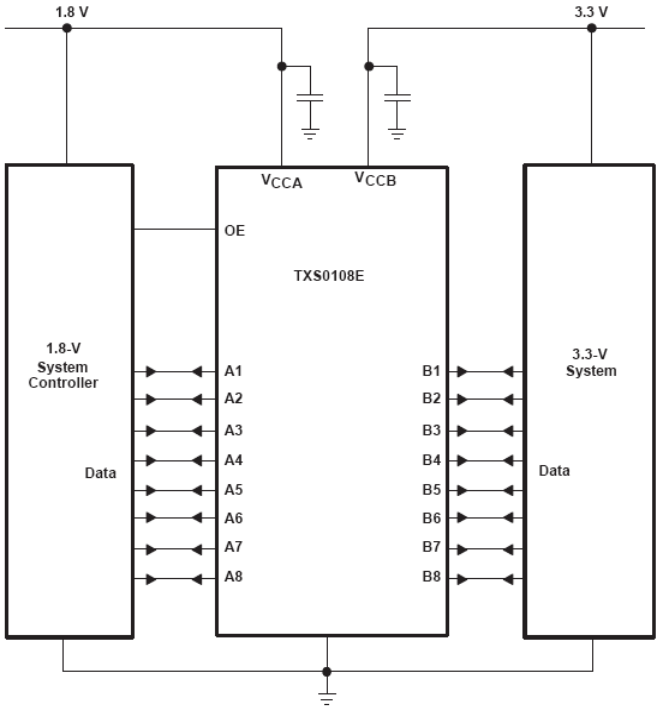


Fig. 33 - Aplicação do TXS0108E (retirada de [24])

Este tradutor foi usado para fazer a conversão dos níveis de tensão das linhas de dados entre o microcontrolador, com níveis de tensão de 5V, e o descodificador de MP3 que possui níveis de tensão de 3.3V.

4.9 A construção do Hardware

A construção do Hardware tem como objectivo a criação de uma placa de circuito impresso (PCB) final com todo este sistema incorporado numa só placa física a aplicar numa rede local. Este processo de criação passou por três fases.

A primeira fase foi a interacção entre o microcontrolador e o módulo de rede construindo inicialmente um PCB e colocá-lo em ligação com uma rede local para interacção entre um cliente e um servidor, integrando os drivers do W3150A disponíveis para download na página da wiznet. A imagem seguinte mostra o seu aspecto físico.

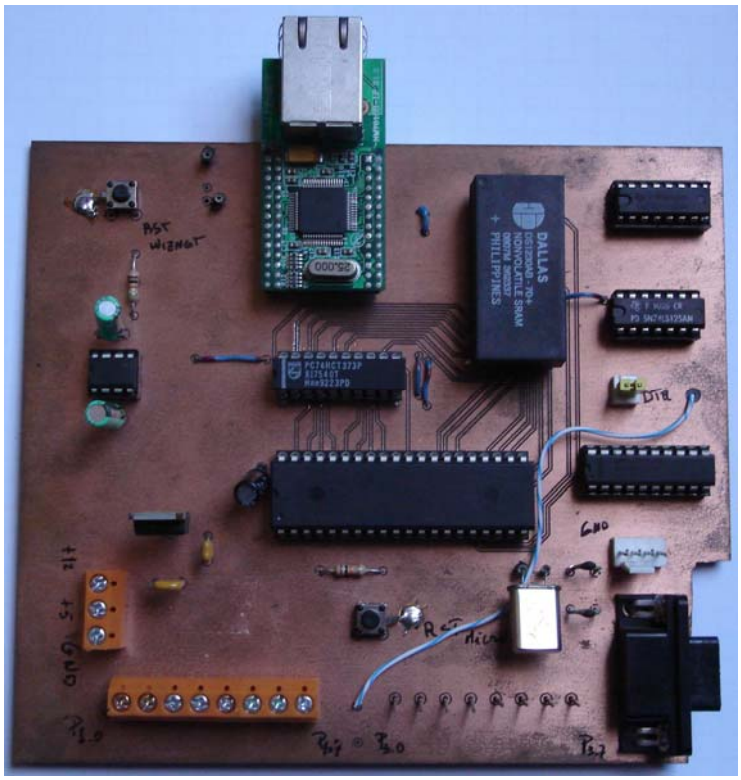


Fig. 34 - 1º PCB elaborado (ligação à rede)

A execução deste PCB foi realizada a partir da ferramenta PAD's 2007, elaborando o esquema de ligações e o desenho do PCB, as imagens elaboradas na ferramenta deste PCB encontram-se no Anexo C. Os componentes que compõem esta placa são:

- Microcontrolador
- Módulo de rede NM7010B
- Possui um circuito de Loader incorporado, para carregamento de software para o microcontrolador
- Memória SRAM, DS1230
- Desmultiplexar 74LS139
- Uma latch 74HCT373
- Conector RS232
- Conector de alimentação, VCC(máximo 35V), 5V e GND
- Conectores para extensão de mais Hardware
- Regulador de tensão para 5V
- Regulador de tensão para 3,3V
- Botões de reset

A segunda fase foi a integração do descodificador de MP3, em que foi criado outro PCB para ser conectado ao primeiro. Neste PCB estão integrados o STA013, o conversor digital analógico de áudio o MAX5556 e um filtro passivo passa baixo.

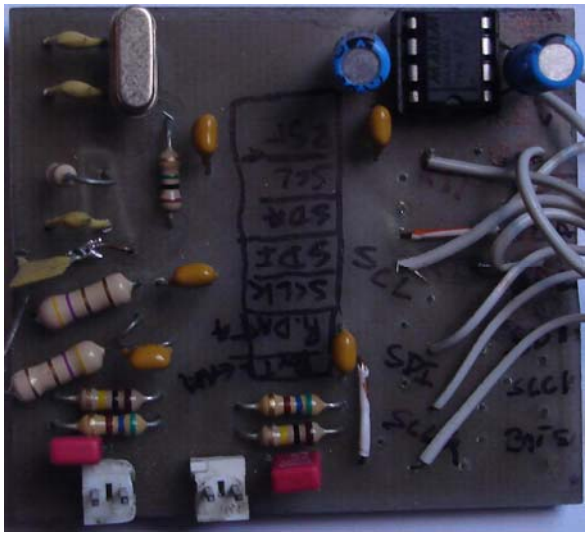


Fig. 35 - 2º PCB elaborado (conversor de MP3)

Nesta segunda placa estão adicionados os componentes para funcionamento do STA013, este esquema de ligações foi obtido a partir de uma *Aplication Note* existente na página do fabricante [20]. O esquema e o desenho do PCB podem ser observados no Anexo C. Esta placa possui apenas os fios de ligação para poderem ser conectados com o primeiro PCB. De notar ainda que entre o interface da primeira placa elaborada e da segunda foram colocados os conversores de níveis de tensão.

A última fase foi a construção de um PCB final com todo o sistema incorporado numa única placa. O Sound-Ether possui assim os seguintes componentes:

- Microcontrolador
- Módulo de rede NM7010B
- Possui um circuito de Loader incorporado, para carregamento de software para o microcontrolador
- Memória SRAM, DS1230
- Desmultiplexar 74LS139
- Uma latch 74HCT373
- Um descodificador de MP3 STA013
- Um DAC MAX5556

- Um conversor de nível de tensão para barramentos I2C o PCA 9306
- Um conversor de nível de tensão TXS0108e
- Filtro passivo passa baixo
- Regulador de tensão para 5V
- Regulador de tensão para 3,3V
- Conector RS232 para carregamento do software
- Conector de alimentação, VCC, 5V e GND
- Conector RJ45 para ligação à rede local
- Conector de saída de áudio
- Botões de reset

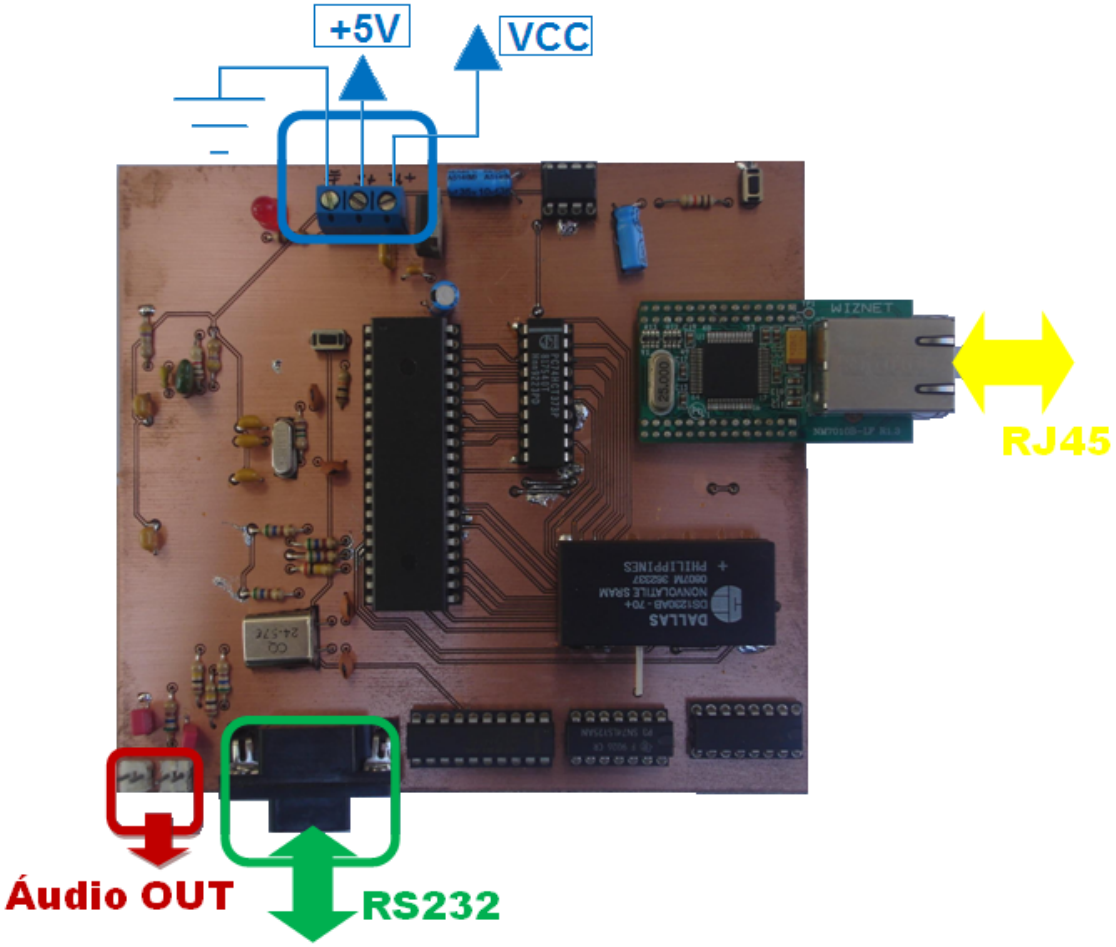


Fig. 36 - PCB Final - Sound Ether

Nesta placa o interface RS232 faz parte do circuito de loader de carregamento de software incorporado na placa.

O conector RJ45 serve para a ligação à rede local, para recepção de dados pela comunicação que o Sound-Ether estabelecerá com o servidor. A saída de áudio permite ligar a um amplificador ou simplesmente a uns auscultadores para ouvir o som. Este hardware possui duas ligações de tensão de alimentação possíveis, ou uma alimentação de 5V contínuos ou ligar ao conector VCC uma tensão máxima de 35V contínuos, que posteriormente é regulada para uma tensão de 5V.

Capítulo V

5 O funcionamento do hardware

Na figura abaixo apresentada pode observar-se o diagrama com o algoritmo implementado pela função *main* do Hardware construído. É a função principal de um software, que contém todo o código necessário para a elaboração da tarefa pretendida. A partir desta função é que serão chamadas todas as funções implementadas para funcionamento e configuração de todo o Hardware. Esta função é responsável por gerir todo o processo da tarefa a executar.

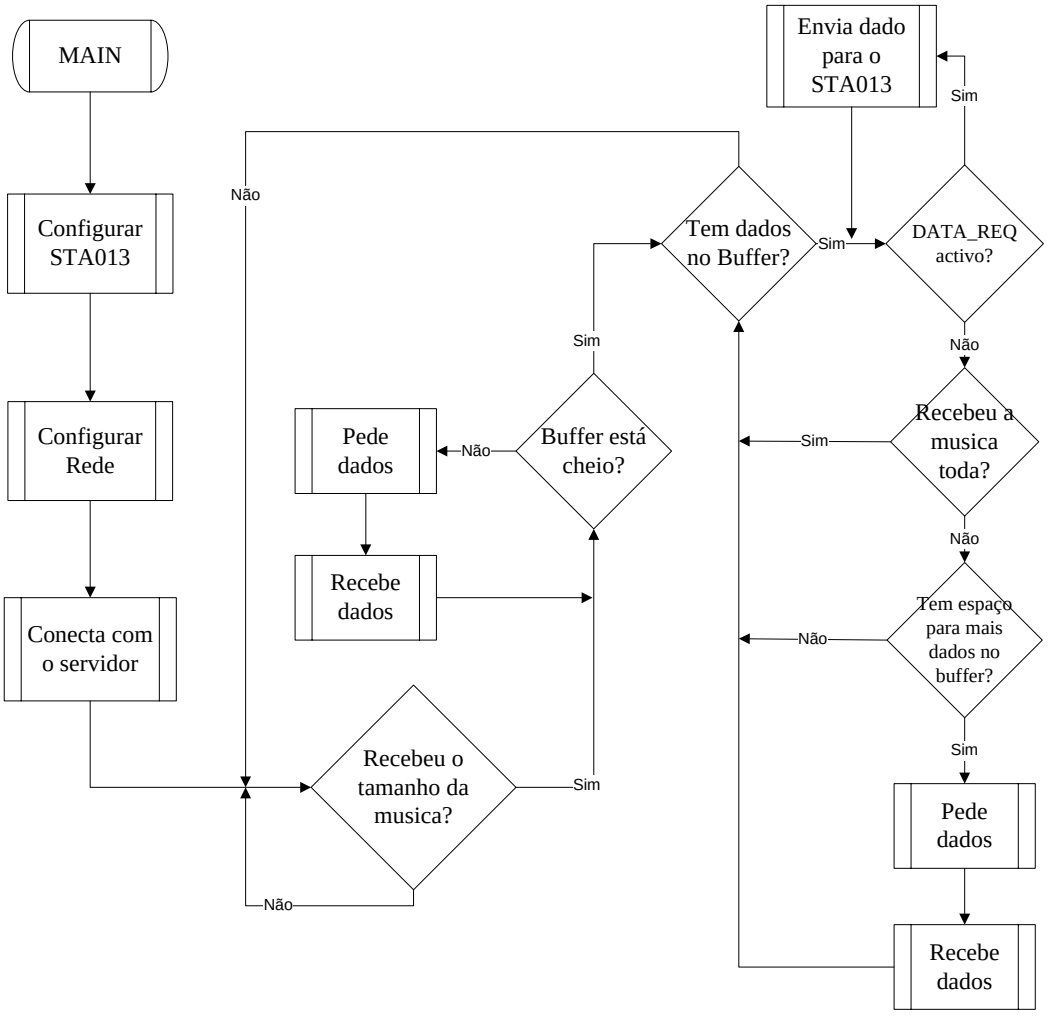


Fig. 37 - Algoritmo implementado pelo Hardware

Neste Projecto a função *main* implementa a configuração do descodificador de MP3 chamando as suas funções de configuração, a configuração do módulo de rede chamando as suas funções de configuração, controla a recepção dos dados de som em formato MP3 enviados pelo servidor colocando-os na memória e controla o fluxo de envio de dados para reprodução.

5.1 A transferência de dados

A transferência de dados é feita através do protocolo de comunicação TCP, em que o cliente é o Hardware construído e o servidor estará instalado num computador que enviará o som que se pretende reproduzir. O cliente fica constantemente à "escuta" da recepção de dados, o primeiro dado que este deve receber é o tamanho do ficheiro MP3, informando o cliente do tamanho de dados correspondentes ao som a reproduzir e sinaliza ainda o início da sua reprodução. O tamanho de dados é enviado em forma de String de dados em que cada dígito corresponde a uma posição da String. Este valor é passado para um inteiro de 32 bits para fazer o controlo da recepção dos dados do ficheiro.

Após o envio do tamanho do ficheiro o cliente envia um pedido através da função Send com uma string a pedir mais dados, tendo assim um controlo do fluxo de dados entre o cliente e o servidor e à medida que o descodificador reproduz o som deixa espaço disponível para transmissão de mais dados. A função *`uint16 send(SOCKET s, const uint8 * buf, uint16 len)`* tem o seguinte algoritmo:

```
Verifica se o tamanho dos dados a enviar não excede o
tamanho máximo do TX

Copia a String para a memória de TX do wiznet

Activa o comando de envio
            COMMAND (0) = CSEND

Espera pela finalização do envio
            (enquanto o registo comando não for zero)
```

O cliente após o pedido de dados fica à espera que o servidor proceda ao envio de uma determinada quantidade de dados. A verificação da recepção de dados é feita através da verificação do registo do tamanho de dados recebido. Determinando que existem dados transferidos pelo servidor estes devem ser copiados para um buffer.

A função que implementa a recepção de dados é a *uint16 recv(SOCKET s, uint8 * buf, uint16 len)*, cujo algoritmo é

```
Copiar os dados da memória RX do wiznet para o buffer

Activar o comando de recepção
            COMMAND(0) = CRECV
```

O buffer deve ser cheio antes de iniciar a reprodução do som, pois como o STA013 faz o pedido de dados e detecta automaticamente se são dados válidos, a quantidade inicial de dados pode ser variável, pois vai depender da frame de dados inicial (TAGv2). Então com o buffer cheio, com cerca de 25000 bytes de dados, haverá sempre dados suficientes para o STA013 não ficar em ciclo fechado a inserir o mesmo conjunto de dados, assim, logo que o STA013 tenha dados suficientes no seu buffer interno ele inicia a reprodução saindo da função de pedidos dados, o que permite executar outra tarefa. Nesse momento inicia-se uma nova transmissão de dados que serão colocados na posição de dados lidos anteriormente pelo descodificador. Este envio só é efectuado se existir espaço para uma nova transmissão, o que vai depender da quantidade de bytes lidos pelo descodificador e pela quantidade de bytes a serem enviados pelo servidor.

A quantidade de dados a enviar pelo servidor, deve ser de acordo com o tempo que este processo demora a executar, isto é, o tempo de recepção dos dados e armazenamento no buffer não pode ser superior ao tempo entre os pedidos de dados do STA013, pois se for superior este vai provocar interrupções no som durante a sua reprodução.

5.2 A reprodução

A reprodução é feita pelo descodificador de MP3, o STA013, que activa um sinal pedindo mais dados, este sinal é feito através do pino 28 do STA013, denominado de Data Request. Este pino é configurável, isto é, pode-se optar por uma activação em nível baixo ou nível alto. Neste projecto este pino é activo alto, ou seja, sempre que o descodificador necessitar de mais dados ele coloca este pino a nível lógico alto, e só o

colocará a nível lógico baixo, quando o seu buffer interno estiver cheio. Este envio é implementado na função *main,* com o seguinte algoritmo:

ENVIO MÚSICA

```
Enquanto DATA_REQUEST==1
      {
            Copia o dado do buffer para uma variável
            Envia o dado
            Incrementa o apontador para o dado seguinte

            Se já passou a posição limite do buffer
                  Aponta para a posição inicial
      }
```

O envio dos dados é através do formato *serial input interface*, em que os dados são enviados bit a bit juntamente com um sinal de relógio para fazer o sincronismo dos dados. Este protocolo foi implementado por software através da seguinte função:

```
void funcaoSTA013(uint8 dado)
{
       int rotacao;
       // faz a rotação dos 8 bits do dado
       for(rotacao=7;rotacao>=0;rotacao--)
       {
             SCLK = 0;           //clock a zero
             SDI = (dado >> rotacao) & 1; //envia bit
             SCLK = 1;           //clock a um
       }
}
```

Sempre que esta função é chamada um byte é enviado.

A reprodução dos dados também é feita de uma forma cíclica, isto é, o descodificador inicia a leitura a partir da posição inicial, e sempre que atingir a posição máxima de dados no buffer na leitura seguinte este volta a ler a partir da posição inicial, pois entretanto o buffer foi recarregado, como será descrito no próximo ponto.

5.3 O Buffer

O armazenamento dos dados é feito num buffer que está disposto na memória externa SDRAM. Este buffer não é mais do que uma área de memória reservada para armazenamento de dados. A forma de armazenamento de dados é feita através da colocação dos dados a partir do endereço 0x1000, e não pode ultrapassar a posição mais

elevada da SRAM, isto é, o buffer pode ter até, 0x7FFF - 0x1000 = 0x6FFF, o que equivale a 28671 bytes de dados no máximo.

Foi adoptado o armazenamento a partir da posição 0x1000, pois durante a execução do processo alguns valores ficam guardados na memória externa como valores auxiliares, tais como o apontador para a posição de memória do RX e do TX, endereços IP, MAC, endereço do servidor, do gateway, variáveis auxiliares da função *main*, então decidiu-se deixar esta área de memória disponível para esse efeito. Ficando assim a restante área de memória da SRAM disponível para o buffer.

A gestão de dados armazenados neste buffer é feita através de uma variável inteira que sabe a quantidade de dados existente no buffer, e sempre que se verificar a existência de espaço para uma nova transmissão esta é efectuada. Garantindo assim que não exista colocação de dados em posições do buffer onde estejam dados que ainda não tenham sido reproduzidos. Esta variável é decrementada à medida que os dados são enviados para reprodução e incrementa a quantidade de dados recebidos pela rede durante cada transmissão.

Estes dados são armazenados nas posições deixadas vazias no buffer durante o envio de som para o descodificador, e sempre que seja atingida a posição limite do buffer os dados voltam a ser armazenados a partir da posição inicial.

Este buffer deixa de ser recarregado quando o ficheiro tiver sido todo enviado, o descodificador reproduz assim os restantes dados do buffer, e fica à espera de um novo som para reprodução. Iniciando novamente mais um ciclo de reprodução de som.

Capítulo VI

6 Testes e resultados em funcionamento

Neste capítulo serão discutidos alguns pontos relacionados com o funcionamento do Sound-Ether, e para o devido efeito foram realizados diversos testes para aferir a funcionalidade do mesmo.

