Universidade de Aveiro
Departamento de
Eletrónica, Telecomunicações e Informática
2013

Paulo Sérgio Nunes Lopes

Interface Homem-Máquina para aplicações de automação residencial

Universidade de Aveiro
Departamento de
Eletrónica, Telecomunicações e Informática
2013

Paulo Sérgio Nunes Lopes

Interface Homem-Máquina para aplicações de automação residencial

Paulo Sérgio Nunes Lopes

Interface Homem-Máquina para aplicações de automação residencial

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Eletrónica e Telecomunicações, realizada sob orientação do Professor Doutor Alexandre Manuel Moutela Nunes da Mota, Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Mestre Rui Miguel de Bernardes Rebelo, Analista de Sistemas na Micro I/O Serviços de Eletrónica Lda.

o júri / the jury

presidente / president

Professor Doutor Paulo Bacelar Reis Pedreiras
Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Joaquim José de Castro Ferreira
Professor Adjunto da Escola Superior de Tecnologia e Gestão de Águeda, Universidade de Aveiro (arguente principal)

Professor Doutor Alexandre Manuel Moutela Nunes da Mota
Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Aos meus pais, pela educação e ensinamentos transmitidos durante toda a minha vida. Sem eles esta etapa não teria sido cumprida.

Ao Professor Doutor Alexandre Mota (orientador), pela oportunidade que me deu em realizar um trabalho na minha área preferencial.

Ao Mestre Rui Rebelo (co-orientador), pelo apoio técnico disponibilizado na realização da vertente prática desta Dissertação. Aproveito também para agradecer ao Mestre Paulo Bartolomeu pelas orientações iniciais.

Aos meus avós, tios e primos todo o apoio e carinho demonstrado.

A todos aqueles que me acompanharam no dia-a-dia, aqueles a quem posso chamar de amigos.

Resumo

A automação em ambientes residenciais é hoje uma realidade emergente, tendo crescido nos últimos anos em função da rápida evolução tecnológica. Existe atualmente uma enorme diversificação na oferta de soluções nesta área, inovadoras e de alto desempenho, que integram o sistema de controlo com uma interface gráfica. Algumas destas soluções apenas se centram na interface com o utilizador, oferecendo elevada qualidade gráfica, assim como uma panóplia de interfaces de comunicação para quase todo o tipo de aplicações. No entanto, muitas destas soluções são proprietárias e pouco flexíveis na adaptação a novos sistemas que possam vir a ser desenvolvidos, além do seu elevado custo. Diante este cenário, o trabalho desenvolvido nesta Dissertação contempla o desenvolvimento de uma *Interface Homem-Máquina* com ecrã tátil, versátil e flexível, de baixo custo, capaz de ser utilizada em diferentes cenários da automação residencial.

Como primeira abordagem, foi estudada uma plataforma de desenvolvimento (*FriendlyArm Tiny6410*) com requisitos idênticos aos pretendidos no trabalho desta Dissertação. Depois desta fase, foi desenvolvida toda a componente de *hardware* e de *software* da *Interface Homem-Máquina* e efetuados os respetivos testes.

Foram também desenvolvidas interfaces gráficas de utilização simples e amigável, que permitem o controlo e monitorização local de um sistema simulado.

Abstract

The *home automation* is an emerging reality today, having evolved on the past few years due to the growing technological evolution. There is currently a huge diversification in offering solutions in this area, very innovative and high performance, which integrate the control system with graphical interface. Some of these solutions focus only on the user interface by providing high quality graphics as well as a plethora of communication interfaces for almost all kinds of applications. However, many of these solutions are proprietary and inflexible in adapting to new systems that can be developed, in addition to its high cost. Given this scenario, the work developed on this dissertation includes the development of a *Human-Machine Interface* with *touchscreen*, versatile and flexible, low cost, capable of being used in different scenarios of home automation.

As a first approach, we studied a development platform (*FriendlyArm Tiny6410*) with identical requirements as required on this dissertation work. After this, all the hardware and software components of the *Human-Machine Interface* have been developed and made the respective tests.

We also developed a simple and user friendly graphical interface that allow the control and monitoring of a simulation system.

Conteúdo

Acrónimos

APO	*Application Object*
APS	*Application Support Sublayer*
BPSK	*Binary Phase Shift Keying*
BOM	*Bill of Material*
CAM	*Computer-Aided Manufacturing*
CAN	*Controller Area Network*
CAP	*Contention Access Period*
CFP	*Contention Free Period*
CLP	*Controlador Lógico Programável*
CPU	*Central Processing Unit*
CSMA/CA	*Carrier Sense Multiple Access with Collision Avoidance*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
DMA	*Direct Memory Access*
DSSS	*Direct Sequence Spread Spectrum*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
FFD	*Full Function Device*
FHS	*File-System Hierarchy Standard*
GB	*GigaByte*
GTS	*Guaranteed Time Slots*
IDE	*Integrated Development Environment*

I2C	*Inter-Integrated Circuit*
IHM	*Interface Homem-Máquina*
ITO	*Indium Tin Oxide*
JVM	*Java Virtual Machine*
LR-WPAN	*Low-Rate Wireless Personal Area Network*
MAC	*Medium Access Control*
Mbps	*Megabits por segundo*
MB	*MegaByte*
MMU	*Memory Management Unit*
MPDU	*MAC Protocol Data Unit*
NRZ	*Non-Return to Zero*
OLED	*Organic Light-Emitting Diode*
O-QPSK	*Offset-Quadrature Phase Shift Keying*
PCB	*Printed Circuit Board*
PPDU	*Physical Protocol Data Unit*
PSDU	*Physical Service Data Unit*
RAM	*Random Access Memory*
RFD	*Reduced Function Device*
RISC	*Reduced Instruction Set Computer*
RTC	*Real Time Clock*
RTR	*Remote Transmission Request*
SAP	*Service Access Point*
SDK	*Software Development Kit*
SoC	*System on Chip*
SPI	*Serial Peripheral Interface*
SSD	*Solid State Drive*

TTL *Transistor-Transistor Logic*

UART *Universal Asynchronous Receiver/Transmitter*

UBIFS *Unsorted Block Image File System*

USB *Universal Serial Bus*

VFS *Virtual File System*

ZDO *ZigBee Device Object*

Capítulo 1

Introdução

A automação em ambientes residenciais é hoje uma realidade emergente, tendo crescido nos últimos anos em função da rápida evolução tecnológica. Existe atualmente uma enorme diversificação na oferta de soluções nesta área, inovadoras e de alto desempenho, que integram o sistema de controlo com uma interface gráfica. Algumas destas soluções apenas se centram na interface com o utilizador, oferecendo elevada qualidade gráfica, assim como uma panóplia de interfaces de comunicação para quase todo o tipo de aplicações. No entanto, muitas destas soluções são proprietárias e pouco flexíveis na adaptação a novos sistemas, além do seu elevado custo. Muitas vezes pretende-se apenas integrar uma interface simples, *user-friendly*, com canais de comunicação dedicados, gráficos personalizados e sem elevados custos, o que é difícil de encontrar no mercado, sendo necessário recorrer a empresas de desenvolvimento de eletrónica para atingir esse fim.

Esta Dissertação tem como objetivo o desenvolvimento de uma plataforma computacional baseada num processador *ARM11*, dotada de várias interfaces de comunicação, tais como, *touchscreen*, *RS-232*, *CAN*, *Ethernet*, *Cartão SD*, capaz de suportar um sistema operativo baseado em *Unix* (*Linux* ou *Android*), destinada a servir de consola de interface entre o utilizador e um conjunto de equipamentos eletrónicos destinados ao controlo de sistemas solares-térmicos para o aquecimento de águas sanitárias e climatização de habitações.

1.1 Organização da Dissertação

A partir deste capítulo, esta Dissertação está organizada da seguinte maneira:

• **Capítulo 2 - Automação Residencial**: Neste capítulo é efetuada uma abordagem ao conceito da *automação residencial*, onde são apresentadas algumas *Interface Homem-Máquina* existentes no mercado, assim como alguns dos protocolos utilizados nesta área com uma relação mais direta com o trabalho desenvolvido nesta Dissertação.

• **Capítulo 3 - Sistemas Embutidos Linux**: Aqui são resumidamente apresentados os *Sistemas Embutidos Linux*. São abordados alguns aspetos no que toca ao seu desenvolvimento, desde os primeiros estágios, até ao produto final.

• **Capítulo 4 - Plataforma de desenvolvimento**: Neste capítulo são apresentadas as ferramentas utilizadas na realização da vertente mais prática desta Dissertação.

• **Capítulo 5 - Implementação**: Esta capítulo discute toda a implementação do trabalho realizado. São abordadas as soluções encontradas para a componente de *hardware* assim como para a componente de *software*, a sua integração e os testes realizados. Finalmente é discutida a implementação das bibliotecas da interface gráfica.

• **Capítulo 6 - O projeto *Unisol***: Aqui é apresentado o projeto *Unisol* e um simulador desenvolvido para o efeito.

• **Capítulo 7 - Conclusões**: Neste último capítulo são apresentadas as principais conclusões, assim como sugestões para trabalho futuro.

• **Anexo A - Procedimentos adicionais**: Este anexo apresenta procedimentos adicionais de instalação de algumas ferramentas utilizadas nesta Dissertação.

• **Anexo B - Esquema elétrico**: Este anexo apresenta o esquema elétrico da *Interface Homem-Máquina* desenvolvida.

Capítulo 2

Automação Residencial

A *automação residencial*, também conhecida como *Domótica*, é um conceito que visa o controlo e a automação inteligente de edifícios, através da utilização de dispositivos com capacidade de comunicação e de seguir um conjunto de instruções previamente estabelecidas. Fenómenos como o envelhecimento da população ou mesmo a necessidade de contenção de custos, por monitorização dos consumos, contribuíram para que esta tecnologia crescesse nos domínios da habitação doméstica, da hotelaria e das superfícies comerciais. Isto porque, cada vez mais, a *domótica* oferece soluções dirigidas a todo o tipo de edifícios, com mais funcionalidades e com um custo de aquisição menor [1].