Este dispositivo foi colocado numa rede ponto a ponto e numa rede do departamento de electrónica ligada através de um gateway.

Para se realizarem testes de funcionamento foi necessário criar um servidor, que enviasse som para o dispositivo, sendo este servidor criado em Visual C++ o qual tem o seguinte aspecto:

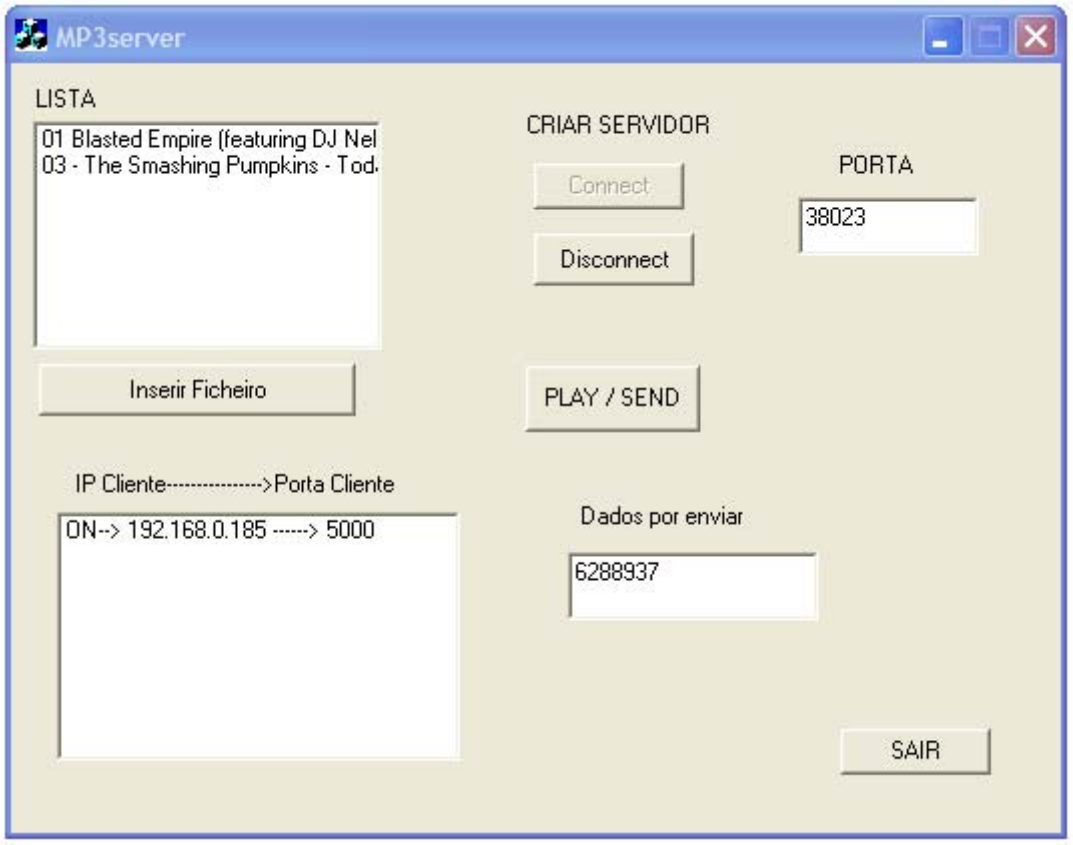


Fig. 38 - Servidor

Este servidor integra a classe CAsyncSocket, que encapsula as funções da API *(Application Program Interface)* do Windows Sockets, para implementação de um

servidor TCP/IP com sockets. Como este servidor espera que um cliente se tente conectar, estabelecida a ligação com o cliente pode-se inserir um som e iniciar a sua reprodução. A reprodução do ficheiro é iniciada com o envio em forma de string do tamanho do som, em seguida é lido do ficheiro para um buffer uma certa quantidade de dados para envio, e o servidor espera por um pedido de dados por parte do cliente, sempre que exista um pedido é enviado para o cliente os dados presentes no buffer. A leitura do ficheiro é através da implementação dos membros da Classe CFile. As principais funções usadas neste servidor encontram-se no Anexo B.

6.1 Ligação Ponto a Ponto

Na ligação Ponto a Ponto, é estabelecida uma conexão entre um servidor e um cliente directamente através de um cabo cruzado UTP, sem a necessidade de concentradores. Neste caso foi estabelecida uma conexão directa entre um Computador e um dispositivo de reprodução, sendo a velocidade de ligação estabelecida entre os dois dispositivos de 100Mbps. E foram verificadas algumas amostras de utilização da rede no envio de uma música de 3 minutos e 42 segundos.

Neste caso, a rede apenas estava a ser usada para envio da música entre o servidor e o cliente, isto é, o computador e o Sound-Ether. Em seguida serão analisadas algumas utilizações da rede durante a reprodução da música com três qualidades de reprodução, a 128kbps, a 256kbps e com a qualidade variável desde 128kbps a 320kbps, neste caso a música está codificada de uma forma aleatória que vai desde os 128kbps aos 320kbps. A frequência de amostragem é sempre de 44.1kHz, equivalente à qualidade de um CD de áudio normal.

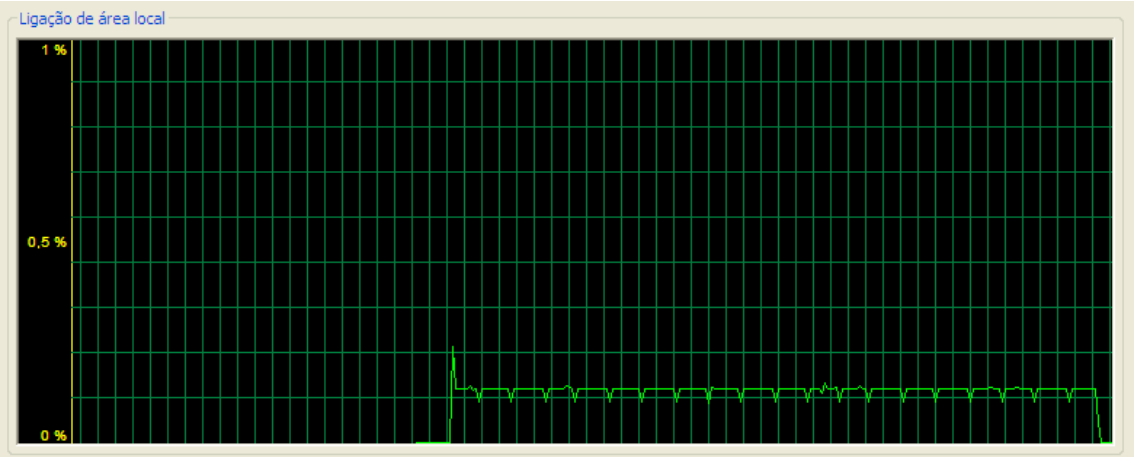


Fig. 39 - Utilização da rede – música de 128kbps

Na figura anterior pode-se verificar a utilização da rede para uma música com qualidade de 128kbps. No eixo das ordenadas verifica-se a utilização da rede, a escala é colocada automaticamente pelo gestor de tarefas do Windows e corresponde a uma percentagem da utilização da rede. Como estamos perante uma rede de 100Mbps, logo 1% corresponde a 1Mbps, então com esta qualidade verifica-se que a utilização de rede é inferior a 0,2%. No eixo das abcissas é mostrado o tempo, em que cada divisão corresponde aproximadamente a 6 segundos, o que podemos verificar que durante a reprodução da música de 3 minutos e 42 segundos (222 segundos) obtemos uma utilização da rede durante aproximadamente 37 divisões.

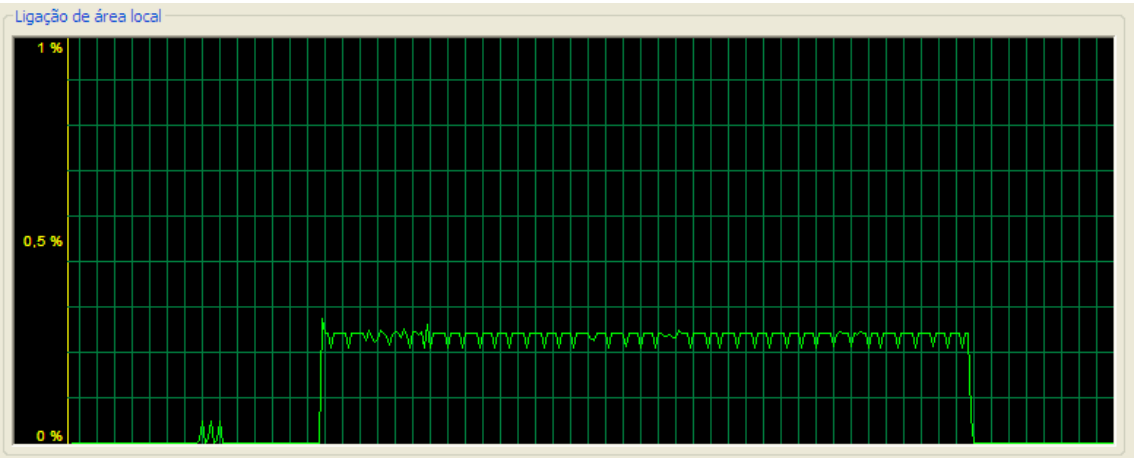


Fig. 40 - Utilização da rede – música de 256kbps

Na Fig. 40, pode-se verificar a utilização da rede para uma música com qualidade de 256kbps. Com esta qualidade verifica-se que a utilização de rede já é superior a 0,2%. Neste caso a utilização é maior que a situação de 128kbps, isto acontece porque cada música tem o mesmo tempo apenas o que muda é quantidade de

bits para codificar a mesma música. Como foi demonstrado no Capítulo I no item 1.2.1 sobre a estrutura de um ficheiro MP3, cada frame de áudio tem sempre o mesmo tempo, que são 26ms, mas o número de bytes da frame depende da frequência de amostragem, e da qualidade da música, que corresponde ao número de bits enviados por segundo. Então pela expressão:

$$Tamanho\quad da\quad Frame = \frac{144 \times BitRate}{Frequência\quad Amostragem} + Padding\,,$$

se aumentar a qualidade (BitRate) e mantendo a frequência de amostragem, o tamanho da frame vai aumentar, logo o tamanho do ficheiro MP3 vai aumentar, no entanto o tempo de reprodução da música é o mesmo, o que faz com que a quantidade de dados a enviar durante a reprodução da música tenha de ser maior. Verifica-se assim que na realidade quanto maior a qualidade da música, maior vai ser a utilização da rede.

No caso seguinte pode-se verificar que numa música de qualidade variável, a utilização da rede tem mais altos e baixos durante a reprodução da mesma, não sendo tão constante a utilização da rede.

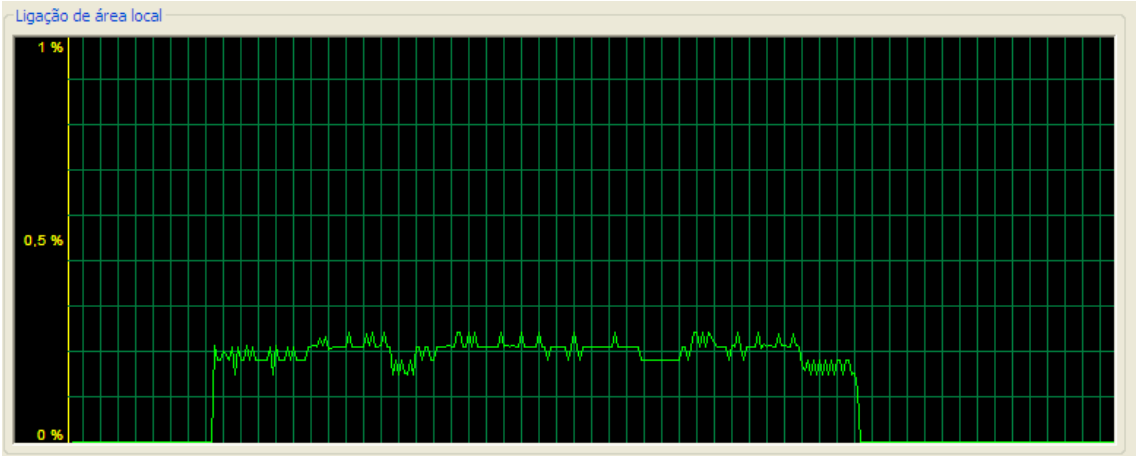


Fig. 41 - Utilização da rede – música de qualidade variável

6.2 Ligação na rede do departamento de electrónica

A ligação em estrela foi realizada na rede interna do departamento, em que o cliente foi colocado num cabo disponível no laboratório de 100Mbps e o servidor num

outro cabo mas de 10Mbps. Estando assim a conexão limitada a 10Mbps. Neste caso estes dispositivos estão ligados através de um gateway.

Nesta ligação foram realizados testes iguais aos testes realizados na ligação ponto a ponto, com a mesma música nos diferentes formatos.

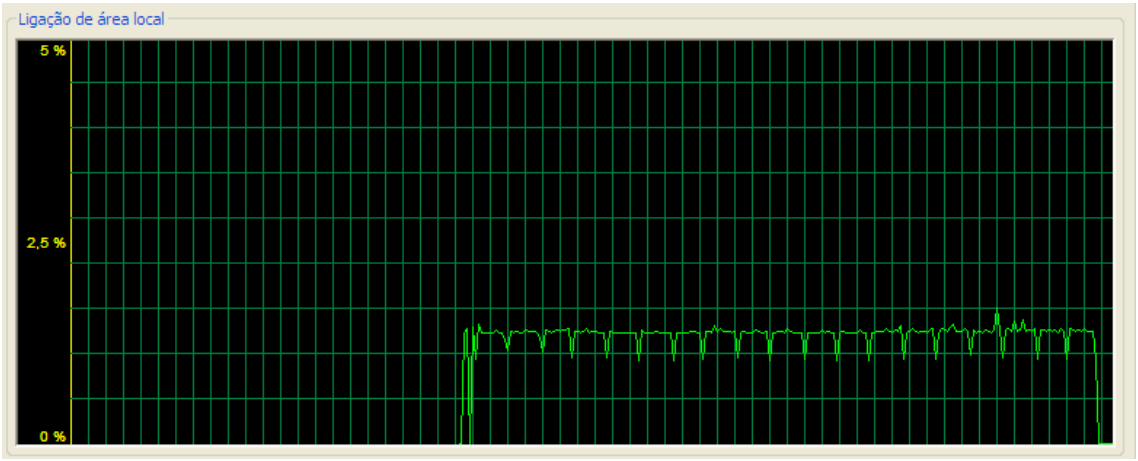


Fig. 42 - Utilização da rede – música de 128kbps

No caso dos 128kbps, pode-se verificar que a utilização da rede anda à volta do 1%. O que está próximo da realidade, pois numa ligação de 10Mbps, e com uma qualidade de música de 128kbps, daria uma utilização de 1,28% de utilização da rede. Mas neste caso nota-se que a transmissão não é tão constante como na ligação ponto a ponto, isto pode acontecer pois estamos perante uma rede que não é totalmente usada para este efeito, mas a nível de som não ocorrem interferências.

Com uma música de 256kbps foi mais difícil obter um resultado, só ao fim de algumas tentativas é que se conseguiu, pois neste caso a obtenção de dados através da rede pretende-se que seja rápida, pois como a qualidade da música é maior logo o pedido de dados pelo STA013 para reprodução também é maior, visto que cada frame tem mais dados, logo a transição entre recepção e reprodução tem de ser mais rápida. Se existir uma falha no envio, a repetição do envio pode ser tardia, ficando a reprodução em ciclo fechado, perdendo-se assim a ligação com o servidor para recepção de dados.

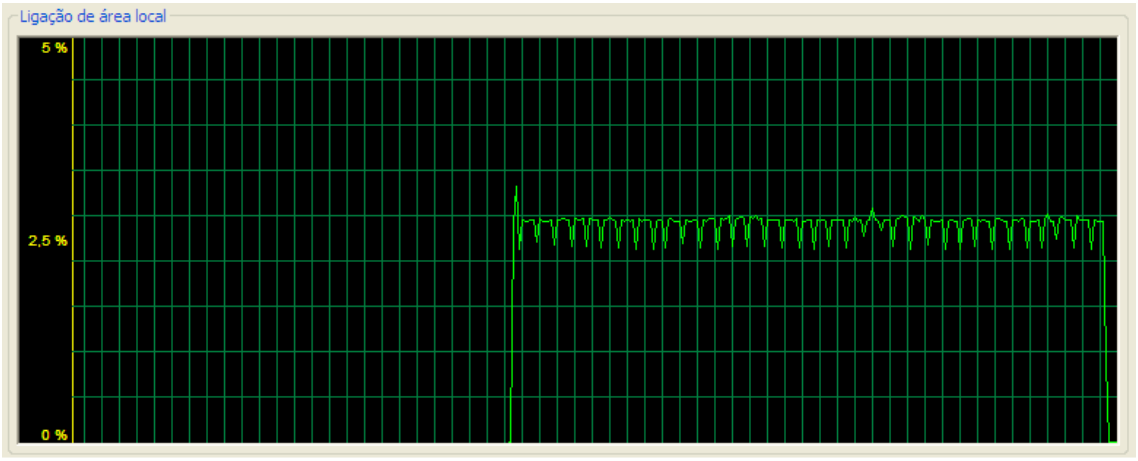


Fig. 43 - Utilização da rede – música de 256kbps

A utilização da rede está à volta dos 2.5%, neste caso também se pode observar que o envio não é tão constante como na ligação ponto a ponto, mas a reprodução da música foi normal e sem interferências.

Na situação seguinte voltamos a ter a mesma música mas com uma qualidade de som variável, isto é, a música varia entre os 128kbps e os 320kbps.

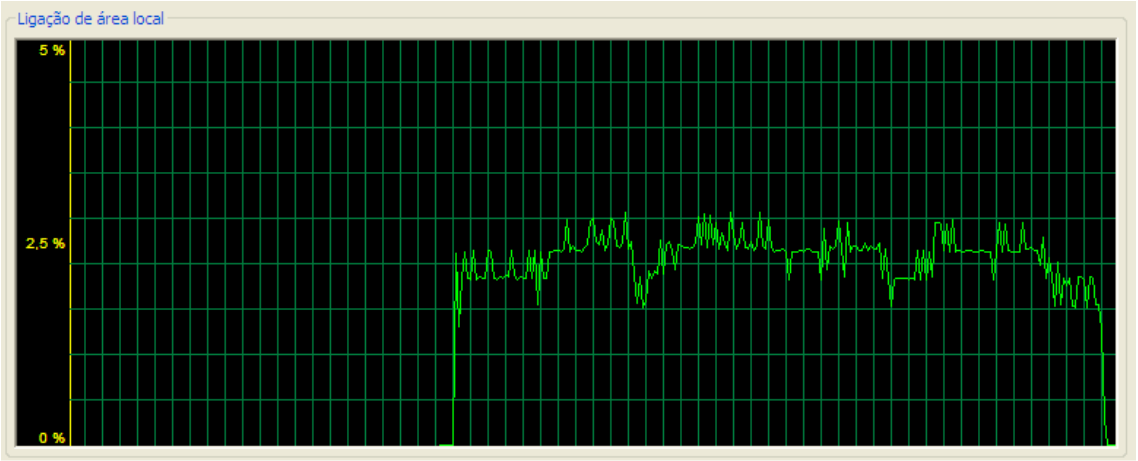


Fig. 44 - Utilização da rede – música de qualidade variável

Nesta situação volta a observar-se altos e baixos durante o envio da música, pois como a qualidade é variável, o pedido de dados também se torna variável.

Capítulo VII

7 Conclusões e Trabalho Futuro

7.1 Conclusões

O Sound-Ether consiste num hardware construído para reproduzir áudio enviado pela Ethernet, idealizado para poder ser inserido numa casa automatizada, em que pode ser inserido na rede local e reproduz o áudio enviado através da mesma. Este hardware deve ser configurado para a rede onde será inserido, através do seu endereço IP, endereço do gateway, e endereço e porta do servidor do qual pretende estabelecer ligação. Com a definição do endereço IP pode-se assim escolher o dispositivo do qual se pretende reproduzir o áudio, no entanto esta selecção terá de ser feita pelo servidor instalado na rede ao qual os dispositivos estabelecem ligação.

O Hardware criado é um cliente numa rede TCP/IP que estabelece ligação com um servidor e recebe os dados do ficheiro MP3 em pequenas porções de uma forma ordenada.

O hardware foi construído e o seu funcionamento testado em dois tipos de ligações distintos, e pelos resultados obtidos conseguiu-se atingir os objectivos inicialmente delineados.

O dispositivo possui no entanto algumas limitações, nomeadamente no que se refere à velocidade de pedido de dados pelo descodificador de MP3 relativos a ficheiros com qualidades elevadas de áudio, isto é, à medida que se aproxima dos 320kbps, que é a qualidade máxima, a reprodução fica limitada pois o microcontrolador não tem a capacidade de receber áudio pela rede e enviar para reprodução tão rápido quanto o pedido de dados. O que acontece é que, o dispositivo descodificador como tem que inserir mais dados por frame, pois se a qualidade aumenta o número de bytes de cada frame aumenta, para a mesma frequência de amostragem, tornando assim o pedido de dados mais rápido colocando em ciclo fechado o estágio de reprodução, impedindo o sistema de sair desse estágio para entrar no estágio de recepção de mais dados. Pelos testes realizados apenas se conseguiram qualidades até 256kbps de codificação do MP3.

No que respeita as dificuldades sentidas na realização deste projecto, pode-se apontar como dificuldade principal o desbravar do caminho até encontrar a resposta mais adequada para a execução da tarefa pretendida, tendo em conta o uso do módulo de rede NM7010B, que foi disponibilizado no início deste caminho como o módulo de rede possível para o efeito pretendido. O recurso a este módulo de rede demonstrou-se um grande desafio em consequência dos inúmeros registos e configurações necessárias para o seu funcionamento numa rede, assim como, a escassa e baixa qualidade de informação disponível. A resolução de muitos dos problemas enfrentados nesta fase só foi possível devido à utilização de um driver em linguagem C disponível pelo fornecedor na sua página da internet. Com este driver tornou-se mais fácil a percepção do seu funcionamento.

Outra dificuldade encontra-se associada à fase inicial da criação deste protótipo, dado que, as velocidades de comunicação entre o microcontrolador e o módulo NM7010B para os testes de funcionamento, interacção e aprendizagem entre comunicação entre ambos dispositivos não foram possíveis nas placas de testes (Bread-Boards) o que obrigou, após inúmeras tentativas nas bread-boards, à construção de um PCB, para as realizar.

7.2 Trabalho futuro

A construção deste trabalho permite a sugestão de algumas melhorias, uma ponte para futuros trabalhos que possam explorar outros aspectos, nomeadamente no que se refere a construção do hardware mediante o meio onde este será inserido, a construção do servidor e a sua fonte de alimentação.

O aspecto mais premente de todos seria ultrapassar a limitação referente a qualidade de codificação do MP3 na reprodução.

Um segundo aspecto a ser explorado seria de acordo com a forma e meio onde será inserido o hardware, tal como o tamanho e robustez necessária para a sua aplicação no meio, sendo ainda possível acrescentar um andar amplificador de áudio para obter directamente o som amplificado para ligação directa às colunas de som.

Outro aspecto a ser melhorado está directamente relacionado com o servidor, sendo possível a criação de uma trama para executar um controlo directo sobre a descodificação de áudio, uma vez que o descodificador de MP3 permite, controlar o

volume, selecção de Mute e Play e controlar o bass e o treable durante a reprodução de áudio.

Neste Hardware seria ainda possível colocar uma fonte de alimentação interna, para que seja possível ser conectado directamente à rede eléctrica sem ser necessário recorrer ao uso de nenhuma fonte de alimentação externa.

Por fim e provavelmente o mais importante seria a criação de um servidor para interagir com este cliente, o qual poderá incorporar mais funções e não limitar-se a enviar porções de áudio em MP3 para o cliente, o que poderia ser possível através do uso de software que permita ao servidor converter outros formatos de áudio para MP3 instantaneamente e permitir que o servidor faça a leitura directa de um microfone e envie em formato MP3 para o cliente.

Capítulo VIII

8 Referências e Bibliografia

8.1 Referencias

[1] Fernando Andrade
Pagina na internet
http://lassommoir.blogspot.com/2007/11/arte-das-musas.html

[2] Ted Painter and Andreas Spanias, Perceptual Coding of Digital Áudio, Department of Electrical Engineering, Telecommunications Research Center Arizona State University, Tempe, Arizona 85287-7206, 2000

[3] K. Brandenburg and H. Popp, An Introduction to MPEG Layer III, Fraunhofer Institut für Integrierte Schaltungen (IIS), 2000

[4] Karlheinz Brandenburg, MP3 AND AAC EXPLAINED, Fraunhofer Institute for Integrated Circuits FhG-IIS A, Erlangen, Germany, 2000

[5] Predrag Supurovic, MPEG Áudio Frame Header,
Página da internet criada em Setembro de 1998,
http://mpgedit.org/mpgedit/mpeg_format/mpeghdr.htm

[6] Autor desconhecido, MP3 File Structure Description,
Página da internet
http://www.multiweb.cz/twoinches/MP3inside.htm#MP3FileStructure

[7] M. Nilsson, id3v2.3.0 – ID3.org,
Página da internet criada a 3 de Fevereiro de 1999
http://id3.org/id3v2.3.0#head-0ef7011e13ae8b3678a676a65b64760b9cedf1de

[8] Paulo Cardoso, Texto de Apoio ao Módulo de Redes de Computadores, Disciplina de Laboratórios integrados III, Departamento de Electrónica Industrial da Universidade do Minho, versão 04/05

[9] 2006-2008 NXP Semiconductors, Facts,
Página da internet
http://www.nxp.com/products/interface_control/i2c//facts/#terminology

[10] Philips Semiconductors, THE I 2C-BUS SPECIFICATION, VERSION 2.1, Janeiro de 2000
Página de internet

http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf

[11] Philips Semiconductors, I^2S bus specification,
Página da internet,
http://www.nxp.com/acrobat_download/various/I2SBUS.pdf

[12] Wiznet Co., NM7010B Datasheet, Versão 1.3, 2005
Página da internet
http://www.wiznet.co.kr

[13] Wiznet Co., W3150A Datasheet, Versão 1.0.2, 2005
Página da internet
http://www.wiznet.co.kr

[14] Philips, PC74HCT373 Datasheet, Octal D-type transparent latch: 3-state

[15] Texas Instruments, SN74LS139 Datasheet, Dual 2-Line to 4-Line Decoders/Demultiplexers

[16] Dallas Semiconductor Maxim, DS89C430/DS89C450 Ultra-High-Speed Flash Microcontrollers Datasheet
Página da internet
http://www.maxim-ic.com

[17] Dallas Semiconductor Maxim, Ultra-High-Speed Flash Microcontrollers User's Guide
Página da internet
http://www.maxim-ic.com

[18] Dallas Semiconductor Maxim, DS1230Y/AB 256k Nonvolatile SRAM
Página da internet
http://www.maxim-ic.com

[19] ST Microelectronics, STA013 Datasheet, MPEG 2.5 LAYER III ÁUDIO DECODER
Página da internet
http://www.st.com/stonline/

[20] ST Microelectronics, AN1090 Application Note STA013 MPEG 2.5 LAYER III SOURCE DECODER
Página da internet
http://www.st.com/stonline/

[21] Paul Stoffregen, MP3 Player, How To Use The STA013 MP3 Decoder Chip, Fevereiro de 2005
Página da internet
http://www.pjrc.com/tech/mp3/sta013.html

[22] Maxim, MAX5556 – MAX5559, Low-Cost Stereo Áudio DACs

Página da internet
http://www.maxim-ic.com

[23] Texas Instruments, PCA9306 Datasheet, DUAL BIDIRECTIONAL I2C BUS AND SMBus VOLTAGE-LEVEL TRANSLATOR

[24] Texas Instruments, TXS0108E Datasheet, 8-BIT BIDIRECTIONAL VOLTAGE-LEVEL TRANSLATOR

[25] EtherSound, Technology: Overview, @2008 Digigram,
Página da internet
http://www.ethersound.com/technology/overview.php

[26] Barix, Barix Extreamer 100, IP Áudio Decoder for commercial, industrial and security applications
Página da internet
http://www.barix.com/index.php?linkid=431

[27] Barix, Barix Extreamer 110, IP Áudio Decoder with display for commercial, industrial and security applications
Página da internet
http://www.barix.com/index.php?linkid=701

[28] Barix, Barix Extreamer 200, Network áudio decoder with built-in amplifier for commercial, industrial and security applications
Página da internet
http://www.barix.com/index.php?linkid=451

[29] Barix, Barix Extreamer Digital, Network áudio decoder with digital output (SPDIF) for commercial, industrial and security applications
Página da internet
http://www.barix.com/index.php?linkid=441

8.2 Bibliografia

[1] Sá, Maurício Cardoso de, *Programação C para Microcontroladores 8051*, Editora Erica Ltda., 1ª Edição, 2005

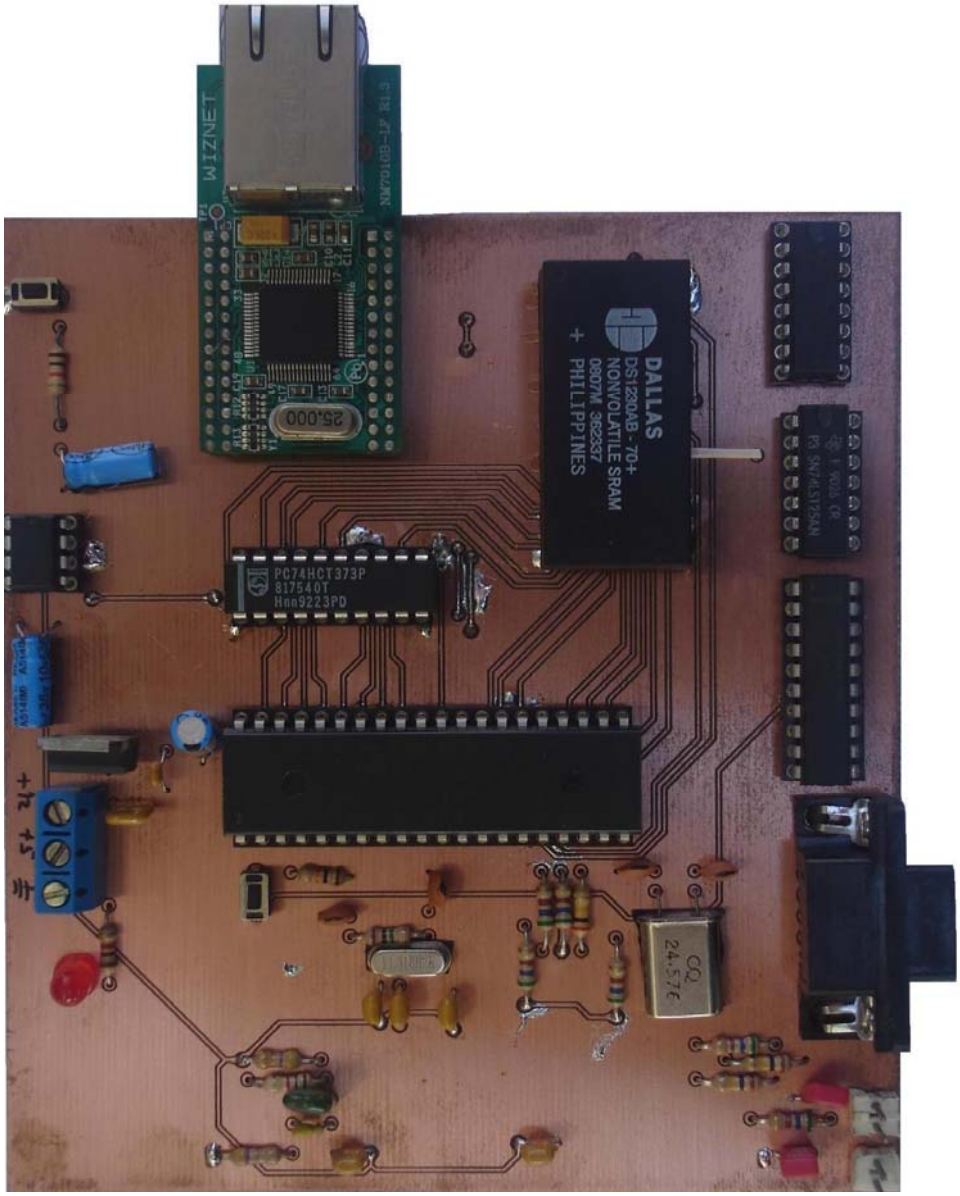
[2] Damas, Luís, *Linguagem C*, FCA – Editora de informática, 3ª Edição

[3] Guerreiro, Pedro, *Programação com Classes em C++*, FCA – Editora de informática, 2ª Edição

[4] Chapman, Davis, *Teach Yourself Visual C++ 6 in 21 Days*, SAMS

ANEXOS

ANEXO A – Manual de Utilização e configuração do Sound-Ether



Trabalho Realizado por:

Cristiano Diogo Pereira Santos

No âmbito da dissertação denominada Sound-Ether, realizada no departamento de Electrónica Industrial da Universidade do Minho sobe a orientação do Professor Doutor Fernando Ribeiro

Versão 1.0

O Sound-Ether

O Sound-Ether conisste num dispositivo de ligação a uma rede local com a finalidade de reproduzir sons em formato MP3 recebidos por um servidor.

Para este dispositivo funcionar correctamente é necessário saber em que rede se vai colocar, isto é, o endereço do gateway e endereço da mascara de rede. É ainda necessário definir um endereço IP e um endereço físico (MAC) por dispositivo. Para que a comunicação com o servidor funcione correctamente configura-se o endereço do servidor e a porta do servidor do qual se pretende ligar, definindo também a porta do cliente.

Para colocar o Sound-Ether em funcionamento é necessário realizar as seguintes tarefas:

1. Configurar os endereços
2. Compilar o software de funcionamento com as definições anteriores
3. Ligar o dispositivo a uma fonte de alimentação
4. Carregar para o microcontrolador o software
5. Ligar o dispositivo numa rede e a saída de áudio
6. Criar um servidor e estabelecer ligação com o dispositivo

As ferramentas necessárias, são:

a) Computador com porta série, ou com cabo conversor USB-série
b) Software de carregamento de programas MicroController Toll Kit da Dallas Semiconductor versão 2.2.0 ou superior
c) Ferramenta de Programação Keil µVision versão 2.40a ou superior, com compilador C51
d) Software Principal do Sound-Ether, ficheiro Sound-Ether.zip

1. Configurar os Endereços

Antes de começar deve-se ter instalado a ferramenta de programação Keil µVision com o compilador C51. Posteriormente deve-se descompactar o ficheiro Sound-Ether.zip para um local conhecido, esta descompactação cria uma pasta Sound-Ether, a qual contém os ficheiros necessários para compilar o nosso software de funcionamento do Hardware. Devem existir os ficheiros como demonstra a imagem seguinte.

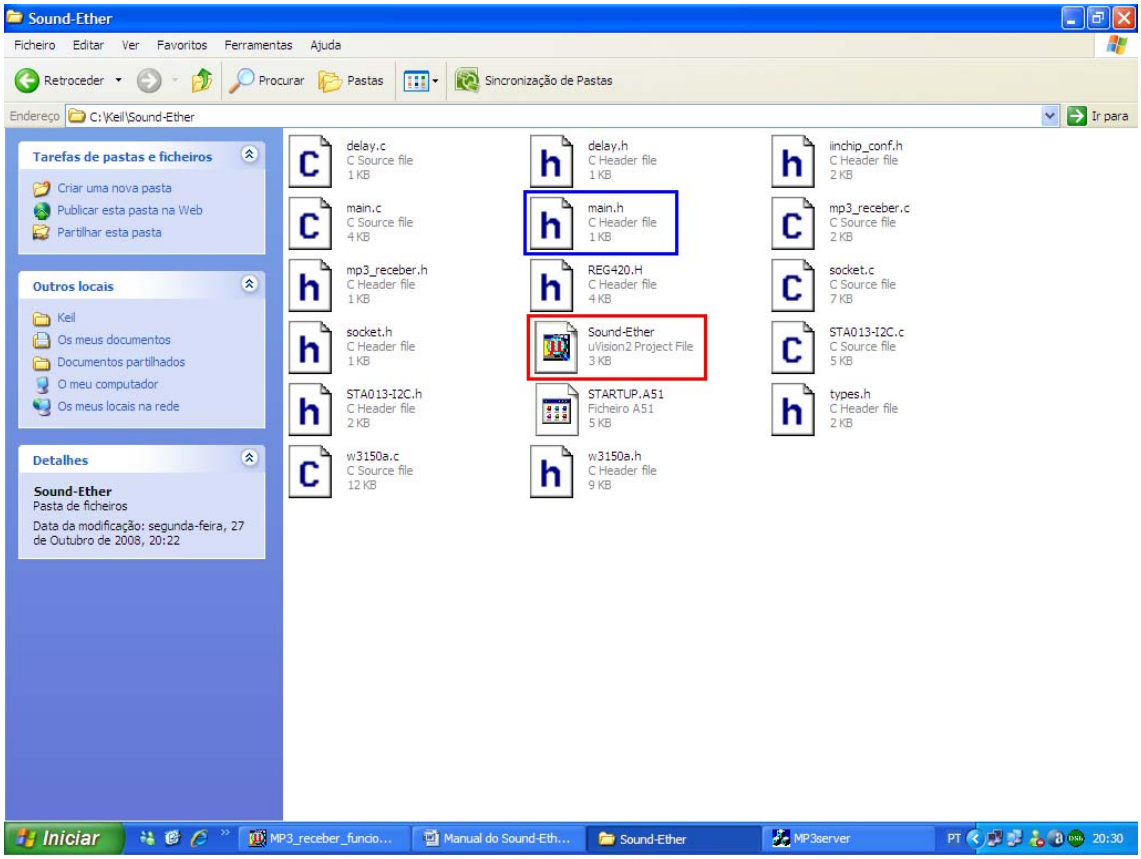


Fig. 45 - demostração dos ficheiros contidos no Sound-Ether.zip

Seguidamente configura-se os endereços necessários para funcionamento na rede onde o Sound-Ether será inserido. Para isso abre-se o ficheiro ***main.h*** (seleccionado a azul na figura anterior) pois é onde se pode inserir todos os dados para configuração da rede. A forma mais simples é abrir o ficheiro com o bloco de notas e nos respectivos campos inserir os dados.

Os campos que devem ser configurados são:

1. Endereço Gateway
2. Endereço da Máscara de rede
3. Endereço IP do dispositivo

4. Endereço Físico (MAC Address)
5. Endereço IP do servidor
6. Porta do servidor
7. Porta do cliente

No ficheiro ***main.h***, estão visiveis com cores os campos seleccionados, tal como demonstra a figura seguinte.

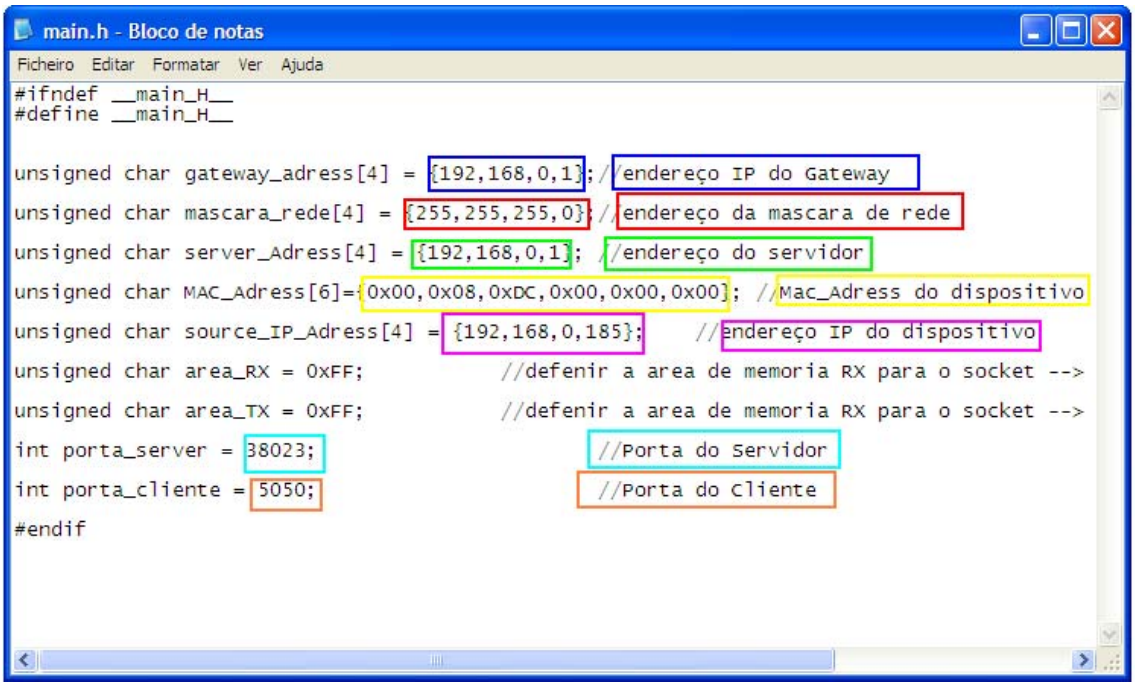


Fig. 46 - Imagem do ficheiro *main.h*

Após termos alterado todos os dados, devem guardar-se as alterações.

2. Compilar o software de funcionamento

Após terem sido definidas as configurações, compila-se o software através da ferramenta keil µVision. Para isso, basta clicar duas vezes no ficheiro Sound-Ether.Uv2, marcado a vermelho na Fig. 45. Este abrirá a ferramenta keil µVision com o software, compila-se o programa para ser criado o ficheiro Hexadecimal, o qual será carregado para o microcontrolador. Esta operação é realizada através desta ferramenta, clicando no botão Rebuild all target files (marcado a vermelho).

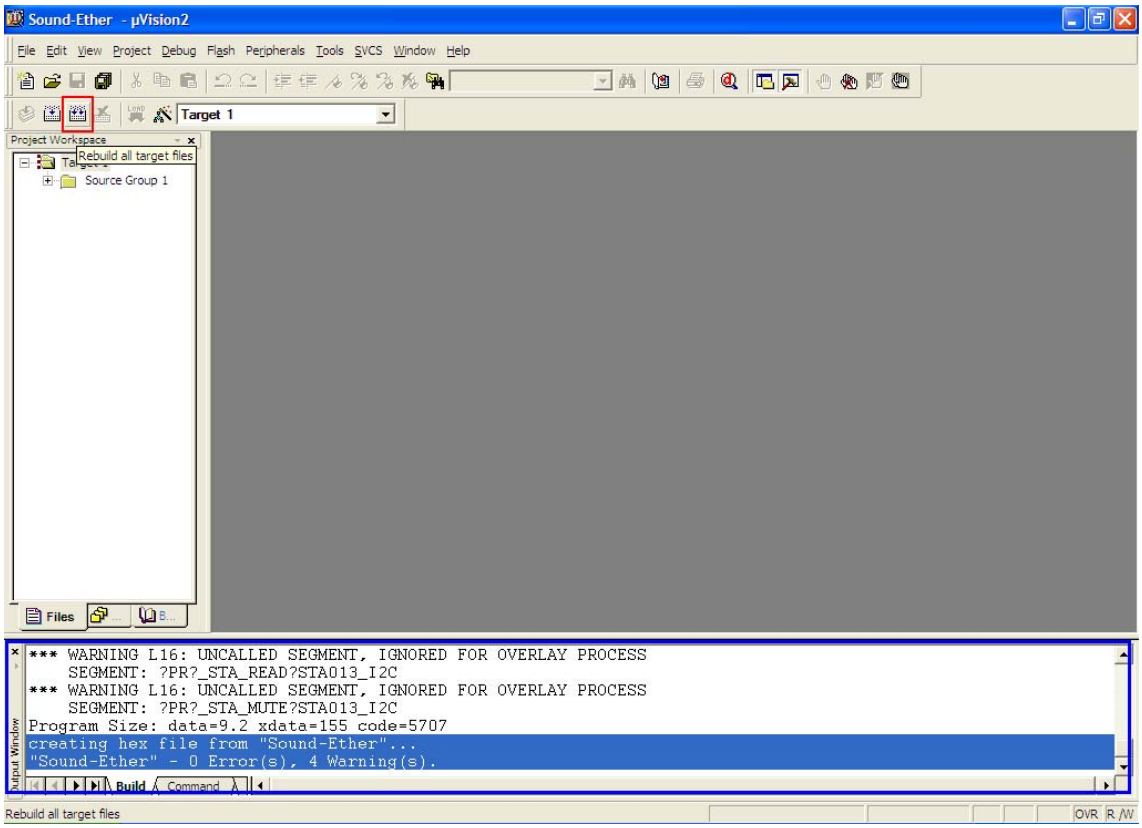


Fig. 47 - Compilar o programa

Dada a ordem para compilar é visivel no Output Window do keil (marcado a azul) a informação da compilação. A compilação deste programa irá gerar na pasta Sound-Ether um ficheiro hexadecimal com o nome Sound-Ether.hex. Este é o ficheiro a carregar para o microcontrolador do dispositivo.