Inúmeras aplicações ou funcionalidades podem ser conseguidas com um sistema de *domótica*. No domínio da energia, a *domótica* permite fazer, por exemplo, uma gestão inteligente da iluminação, da climatização, dos sistemas de rega e do funcionamento dos diversos eletrodomésticos, aproveitando melhor os recursos naturais e utilizando as tarifas horárias de menor custo. Facilita o acesso aos diferentes espaços da habitação a pessoas com problemas de mobilidade, ajustando-se às necessidades individuais de cada utilizador, oferecendo também serviços de assistência remota. No que respeita à segurança, a *domótica* permite controlar os acessos aos edifícios, dispondo também de alarmes de deteção de incêndios, fugas de gás ou de água. Permite aceder remotamente, através do telemóvel ou via *internet*, aos diferentes elementos do edifício, como câmeras de segurança, sensores ou atuadores. Resumindo, a *domótica* oferece uma maior qualidade de vida, reduz o trabalho doméstico e aumenta o bem-estar e a segurança dos utilizadores.

Um sistema de *domótica* compreende 3 elementos básicos [2]: o *controlador*, responsável por gerir todo o sistema, o *atuador*, que recebe as instruções do controlador e realiza a ação pretendida e o *sensor*, que monitoriza o ambiente e gera informação que será processada pelo *controlador*. Um elemento que também é importante e distinto num sistema de *domótica* atual é a *interface de controlo*, que é implementada[1] por sistemas computacionais com ou sem teclado e/ou ecrãs táteis. É esta interface que permite ao utilizador interagir com o

[1] Referindo às mais avançadas tecnologicamente.

sistema, visualizando o seu estado e dando ordens de comando.

2.1 Interface Homem-Máquina

Os sistemas atuais de automação requerem que o acesso à informação seja preciso, quer em termos de espaço ou de tempo. O diálogo entre o *Homem* e a *Máquina* tem que reunir todas as funções de que o operador necessita para controlar ou supervisionar o sistema, pois todas as ações tomadas por este têm que garantir o correto funcionamento do mesmo, garantindo também a segurança e disponibilidade do serviço. Assim, é indispensável que o desenvolvimento de interfaces para o diálogo *Homem - Máquina* seja de qualidade, por forma a garantir um controlo correto e seguro do sistema em qualquer circunstância [3].
Estas interfaces tiveram uma grande evolução nos últimos tempos, sobretudo devido ao aparecimento dos sistemas computacionais. A utilização do *botão de pressão*[2] tem vindo a ser substituída por dispositivos eletrónicos (principalmente por ecrãs táteis), que podem ser personalizados de modo a responderem a novas exigências. Têm como funções, a visualização dos dados vindos dos sistemas de controlo, a modificação dos parâmetros e variáveis de controlo, ou comandar um determinado processo do sistema. A comunicação entre este tipo de sistemas e a interface de controlo é, regra geral, feita através de uma ligação série assíncrona *RS-232/RS-422/RS-485*, via porta *USB*, *CAN* ou em alguns casos via *Ethernet*. O mercado oferece uma vasta gama de soluções perfeitamente adequadas a qualquer nível de diálogo.

2.1.1 Interfaces Homem-Máquina Comerciais

Grande parte das soluções comerciais são destinadas ao ambiente industrial. Gigantes da automação como a *Siemens* (`http://www.automation.siemens.com/`), a *Schneider Electric* (`www.schneider-electric.com`), a *Mitsubishi* (`www.mitsubishi-automation.com`), a *Hitachi* (`www.hitachi-ds.com`), a *Beckoff* (`www.beckhoff.com`), entre muitas outras, oferecem uma ou mais famílias de *Interface Homem-Máquina* (IHM), cada uma delas com determinadas caraterísticas e particularidades, mas que na sua maioria possuem similaridades, pois o conceito de concorrência está bem presente neste mercado. Diferenciam-se sobretudo nos protocolos de comunicação que suportam (*ProfiBus*, *ModBus*, *EtherCat*) e a sua escolha depende do destino final de implementação.
Embora possam ser utilizadas em ambiente residencial, devido ao conjunto de certificações para garantir o bom funcionamento em ambiente industrial, estas apresentam um custo demasiado elevado para que possam ser opção em uso habitacional. Por outro lado, as IHM para ambiente residencial estão muito confinadas ao produto para o qual estão destinadas a interagir. Posto isto, é difícil encontrar um sistema residencial automático que albergue apenas uma IHM. É muito comum encontrar uma IHM para o sistema de segurança, outra

[2] A interface mais básica e primordial utilizada numa *Interface Homem-Máquina*.

para o sistema de rega, incêndios ou iluminação e estores.

Um exemplo de um sistema de *domótica* residencial é o *QBus* (`http://www.qbus.be/`). Este sistema é baseado num controlador central, ligado a todos os módulos através de um barramento de dois condutores. Este barramento transporta tanto a energia como os dados de comunicação, o que torna o sistema relativamente simples de implementar. Integra módulos de relés (ON/OFF), módulos de posicionamento de estores, módulos de regulação de iluminação, diversos tipos de sensores (movimento, presença, luz, temperatura, qualidade do ar, etc), entre muitos outros.
A Figura 2.1 apresenta uma das interfaces de controlo oferecidas por este sistema, o *ViZiR Room Controller*.

Figura 2.1: *ViZiR Romm Controller*

Esta interface consiste num ecrã com tecnologia *OLED* (*Organic Light-Emitting Diode*), rodeado de uma tampa frontal capacitiva. Ao tocar na mesma, o ecrã irá mostrar os diferentes menus dos diferentes módulos que pode controlar. Apenas dispõe de uma ligação para o barramento *QBus*.
Outra solução mais completa e versátil oferecida pelo sistema *QBus* é o *Navigator* (Figura 2.2), um painel tátil que pode ser embutido numa parede ou num móvel. Incorpora uma câmera, microfone, altifalantes, podendo funcionar como monitor de vídeo-porteiro. É ideal para executar o software *EQOmmand* (proprietário do *QBus*) que permite visualizar/controlar o estado da habitação, assim como fazer uma gestão dos consumos de energia. Além da ligação ao barramento *QBus*, este painel oferece também interface *RS-232*, *Ethernet* e *USB*. Como sistema operativo, utiliza o *Windows XP Embedded*.

Figura 2.2: *Navigator*

Outros sistemas de *domótica* são oferecidos pela *ELK Products Inc.* (`http://www.elkproducts.com/`). Estes sistemas incorporam segurança (deteção de incêndios, intrusão), controlo de entradas/saídas, controlo de iluminação e estores e gestão de energia. Uma das interfaces disponibilizadas é a *ELK-TS07* (Figura 2.3). Esta interface dispõe de um ecrã tátil com 7 polegadas, um processador *ARM* (com frequência de operação a 200 MHz) e 64 *MegaByte*s de RAM. Como interfaces de comunicação, disponibiliza *RS-232* e *Ethernet*. Executa o sistema operativo *Windows CE Embedded* e corre uma aplicação gráfica proprietária (*ElkRM PC Edition Remote Management*) que permite fazer o controlo de todas as funcionalidades anteriormente referidas. Esta interface pode ligar-se ao sistema de controlo através de uma rede *Ethernet* ou diretamente através da porta série.

Figura 2.3: *ELK-TS07*

A *ComFile Technology* (`http://cubloc.com/`) oferece uma gama de produtos direcionados a servir de IHM. Por exemplo, a gama *CUWIN* oferece as séries *5000/6000/CWV* (Figura 2.4). Vêm equipadas com o sistema operativo *Windows CE 6.0 Embedded*, muito utilizado em aplicações de automação, executado num sistema com um processador *ARM* de *32 bits* (a uma frequência de 533 MHz) e 128 *MegaBytes* (MBs) de memória RAM. Suporta aplicações desenvolvidas no *Microsoft Visual Studio*, o que torna o desenvolvimento para esta plataforma muito similar ao desenvolvimento para um *PC*. O painel frontal apresenta um ecrã a cores, com *touchscreen* e uma resolução de 800x400. Como interfaces de comunicação para o exterior, esta gama oferece *RS-232*, *RS-485*, *Ethernet*, *USB*, *audio* e *SD Card*. Uma aplicação típica destas consolas é apresentada na Figura 2.5.

Figura 2.4: *CUWIN 5000/6000/CWV Series*

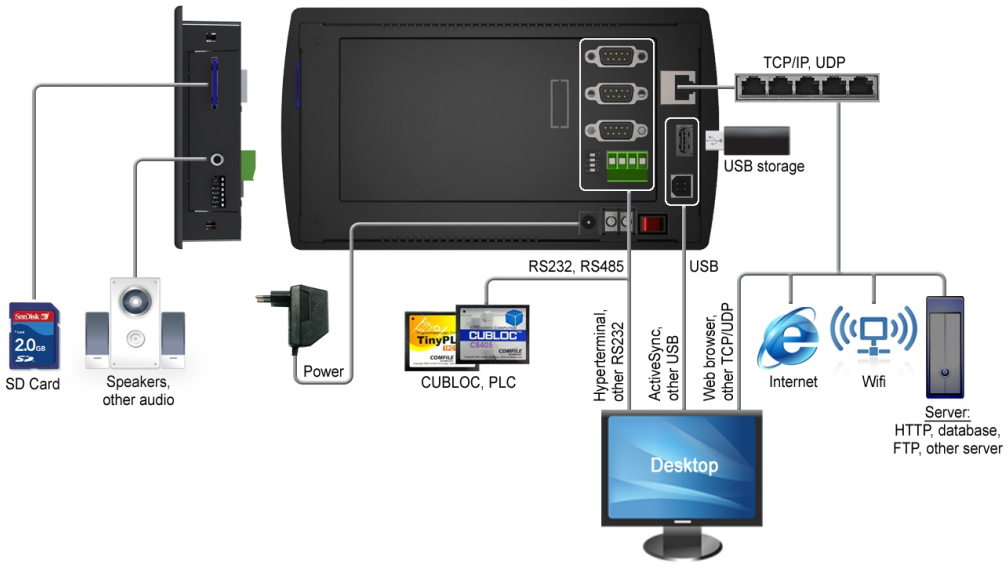


Figura 2.5: Aplicação típica da gama *CUWIN*

A *Advantech* (`http://www.advantech.com/`) oferece, para aplicações que requerem uma *Interface Homem-Máquina*, o *PPC-L61T*. Com um ecrã tátil de 6.5 polegadas, incorpora um processador de baixo consumo (que opera a uma frequência de 500 MHz) e 1 *GigaByte* (GB) de memória RAM. Para comunicação com o exterior, esta *Interface Homem-Máquina* oferece *RS-232*, *RS-485*, *Ethernet*, *USB* e uma porta *VGA*, caso seja necessário adicionar um ecrã de maiores dimensões. Estão disponíveis os sistemas operativos *Windows CE 6.0 Embedded* e *Windows XP Embedded*.

Figura 2.6: *PPC-L61T*

Muitos outros exemplos de interfaces podem ser encontrados pelos mais diversos fabricantes. A sua lista é tão extensa que torna difícil, neste contexto, fazer qualquer tipo de enumeração ou comparação.

2.2 Normas e protocolos utilizados na *domótica*

Nesta secção serão apresentados algumas normas e os protocolos utilizados no domínio da automação residencial.

2.2.1 RS-422/RS-485

Oficialmente denominadas de *TIA/EIA-422* e *TIA/EIA-485*, as normas *RS-422* e *RS-485* são uma solução robusta e fiável para a transmissão de dados a longas distâncias e em meios ruidosos [4]. Ambas se caraterizam pela utilização de um meio de transmissão diferencial ou balanceado[3], isto é, a informação é codificada pela diferença de tensão nas duas linhas de

[3]Normalmente são utilizados cabos de par entrelaçado.

comunicação. Dependendo da polaridade dessa diferença, é codificada a informação binária respetiva. Caso a polaridade seja positiva (a tensão do condutor positivo é maior do que a do condutor negativo) é codificado o nível lógico `1` e caso seja negativa (a tensão do condutor negativo é maior do que a do condutor positivo) é codificado o nível lógico `0`. Uma margem de `0.2 volts` é introduzida na descodificação da informação binária, tornando assim o sistema imune a possíveis níveis de ruído introduzidos no canal de comunicação.

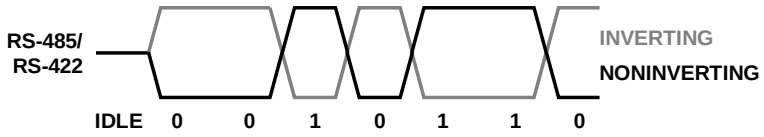


Figura 2.7: Representação digital nas normas *RS-422/RS-485* [5]

RS-422

Esta norma permite interligar um dispositivo `transmissor` e até dez dispositivos `recetores` num único barramento [4]. A Figura 2.8 apresenta o circuito de interface digital da norma *RS-422*. O `transmissor` está identificado como D e o `recetor` como R. A impedância Z_T, usada apenas uma vez no final da linha, é utilizada para igualar as impedâncias do `recetor` e da linha (Z_0). Isto é necessário sempre que a taxa de transferência seja superior a `200 Kbps`, minimizando assim as reflexões. O valor de Z_T pode variar até $\pm 20\%$ sobre o valor de Z_0 [4].

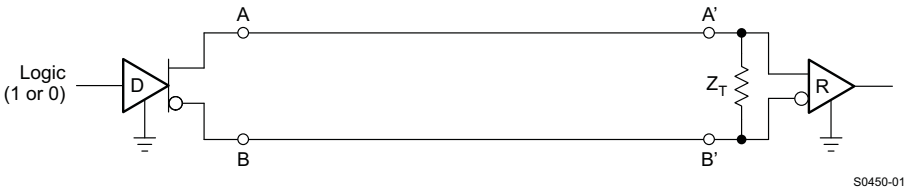


Figura 2.8: Circuito de interface digital da norma *RS-422* [4]

Uma das vantagens desta norma é que não impõe restrições quanto ao comprimento máximo do cabo. No entanto, existe uma relação direta entre o comprimento do mesmo e a taxa máxima de transferência: quanto maior o cabo, menor será a taxa de transferência. Uma aproximação que pode ser usada é que a taxa de transferência multiplicada pelo comprimento do cabo não deve exceder o valor de 10^8 [4].

RS-485

Esta norma é uma expansão da norma *RS-422* e permite que sejam interligados, no mesmo barramento, até `32 transmissores` e `32 recetores` [6]. As caraterísticas elétricas dos `transmissores` e dos `recetores` foram projetadas por forma a que seja possível utilizar dispositivos *RS-485* em aplicações *RS-422*. A Figura 2.9 apresenta o circuito de interface digital da norma *RS-485*. A possibilidade de inclusão de mais do que um `transmissor` no barramento deve-se ao facto de que os `transmissores` podem agora operar em `3` estados diferentes (operação *tri-state*): `lógica 1`, `lógica 0` e `alta impedância`. Em `alta impedância`, o `transmissor` consome uma corrente virtualmente nula, aparentando estar desligado do barramento. Estando neste estado todos os `transmissores` (à exceção daquele que pretende transmitir), o barramento *RS-485* assemelha-se a um barramento *RS-422*.

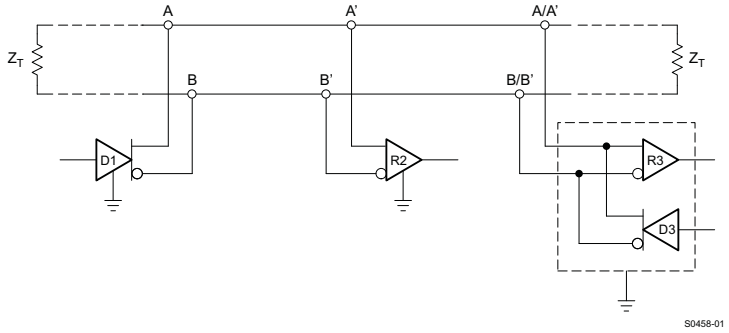


Figura 2.9: Circuito de interface digital da norma *RS-485* [4]

O barramento de comunicação pode ser implementado em *Half-Duplex* ou *Full-Duplex* (Figura 2.10).

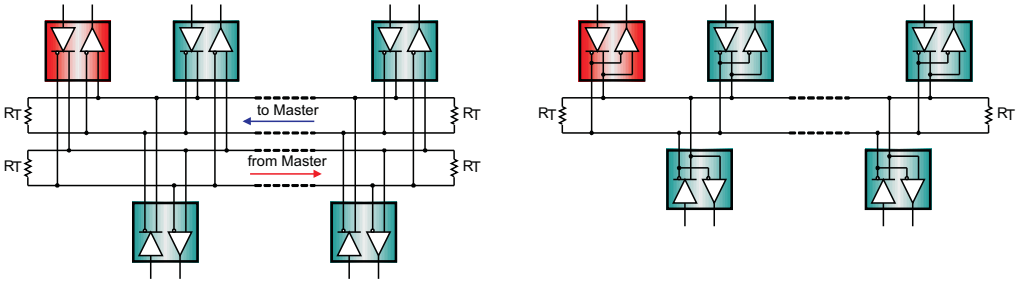


Figura 2.10: Estrutura do barramento *Full-Duplex* e *Half-Duplex* [6]

A implementação em *Full-Duplex* requer que sejam utilizados 2 pares de cabos e permite que a transmissão e receção de dados seja feita em simultâneo, sendo 1 par de cabos utilizado

para a transmissão e o outro para a receção. Na implementação em *Half-Duplex*, apenas é utilizado um par de cabos, o que faz com que a transmissão e a receção sejam feitas em alturas diferentes. Mecanismos de controlo de acesso ao barramento têm que ser implementados por forma a evitar colisões de informação.
As linhas de transmissão devem ser sempre terminadas com uma resistência (R_T) de valor idêntico à impedância caraterística da linha (Z_0), evitando assim reflexões do sinal (Figura 2.11). Normalmente são utilizadas 2 resistências $R_T = 120\ \Omega$ (uma em cada extremidade do cabo). Este valor vem do facto de que, normalmente, o cabo utilizado em instalações *RS-485* possui uma impedância caraterística $Z_0 = 120\ \Omega$.

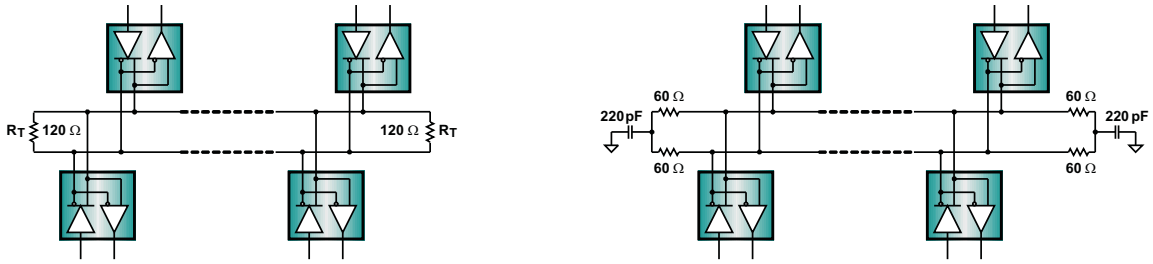


Figura 2.11: Terminações do cabo em RS-485 [6]

Quando a aplicação é feita em meios ruidosos, as terminações resistivas da linha são substituídas por filtros *passa-baixo*, filtrando assim o ruído em modo comum [6]. A Tabela 2.1 resume as caraterísticas gerais das duas normas *RS-422* e *RS-485*.