3. Ligar a alimentação ao dispositivo

A alimentação deve ser efectuada a partir de uma fonte de alimentação externa de tensões contínuas. O Sound-Ether funciona a alimentações de 5V contínuos, podendo ser alimentado directamente a 5V contínuos ou a uma tensão máxima de 35V contínuos, pois o Sound-Ether possui um regulador de tensão para 5V. Os pontos de ligação estão descritos na figura seguinte.

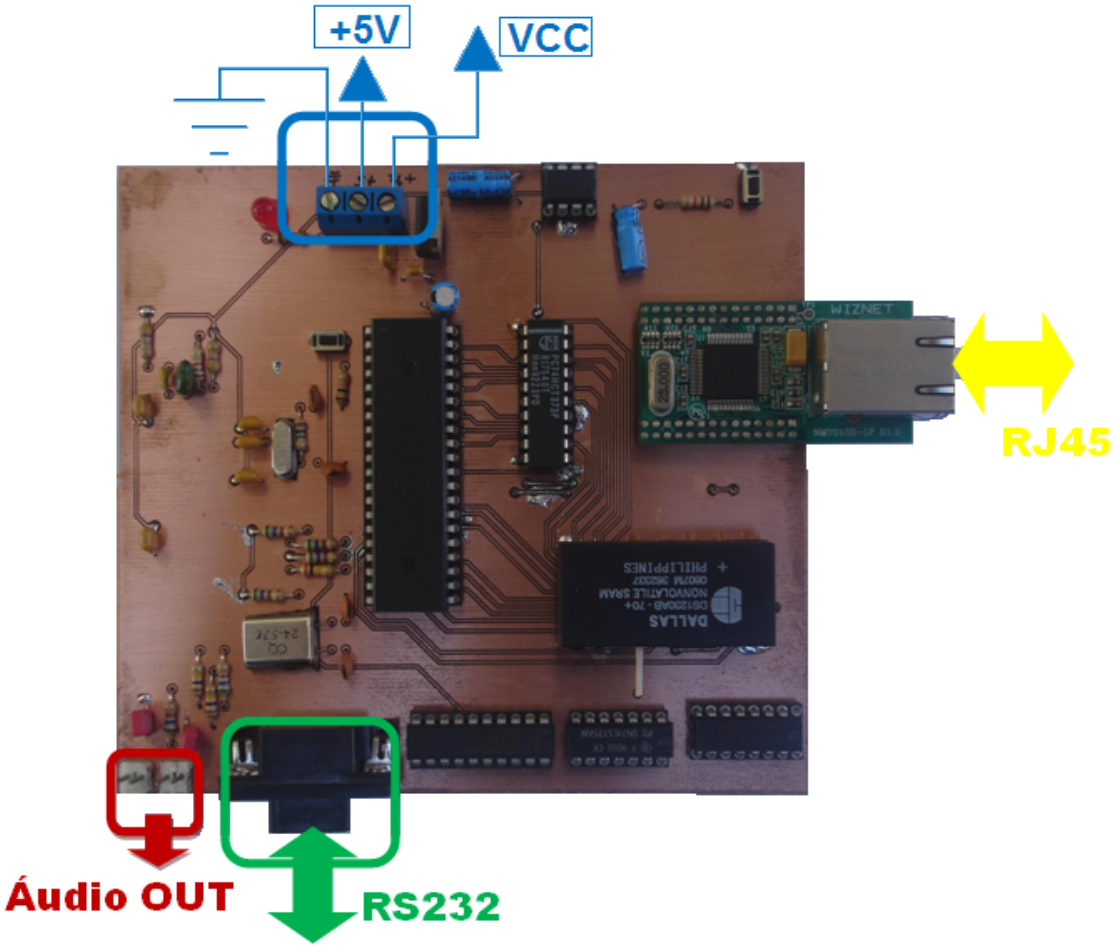


Fig. 48 - Sound-Ether

4. Carregar o software para o microcontrolador

O carregamento do programa para o microcontrolador é feito através da aplicação Microcontroller Tool Kit (MTK) usada para comunicar com a memória de programa da maior parte dos microcontroladores da *Dallas Semiconductor.* Esta aplicação tem como principal actividade configurar, carregar e descarregar código para o dispositivo conectado através de uma porta série de um computador.

Antes de começar, deve-se ligar o cabo série do computador ao conector RS232 do Sound-Ether e depois ligar a alimentação do Sound-Ether.

Executa-se a aplicação MTK e selecciona-se o microcontrolador DS89C450.

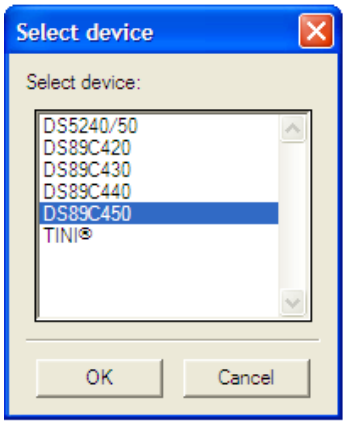


Fig. 49 - Seleccionar o dispositivo

Depois aparece uma janela e configura-se a porta série para um baud-rate de 9600. Para isso seleciona-se Options -> Configure serial Port

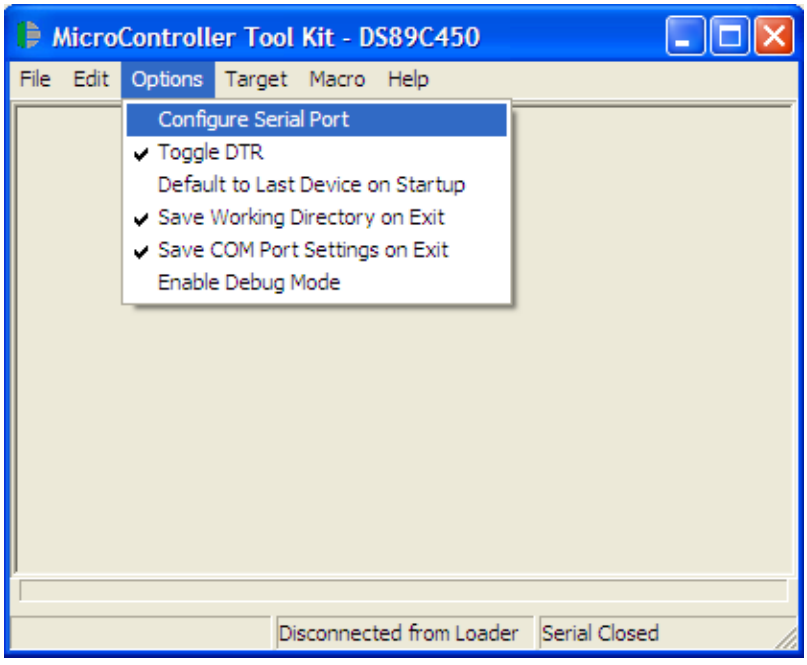


Fig. 50 - Configurar porta serie

Configurada a porta abre-se a porta, clicando no separador Target ->Open COMX at 9600 baud, ou então clica-se simultaneamente nas teclas Control + O e a porta abre.

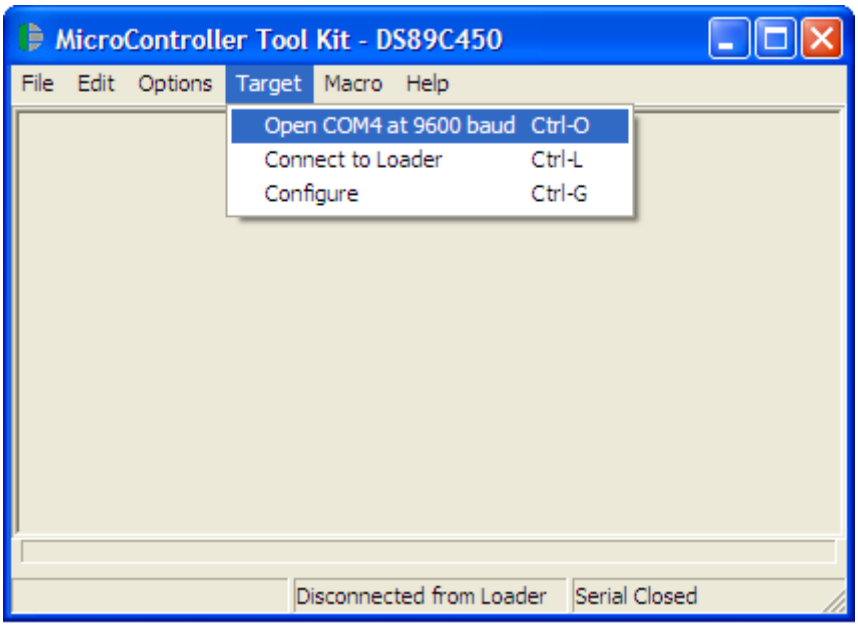


Fig. 51 - Abrir a porta serie

Para conectar com o microcontrolador executa-se o comando Control + L simultaneamente ou então no separador Target escolhe-se a opção Connect to Loader. Se este se conectar com o microcontrolador aparece uma mensagem como se pode verificar na imagem seguinte.

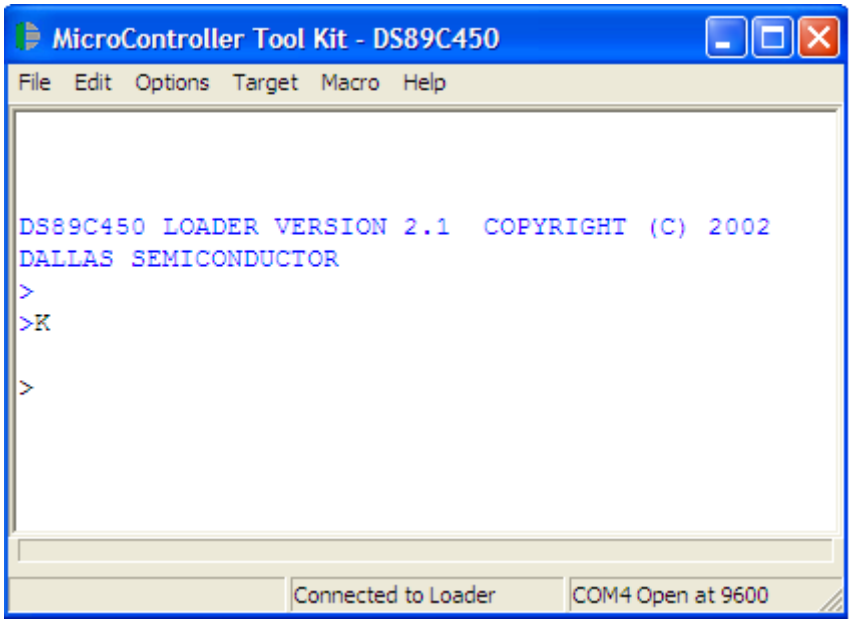


Fig. 52 - Conectado com o microcontrolador

Após estar conectado, deve-se apagar a memória do microcontrolador e para isso clica-se em **k** e posteriormente em **Enter**, assim está apagada a memória. Agora

procede-se ao carregamento do ficheiro hexadecimal criado na compilação. Para isso usa-se o separador File, selecciona-se a opção Load Flash, ou através do comando Control + F, que irá abrir uma janela onde se deve ir à directoria onde se encontra o ficheiro hexadecimal (pasta Sound-Ether). Aí, pode-se ver um ficheiro com o nome Sound-Ether.hex, selecciona-se este ficheiro e faz-se abrir, este ficheiro é carregado automaticamente.

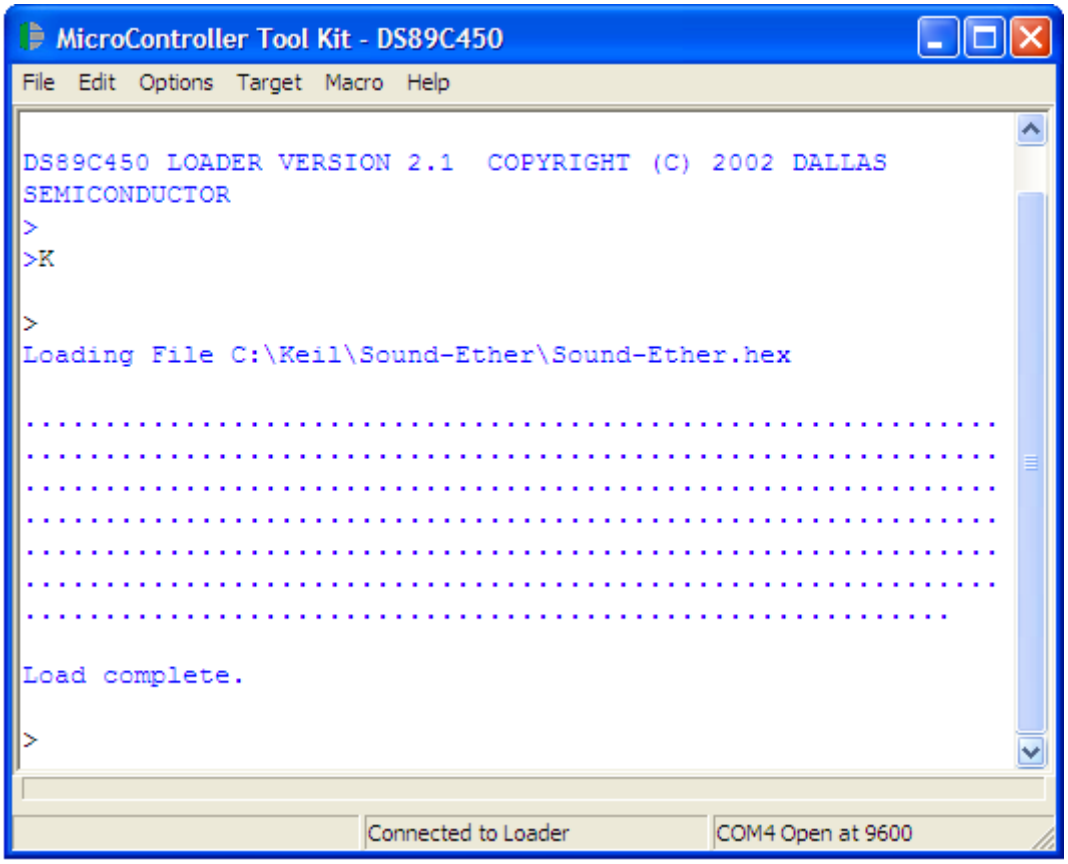


Fig. 53 - Carregamento do ficheiro hexadecimal

Quando este estiver carregado fecha-se a aplicação MTK e desliga-se o cabo série pois já se pode colocar o dispositivo na rede.

5. Ligar o Sound-Ether numa rede e a saída de áudio

Para ligar o dispositivo na rede usa-se um cabo de rede com ligação RJ45 disponível para conectar ao dispositivo, liga-se a saída de áudio a um amplificador ou a uns auscultadores para se poder ouvir os sons transmitidos pela rede e por fim liga-se a alimentação para o dispositivo entrar em funcionamento. Este quando ligado fisicamente à rede, informa o gateway da sua presença na rede para a actualização da sua tabela ARP. Podemos observar os locais de ligação dos cabos através da Fig. 48.

6. Criar um servidor e estabelecer ligação com o dispositivo

Por fim servidor estabelece a conexão com o dispositivo. O servidor pode ser feito de várias formas e em várias linguagens de programação, apenas tem de obedecer a algumas regras para que a troca de informação seja possível. A principal regra é que tem de ser um servidor com Sockets TCP em que a porta de ligação e o seu endereço IP tem de ser iguais ao configurado anteriormente nos dispositivos, pois será para onde se irá efectuar o pedido de ligação.

O servidor deve ficar à escuta do pedido de conexão por parte do cliente e permitir o estabelecimento da conexão.

A transmissão de áudio inicia com o envio do tamanho do ficheiro em forma de string pelo servidor. Esta operação indica ao cliente o início de uma transmissão de áudio.

Após esta indicação, o servidor deve esperar por um pedido de dados vindo do cliente. Este pedido de dados é feito através do envio de uma string com a palavra **SEND**. Após a recepção de pedido de dados o servidor deve proceder ao envio de dados para o cliente.

Os dados devem ser lidos do ficheiro MP3, e enviados em pequenas porções, isto é, a quantidade de dados enviada em cada transmissão do servidor para o cliente não deve ser maior que a área de memoria do RX. Neste caso está configurada para 8kbytes de dados.

Sempre que seja terminado o envio de um ficheiro, o cliente fica à espera que uma nova transmissão de áudio lhe seja pedida, isto é, espera pelo tamanho do ficheiro seguinte que se pretende reproduzir e todo o processo se irá repetir.

Exemplo de um servidor

Este é um servidor exemplo, que foi criado para teste de funcionamento durante o desenvolvimento do Sound-Ether, em linguagem de programação C++ na ferramenta Visual Studio 6. Neste servidor define-se a porta de ligação do servidor.

Clica-se em connect, e o servidor fica à escuta de um pedido de conexão. Inserem-se os ficheiros MP3 clicando em Inserir Ficheiro. Clicando em PLAY/SEND o servidor inicia a transmissão de dados.

Neste software pode-se observar o endereço IP e a porta de ligação do cliente conectado e analisar a quantidade de bytes que ainda faltam para terminar o envio do som para reprodução.

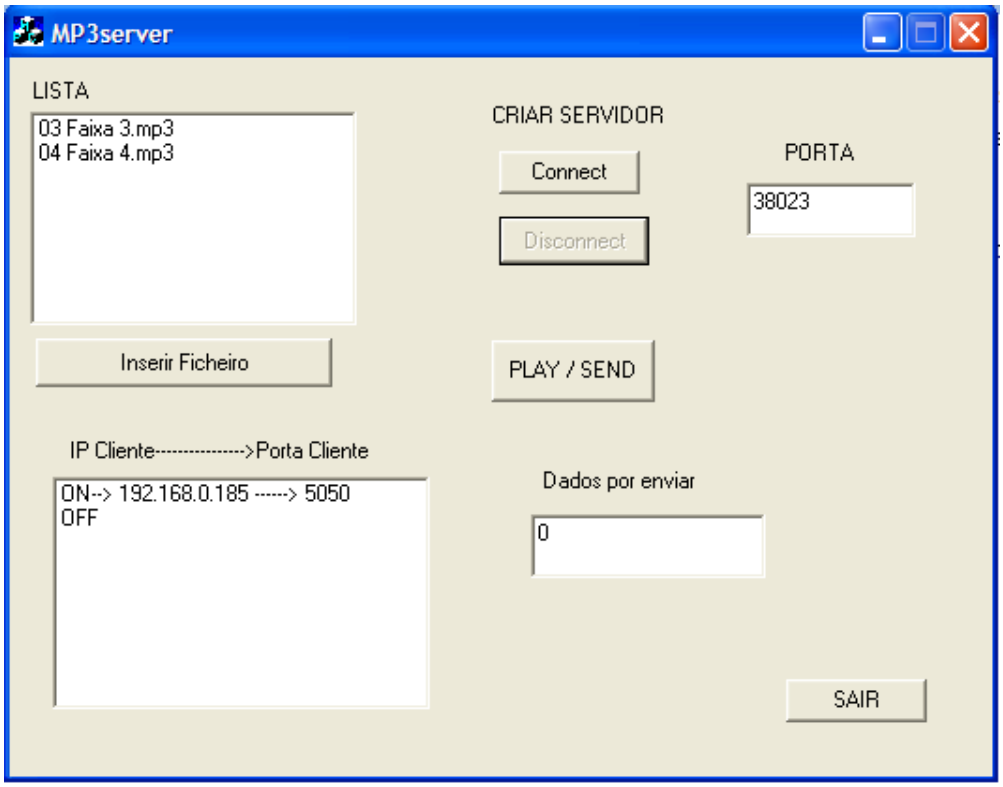


Fig. 54 - Exemplo de um servidor

ANEXO B – Código Fonte aplicado no microcontrolador em Linguagem C

MP3_receber.c

```
#include <stdio.h>
#include "socket.h"
#include "mp3_receber.h"


uint16 MP3_RECEBER(uint8 * addr,u_char xdata * posicao,uint16 porta,
uint16 size)
{
//adrr --> endereço servidor
//posicao --> posiçao da memória para onde irão ser guardados os dados
//porta --> porta correspondente ao cliente
//porta_servidor --> Porta do servidor

      int len;
      int porta_servidor = 38023;
      switch (select(0, SEL_CONTROL))                /* Verificar o
estado da ligação */
      {
      case SOCK_ESTABLISHED:                                  /* Se a
ligação estiver estabelecida */

            if ((len = select(0, SEL_RECV)) > 0)                /*
verificar se recebeu dados */
            {
                  if (len > size) len = size;   /* se o tamanho dos
dados é maior que o espaço disponivel em size */

                  len = recv(0, posicao, len);  /* lê e armazena os
dados recebidos no endereço passado por posicao */

            }
            break;
      case SOCK_CLOSE_WAIT:   /* Caso o servidor peça para desligar */
            disconnect(0);
            len=0;
            break;
      case SOCK_CLOSED:                         /* se o socket não
estiver criado */
            if(socket(0,SOCK_STREAM,porta,0x00) == 1)   /* reiniciar o
socket TCP */
            {
                  connect(0, addr, porta_servidor);   //conecta com o
servidor
                  len =0;

            }
            break;
      }
      return len;
}
```