	RS-422	**RS-485**
Modo de transmissão	Diferencial	Diferencial
Comprimento do cabo @90Kbps	1200 m	1200 m
Comprimento do cabo @10Mbps	15 m	15 m
Taxa de transmissão máx.	10 Mbps	10 Mbps
Saída diferencial mín.	±2V	±1.5V
Saída diferencial máx.	±10V	±6V
Sensibilidade recetor	±0.2V	±0.2V
Transmissores (máx.)	1	32
Recetores (máx.)	10	32

Tabela 2.1: Caraterísticas gerais das normas *RS-422* e *RS-485*

2.2.2 CAN - Controller Area Network

O *Controller Area Network* (CAN) é um protocolo robusto de comunicação série, que permite que múltiplos processadores num sistema comuniquem entre si. Desenvolvido nos anos 80 por *Robert Bosh* para a indústria automóvel [7], rapidamente ganhou aplicação nos domínios da automação industrial, residencial e em aplicações médicas. É um protocolo baseado em mensagens, ou seja, o envio de informação de um nó para outro baseia-se no identificador da mensagem e não no endereço do nó destinatário. Cada mensagem enviada para a rede é recebida por todos os nós, mas apenas consumida pelos interessados. A vantagem desta abordagem é que um novo nó pode ser adicionado ao sistema, sem que seja necessário reprogramar os nós já existentes. Este novo nó irá receber todas as mensagens a circular no barramento e, com base no identificador da mensagem, processará ou não a informação contida na mesma.

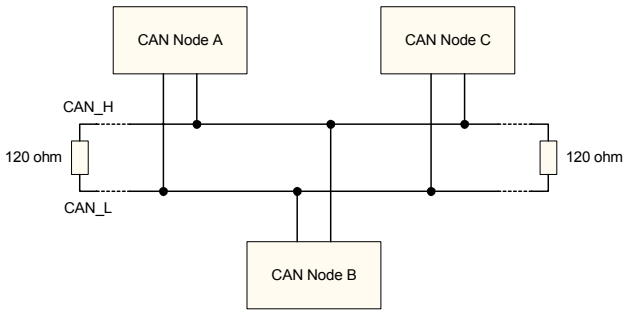


Figura 2.12: Exemplo de um barramento CAN [7]

Como meio físico, o CAN utiliza um barramento diferencial terminado por resistências[4] de 120Ω (Figura 2.12). Existem apenas 2 estados de tensão no barramento: `recessivo`, onde as saídas estão no mesmo potencial (`CAN_L = CAN_H = 2.5V`) e `dominante`, onde as saídas estão com uma diferença de potencial (`CAN_L = 1.5V and CAN_H = 3.5V`). O estado `recessivo` indica a transmissão do `1 lógico` e o estado `dominante` indica a transmissão do `0 lógico` (Figura 2.13).

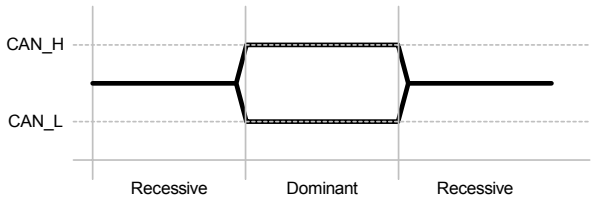


Figura 2.13: Estados recessivo e dominante no barramento CAN [7]

[4]Tal como no barramento `RS-422/485`, estas resistências são utilizadas para minimizar as reflexões.

Estrutura das mensagens

O protocolo CAN utiliza pacotes de informação que contêm uma mensagem completa pronta a ser enviada pelo transmissor (*frame*). Existem 4 tipos de *frames*: *Data Frame*, *Remote Frame*, *Error Frame* e *Overload Frame*.
O tipo de mensagem mais comum é o ***Data Frame***, utilizado para enviar dados para a rede. Existem duas versões do *Data Frame*: o *Standard Data Frame*, que contém um identificador de 11 bits e o *Extended Data Frame*, que contém um identificador de 29 bits. Este último foi introduzido na especificação CAN 2.0B [7] e veio resolver o problema da falta de identificadores em determinados sistemas CAN, devido ao elevado número de mensagens a circular nos mesmos. Assim, ao adicionar um identificador maior (29 bits), os sistemas CAN podem agora gerar, aproximadamente, até 536 milhões de mensagens (2^{29}).

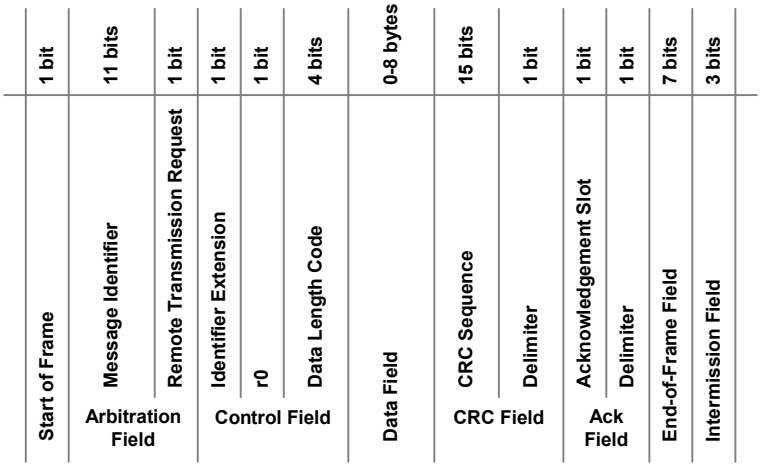


Figura 2.14: Standard Data Frame [7]

A Figura 2.14 apresenta a estrutura do *Standard Data Frame*. Nesta estrutura, existem 7 campos de informação: *Start Of Frame*, *Arbitration Field*, *Control Field*, *Data Field*, *CRC Field*, *Acknowledge Field* e *End Of Frame*.
O primeiro campo, ***Start Of Frame***, é composto por 1 bit *dominante* e é utilizado por todos os nós do barramento para sincronização dos relógios internos.
O campo ***Arbitration Field*** contém o identificador da mensagem e o bit de *Remote Transmission Request* (RTR). O identificador da mensagem é composto por 11 bits, o que torna possível o uso de 2048 (2^{11}) mensagens diferentes, sendo que o menor dos identificadores, representa a mensagem de mais alta prioridade. O bit RTR é utilizado para identificar o tipo de mensagem: caso este bit seja *dominante*, estamos perante um *Data Frame* e caso seja *recessivo* estamos perante um *Remote Frame*.
O ***Control Field*** é composto pelo bit *Identifier Extension* que, caso seja *dominante*, estamos perante a versão *Standard Data Frame*. Caso contrário é *Extended Data Frame*. O bit *r0* não

é usado e é sempre dominante. Por último, o *Data Length Code* indica o número de bytes de dados da mensagem.

O campo ***Data Field*** aloja os dados da mensagem, que podem ser de 0 bytes a 8 bytes. Estes são transmitidos do bit mais significativo para o bit menos significativo.

Para fins de verificação de consistência dos dados, é utilizado o valor presente no campo *CRC Field*.

O campo ***Acknowledgement Field*** é utilizado para indicar se a mensagem enviada foi recebida corretamente. O transmissor envia a mensagem com o bit *Acknowledgement Slot* no estado *recessivo* e espera que pelo menos um nó receba a mensagem corretamente e ponha este bit no estado *dominante*. Caso nenhum nó receba a mensagem, ou seja, o *Acknowledgement Slot* permaneça sempre *recessivo*, é gerado um *Error Frame*. O bit *Acknowledge Delimiter* permanece sempre *recessivo* por forma a distinguir entre um recebimento positivo de mensagem e o início do *Error Frame*.

O campo ***End Of Frame***, composto por 7 bits, indica o fim da mensagem e é enviado totalmente *recessivo* caso esta não contenha erros. Caso o bit *Acknowledgement Delimiter* ou qualquer um dos bits do *End Of Frame* sejam transmitidos *dominantes*, estamos perante um *Error Frame* ou *Overload Frame*.

Finalmente, o campo ***Intermission Field***, composto por 3 bits *recessivos*, faz com que haja um pequeno espaço de tempo em que o barramento permaneça sem qualquer tipo de dados a circular, tempo esse utilizado para os nós processarem a mensagem previamente recebida. No entanto, caso os 2 primeiros bits sejam *dominantes*, estamos perante a transmissão de um *Overload Frame*.

Muito semelhante ao *Standard Data Frame*, a Figura 2.15 apresenta a estrutura do *Extended Data Frame*. Ambas as versões podem coexistir num sistema CAN, sendo que o *Standard Data Frame* tem prioridade sobre o *Extended Data Frame*. Os únicos campos que diferem entre ambos os tipos são os campos *Arbitration Field* e o *Control Field*.

O ***Arbitration Field*** é agora composto por 32 bits distribuídos por 5 subcampos. O *Message Identifier* é igual ao da versão *Standard* e contém os 11 bits mais significativos do identificador da mensagem. De seguida, o bit *Substitute Remote Request* (sempre *recessivo*) é utilizado para que o bit seguinte, o *Identifier Extension* esteja no mesmo sítio em ambas as versões, garantido assim a compatibilidade. O subcampo *Extended Message Identifier* contém os restantes 18 bits do identificador da mensagem.

O ***Control Field*** apresenta apenas um bit adicional na sua composição, o bit *r1*, que deve ser transmitido no estado *dominante*. Os subcampos restantes são utilizados de igual modo.

Outro tipo de mensagens é o ***Remote Frame***. Este é utilizado quando um nó pretende receber informação de outro determinado nó (por exemplo, pedir o valor de um determinado sensor). Em comparação com um *Data Frame*, o *Remote Frame* transmite o bit RTR no estado *recessivo* e o campo *Data Length Code* indica sempre uma transmissão de 0 bytes de dados. Logicamente, não existe o campo *Data Field*.

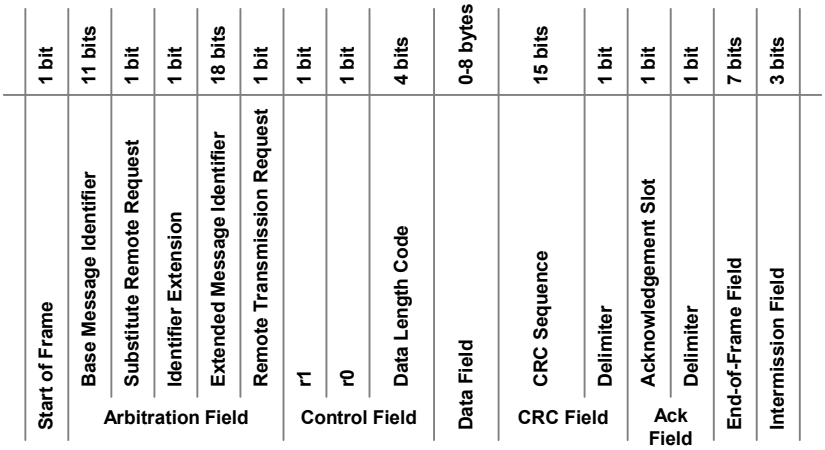


Figura 2.15: Extended Data Frame [7]

Quando um nó recetor deteta um erro num *frame* a receber, este transmite o ***Error Frame***. Este *frame* é enviado assim que o erro é detetado e sempre antes do fim de uma transmissão completa de um *Data Frame* ou *Remote Frame*. Uma vez que o transmissor está constantemente a monitorizar o barramento, ao detetar um *Error Frame*, aborta imediatamente a transmissão e retransmite quando o barramento estiver inativo.
O último tipo de mensagem utilizado no protocolo CAN é o ***Overload Frame*** e é usado quando é necessário que um transmissor adie a transmissão de possíveis mensagens que tenha por enviar ou para sinalizar problemas no *Intermission Field*.

Arbitragem

O protocolo CAN utiliza um método de acesso ao barramento denominado de *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD), ou seja, antes de transmitir, cada nó verifica se existe tráfego no barramento. Caso não exista e na possibilidade de 2 nós tentarem transmitir ao mesmo tempo, ambos param a sua transmissão e recomeçam após um determinado tempo aleatório. Além deste método, o CAN utiliza uma arbitragem de mensagens, baseada na prioridade de cada mensagem (*Non-Destructive Bit Wise Arbitration*). Esta arbitragem *não destrutiva*, faz com que o nó que ganhou o acesso ao barramento continue a transmissão da mensagem, sem que esta tenha sido afetada por outro nó.
Simultaneamente com a transmissão do campo de arbitragem (*Arbitration Field*), cada nó escuta o barramento e caso detete um bit *dominante* quando está a tentar transmitir um bit *recessivo*, cessa a sua transmissão e torna-se um recetor da mensagem de maior prioridade. A Figura 2.16 apresenta um exemplo de um processo de arbitragem num barramento CAN. No *Ponto 1*, ambos os nós começam a transmissão do campo de arbitragem. Até ao *Ponto 2*, ambos os nós transmitem a mesma informação, altura em que o *nó 1* deteta a presença de

um bit *dominante* no barramento, enquanto tenta transmitir um bit *recessivo*, o que origina uma paragem imediata da sua transmissão. No *Ponto 3*, após o processo de arbitragem, o *nó 2* ganha o acesso ao barramento e continua a transmissão dos restantes dados da mensagem.

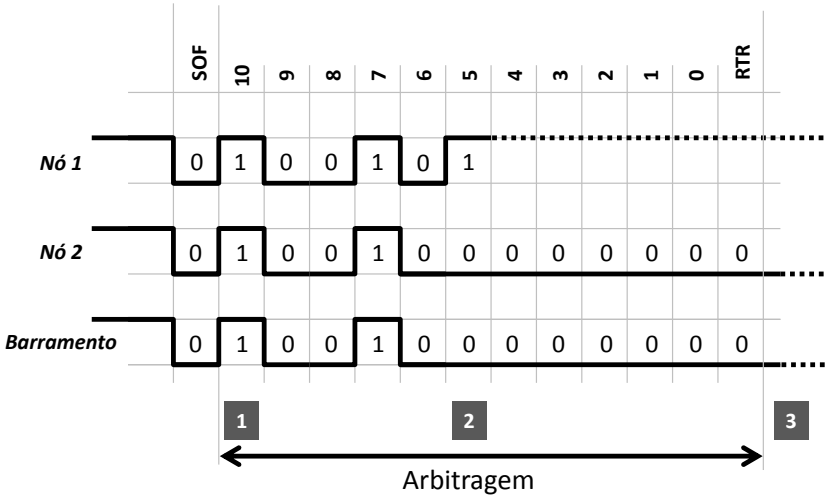


Figura 2.16: Processo de arbitragem no barramento CAN [7]

Técnica de `bit stuffing`

O protocolo CAN utiliza codificação *Non-Return to Zero* (NRZ) para representar os sinais digitais. Nesta codificação, os bits permanecem constantes durante todo o intervalo de tempo que os representa, o que leva a que uma longa sequência de bits iguais faça com que o sinal não varie. Uma vez que os relógios internos de cada dispositivo CAN utilizam as transições do sinal para a sua sincronização, uma longa sequência igual pode levar a um ligeiro desvio dos mesmos, o que origina uma incorreta descodificação do sinal. A técnica de *bit stuffing* vem resolver este problema, pois o seu propósito é introduzir uma transição do sinal sempre que sejam transmitidos 5 bits iguais (Figura 2.17).

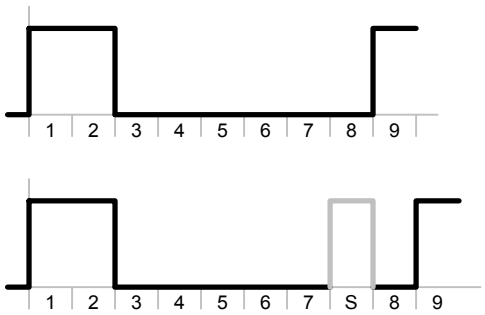


Figura 2.17: *Bit stuffing* [7]

Taxa de transferência e comprimento do barramento

A Tabela 2.2 apresenta os valores máximos da taxa de transferência dos dados em função do tamanho máximo do barramento. Logicamente, quanto maior o barramento, menor a taxa de transferência. Isto para garantir que os dispositivos presentes no barramento, mais distantes do ponto de transmissão, possam detetar colisão do *bit*.

Taxa de transferência	**Comprimento**
1 Mbit/s	25 m
500 Kbit/s	100 m
250 Kbit/s	250 m
125 Kbit/s	500 m
50 Kbit/s	1000 m
10 Kbit/s	5000 m

Tabela 2.2: Taxa de transferência em função do comprimento do barramento CAN [7]

2.2.3 IEEE 802.15.4 e ZigBee

Fundada em 2002, a *Zigbee Alliance*, uma associação sem fins lucrativos e que envolve um conjunto de empresas nas diversas áreas da engenharia e das energias verdes, trabalha no desenvolvimento de normas e produtos para redes sem-fios, de baixo custo e baixo consumo. Deste trabalho, resultou o *ZigBee* [8], uma tecnologia que permite a implementação de redes sem-fios de baixa potência, baixa taxa de transmissão e de baixo custo. O *ZigBee*, que define a camada de rede para as topologias *star* e *peer-to-peer*, fornecendo também um conjunto de ferramentas para a programação de aplicações na camada de *aplicação*, é implementado sobre a norma *IEEE 802.15.4*, que define as camadas física (PHY) e *Medium Access Control* (MAC) para redes sem-fios de baixo custo e baixa taxa de transferência [9].

IEEE 802.15.4

Esta norma tem como principais caraterísticas a flexibilidade na construção de redes sem-fios, com baixos consumos energéticos, onde a transferência massiva de dados não é um requisito forte. Estas redes, denominadas *Low-Rate Wireless Personal Area Network*s (LR-WPANs),

têm como principais caraterísticas [9]:

- Taxas de transferência até 250 kb/s
- Topologias *star* ou *peer-to-peer*
- Endereços de 16 ou 64 bits
- Alocação de *Guaranteed Time Slots* (GTSs)
- Acesso ao meio através de CSMA/CA
- Baixo consumo
- Deteção de energia (*Energy Detection (ED)*)
- *Link quality indication (LQI)*

• Camada física (PHY)

A camada física é responsável pela ativação/desativação do *transceiver* rádio, pela estimação da potência do sinal recebido (*Energy Detection*)[5], pela medição da qualidade do sinal recebido (*Link Quality Indication*) e por sintonizar o *transceiver* no canal pedido pelas camadas superiores.
A norma *IEEE 802.15.4* oferece três opções para a camada física. Ambas são baseadas na técnica de modulação *Direct Sequence Spread Spectrum* (DSSS), que consiste no espalhamento do sinal numa faixa de frequência elevada, utilizando uma menor densidade espectral de potência do sinal. Isto faz com que seja possível a partilha de um único canal por vários utilizadores, resultando numa implementação de menor custo [10]. Estas opções apenas diferem na frequência de operação: 2.4 GHz (Universal), 915 MHz (Estados Unidos da América) e 868 MHz (Europa).
A banda dos 2.4 GHz utiliza uma modulação *Offset-Quadrature Phase Shift Keying* (O-QPSK), atingindo taxas de transferência até 250 Kbps e, devido à sua utilização internacional, esta banda oferece vantagens em termos de um maior mercado e menor custo de produção. As bandas 868/915 MHz oferecem uma alternativa para ambientes onde a interferência seja elevada (pois muitas outras tecnologias funcionam na banda dos 2.4 GHz, como por exemplo, o *WiFi* ou o *Bluetooth*) e utilizam *Binary Phase Shift Keying* (BPSK) como modulação. A taxa de transferência é de 20 Kbps para a frequência de 868 MHz e de 40 Kbps para a frequência de 915 MHz [10].
Estão disponíveis 27 canais distribuídos pelas 3 bandas (Figura 2.18): a camada física dos 868 MHz suporta apenas um canal entre os 868.0 e os 868.6 MHz e a dos 915 MHz suporta 10 canais entre os 902.0 e os 928.0 MHz. Nos 2.4 GHz, são suportados 16 canais entre os 2.4 GHz e os 2.4835 GHz, separados em 5 MHz.

[5]Uma medida utilizada pela camada de rede no processo de seleção do canal ou para determinar se um canal está livre ou ocupado.