MP3_receber.h

```
#ifndef _mp3_receber_H
#define _mp3_receber_H


#include "types.h"

uint16 MP3_RECEBER(uint8 * addr,u_char xdata * posicao,uint16 porta,
uint16 size);
#endif
```

socket.c

```
/*
 * (c)COPYRIGHT
 * ALL RIGHT RESERVED
 *
 * FileName : socket.c
 * Revision History :
 * ----------  -------  -----------  ---------------------------------
------------
 * Date        version  Name         Description
 * ----------  -------  -----------  ---------------------------------
------------
 * 09/23/2005  1.0      Bong         Create version
 * ----------  -------  -----------  ---------------------------------
------------
 * 10/12/2005  2.0      Woo          Release version
 * ----------  -------  -----------  ---------------------------------
------------
 * 10/13/2005  2.0.1    Bong         modify sendto() function for UDP
defection
 * ----------  -------  -----------  ---------------------------------
------------
 * 10/25/2005  2.0.2    Bong         modify connect(),sendto()
function fix subnet bug.
 * ----------  -------  -----------  ---------------------------------
------------
 * 11/03/2005  2.0.3    Bong         modify send() function fix send
completion bug.
 * ----------  -------  -----------  ---------------------------------
------------
 * 11/11/2005  2.0.4    Bong         #define _20051111_
 *                                   modify send() function fix send
completion bug.
 * ----------  -------  -----------  ---------------------------------
------------
 */

#include "socket.h"

#include "delay.h" // for wait function

extern uint8 xdata GW_MAC[6]; // gateway mac address defined in
w3150a.c
extern uint8 xdata GW_IP[6]; // gateway mac address defined in
w3150a.c
extern uint8 xdata is_gw_samenet;
uint16 xdata local_port;

/**
```

```
* Internal Functions
*/

/**
 * \brief socket initialization
 * This function initializes a specified socket and waits until the
W3150 has done.
 * \param s socket number
 * \param protocol the protocol for the socket
 * \param port the source port for the socket
 * \param flag the option for the socket
 * \return When succeeded : 1, failed :0
*/
uint8 socket(SOCKET s, uint8 protocol, uint16 port, uint8 flag)
{
      uint8 ret;
      if ((protocol == SOCK_STREAM) || (protocol == SOCK_DGRAM) ||
(protocol == SOCK_ICMPM) || (protocol == SOCK_IPL_RAWM) || (protocol
== SOCK_MACL_RAWM) || (protocol == SOCK_PPPOEM))
      {
            ret = 1;
            if (IINCHIP_READ(SOCK_STATUS(s)) != SOCK_CLOSED) close(s);
            IINCHIP_WRITE(OPT_PROTOCOL(s),protocol | flag);
            if (port != 0) {
                  IINCHIP_WRITE(SRC_PORT_PTR(s),(uint8)((port & 0xff00)
>> 8));
                  IINCHIP_WRITE((SRC_PORT_PTR(s) + 1),(uint8)(port &
0x00ff));
            } else {
                  local_port++; //if don't set the source port, set
local_port number.
                  IINCHIP_WRITE(SRC_PORT_PTR(s),(uint8)((local_port &
0xff00) >> 8));
                  IINCHIP_WRITE((SRC_PORT_PTR(s) +
1),(uint8)(local_port & 0x00ff));
            }
            IINCHIP_WRITE(COMMAND(s),CSOCKINIT);
      }
      else
      {
            ret = 0;
      }
      return ret;
}

/**
 * \brief close socket
 * This function is to close the socket.
 * \param s socket number
*/
void close(SOCKET s)
{

      IINCHIP_WRITE(COMMAND(s),CCLOSE);

}

/**
 * \brief establish connection (active)
 * This function establishs a connection to the peer,
```

```
 * and wait until the connection is established successfully. (TCP
client mode)
 * \param s socket number
 * \param addr the peer's IP address
 * \param port the peer's port number
 * \return when succeeded : 1, failed : 0
*/
uint8 connect(SOCKET s, uint8 * addr, uint16 port)
{
      uint8 ret;
      if
            (
                  ((addr[0] == 0xFF) && (addr[1] == 0xFF) && (addr[2]
== 0xFF) && (addr[3] == 0xFF)) ||
                  ((addr[0] == 0x00) && (addr[1] == 0x00) && (addr[2]
== 0x00) && (addr[3] == 0x00)) ||
                  (port == 0x00)
            )
      {
            ret = 0;
      }
      else
      {
            ret = 1;
            // for UDP defection
            if (issubnet(addr) == 1)
            {
                  // 2005.10.25 added fix subnet check error
                  if (is_gw_samenet == 0)
                  {

                        IINCHIP_WRITE((GATEWAY_PTR + 0),addr[0]);
                        IINCHIP_WRITE((GATEWAY_PTR + 1),addr[1]);
                        IINCHIP_WRITE((GATEWAY_PTR + 2),addr[2]);
                        IINCHIP_WRITE((GATEWAY_PTR + 3),addr[3]);
                  }
            }

            // set destination IP
            IINCHIP_WRITE(DST_IP_PTR(s),addr[0]);
            IINCHIP_WRITE((DST_IP_PTR(s) + 1),addr[1]);
            IINCHIP_WRITE((DST_IP_PTR(s) + 2),addr[2]);
            IINCHIP_WRITE((DST_IP_PTR(s) + 3),addr[3]);
            IINCHIP_WRITE(DST_PORT_PTR(s),(uint8)((port & 0xff00) >>
8));
            IINCHIP_WRITE((DST_PORT_PTR(s) + 1),(uint8)(port &
0x00ff));
            IINCHIP_WRITE(COMMAND(s),CCONNECT);
            // wait for completion
            while (IINCHIP_READ(COMMAND(s)))
            {
                  if (IINCHIP_READ(SOCK_STATUS(s)) == SOCK_CLOSED)
                  {

                        ret = 0; break;
                  }
            }
            // 2005.10.25 added fix subnet check error
            if (is_gw_samenet == 0)
            {
```

```
                  IINCHIP_WRITE((GATEWAY_PTR + 0),GW_IP[0]);
                  IINCHIP_WRITE((GATEWAY_PTR + 1),GW_IP[1]);
                  IINCHIP_WRITE((GATEWAY_PTR + 2),GW_IP[2]);
                  IINCHIP_WRITE((GATEWAY_PTR + 3),GW_IP[3]);
            }
      }


      return ret;
}

/**
 * \brief close socket
 * This function is to close the connection of the socket.
 * \param s socket number
*/
void disconnect(SOCKET s)
{

      IINCHIP_WRITE(COMMAND(s),CDISCONNECT);

}
/**
 * \brief receive tcp data packet
 * This function is to receive TCP data.
 * The recv() function is an application I/F function.
 * It continues to waits for as much data as the application wants to
receive.
 * \param s socket number
 * \param buf a pointer to copy the data to be received
 * \param len the size of the data to read
 * \return Succeed: received data size, Failed:  -1
*/
uint16 recv(SOCKET s, uint8 * buf, uint16 len)
{
      uint16 ret=0;

      if ( len > 0 )
      {
            recv_data_processing(s, buf, len);
            IINCHIP_WRITE(COMMAND(s),CRECV);
            ret = len;
      }
      return ret;
}

/**
 * \brief send tcp data packet
 * This function sends TCP data.
 * \param s socket number
 * \param buf a pointer to data
 * \param len the data size to send
 * \return Succeed: sent data size, Failed:  0
*/
uint16 send(SOCKET s, const uint8 * buf, uint16 len)
{
      uint8 status=0;
      uint16 ret=0;
      uint16 freesize=0;
```

```
      if (len > getIINCHIP_TxMAX(s)) ret = getIINCHIP_TxMAX(s); //
check size not to exceed MAX size.
      else ret = len;

      // if freebuf is available, start.
      do
      {
            freesize = IINCHIP_READ(TX_FREE_SIZE_PTR(s));
            freesize = (freesize<<8) +
IINCHIP_READ(TX_FREE_SIZE_PTR(s)+1);
            status = IINCHIP_READ(SOCK_STATUS(s));
            if ((status != SOCK_ESTABLISHED) && (status !=
SOCK_CLOSE_WAIT)){ret = 0; break;}

      } while (freesize < ret);

      if (ret != 0)
      {
            // copy data
            send_data_processing(s, (uint8 *)buf, ret);
            IINCHIP_WRITE(COMMAND(s),CSEND);

            // wait for completion
            while (IINCHIP_READ(COMMAND(s)))
            {
                  status = IINCHIP_READ(SOCK_STATUS(s));
                  if (status == SOCK_CLOSED)
                  {

                  ret = 0; break;
                  }
            }


      }
      return ret;
}
```

socket.h

```
#ifndef     _SOCKET_H_
#define     _SOCKET_H_

#include "types.h"
#include "w3150a.h"

/***************************************************
*
* define function of socket API
*
***************************************************/
uint8 socket(SOCKET s, uint8 protocol, uint16 port, uint8 flag); //
Opens a socket(TCP or UDP or IP_RAW mode)
void close(SOCKET s); // Close socket
uint8 connect(SOCKET s, uint8 * addr, uint16 port); // Establish TCP
connection (Active connection)
void disconnect(SOCKET s); // disconnect the connection
uint16 recv(SOCKET s, uint8 * buf, uint16 len);// Receive data (TCP)
uint16 send(SOCKET s, const uint8 * buf, uint16 len);
#endif
```

w3150a.c

```
/*
 * (c)COPYRIGHT
 * ALL RIGHT RESERVED
 *
 * FileName : w3150a.c
 * Revision History :
 * ----------  -------  -----------  ---------------------------------
------------
 * Date        version  Name         Description
 * ----------  -------  -----------  ---------------------------------
------------
 * 09/23/2005  1.0      Bong         Create version
 * ----------  -------  -----------  ---------------------------------
------------
 * 10/12/2005  2.0      Woo          Release version
 * ----------  -------  -----------  ---------------------------------
------------
 * 10/13/2005  2.0.1    Bong         added function getting GW_IP,
GW_MAC for UDP defection
 * ----------  -------  -----------  ---------------------------------
------------
 * 10/25/2005  2.0.2    Bong         added function fix subnet bug.
 * ----------  -------  -----------  ---------------------------------
------------
 * 11/03/2005  2.0.3    Bong         modify issubnet() function.
 * ----------  -------  -----------  ---------------------------------
------------
 * 11/22/2005  2.0.4    Bong         #define _20051121_
 *                                   modify pppinit_in() function.
 *                                   support CHAP mode
 *                                   added md5.c, md5.h
 * ----------  -------  -----------  ---------------------------------
------------
 */

#include "w3150a.h"
#include "delay.h" // for wait function


uint8 xdata L_IP[4]; // local ip
uint8 xdata SUBNET[4]; // subnet
uint8 xdata GW_IP[4]; // gateway ip address
uint8 xdata GW_MAC[6]; // gateway mac address
uint8 xdata is_gw_samenet;
uint8 xdata I_STATUS[MAX_SOCK_NUM];
uint16 xdata SMASK[MAX_SOCK_NUM]; /* Variable for Tx buffer MASK in
each channel */
uint16 xdata RMASK[MAX_SOCK_NUM]; /* Variable for Rx buffer MASK in
each channel */
uint16 xdata SSIZE[MAX_SOCK_NUM]; /* Max Tx buffer size by each
channel */
uint16 xdata RSIZE[MAX_SOCK_NUM]; /* Max Rx buffer size by each
channel */
uint16 xdata SBUFBASEADDRESS[MAX_SOCK_NUM]; /* Tx buffer base address
by each channel */
uint16 xdata RBUFBASEADDRESS[MAX_SOCK_NUM]; /* Rx buffer base address
by each channel */
```

```
uint16 getIINCHIP_RxMAX(uint8 s)
{
   return RSIZE[s];
}
uint16 getIINCHIP_TxMAX(uint8 s)
{
   return SSIZE[s];
}
uint16 getIINCHIP_RxMASK(uint8 s)
{
   return RMASK[s];
}
uint16 getIINCHIP_TxMASK(uint8 s)
{
   return SMASK[s];
}
uint16 getIINCHIP_RxBASE(uint8 s)
{
   return RBUFBASEADDRESS[s];
}
uint16 getIINCHIP_TxBASE(uint8 s)
{
   return SBUFBASEADDRESS[s];
}

uint8 IINCHIP_WRITE(uint16 addr,uint8 val)
{
      *((vuint8 xdata *)(addr))=val;

      return 1;
}

uint8 IINCHIP_READ(uint16 addr)
{
      uint8 val;
      val = *((vuint8 xdata*)(addr));
      return val;
}

uint16 wiz_read_buf(uint16 addr, uint8* buf,uint16 len)
{


      memcpy(buf, (uint8 xdata*)addr, len);
      return len;
}

uint16 wiz_write_buf(uint16 addr,uint8* buf,uint16 len)
{

      memcpy((uint8 xdata*)addr, buf, len);
      return len;
}

/*
* Initializes the iinchip
* This function is for resetting of the iinchip.
*/
void iinchip_init(void)
{
      *((volatile uint8 xdata *)(TMODE)) = TMODE_SWRESET;
```

```
}

/**
 * Set the flexible memory
 * This function sets the Tx, Rx memory size by each channel
 * \param tx_size Tx memory size (00 - 1KByte, 01- 2KBtye, 10 -
4KByte, 11 - 8KByte)
 * \param rx_size Rx memory size (00 - 1KByte, 01- 2KBtye, 10 -
4KByte, 11 - 8KByte)
 * \note
 * bit 1-0 : memory size of channel #0 \n
 * bit 3-2 : memory size of channel #1 \n
 * bit 5-4 : memory size of channel #2 \n
 * bit 7-6 : memory size of channel #3 \n
 * Maximum memory size for Tx, Rx in the W3150 is 8K Bytes,
 * In the range of 8KBytes, the memory size could be allocated
dynamically by each channel.
 * Be attentive to sum of memory size shouldn't exceed 8Kbytes
 * and to data transmission and receiption from non-allocated channel
may cause some problems.
 * If the 8KBytes memory is already  assigned to centain channel,
 * other 3 channels couldn't be used, for there's no available memory.
 * If two 4KBytes memory are assigned to two each channels,
 * other 2 channels couldn't be used, for there's no available memory.
 *
 */
void sysinit(uint8 tx_size, uint8 rx_size)
{

      IINCHIP_WRITE(TX_DMEM_SIZE,tx_size); /* Set Tx memory size for
each channel */
      IINCHIP_WRITE(RX_DMEM_SIZE,rx_size);      /* Set Rx memory size
for each channel */

      SBUFBASEADDRESS[0] = (uint16)(__DEF_IINCHIP_MAP_TXBUF__);  /*
Set base address of Tx memory for channel #0 */
      RBUFBASEADDRESS[0] = (uint16)(__DEF_IINCHIP_MAP_RXBUF__);  /*
Set base address of Rx memory for channel #0 */



/* Set maximum memory size for Tx and Rx, mask, base address of memory
by each channel */

/* Set maximum Tx memory size */
            SSIZE[0] = (int16)(8192);
            SMASK[0] = (uint16)(0x1FFF);


/* Set maximum Rx memory size */
            RSIZE[0] = (int16)(8192);
            RMASK[0] = (uint32)(0x00001FFF);

   // for updating gw arp cache
      getGWMAC_processing();
      is_gw_samenet=0;
      is_gw_samenet = issubnet_gw();

}
```

```
/**
 * \brief Sets up Gateway IP address
 * This function sets up gateway IP address.
 * \param addr A pointer to a 4-byte array that will be filled in with
 * the Gateway IP address.
 */
void setgateway(uint8 * addr)
{
      GW_IP[0] = addr[0];
      GW_IP[1] = addr[1];
      GW_IP[2] = addr[2];
      GW_IP[3] = addr[3];
      IINCHIP_WRITE((GATEWAY_PTR + 0),addr[0]);
      IINCHIP_WRITE((GATEWAY_PTR + 1),addr[1]);
      IINCHIP_WRITE((GATEWAY_PTR + 2),addr[2]);
      IINCHIP_WRITE((GATEWAY_PTR + 3),addr[3]);
}

void getGWMAC_processing(void)
{
   uint8 i = 0;
   uint8 j = 0;
      do
      {
            IINCHIP_WRITE(OPT_PROTOCOL(0),SOCK_STREAM);
            IINCHIP_WRITE(COMMAND(0),CSOCKINIT);
            IINCHIP_WRITE(DST_IP_PTR(0),L_IP[0]+1);
            IINCHIP_WRITE(COMMAND(0),CCONNECT);
            wait_10ms();
            for (i = 0; i < 6; i++)GW_MAC[i] =
IINCHIP_READ(DST_HA_PTR(0)+i);
            IINCHIP_WRITE(COMMAND(0),CCLOSE);
            IINCHIP_WRITE(DST_IP_PTR(0),0x00);



   } while ((IINCHIP_READ(DST_HA_PTR(0)) == 0xff) && j++ <3);
}
/**
 * \brief Sets up subnet mask
 * This function sets subnet mask.
 * \param addr A pointer to a 4-byte array that will be filled in with
 * the subnet mask.
 */
void setsubmask(uint8 * addr)
{
      SUBNET[0] = addr[0];
      SUBNET[1] = addr[1];
      SUBNET[2] = addr[2];
      SUBNET[3] = addr[3];
      IINCHIP_WRITE((SUBNET_MASK_PTR + 0),addr[0]);
      IINCHIP_WRITE((SUBNET_MASK_PTR + 1),addr[1]);
      IINCHIP_WRITE((SUBNET_MASK_PTR + 2),addr[2]);
      IINCHIP_WRITE((SUBNET_MASK_PTR + 3),addr[3]);
}
/*
* return
* 0 => different subnet
* 1 => same subnet
*/
uint8 issubnet(uint8 *addr)
```

```
{
   if (
      (   ((addr[0] & SUBNET[0]) == (L_IP[0] & SUBNET[0]))
          && ((addr[1] & SUBNET[1]) == (L_IP[1] & SUBNET[1]))
          && ((addr[2] & SUBNET[2]) == (L_IP[2] & SUBNET[2]))
          && ((addr[3] & SUBNET[3]) == (L_IP[3] & SUBNET[3]))
      ) ||
      ( (addr[0] == 0xFF) && (addr[1] == 0xFF) && (addr[2] == 0xFF) &&
(addr[3] == 0xFF) )
   )
   return 1;
   else return 0;
}

// 2005.10.25 added fix subnet check error
/*
* return
* 0 => different subnet
* 1 => same subnet
*/
uint8 issubnet_gw(void)
{
   if (   ((L_IP[0] & SUBNET[0]) == (GW_IP[0] & SUBNET[0]))
       && ((L_IP[1] & SUBNET[1]) == (GW_IP[1] & SUBNET[1]))
       && ((L_IP[2] & SUBNET[2]) == (GW_IP[2] & SUBNET[2]))
       && ((L_IP[3] & SUBNET[3]) == (GW_IP[3] & SUBNET[3]))
   ) return 1;
   else return 0;
}
/**
 * \brief Sets up source MAC address
 * This function sets up the source MAC Address.
 * \param addr A pointer to a 6-byte array that will be filled in with
 * the host MAC address.
 */
void setMACAddr(uint8 * addr)
{
      IINCHIP_WRITE((SRC_HA_PTR + 0),addr[0]);
      IINCHIP_WRITE((SRC_HA_PTR + 1),addr[1]);
      IINCHIP_WRITE((SRC_HA_PTR + 2),addr[2]);
      IINCHIP_WRITE((SRC_HA_PTR + 3),addr[3]);
      IINCHIP_WRITE((SRC_HA_PTR + 4),addr[4]);
      IINCHIP_WRITE((SRC_HA_PTR + 5),addr[5]);
}
/**
 * \brief Sets up source IP address
 * This function sets up the source IP Address
 * \param addr A pointer to a 4-byte array that will be filled in with
 * the host IP address.
 */
void setIP(uint8 * addr)
{
      L_IP[0] = addr[0];
      L_IP[1] = addr[1];
      L_IP[2] = addr[2];
      L_IP[3] = addr[3];
      IINCHIP_WRITE((SRC_IP_PTR + 0),addr[0]);
      IINCHIP_WRITE((SRC_IP_PTR + 1),addr[1]);
      IINCHIP_WRITE((SRC_IP_PTR + 2),addr[2]);
      IINCHIP_WRITE((SRC_IP_PTR + 3),addr[3]);
```

```
}

/**
 * This function returns socket information.
 * socket status or Tx free buffer size or received data size
 * func
 * SEL_CONTROL(0x00) -> return socket status
 * SEL_SEND(0x01)    -> return Tx free buffer size
 * SEL_RECV(0x02)    -> return received data size
 */
uint16 select(SOCKET s, uint8 func)
{
      uint16 xdata val=0;
      switch (func)
      {
      case SEL_CONTROL :
            val = (uint16)IINCHIP_READ(SOCK_STATUS(s));
            break;
      case SEL_SEND :
            val = IINCHIP_READ(TX_FREE_SIZE_PTR(s));
            val = (val << 8) + IINCHIP_READ(TX_FREE_SIZE_PTR(s) + 1);
            break;
      case SEL_RECV :
            val = IINCHIP_READ(RX_RECV_SIZE_PTR(s));
            val = (val << 8) + IINCHIP_READ(RX_RECV_SIZE_PTR(s) + 1);
            break;
      default :
            val = 0;
            break;
      }

      return val;
}

/*
* data, pointer processing
*/
void send_data_processing(SOCKET s, uint8 *val, uint16 len)
{
      uint16 ptr;
      ptr = IINCHIP_READ(TX_WR_PTR(s));
      ptr = ((ptr & 0x00ff) << 8) + IINCHIP_READ(TX_WR_PTR(s) + 1);
      write_data(s, val, (uint8 *)(ptr), len);
      ptr += len;
      IINCHIP_WRITE(TX_WR_PTR(s),(uint8)((ptr & 0xff00) >> 8));
      IINCHIP_WRITE((TX_WR_PTR(s) + 1),(uint8)(ptr & 0x00ff));
}

/*
* data, pointer processing
*/
void recv_data_processing(SOCKET s, uint8 *val, uint16 len)
{
      uint16 ptr;
      ptr = IINCHIP_READ(RX_RD_PTR(s));
      ptr = ((ptr & 0x00ff) << 8) + IINCHIP_READ(RX_RD_PTR(s) + 1);

      read_data(s, (uint8*)ptr, val, len); // read data
      ptr += len;
      IINCHIP_WRITE(RX_RD_PTR(s),(uint8)((ptr & 0xff00) >> 8));
      IINCHIP_WRITE((RX_RD_PTR(s) + 1),(uint8)(ptr & 0x00ff));
```

```
}

/*
* write data from src to dst
*/
void write_data(SOCKET s, vuint8 * src, vuint8 * dst, uint16 len)
{
      uint16 size;
      uint16 dst_mask;
      uint8 * xdata dst_ptr;

      dst_mask = (uint16)dst & getIINCHIP_TxMASK(s);
      dst_ptr = (uint8 xdata *)(getIINCHIP_TxBASE(s) + dst_mask);

      if (dst_mask + len > getIINCHIP_TxMAX(s))
      {
            size = getIINCHIP_TxMAX(s) - dst_mask;
            wiz_write_buf((uint16)dst_ptr, (uint8*)src, size);
            src += size;
            size = len - size;
            dst_ptr = (uint8 xdata *)(getIINCHIP_TxBASE(s));
            wiz_write_buf((uint16)dst_ptr, (uint8*)src, size);
      }
      else
      {
            wiz_write_buf((uint16)dst_ptr, (uint8*)src, len);
      }
}

/*
* read data from src to dst
*/
void read_data(SOCKET s, vuint8 * src, vuint8 * dst, uint16 len)
{
      uint16 size;
      uint16 src_mask;
      uint8 * xdata src_ptr;

      src_mask = (uint16)src & getIINCHIP_RxMASK(s);
      src_ptr = (uint8 xdata *)(getIINCHIP_RxBASE(s) + src_mask);

      if( (src_mask + len) > getIINCHIP_RxMAX(s) )
      {
            size = getIINCHIP_RxMAX(s) - src_mask;
            wiz_read_buf((uint16)src_ptr, (uint8*)dst,size);
            dst += size;
            size = len - size;
            src_ptr = (uint8 xdata *)(getIINCHIP_RxBASE(s));
            wiz_read_buf((uint16)src_ptr, (uint8*) dst,size);
      }
      else
      {
            wiz_read_buf((uint16)src_ptr, (uint8*) dst,len);
      }
}
```