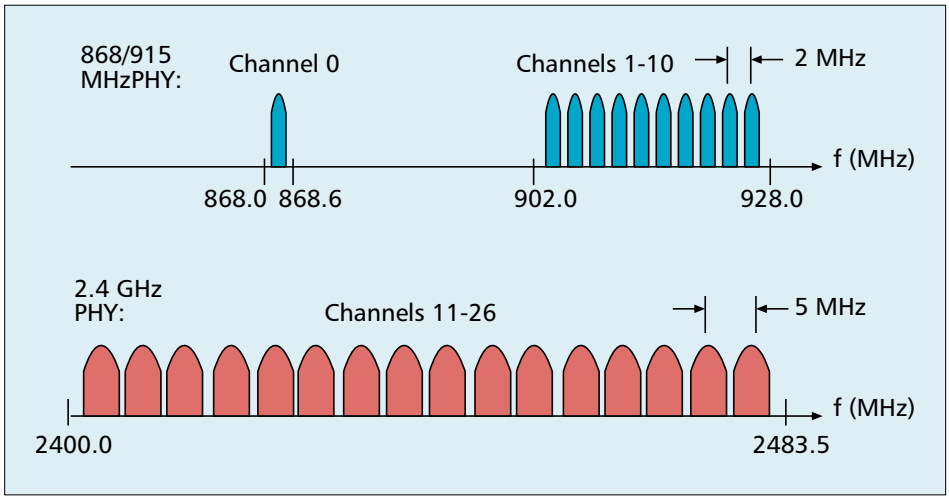


Figura 2.18: Posicionamento do canais de comunicação no espetro de frequência [10]

Para manter uma interface comum com a camada MAC, todas as camadas físicas têm a mesma estrutura do pacote de dados, denominado de *Physical Protocol Data Unit* (PPDU).

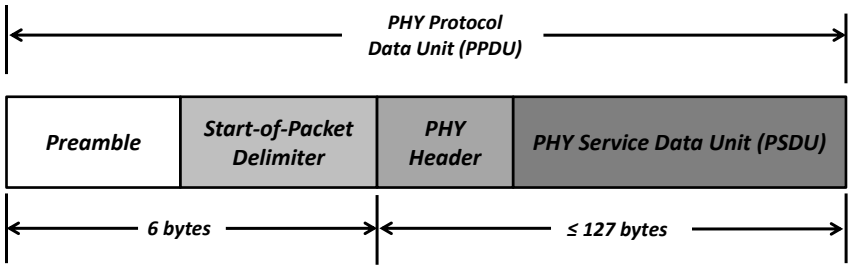


Figura 2.19: *Physical Protocol Data Unit* [10]

O PPDU é composto por um *Synchronization Header* (*Preamble + Start-of-Packet delimiter*), utilizado na sincronização, por um *PHY Header*, que indica o tamanho do *Physical Service Data Unit* (PSDU) em bytes e pelo *payload* de dados, o PSDU.

- **Camada MAC**

A camada MAC define como os rádios dos diferentes dispositivos irão partilhar o espaço livre, coordenando o acesso dos *tranceivers* aos diferentes canais de rádio e fazendo o encaminhamento das tramas de dados. Esta camada também incorpora funções de associação e dissociação da rede. Os dados de informação são transportados numa trama, denominada de *MAC Protocol Data Unit* (MPDU) (Figura 2.20). Esta estrutura é muito flexível, adaptando-se às

necessidades das diferentes aplicações e topologias de rede, enquanto mantém a simplicidade do protocolo. A sua composição compreende o *MAC Header*, o *MAC Service Data Unit* e o *MAC Footer*. O *MAC Header* é composto pelo *Frame Control*, responsável por especificar o tipo de trama a ser transmitida, o formato do endereço e o que esta transporta. O tamanho[6] do endereço pode variar entre 0 e 20 bytes. O *Sequence Number* é utilizado para questões de verificação de transmissão bem sucedida.
O *MAC Service Data Unit* é composto pelo *Payload*, ou seja, pelo campo que transporta a informação. Este campo tem tamanho variável, dependendo do tipo de trama e da informação que transporta. No entanto a trama MAC não pode exceder os 127 bytes de tamanho. A camada MAC define quatro tipos de tramas: *beacon frame*, *data frame*, *acknowledgement frame* e *command frame*. Apenas a *beacon frame* e a *data frame* contêm informação a ser enviada para as camadas mais elevadas. A *acknowledgement frame* e a *command frame* são utilizadas para comunicações *peer-to-peer*.
Finalmente, o *MAC Footer* é composto pelo *Frame Check Sequence*, utilizado para verificar a integridade de toda a trama.

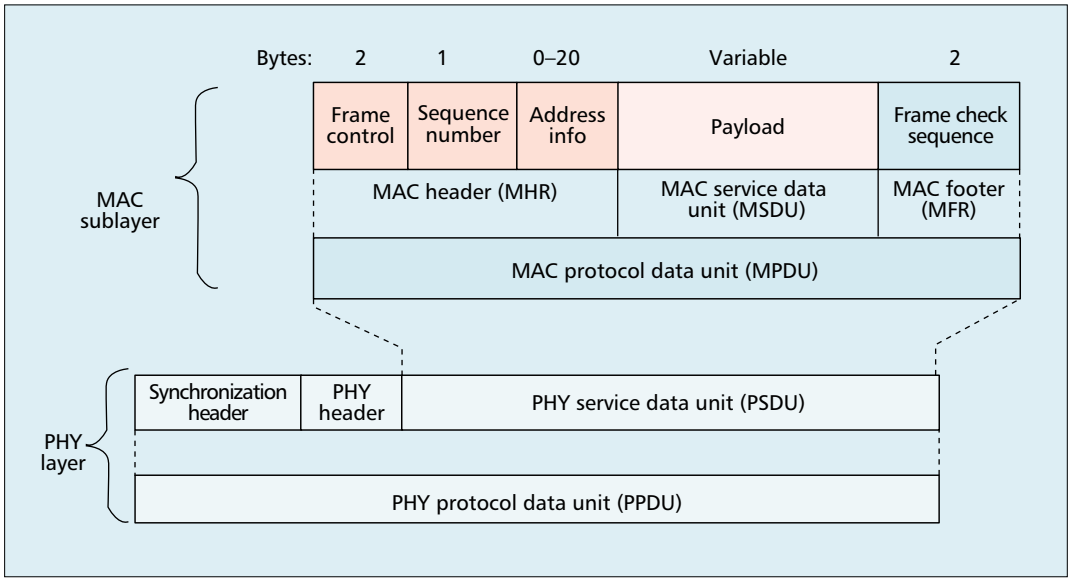


Figura 2.20: *Physical Service Data Unit* [10]

Existem dois tipos de dispositivos definidos pela camada MAC: os *Full Function Device*s (FFDs) e os *Reduced Function Device*s (RFDs). Os FFDs estão equipados com toda a implementação da camada MAC, tornando-os capazes de agir como coordenadores de rede ou como dispositivos terminais. Quando agem como coordenadores de rede, os FFDs enviam *beacons* com o objetivo de sincronizar, comunicar e fornecer serviços de *join* (registo e ligação) na

[6]No caso de uma trama de *Ack*, tamanho do endereço é 0.

rede. Os dispositivos RFD apenas podem atuar como dispositivos terminais, normalmente equipados com sensores ou atuadores e podem apenas interagir com um FFD.

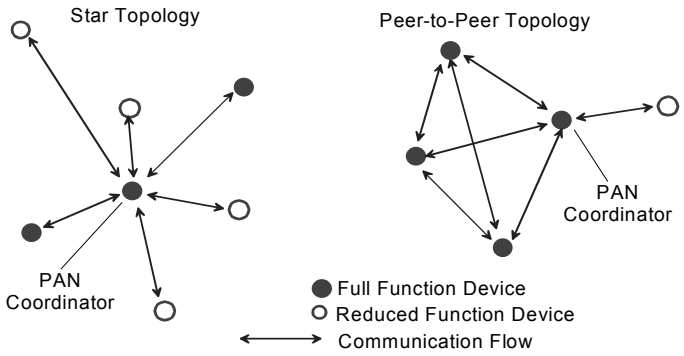


Figura 2.21: Topologias na norma IEEE 802.15.4 [9]

A norma IEEE 802.15.4 define 2 tipos de topologia de rede [9]: a topologia *star* e a topologia *peer-to-peer* (Figura 2.21). Na topologia *star*, é adotado um modelo *Master-Slave*, onde o FFD assume o papel de coordenador da rede. Os outros dispositivos podem ser tanto FFDs ou RFDs e apenas comunicam com o dispositivo coordenador. Na topologia *peer-to-peer*, um dispositivo FFD pode comunicar diretamente com outros dispositivos FFD que estejam na área de alcance do seu rádio e com dispositivos que estão fora do alcance, através de FFDs intermediários. Nesta topologia, é selecionado um FFD para assumir o papel de coordenador da rede. Algumas aplicações podem requerer largura de banda dedicada para alcançar latências mais baixas [10]. Assim, a norma *IEEE 802.15.4* permite que uma rede possa operar em modo *superframe*. Neste modo, o coordenador da rede transmite um *superframe beacon* (Figura 2.22) em intervalos pré determinados, que podem variar de $15ms$ a $245s$ [10]. Este *beacon* serve de mecanismo de sincronização, assim como para descrever a estrutura do *superframe* e enviar informação de controlo para a rede.
O *superframe* está dividido numa parte inativa, que permite ao coordenador entrar em modo *sleep*, poupando assim energia e numa parte ativa, dividida em porções iguais, que contém o *Contention Access Period* (CAP) e o *Contention Free Period* (CFP). Durante o CAP, os dispositivos competem para aceder ao canal utilizando o mecanismo de acesso ao meio *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA). Durante o CFP, os dispositivos transmitem através de porções de tempo exclusivamente dedicadas para o efeito (*Guaranteed Time Slots* (GTS)) e cedidas pelo coordenador da rede.
Sempre que um dispositivo necessita transmitir dados ao coordenador (caso não tenha GTS), tem que esperar pelo próximo *beacon*, sincronizar e competir pelo acesso ao canal.
Por outro lado, a comunicação pode ser feita sem *superframe*. Neste caso, o coordenador nunca envia o *beacon* e a comunicação baseia-se apenas sobre o mecanismo de CSMA/CA. O coordenador permanece sempre ligado e pronto a receber dados. Periodicamente, os disposi-

tivos terminais pedem informação que esteja pendente ao coordenador.

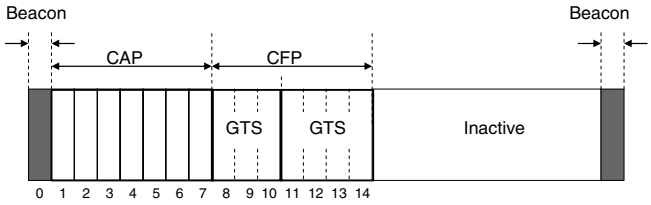


Figura 2.22: Estrutura do *superframe* [10]

ZigBee

A arquitetura da *stack* do ZigBee é composta por diversas camadas (Figura 2.23). Cada uma destas camadas fornece um conjunto de serviços para as camadas superiores, como, por exemplo, o *data service*, que possibilita a transmissão de dados. Estes serviços são expostos às camadas superiores através de uma interface denominada de *Service Access Point* (SAP). Como já referido anteriormente, a norma *IEEE 802.15.4* definiu as camadas *física* (PHY) e MAC, sendo da responsabilidade do *ZigBee* as camadas de *rede* (NWK) e da *aplicação* (APL).

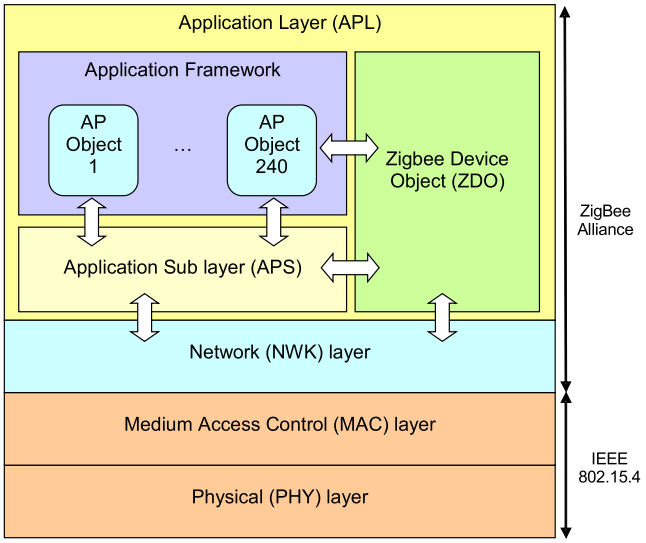


Figura 2.23: Arquitetura da camada funcional ZigBee e pilha de protocolos [11]

A camada de rede (NWK) é responsável pelos procedimentos de gestão da rede (por exemplo, entrada e saída de novos dispositivos na rede), segurança e encaminhamento de dados. Nesta camada, são suportadas as topologias *star*, *tree* e *mesh* (Figura 2.24). Nestas topologias, podem existir 3 tipos de dispositivos. O *ZigBee End-Device* ou *ZED* corresponde a um FFD ou RFD, definido pela norma *IEEE 802.15.4*, agindo como um dispositivo terminal simples

(sensor ou atuador). Este dispositivo não permite que outros dispositivos se associem com ele e não participa no encaminhamento de informação. O *ZigBee Router* ou *ZR* é um FFD que participa no encaminhamento da informação nas redes *mesh* e *tree*. Por último, o *ZigBee Coordinator* ou *ZC* é um FFD que faz a gestão de toda a rede.
Na topologia *star*, apenas um único dispositivo opera como *ZC*. Este escolhe um identificador para a rede (que não pode estar a ser utilizado por outra qualquer rede na vizinhança) e faz o reencaminhamento da informação de um dispositivo para outro. Esta topologia não é a mais adequada para redes de sensores, pois, por um lado, o dispositivo que for escolhido para *ZC* rapidamente irá ficar sem bateria e por outro, haverá problemas de escalabilidade.
A topologia *mesh* também inclui apenas um dispositivo que opera como *ZC* e que identifica toda a rede, mas, no entanto, a comunicação é descentralizada, ou seja, cada dispositivo na rede pode comunicar diretamente com outro, desde que esteja ao seu alcance. Em termos de consumo energético, esta topologia, em relação à topologia *star*, é mais eficiente, uma vez que o processo de comunicação não depende de um dispositivo central. A topologia *tree* é uma particularidade da topologia *mesh*, onde apenas existe um caminho de fluxo de informação em cada dois dispositivos.
A camada de *aplicação* (APL) fornece uma *framework* para o desenvolvimento de aplicações e comunicação. Consiste na *Application Support Sublayer* (APS), no *ZigBee Device Object* (ZDO) e na *Application Framework*. A *Application Framework* pode ter até 240 *Application Object*s (APOs), que são aplicações definidas pelos desenvolvedores e que controlam unidades de *hardware*, como por exemplo, interruptores, lâmpadas ou sensores. O ZDO é um objeto que auxilia as APOs na descoberta de novos dispositivos na rede e fornece serviços de comunicação, gestão de rede e segurança. A APS fornece serviços de transferência de dados para os ZDOs e APOs.

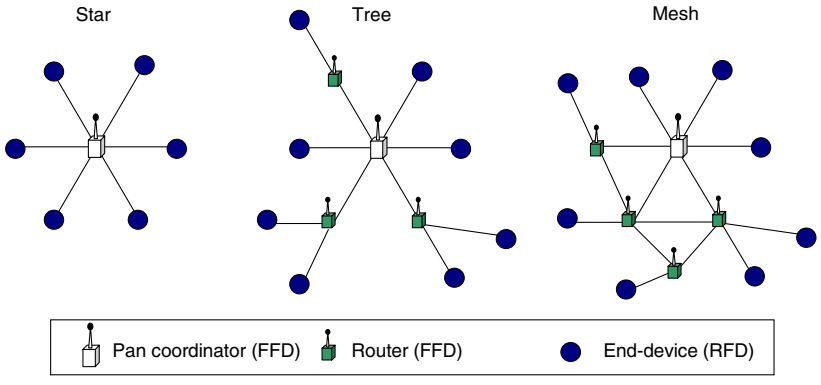


Figura 2.24: Topologias de rede no Zigbee [11]

Este capítulo introduziu o conceito da *automação residencial* e nele foram apresentadas algumas das soluções (quer a nível de interfaces, quer a nível de protocolos) utilizadas neste domínio que mais se enquadram nas caraterísticas e funcionalidades pretendidas no traba-

lho desta Dissertação. De retirar que o mercado oferece soluções para quase todo o tipo de aplicações pretendidas, com inúmeras funcionalidades, mas que muitas vezes, por serem proprietárias, são pouco flexíveis na adaptação a outros sistemas.

Capítulo 3

Sistemas Embutidos *Linux*

Neste capítulo será dada uma visão geral dos sistemas embutidos, nomeadamente dos *Sistemas Embutidos Linux*. Estes últimos tiveram um grande crescimento nos últimos anos, estando cada vez mais presentes nas nossas vidas quotidianas, muitas vezes sem darmos pela sua presença. São exemplos de *Sistemas Embutidos Linux* os recetores digitais de televisão, sistemas de entretenimento em casa, telemóveis, até sistemas menos óbvios como caixas de multibanco, sistemas médicos e automóveis.

3.1 O *Linux*

O *Linux* é o núcleo (que passaremos a tratar pelo termo inglês *Kernel*) de um sistema operativo, originalmente escrito por Linus Torvalds [12]. Não é um sistema operativo completo, como muitas vezes é referido, pois não inclui, por exemplo, o sistema de ficheiros ou o ambiente gráfico [13].
O arquitetura do *Kernel* é monolítica, ou seja, as suas funções (*scheduling* de processos, gestão de memória, operações de entrada/saída, acesso ao sistema de ficheiros) são executadas num único processo e num único espaço de endereçamento. Embora tenha esta particularidade, o *Kernel* permite que algumas das funções (por exemplo, controladores de dispositivos, suporte à rede ou sistema de ficheiros) possam ser compiladas e executadas como módulos (denominados *loadable kernel modules*). Estes módulos, compilados separadamente do *Kernel*, são posteriormente inseridos durante o arranque ou durante a execução do mesmo. A Figura 3.1 ilustra a arquitetura genérica de um sistema *Linux* e todos os componentes envolvidos.
Como camada base, temos a camada de *Hardware*. Têm que fazer parte obrigatoriamente desta camada um processador com *Memory Management Unit* (MMU), *Random Access Memory* (RAM) suficiente para acomodar o sistema e uma unidade de armazenamento capaz de ser acedida pelo *Kernel* para carregar o sistema de ficheiros.
Acima da camada *Hardware* temos a camada do *Kernel*, que como já referido, é o núcleo do sistema operativo responsável por fazer a gestão do hardware de maneira coerente e fornecer

abstrações do mesmo ao software na camada do utilizador. As duas sub-camadas presentes, *low-level interfaces* e *high-level abstractions*, são responsáveis por fornecer serviços à camada de aplicações. A primeira permite que as aplicações acedam aos registos internos ou à memória do microprocessador, enquanto que a segunda providencia os *processos*, os *ficheiros* e os *sockets*. Para que os dados provenientes de dispositivos específicos (dispositivos de armazenamento ou de rede) sejam corretamente interpretados, o *Kernel* necessita de recorrer a dois componentes: o sistema de ficheiros e os protocolos de rede.

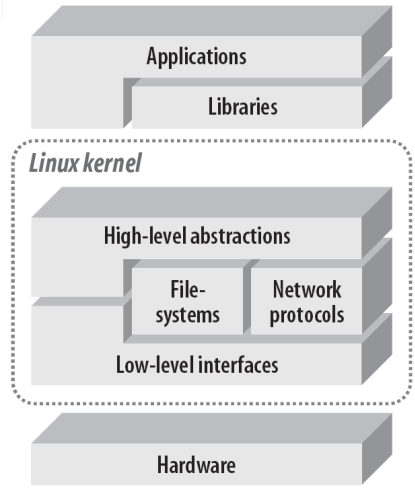


Figura 3.1: Arquitetura genérica de um sistema *Linux* [12].