w3150a.h

```
#ifndef     _W3150A_H_
#define     _W3150A_H_

#include "iinchip_conf.h"
#include "types.h"

#define TMODE __DEF_IINCHIP_MAP_BASE__
#define IDM_OR ((__DEF_IINCHIP_MAP_BASE__ + 0x00))
#define IDM_AR0 ((__DEF_IINCHIP_MAP_BASE__ + 0x01))
#define IDM_AR1 ((__DEF_IINCHIP_MAP_BASE__ + 0x02))
#define IDM_DR ((__DEF_IINCHIP_MAP_BASE__ + 0x03))

/*
* Maxmium number of socket
*/
#define     MAX_SOCK_NUM            1

/**
 * \brief Gateway IP Register address
 */
#define GATEWAY_PTR                         (COMMON_BASE + 0x0001)
/**
 * \brief Subnet mask Register address
 */
#define SUBNET_MASK_PTR             (COMMON_BASE + 0x0005)
/**
 * \brief Source MAC Register address
 */
#define SRC_HA_PTR                          (COMMON_BASE + 0x0009)
/**
 * \brief Source IP Register address
 */
#define SRC_IP_PTR                          (COMMON_BASE + 0x000F)
/**
 * \brief Interrupt Register
 */
#define INT_REG                             (COMMON_BASE + 0x0015)
/**
 * \brief Interrupt mask register
 */
#define INTMASK                             (COMMON_BASE + 0x0016)
/**
 * \brief Timeout register address
 *
 * 1 is 100us
 */
#define TIMEOUT_PTR                         (COMMON_BASE + 0x0017)
/**
 * \brief Retry count reigster
 */
#define RCR                                 (COMMON_BASE + 0x0019)
/**
 * \brief Receive memory size reigster
 */
#define RX_DMEM_SIZE                (COMMON_BASE + 0x001A)
/**
 * \brief Transmit memory size reigster
 */
#define TX_DMEM_SIZE                (COMMON_BASE + 0x001B)
```

```
/**
 * \brief Authentication type register address in PPPoE mode
 */
#define PPPAUTH                             (COMMON_BASE + 0x001C)
#define PPPALGO                             (COMMON_BASE + 0x001D)
#define PPP_TIMEOUT                         (COMMON_BASE + 0x0028)
#define PPP_MAGIC                   (COMMON_BASE + 0x0029)
/**
 * \brief Unreachable IP register address in UDP mode
 */
#define UNREACH_IP                          (COMMON_BASE + 0x002A)
/**
 * \brief Unreachable Port register address in UDP mode
 */
#define UNREACH_PORT                (COMMON_BASE + 0x002E)

/* socket register */
#define CH_BASE (COMMON_BASE + 0x0400)
/**
 * size of each channel register map
 */
#define CH_SIZE         0x0100
/**
 * \brief socket option register
 */
#define OPT_PROTOCOL(ch)            (CH_BASE + ch * CH_SIZE + 0x0000)
/**
 * \brief channel command register
 */
#define COMMAND(ch)                         (CH_BASE + ch * CH_SIZE +
0x0001)
/**
 * \brief channel interrupt register
 */
#define INT_STATUS(ch)              (CH_BASE + ch * CH_SIZE + 0x0002)
/**
 * \brief channel status register
 */
#define SOCK_STATUS(ch)             (CH_BASE + ch * CH_SIZE + 0x0003)
/**
 * \brief source port register
 */
#define SRC_PORT_PTR(ch)            (CH_BASE + ch * CH_SIZE + 0x0004)
/**
 * \brief Peer MAC register address
 */
#define DST_HA_PTR(ch)              (CH_BASE + ch * CH_SIZE + 0x0006)
/**
 * \brief Peer IP register address
 */
#define DST_IP_PTR(ch)              (CH_BASE + ch * CH_SIZE + 0x000C)
/**
 * \brief Peer port register address
 */
#define DST_PORT_PTR(ch)            (CH_BASE + ch * CH_SIZE + 0x0010)
/**
 * \brief Maximum Segment Size(MSS) register address
 */
#define MSS(ch)                             (CH_BASE + ch * CH_SIZE +
0x0012)
/**
```

```
 * \brief Protocol of IP Header field register in IP raw mode
 */
#define IP_PROTOCOL(ch)             (CH_BASE + ch * CH_SIZE + 0x0014)

/* \brief IP Type of Service(TOS) Register
 */
#define IP_TOS(ch)                                      (CH_BASE + ch *
CH_SIZE + 0x0015)
/**
 * \brief IP Time to live(TTL) Register
 */
#define IP_TTL(ch)                                      (CH_BASE + ch *
CH_SIZE + 0x0016)

/**
 * \brief Transmit free memory size register
 */
#define TX_FREE_SIZE_PTR(ch)  (CH_BASE + ch * CH_SIZE + 0x0020)
/**
 * \brief Transmit memory read pointer register address
 */
#define TX_RD_PTR(ch)               (CH_BASE + ch * CH_SIZE + 0x0022)
/**
 * \brief Transmit memory write pointer register address
 */
#define TX_WR_PTR(ch)               (CH_BASE + ch * CH_SIZE + 0x0024)
/**
 * \brief Received data size register
 */
#define RX_RECV_SIZE_PTR(ch)  (CH_BASE + ch * CH_SIZE + 0x0026)
/**
 * \brief Read point of Receive memory
 */
#define RX_RD_PTR(ch)               (CH_BASE + ch * CH_SIZE + 0x0028)
/**
 * \brief Write point of Receive memory
 */
#define RX_WR_PTR(ch)               (CH_BASE + ch * CH_SIZE + 0x002A)
/** @} */

/* TMODE register values */
#define TMODE_INDIRECT              0x01
#define TMODE_AUTOINC               0x02
#define TMODE_LITTLEENDIAN          0x04
#define TMODE_PPPOE                       0x08
#define TMODE_PINGBLOCK             0x10
#define TMODE_MEMTEST               0x20
#define TMODE_STATIC_ISN            0x40
#define TMODE_SWRESET               0x80

/* INT_REG register values */
#define INT_CH(ch)                        (0x01 << ch)
#define INT_PPPTERM                       0x20
#define INT_UNREACH                       0x40
#define INT_IPCONFLICT              0x80

/* OPT_PROTOCOL values */
#define     SOCK_CLOSEDM                  0x00        // unused socket
#define     SOCK_STREAM                   0x01        // TCP
#define     SOCK_DGRAM                    0x02        // UDP
#define     SOCK_ICMPM                    0x03        // icmp
```

```
#define     SOCK_IPL_RAWM           0x03        // IP LAYER RAW SOCK
#define     SOCK_MACL_RAWM          0x04        // MAC LAYER RAW SOCK
#define     SOCK_PPPOEM             0x05        // PPPoE
#define SOCKOPT_ZEROCHKSUM          0x10
#define SOCKOPT_NDACK               0x20        // No Delayed Ack(TCP)
flag
#define SOCKOPT_MULTI               0x80        // support multicating

/* COMMAND values */
#define CSOCKINIT             0x01        // initialize or open socket
#define CLISTEN               0x02        // wait connection request in
tcp mode(Server mode)
#define CCONNECT              0x04        // send connection request in
tcp mode(Client mode)
#define CDISCONNECT           0x08        // send closing reqeuset in
tcp mode
#define CCLOSE                0x10        // close socket
#define CSEND           0x20        // updata txbuf pointer, send data
#define CSENDMAC        0x21        // send data with MAC address
#define CSENDKEEPALIVE        0x22        // send keep alive message
#define CRECV           0x40        // update rxbuf pointer, recv data


/* INT_STATUS values */
#define ISR_CON               0x01        // established connection
#define ISR_DISCON            0x02        // closed socket
#define ISR_RECV              0x04        // receiving data
#define ISR_TIMEOUT           0x08        // assert timeout



/* SOCK_STATUS values */
#define SOCK_CLOSED                 0x00        // closed
#define SOCK_INIT                   0x13        // init state
#define SOCK_LISTEN                 0x14        // listen state
#define SOCK_SYNSENT                0x15        // connection state
#define SOCK_SYNRECV                0x16        // connection state
#define SOCK_ESTABLISHED            0x17        // success to connect
#define SOCK_FIN_WAIT1              0x18        // closing state
#define SOCK_FIN_WAIT2              0x19        // closing state
#define SOCK_CLOSING                0x1A        // closing state
#define SOCK_TIME_WAIT              0x1B        // closing state
#define SOCK_CLOSE_WAIT             0x1C        // closing state
#define SOCK_LAST_ACK               0x1D        // closing state
#define SOCK_UDP                    0x22        // udp socket
#define SOCK_IPL_RAW                0x32        // ip raw mode socket
#define SOCK_MACL_RAW               0x42        // mac raw mode socket
#define SOCK_PPPOE                  0x5F        // pppoe socket

/* IP PROTOCOL */
#define IPPROTO_IP        0           /* Dummy for IP */
#define IPPROTO_ICMP      1           /* Control message protocol */
#define IPPROTO_IGMP      2           /* Internet group management
protocol */
#define IPPROTO_GGP       3           /* Gateway^2 (deprecated) */
#define IPPROTO_TCP       6           /* TCP */
#define IPPROTO_PUP       12          /* PUP */
#define IPPROTO_UDP       17          /* UDP */
#define IPPROTO_IDP       22          /* XNS idp */
#define IPPROTO_ND        77          /* UNOFFICIAL net disk protocol
*/
```

```
#define IPPROTO_RAW         255         /* Raw IP packet */

/*****************************************************
* iinchip access function
*****************************************************/
uint8 IINCHIP_READ(uint16 addr);
uint8 IINCHIP_WRITE(uint16 addr,uint8 val);
uint16 wiz_read_buf(uint16 addr, uint8* buf,uint16 len);
uint16 wiz_write_buf(uint16 addr,uint8* buf,uint16 len);

void iinchip_init(void); // reset iinchip
void sysinit(uint8 tx_size, uint8 rx_size); // setting tx/rx buf size

uint16 getIINCHIP_RxMAX(uint8 s);
uint16 getIINCHIP_TxMAX(uint8 s);
uint16 getIINCHIP_RxMASK(uint8 s);
uint16 getIINCHIP_TxMASK(uint8 s);
uint16 getIINCHIP_RxBASE(uint8 s);
uint16 getIINCHIP_TxBASE(uint8 s);
void setgateway(uint8 * addr); // set gateway address
void setsubmask(uint8 * addr); // set subnet mask address
uint8 issubnet(uint8 *addr);
uint8 issubnet_gw(void);
void setMACAddr(uint8 * addr); // set local MAC address
void setIP(uint8 * addr); // set local IP address
void getGWMAC_processing(void);
void getDestPort(SOCKET s, uint8 * addr);
uint16 select(SOCKET s, uint8 func); // Get socket status/Tx free
buffer size/ Rx buffer size
void send_data_processing(SOCKET s, uint8 *val, uint16 len);
void recv_data_processing(SOCKET s, uint8 *val, uint16 len);
void read_data(SOCKET s, vuint8 * src, vuint8 * dst, uint16 len);
void write_data(SOCKET s, vuint8 * src, vuint8 * dst, uint16 len);

/* select func value */
#define SEL_CONTROL 0 // socket status
#define SEL_SEND 1 // free size in tx buf
#define SEL_RECV 2 // receving data size in rx buf

#endif
```

iinchip_conf.h

```
#ifndef     _IINCHIP_CONF_H_
#define     _IINCHIP_CONF_H_



#define __DEF_IINCHIP_DIRECT_MODE__ 1
#define __DEF_IINCHIP_INDIRECT_MODE__ 2
#define __DEF_IINCHIP_BUS__ __DEF_IINCHIP_DIRECT_MODE__
//#define __DEF_IINCHIP_BUS__ __DEF_IINCHIP_INDIRECT_MODE__

/**
* __DEF_IINCHIP_MAP_xxx__ : define memory map for iinchip
*/
#define __DEF_IINCHIP_MAP_BASE__ 0x8000
#if (__DEF_IINCHIP_BUS__ == __DEF_IINCHIP_DIRECT_MODE__)
 #define COMMON_BASE __DEF_IINCHIP_MAP_BASE__
#else
```

```
 #define COMMON_BASE 0x0000
#endif
#define __DEF_IINCHIP_MAP_TXBUF__ (COMMON_BASE + 0x4000) /* Internal
Tx buffer address of the iinchip */
#define __DEF_IINCHIP_MAP_RXBUF__ (COMMON_BASE + 0x6000) /* Internal
Rx buffer address of the iinchip */

/**
* __DEF_MCU_xxx__ : define option related to MCU
*/
#define __DEF_MCU_Dallas__

#ifdef __DEF_MCU_Dallas__
   #include "reg420.h"
   #include <stdio.h>
   #include <string.h>

   #ifdef __DEF_IINCHIP_INT__
      // iinchip use external interrupt 4
      #define IINCHIP_ISR_DISABLE()(EX0 = 0)
      #define IINCHIP_ISR_ENABLE()  (EX0 = 1)
      #define IINCHIP_ISR_GET(X)    (X = EX0)
      #define IINCHIP_ISR_SET(X)    (EX0 = X)
   #else
      #define IINCHIP_ISR_DISABLE()
      #define IINCHIP_ISR_ENABLE()
      #define IINCHIP_ISR_GET(X)
      #define IINCHIP_ISR_SET(X)
   #endif
#endif

#endif
```

types.h

```
#ifndef _TYPE_H_
#define _TYPE_H_

typedef code      code_area;

#ifndef NULL
#define NULL            ((void *) 0)
#endif

typedef enum { false, true } bool;

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

/**
 * The 8-bit signed data type.
 */
typedef char int8;
/**
 * The volatile 8-bit signed data type.
 */
typedef volatile char vint8;
```

```
/**
 * The 8-bit unsigned data type.
 */
typedef unsigned char uint8;
/**
 * The volatile 8-bit unsigned data type.
 */
typedef volatile unsigned char vuint8;

/**
 * The 16-bit signed data type.
 */
typedef int int16;
/**
 * The volatile 16-bit signed data type.
 */
typedef volatile int vint16;
/**
 * The 16-bit unsigned data type.
 */
typedef unsigned int uint16;
/**
 * The volatile 16-bit unsigned data type.
 */
typedef volatile unsigned int vuint16;
/**
 * The 32-bit signed data type.
 */
typedef long int32;
/**
 * The volatile 32-bit signed data type.
 */
typedef volatile long vint32;
/**
 * The 32-bit unsigned data type.
 */
typedef unsigned long uint32;
/**
 * The volatile 32-bit unsigned data type.
 */
typedef volatile unsigned long vuint32;

/* bsd */
typedef uint8                u_char;           /* 8-bit value */
typedef uint8                SOCKET;
typedef uint16               u_short;    /* 16-bit value */
typedef uint16               u_int;            /* 16-bit value */
typedef uint32               u_long;           /* 32-bit value */

typedef union _un_l2cval {
      u_long      lVal;
      u_char      cVal[4];
}un_l2cval;

typedef union _un_i2cval {
      u_int iVal;
      u_char      cVal[2];
}un_i2cval;


#endif            /* _TYPE_H_ */
```

delay.c

```
#include "delay.h"
#include <reg420.h>

void init_timer()       //configurar o timer 0 no modo 1
{
      TMOD |= 0x01;
      TH0=0xBE;               // configurar o timer para 10ms com um
cristal de 20MHz
      TL0=0xE6;
/*
      TH0=0xB0;               // configurar o timer para 10ms com um
cristal de 24,576MHz
      TL0=0x00;

      TH0=0xDC;               // configurar o timer para 10ms com um
cristal de 11,0592MHz
      TL0=0x00;*/
}

//delay de 10ms
void wait_10ms()
{
TR0=1;
while(!TF0);
TF0=0;
TR0=0;
}
```

delay.h

```
#ifndef _DELAY_H
#define _DELAY_H

#include "types.h"

void init_timer();

void wait_10ms();

#endif
```

STA013-I2C.c

```
#include<STA013-I2C.h>

//pequeno delay para o I2C
void I2C_delay(void)
{
      unsigned char i;
      for(i=0; i<I2C_DELAY; i++);
}

//Clock do I2C
void I2C_clock(void)
{
```

```
	I2C_delay();
	SCL = 1;
	I2C_delay();
	SCL = 0;
}

//Sinal de start
void I2C_start(void)
{
	SDA = 1;
	SCL = 1;
	I2C_delay();
	SDA = 0;
	I2C_delay();
	SCL = 0;
	I2C_delay();
}

//sinal de Stop
void I2C_stop(void)
{
	if(SCL)
	SCL = 0;
	SDA = 0;
	I2C_delay();
	SCL = 1;
	I2C_delay();
	SDA = 1;
I2C_delay();
}

//escrita de um byte
bit I2C_write(unsigned char dat)
{
	bit data_bit;
	unsigned char i;

	for(i=0;i<8;i++)
	{
		data_bit = dat & 0x80;
		SDA = data_bit;
			I2C_delay();
			SCL = 1;
			I2C_delay();
			SCL = 0;
			I2C_delay();
			dat = dat<<1;
	}

	SDA = 1;
	I2C_delay();
	SCL = 1;
	I2C_delay();
	data_bit = SDA;
	SCL = 0;
	I2C_delay();
	return data_bit;
}

//leitura de 1 byte
unsigned char I2C_read(void)
```

```
{
	bit rd_bit;
	unsigned char i, dat;
	SDA=1;
	dat = 0x00;

	for(i=0;i<8;i++)
	{
		I2C_delay();
		SCL = 1;
		I2C_delay();
		rd_bit = SDA;
		dat = dat<<1;
		dat = dat | rd_bit;
		I2C_delay();
		SCL = 0;
	}

	return dat;
}

//Acknowledgment
void I2C_ack()
{
	SDA = 0;
	I2C_delay();
	I2C_clock();
	SDA = 1;
}

//not Acknowledgment
void I2C_noack()
{
	SDA = 1;
	I2C_delay();
	I2C_clock();
	SCL = 1;
}

// função para escrita de um byte num determinado endereço do STA013
void STA_WRITE(uint8 endereco,uint8 dado)
{
I2C_start();
I2C_write(STA013_W);
I2C_write(endereco);
I2C_write(dado);
I2C_stop();
}

// função para leitura de um byte num determinado endereço do STA013
int STA_READ(uint8 endereco)
{
int dado;
I2C_start();
I2C_write(STA013_W);
I2C_write(endereco);
I2C_start();
I2C_write(STA013_R);
dado=I2C_read();
I2C_stop();
return dado;
```

```
}

// configuração do PCM Divider
void PCM_Divider()
{
STA_WRITE(PCMDIVIDER,0x03);
}

// configuração do PCM config
void PCM_CONFIG()
{
STA_WRITE(PCMCONF,0x27);
}
/*
  ###  Configuração da PLL  ###
//#### Para um cristal de 14.318MHz oversampling de 512
*/
void SET_PLL()
{
STA_WRITE(PLLCTL_N,0);
STA_WRITE(PLLCTL_M,11);
STA_WRITE(11,3);

STA_WRITE(MFSDF_(X),6);
STA_WRITE(MFSDF_441,7);
STA_WRITE(PLLFRAC_441_L,157);
STA_WRITE(PLLFRAC_441_H,157);
STA_WRITE(PLLFRAC_L,211);
STA_WRITE(PLLFRAC_H,3);

STA_WRITE(PLLCTL,161);

}
//Configurar a polarazição do sinal de relogio para envio do dado
void SET_SCLK_POL()
{
STA_WRITE(SCKL_POL,4);
}

//Configurar o modo de envio de dados
void SET_DATA_REQUEST()
{
STA_WRITE(DATA_REQ_ENABLE,4);

}

//configurar a polarização de envio de dados
void SET_REQ_POL()
{
STA_WRITE(REQ_POL,1);

}

//configuração inicial para funcionamento do STA013
void INIT_STA013()
{
      PCM_Divider();
      PCM_CONFIG();
      SET_PLL();
      SET_SCLK_POL();
      SET_DATA_REQUEST();
```

```
      SET_REQ_POL();
}

//activar o modo RUN
void STA_RUN(uint8 dado)
{
      STA_WRITE(RUN,dado);
}

//activar o modo PLAY
void STA_PLAY(uint8 dado)
{
      STA_WRITE(PLAY,dado);
}

//activar o modo MUTE
void STA_MUTE(uint8 dado)
{
      STA_WRITE(MUTE,dado);
}

//função de envio de um byte de uma forma serie sincrono
void funcaoSTA013(uint8 dado)
{
int rotacao;
for(rotacao=7;rotacao>=0;rotacao--)
{
      SCLK = 0;
    SDI = (dado >> rotacao) & 1;
      SCLK = 1;
}
}
```

STA013-I2C.h

```
#ifndef __I2C_H__
#define __I2C_H__

#include "types.h"
#include<reg420.h>

sbit SDA= P3^7;
sbit SCL= P3^6;

sbit RST_STA013=P3^5;

sbit SDI=P3^4;
sbit SCLK=P3^3;
sbit DATA_REQ = P3^2;

#define I2C_DELAY 0x0F

void I2C_delay(void);
void I2C_clock(void);
void I2C_start(void);
void I2C_stop(void);
bit I2C_write(unsigned char dat);
unsigned char I2C_read(void);
void I2C_ack();
void I2C_noack();
```

```
#define STA013_W  0x86  //  Endereço do STA013, endereço de Escrita
#define STA013_R  0x87  //  Endereço do STA013, endereço de Leitura

//############  DEFENIR OS ENDEREÇOS DOS REGISTOS DE CONFIGURAÇÃO
##############

#define VERSION   0x00
#define IDENT     0x01
#define PLLCTL    0x05
#define PLLCTL_M  0x06
#define PLLCTL_N  0x07
#define REQ_POL   0x0C
#define SCKL_POL  0x0D
#define ERROR_CODE      0x0F
#define SOFT_RESET      0x10
#define PLAY      0x13
#define MUTE      0x14
#define CMD_INTERRUPT   0x16
#define DATA_REQ_ENABLE 0x18
#define SYNCSTATUS 0x40
#define DLA       0x46
#define DLB       0x47
#define DRA       0x48
#define DRB       0x49
#define MFSDF_441 0x50
#define PLLFRAC_441_L   0x51
#define PLLFRAC_441_H   0x52
#define PCMDIVIDER      0x54
#define PCMCONF   0x55
#define PCMCROSS 0x56
#define MFSDF_(X) 0x61
#define DAC_CLK_MODE    0x63
#define PLLFRAC_L 0x64
#define PLLFRAC_H 0x65
#define RUN       0x72
#define TREBLE_FREQUENCY_LOW  0x77
#define TREBLE_FREQUENCY_HIGH 0x78
#define BASS_FREQUENCY_LOW    0x79
#define BASS_FREQUENCY_HIGH   0x7A

//##########  FUNÇOES STA013   ############


void STA_WRITE(uint8 endereco,uint8 dado);
int STA_READ(uint8 endereco);
void PCM_Divider();
void PCM_CONFIG();
void SET_PLL();
void SET_SCLK_POL();
void SET_DATA_REQUEST();
void SET_REQ_POL();
void INIT_STA013();
void STA_RUN(uint8 dado);
void STA_PLAY(uint8 dado);
void STA_MUTE(uint8 dado);

void funcaoSTA013(uint8 dado);

#endif
```

main.c

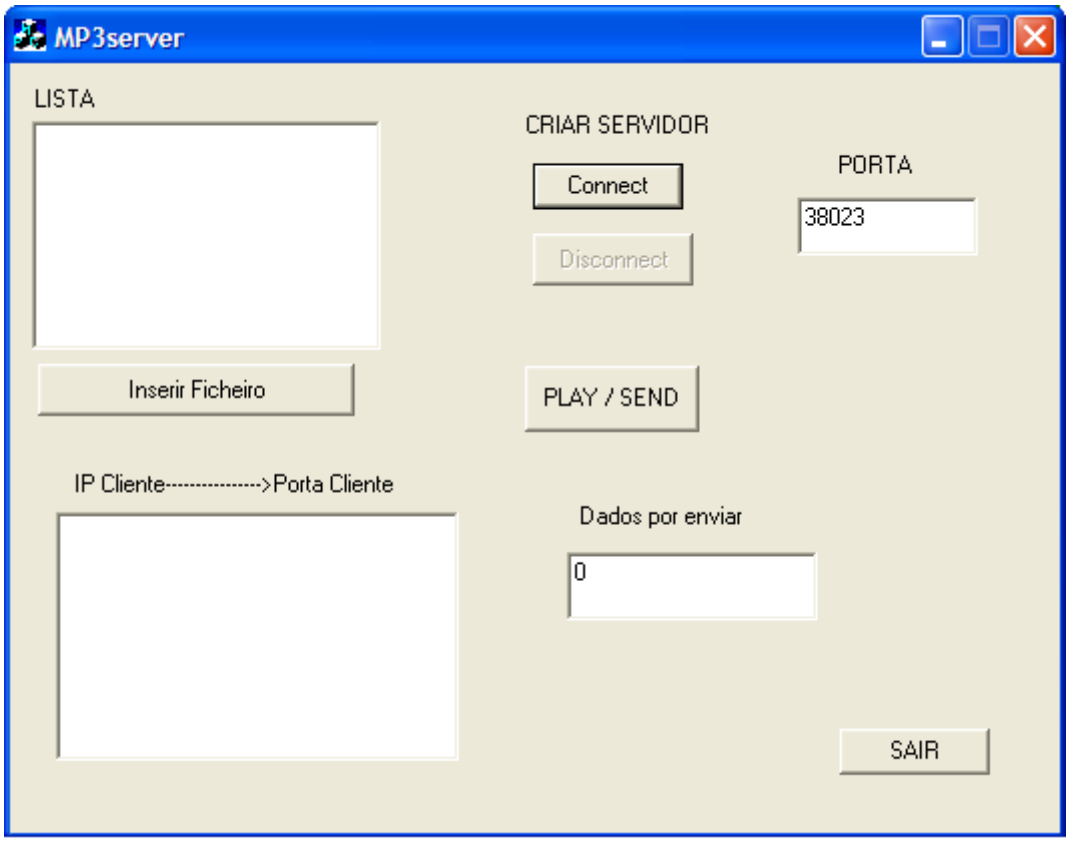
```
#include <stdio.h>
#include "reg420.h"
#include "types.h"
#include "socket.h"
#include "mp3_receber.h"
#include "STA013-I2C.h"
#include "delay.h"


sbit led= P1^7;
void mcu_init(void)
{
      EA = 0;           // Desabilitar todas a interrupçoes
      init_timer();  // configurar o timer
}

void net_init(void)
{

      uint8 addr[4] = {192,168,0,1};           //Gateway Ligação ponto
a ponto
//    uint8 addr[4] = {193,136,12,254};   //Gateway da UM
      iinchip_init();
      setMACAddr("\x00\x08\xDC\x00\x00\x00");
      setgateway(addr); // Configurar o endereço IP do Gateway
      addr[3] = 185;
      setIP(addr);            // Configurar o endereço IP do Hardware
      addr[0]=addr[1]=addr[2]=255;
      addr[3] = 0;
      setsubmask(addr); // Configurar o endereço da mascara de rede
255.255.255.0
      sysinit(0xFF,0xFF); //defenir RX e TX memória em 8K
}

int main(void)
{
int tamanho=0;
u_char enviar[4]={'S','E','N','D'}; //palavra de pedido de dados
long sizefile,auxfile,variavel;
unsigned char Tfile[10];
int i,j;
uint8 dado;
int contador;
int bufersize;
int limite;
int memoposicao;
int apontador;


//uint8 addr1[4] = {193,136,12,171};  // endereço do servidor ---> UM
uint8 addr1[4] = {192,168,0,1};  // endereço do servidor ---> CASA
(ligação ponto a ponto)

      RST_STA013=0;
      mcu_init();

for(i=0;i<300;i++)           //criar um delay inicial 30s
{ wait_10ms();}
```

```
      RST_STA013=1;
      INIT_STA013();    // configurar o STA013

      net_init();       //configurar a NM7010B para a rede


      STA_RUN(1);       //colocar o STA013 no modo Run
      STA_PLAY(1);      // apto a reproduzir

      while (1)
      {
            bufersize=17000;
            contador=0;
            memoposicao = 4096;
            apontador = 4096;

//espera pelo inicio da transmissão de uma música ou som
            while(tamanho==0)
            {     tamanho = MP3_RECEBER(addr1,Tfile, 5000,100);  }


            sizefile=0;
            for(i=0;i<tamanho;i++)
            {
                  auxfile= (char)Tfile[i] - 48;
                  variavel=1;
                  for(j=0;j<(tamanho-i-1);j++)
                  {
                        variavel=variavel*10;
                  }

            sizefile=sizefile + (auxfile * variavel); //adquire o
tamanho do ficheiro de som
            }
            tamanho=0;

//enchimento do buffer
            while(contador<bufersize)
            {     send(0,enviar,4); //envia pedido de dados
                  tamanho = MP3_RECEBER(addr1,(uint8 xdata*)
memoposicao, 5000,8192);
                  contador=contador + tamanho;
                  sizefile=sizefile - tamanho;
                  if((memoposicao = memoposicao + tamanho) > 24500)
                  {
                        limite=memoposicao;
                        memoposicao=4096;
                  }
                  tamanho=0;
            }

            {
            while(contador>0)
                  {
                        while(DATA_REQ)
                        {
                              dado = *((uint8 xdata*)apontador);
//leitura do dado do buffer
                              funcaoSTA013(dado); //envia dado para o
conversor
                              contador--;
```

```
                              apontador++;
                              if(apontador>=limite)
                              {
                                    apontador=4096;
                              }
                        }

                        if(sizefile>0)
                        {
                              if(contador<(bufersize-8192))
                              {send(0,enviar,4); //pede mais dados ao
servidor
                              while(tamanho==0)//espera para receber os
dados e coloca-os nas posiçoes livres do buffer
                                    {tamanho = MP3_RECEBER(addr1,(uint8
xdata*) memoposicao, 5000,8192);}
                                    sizefile=sizefile - tamanho;
                                    contador=contador + tamanho;
                                    if((memoposicao = memoposicao +
tamanho) > 24500)
                                    {
                                          limite=memoposicao;
                                          memoposicao=4096;
                                    }
                                    tamanho=0;
                              }
                        }
                  }
            }
            }
      return 0;
}
```

ANEXO C – Principais funções usadas no servidor

Imagem do Servidor



Função Connect (liga o servidor, e espera por uma conexão, e aceita a ligação)

```
void CMP3serverDlg::OnConnect()
{
	// TODO: Add your control notification handler code here

	UpdateData(FALSE);

	socket.Create(m_Porta); //criar um socket

	if (OnListen()==FALSE)
	{
		AfxMessageBox("Impossivel usar estar porta, por favor escolha
outra porta");
		OnClose();
		return;
	}

	OnAccept(); //vai para a função aceitar

	UpdateData(FALSE);
	GetDlgItem(ID_Connect)->EnableWindow(FALSE);
	GetDlgItem(IDC_Disconnect)->EnableWindow(TRUE);
```

```
	}
```

Função Listen (fica à escuta de uma ligação)

```
BOOL CMP3serverDlg::OnListen()
{

	if ( socket.Listen()==TRUE)
	{
		 AfxMessageBox("entrou no listen");
		 return TRUE;
	}
	else
	{
		return FALSE;
	}
}
```

Função Accept (aceita uma ligação)

```
void CMP3serverDlg::OnAccept()
{
	CString strIP;
	UINT port;
	if(socket.Accept(socketConect)==TRUE) //foi aceite a ligação?
	{
		socketConect.GetPeerName(strIP,port);
		TCHAR ligacoes[256];
		wsprintf(ligacoes, "ON--> %s ------> %d", strIP,port);

		m_IpPortaControl.AddString(ligacoes); //mostra a ligação
estabelecida com o cliente
		UpdateData(FALSE);
		UpdateData(TRUE);
	}
	else
	{
		AfxMessageBox("impossivel aceitar a ligação");
	}
}
```

Função PLAY/SEND (transmissão de dados)

```
void CMP3serverDlg::OnPlaySend()
{
	// TODO: Add your control notification handler code here
	int tamanhofile;
	int tamanhoenviar;
	int tamanhoporenviar;
	int enviado;
	int recebido;
	BYTE * pedido = NULL;
	BYTE * bufer = NULL;
	TCHAR recebe[20];
	TCHAR sizefile[10];

	UpdateData(TRUE);
	//abrir o ficheiro no modo leitura
	if(fileselect.Open(directorio,CFile::modeRead)==TRUE)
	{
		AfxMessageBox("Iniciar nova reprodução");
		tamanhofile=fileselect.GetLength();
		m_tamanho=tamanhofile;
		UpdateData(FALSE);
		wsprintf(sizefile,"%d",tamanhofile);
```

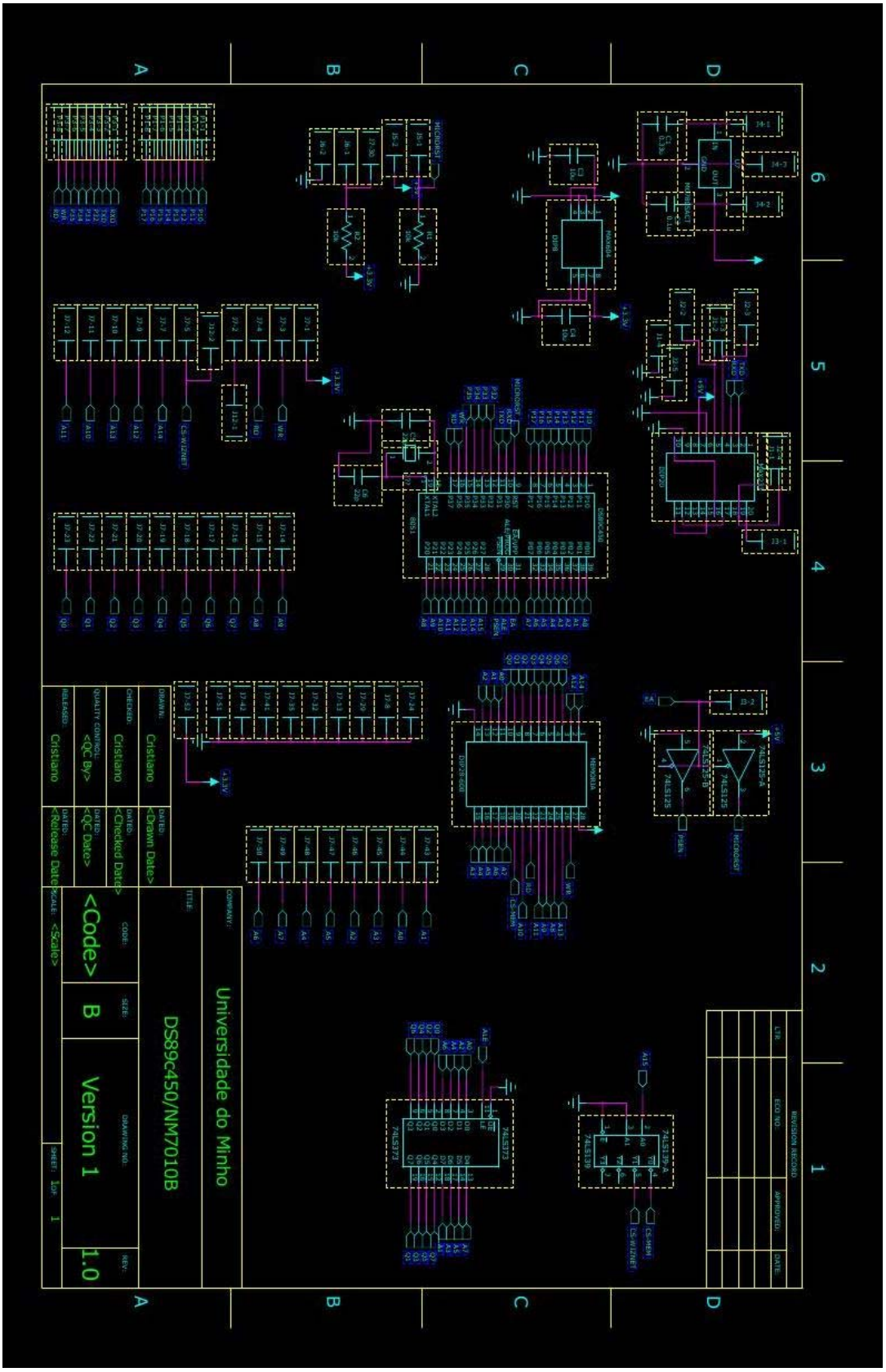
```
		AfxMessageBox(sizefile);
		//envia o tamanho do ficheiro
		socketConect.Send(sizefile,strlen(sizefile));

		tamanhoporenviar=tamanhofile;

			bufer = new BYTE[SEND_BUFFER_SIZE];
		do
		{
			//lê do ficheiro para um buffer os dados
			tamanhoenviar=fileselect.Read(bufer,SEND_BUFFER_SIZE);
			//espera que o cliente peça mais dados
			recebido=socketConect.Receive(recebe,20);
			recebe[recebido] = 0;

			//envia os dados do buffer para o cliente
			enviado=socketConect.Send(bufer,tamanhoenviar);

			//verifica se foi cancelado o envio
			if (enviado == (-1))
			{
				AfxMessageBox("falha no envio do dado");
				tamanhoporenviar = 0;
			}
			else
			{
				tamanhoporenviar = tamanhoporenviar - tamanhoenviar;
				m_tamanho=tamanhoporenviar;
			}

			UpdateData(FALSE);

		} while (tamanhoporenviar>0);

	AfxMessageBox("fim do envio do ficheiro");

	}
	else AfxMessageBox("erro a iniciar o ficheiro");
	fileselect.Close();
}
```

Função OnFile (abrir o ficheiro MP3)

```
void CMP3serverDlg::OnFile()
{
	// TODO: Add your control notification handler code here

	CFileDialog
m_file(TRUE,NULL,NULL,OFN_HIDEREADONLY|OFN_FILEMUSTEXIST,"MP3
Files(*.mp3)|*.mp3||",this);

	if (m_file.DoModal()== TRUE)
	{
		m_ControlList.AddString(m_file.GetFileName());
		directorio=m_file.GetPathName();
	}
	else AfxMessageBox("Erro ao abrir o ficheiro",MB_OK);

}
```

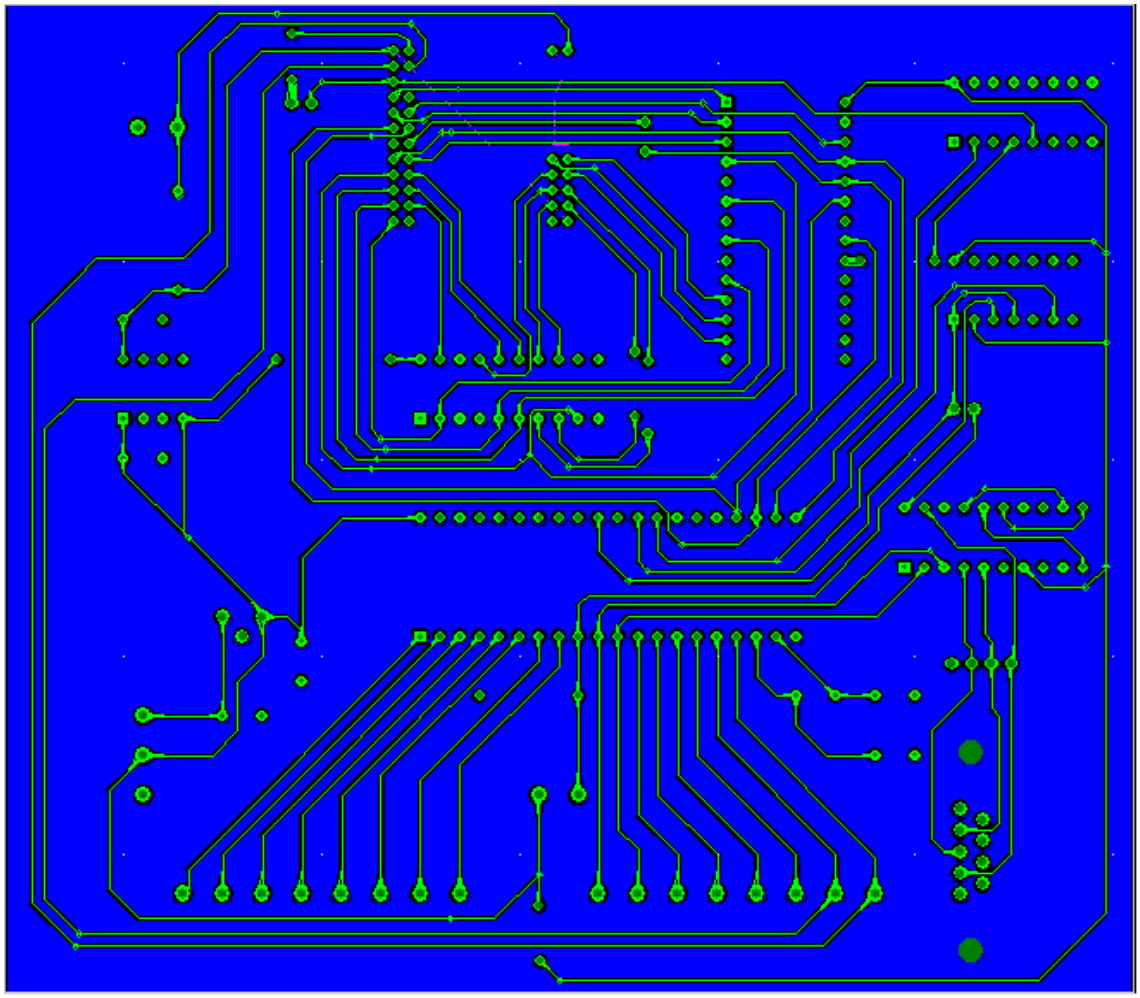
ANEXO D – Esquemáticos e desenhos dos PCBs

Esquema de ligações do 1º PCB criado



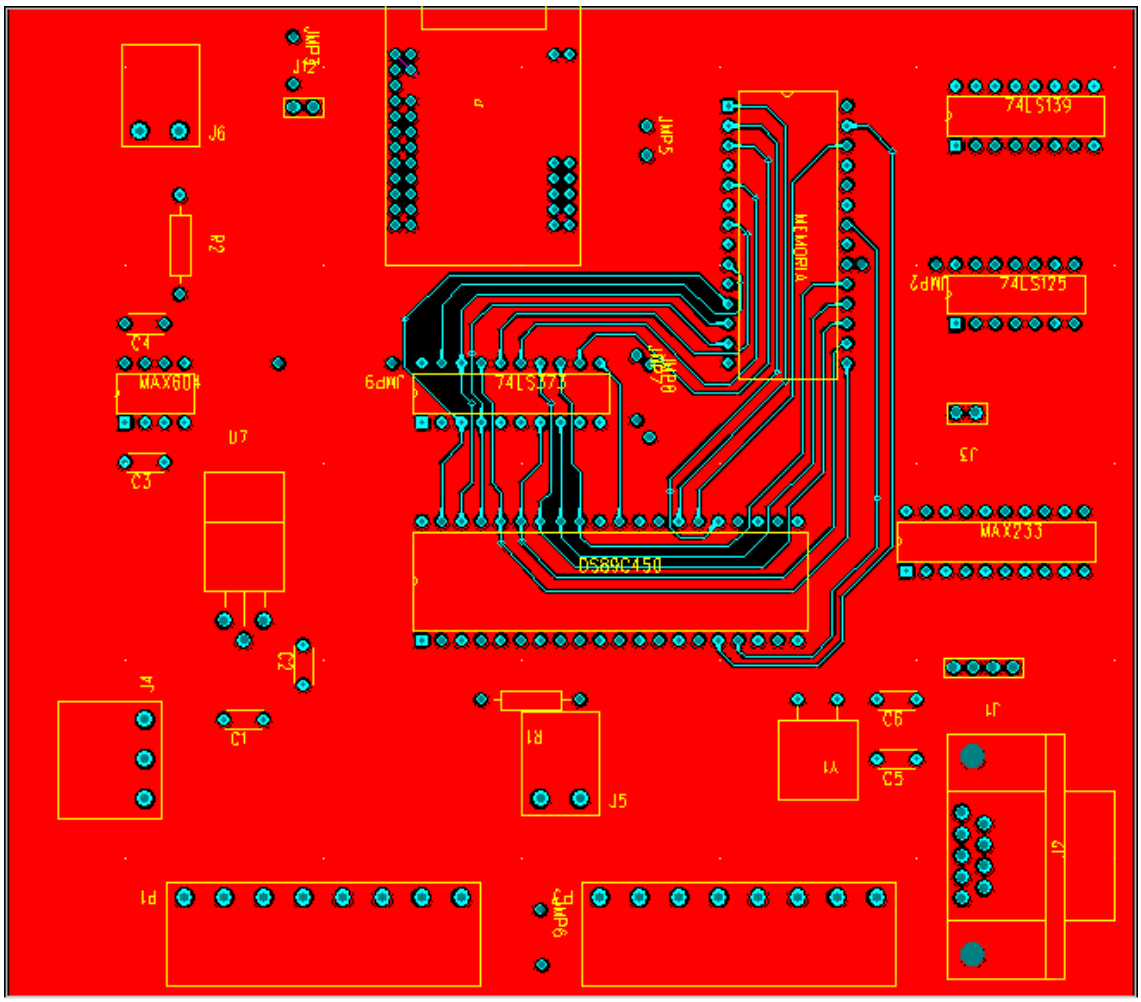
Imagens retiradas do Pads do 1º PCB criado

Face de baixo

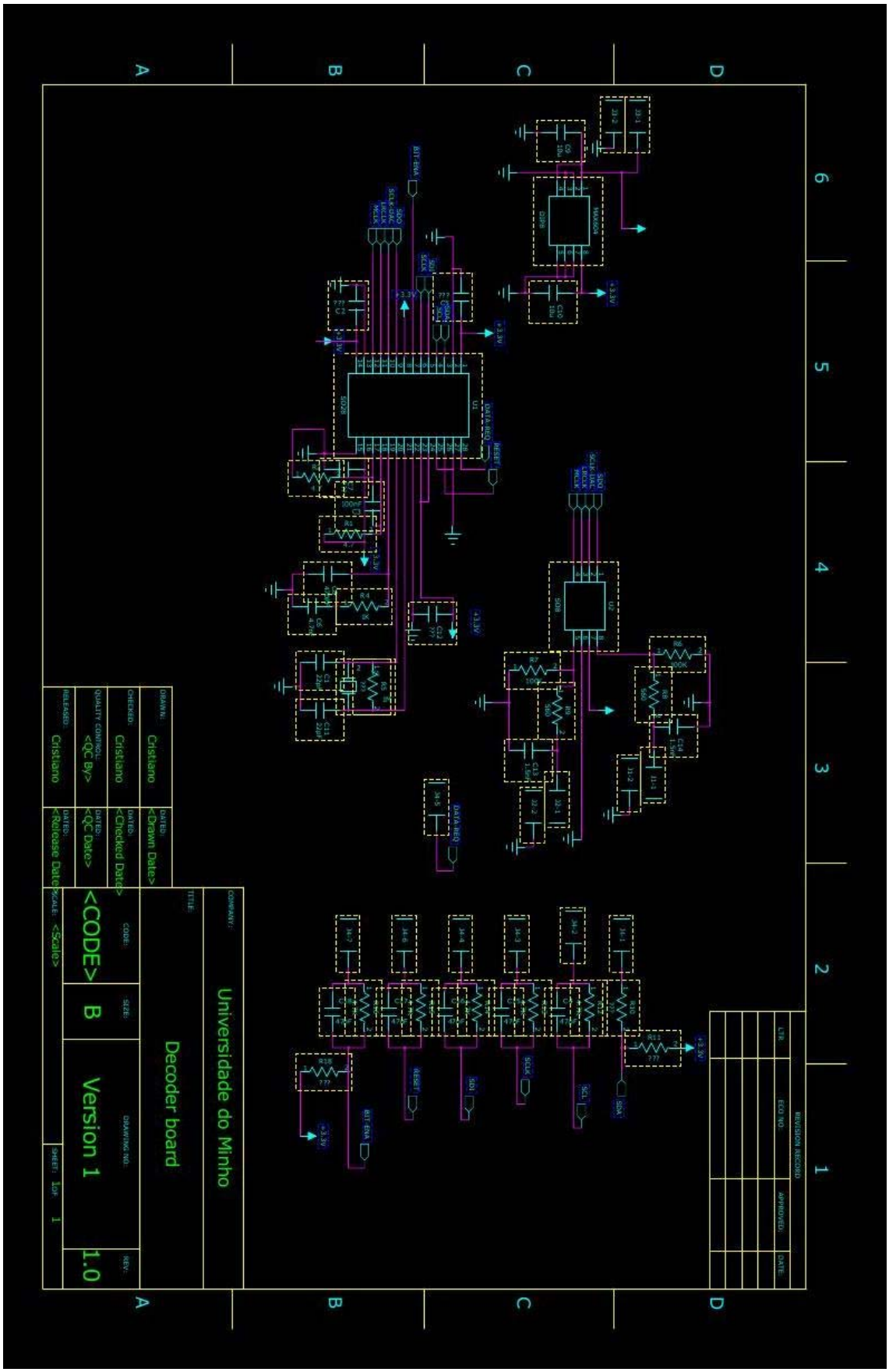


L

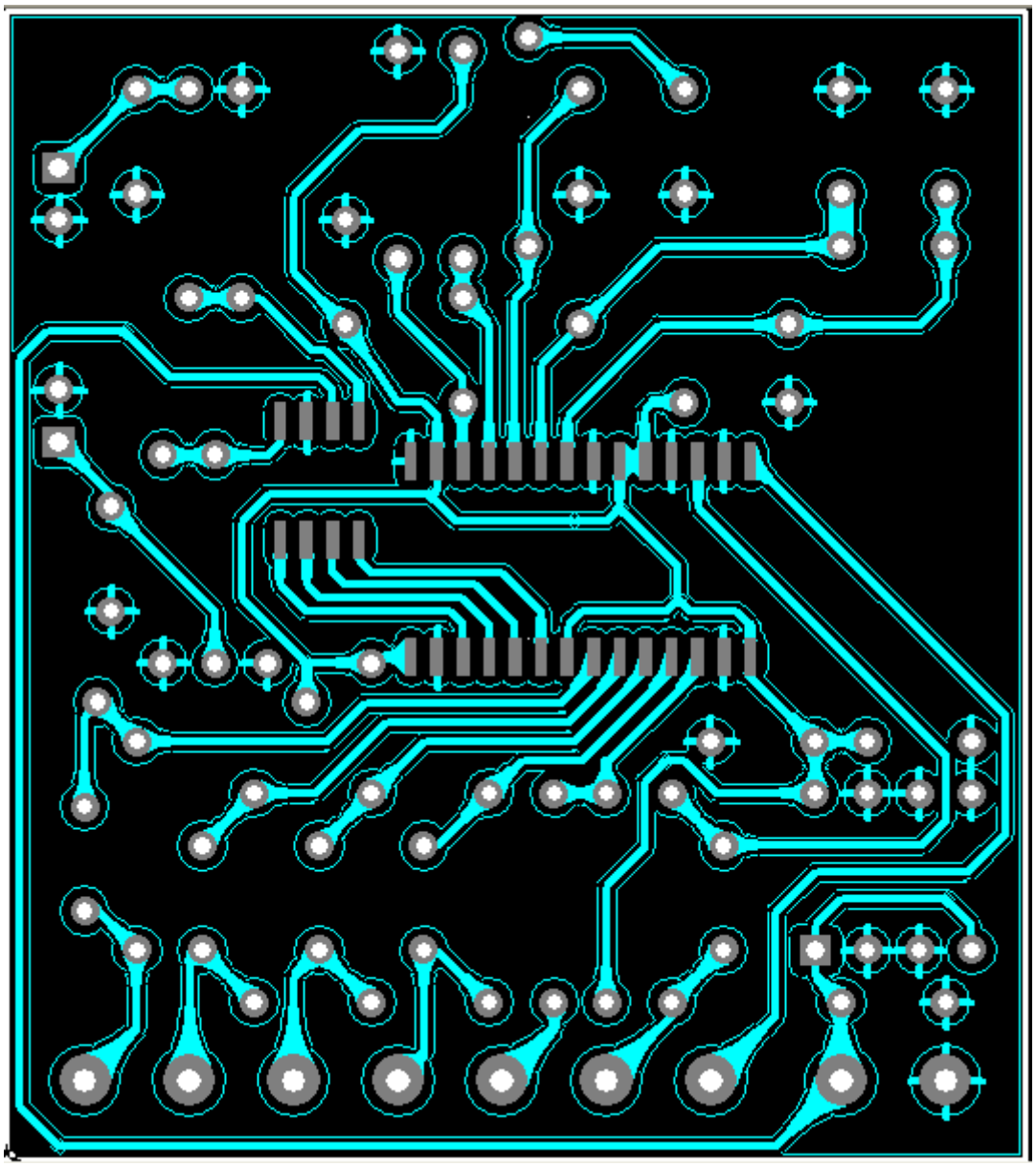
Face de cima



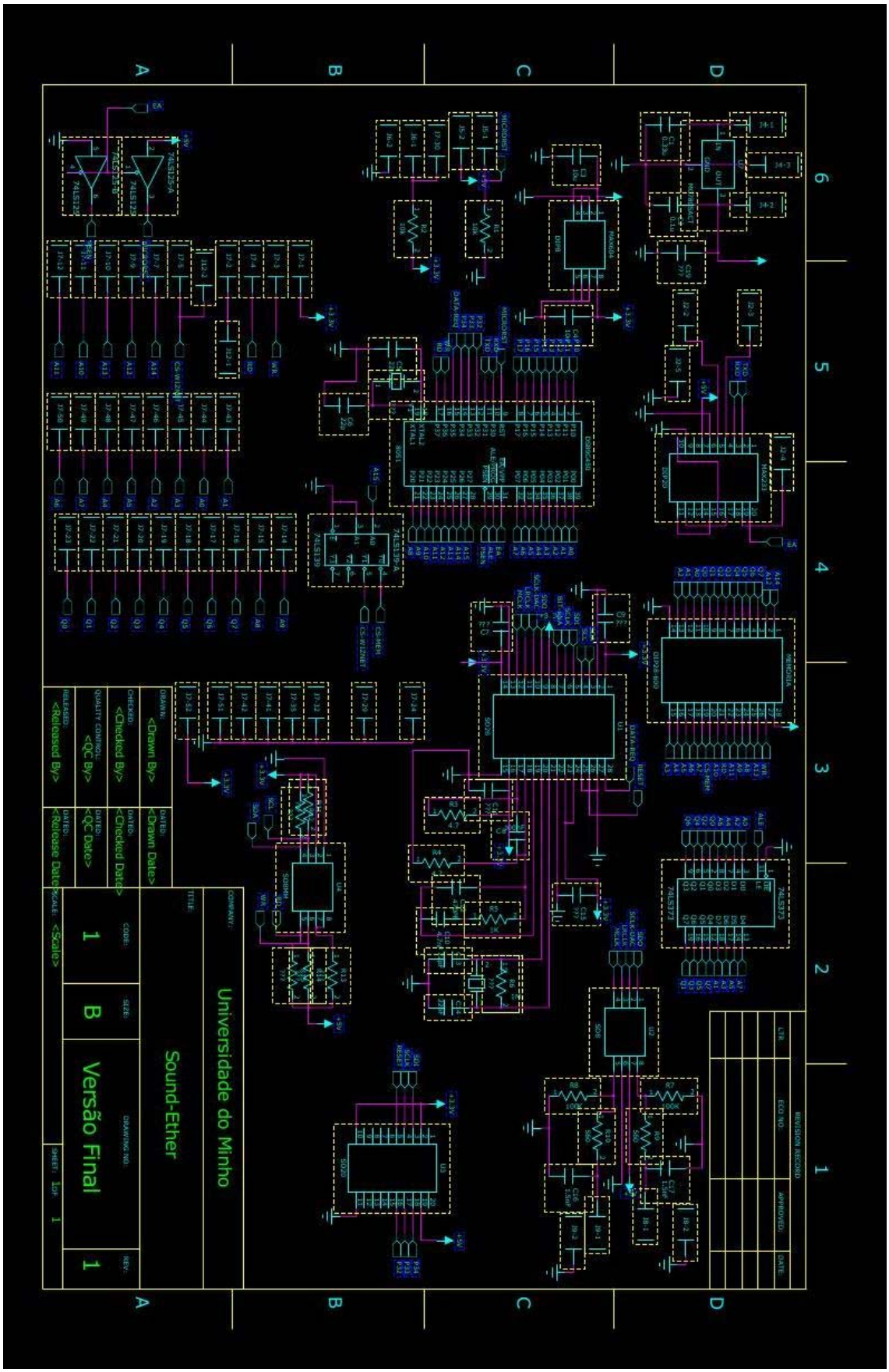
Esquema de ligações do 2º PCB criado



Imagens retiradas do Pads do 2º PCB criado

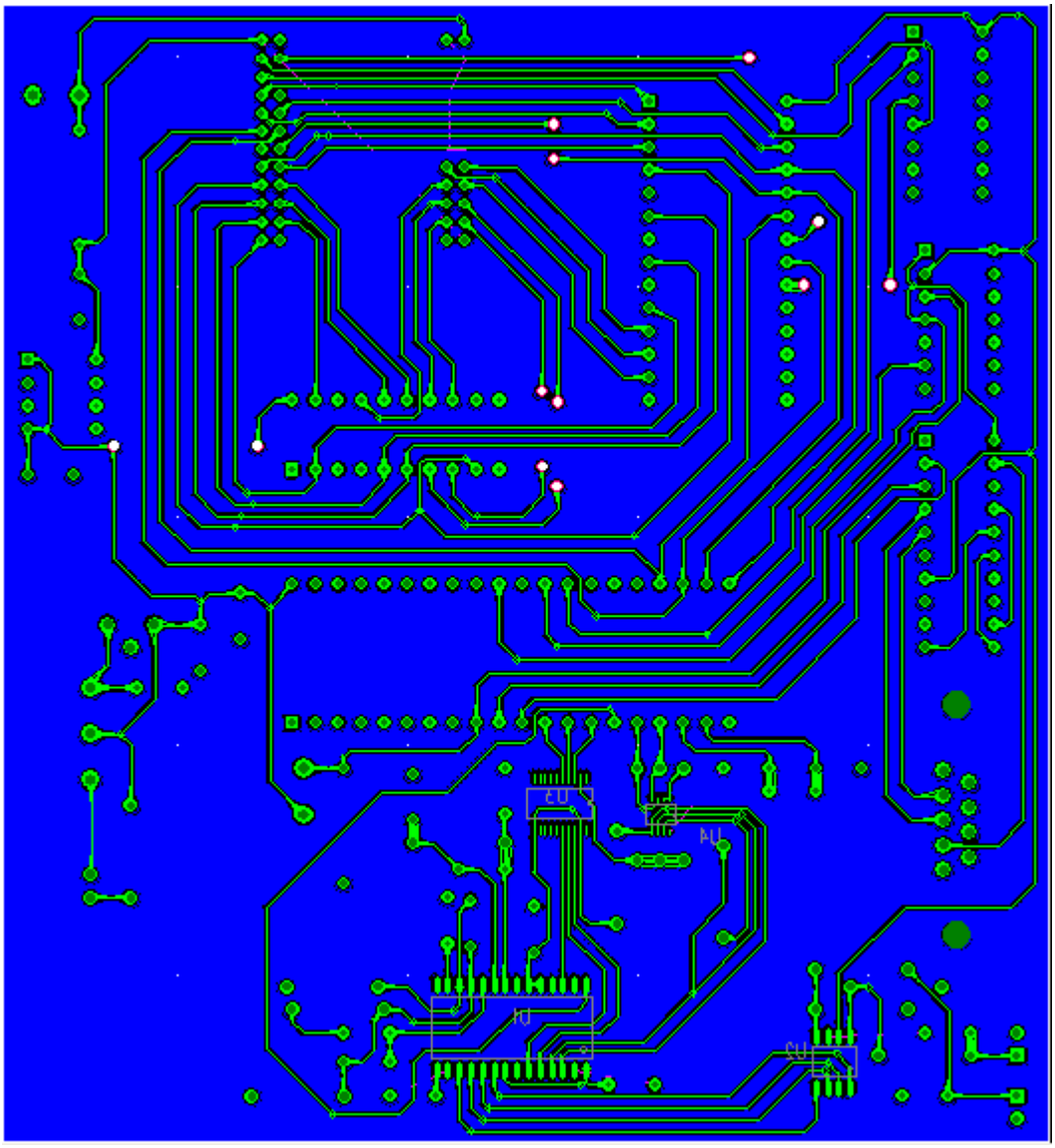


Esquema de ligações do PCB final

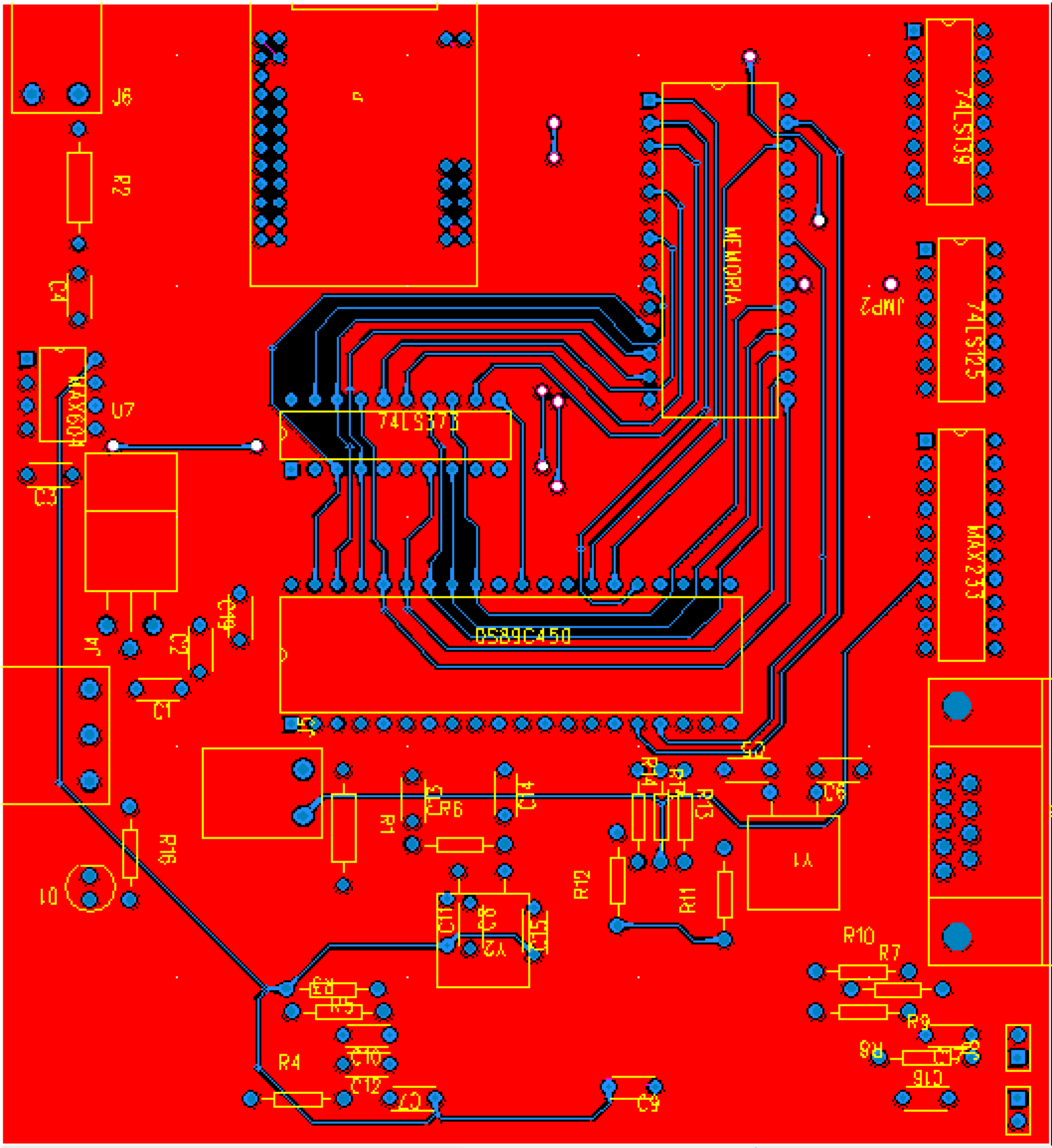


Imagens do PCB final criado

Face de baixo



Face de cima


J6
R2
C4
MAX608
U7
C3
J4
C2
C1
C16
74LS373
DS89C450
MEMORIA
JMP2
74LS139
74LS125
MAX233
R1
R6
C5
C14
R16
R14
R13
R12
R11
Y1
C6
C11
Y2
C15
R3
R5
R4
C10
C12
C7
C9
R10
R7
R9
R8
C16