Antes de chegar às aplicações que correm no nível do utilizador, existe uma camada de bibliotecas e serviços que fornecem uma abstração entre as aplicações e o *Kernel*. A principal biblioteca dos sistemas *Linux* é a *GNU C*, *glibc*. Para sistemas embutidos, poderão ser utilizadas outras bibliotecas, como por exemplo, a *uglibc*, que tem vantagens a nível de tamanho. As bibliotecas podem ser *ligadas dinamicamente* às aplicações, isto é, não fazem parte do ficheiro executável, mas são carregadas para memória assim que o programa arranca. A biblioteca *glibc* é carregada na RAM apenas uma vez e é partilhada por todas as aplicações que a usam. Em sistemas embutidos, por questões de espaço de memória RAM, as bibliotecas são *ligadas estaticamente* às aplicações, isto é, fazem parte do ficheiro executável [12].

3.2 Sistema Embutido *Linux*

A expressão *Sistema Embutido Linux* refere-se a um sistema completo[1] *Linux*, ou seja, um sistema que utiliza um *Kernel Linux* e uma variedade de outras aplicações (que serão exploradas adiante) e que tem como alvo os sistemas embutidos. A Figura 3.2 ilustra o diagrama de um

[1]Referindo-se a *software*.

sistema embutido capaz de suportar *Linux embutido*. Este sistema é composto por um processador de 32-bits com uma arquitetura *Reduced Instruction Set Computer* (RISC)[2], memória *Flash* para armazenamento de dados, memória RAM, interface de comunicação série *Universal Asynchronous Receiver/Transmitter* (UART), comunicação *Ethernet* e comunicação sem fios (IEEE 802.11). Também possui uma porta *Universal Serial Bus* (USB) e um *Real Time Clock* (RTC).

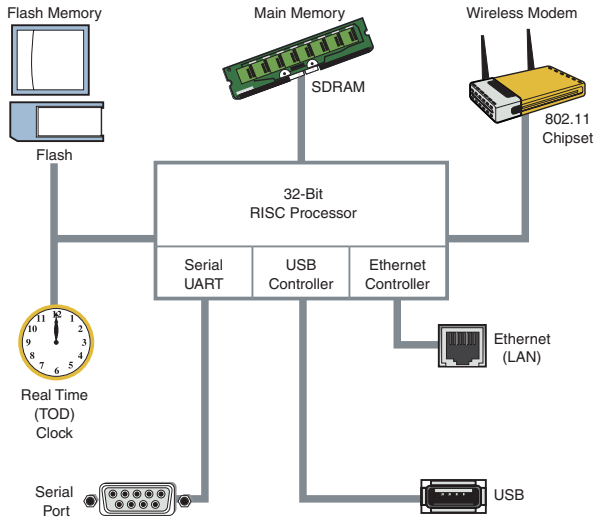


Figura 3.2: Exemplo de um sistema embutido [14].

Apesar das suas particularidades, nomeadamente a comunicação sem-fios, este sistema ilustra bem o protótipo básico de um sistema embutido: possui um microprocessador, que além do processamento de instruções, vem dotado de inúmeros periféricos, neste caso, UART, controlador *Ethernet* e controlador USB; foi desenvolvido para uma aplicação específica; não inclui, mas poderia incluir interface com o utilizador (*botões*, *leds* ou *ecrã*); tem recursos limitados (por exemplo, a memória é apenas projetada de acordo com os requisitos).

3.3 Pré-Desenvolvimento de um sistema embutido

Com o exemplo da Figura 3.2, percebemos que projetar um sistema embutido é um grande desafio para os seus projetistas. Questões como qual o processador a escolher, que periféricos deve conter, qual o tamanho da memória RAM e física, entre outros, devem ser bem analisadas por forma a obter um produto robusto, funcional e com custo mínimo.

[2]De notar que este processador deve possuir os requisitos anteriormente apresentados para suportar o *Kernel*.

Nas secções seguintes, serão abordadas alguns pontos a ter em consideração antes de desenvolver um sistema embutido.

3.3.1 Processador

O *Kernel Linux* suporta atualmente mais de 20 arquiteturas diferentes de processadores para aplicações em sistemas embutidos, sendo que nem todas são preferencialmente utilizadas. A maioria dos sistemas embutidos utiliza processadores integrados, também conhecidos como *System on Chip* (SoC) com as arquiteturas *Power Architecture* (`www.power.org`), *ARM* (`www.arm.com/`), *MIPS* (`www.imgtec.com/`) e *Intel X86* (`www.intel.com`). A arquitetura *Power Architecture* é mais utilizada em aplicações de telecomunicações e redes, a *ARM* em telemóveis e a *MIPS* em equipamento doméstico.
Estes processadores[3] possuem diversas funções integradas que vão desde módulos de comunicação sem fios, aos tradicionais módulos de comunicação *UART*, *SPI* e *I2C*, codificadores/descodificadores de vídeo e áudio, entre muitos outros.

3.3.2 Armazenamento

Como referido anteriormente, uma das caraterísticas de um sistema embutido é que este possui recursos limitados e a memória física não é excepção. Preferencialmente encontramos discos rígidos como meio de armazenamento para grandes quantidades de dados[4], mas devido a vários fatores, como o seu peso, sensibilidade a vibrações e por requererem múltiplas fontes de alimentação, estes não são a escolha preferencial na maior parte dos sistemas embutidos. A solução passa por utilizar dispositivos de armazenamento mais pequenos e mais baratos, denominados como dispositivos de armazenamento *Solid State Drive* (SSD). Estes dispositivos, por não terem partes móveis, são imunes às vibrações e são alimentados apenas por uma única fonte de alimentação, o que os torna a melhor escolha para a maioria dos sistemas embutidos.
Os dispositivos SSD são baseados na tecnologia de memória *Flash*[5]. A sua estrutura interna está dividida em vários blocos, denominados *erase blocks*, o que faz com que a necessidade de apagar uma célula de dados leve a apagar um bloco inteiro de dados. Além desta desvantagem, as memórias *Flash* têm um número limitado de ciclos de escrita, o que um erro de software pode levar a que rapidamente seja danificada.

Podemos encontrar 2 tipos de memórias *Flash*: *NOR Flash* e *NAND Flash*. As memórias do tipo *NOR* foram as primeiras a surgir no mercado e possuem *erase blocks* maiores que as

[3] Referindo aos SoC.

[4] O computador de secretária tradicional possui hoje em dia pelo menos 500 GB de capacidade de armazenamento.

[5] A denominação de *Flash* foi dada pelo seu inventor, pois segundo este, a forma como se apagam estas memórias lembra o "flash"de uma máquina fotográfica.

memórias do tipo *NAND*, o que leva a uma demora mais acentuada a apagar dados. As suas células são acedidas pelo processador sequencialmente enquanto que as *NAND* são acedidas aleatoriamente [14].

3.3.3 Ferramentas de desenvolvimento

Uma prática comum antes de desenvolver um *Sistema Embutido Linux* consiste em adquirir uma plataforma de desenvolvimento, baseada numa determinada arquitetura e com características semelhantes ao produto final que se pretende desenvolver. Normalmente estas plataformas, além do hardware, vêm também com a componente de software, o *Linux embutido*, que traz todas as ferramentas necessárias para começar o desenvolvimento de código (controladores de dispositivos e aplicações), compilação, depuração de erros, entre outros[6]. Uma dessas ferramentas é o *cross-compiler*, que normalmente vem integrada numa *cross-toolchain*[7] e permite compilar código numa arquitetura e ser executado noutra. Por exemplo, podemos compilar código para a arquitetura *ARM* numa máquina com arquitetura *X86*.
Outras ferramentas que poderão ser necessárias dependem das aplicações que o sistema a ser desenvolvido terá. Por exemplo, poderá ser necessária uma interface gráfica e neste caso é necessário adquirir as bibliotecas respetivas, assim como um ambiente de desenvolvimento gráfico. No Capítulo 4 serão abordados alguns dos aspetos aqui referidos e com especial foco nas ferramentas utilizadas no desenvolvimento desta Dissertação.

3.4 *Character Device Drivers*

Neste ponto iremos abordar os *Character Device Drivers*. Este grupo de *controladores* abrange a maioria dos dispositivos, pois grande parte deles são orientados ao *byte*.

Um módulo (*Character Device Driver*) pode ser inserido estaticamente ou dinamicamente no *Kernel*. Da sua compilação, resulta um ficheiro com extensão *.ko* (*kernel object*). Uma vez que vai ser carregado e *ligado* ao *Kernel*, o módulo tem que ser compilado da mesma forma que este, ou seja, com a mesma *toolchain* e os *headers* (ficheiros de pré-compilação) a utilizar têm que ser aqueles existentes no diretório fonte do *Kernel*.
Existem *headers* que são comuns em todos os módulos: `linux/module.h` e `linux/init.h`. O *header* `module.h` contém grande parte das definições de símbolos e funções necessárias pelos módulos dinâmicos e o *header* `init.h` é necessário para especificar as funções de inicialização e remoção do módulo. São também utilizadas as seguintes definições, normalmente no final do código e que identificam o respetivo *device driver*:

[6] Respeitante ao *Kernel*.

[7] Conjunto de ferramentas de programação que são usadas para criar um ficheiro executável numa determinada arquitetura e que tem como alvo outra arquitetura diferente.

- `MODULE_LICENCE("GPL")` - define qual a licença aplicada ao código;
- `MODULE_AUTHOR("autor do módulo");`
- `MODULE_DESCRIPTION("breve descrição do módulo");`

A Figura 3.3 ilustra a iteração do *Character Device Driver* e o sistema computacional. Uma aplicação (*user-space*) comunica com um dispositivo (*hardware-space*) utilizando o correspondente *Character Device Driver* (*kernel-space*). Esta aplicação executa as tradicionais operações sobre ficheiros no *Character Device File* (CDF) (*open*, *close*, *write* e *read*), sendo o *Virtual File System* responsável por fazer a correspondente tradução da operação no *Dharacter Device Driver*.

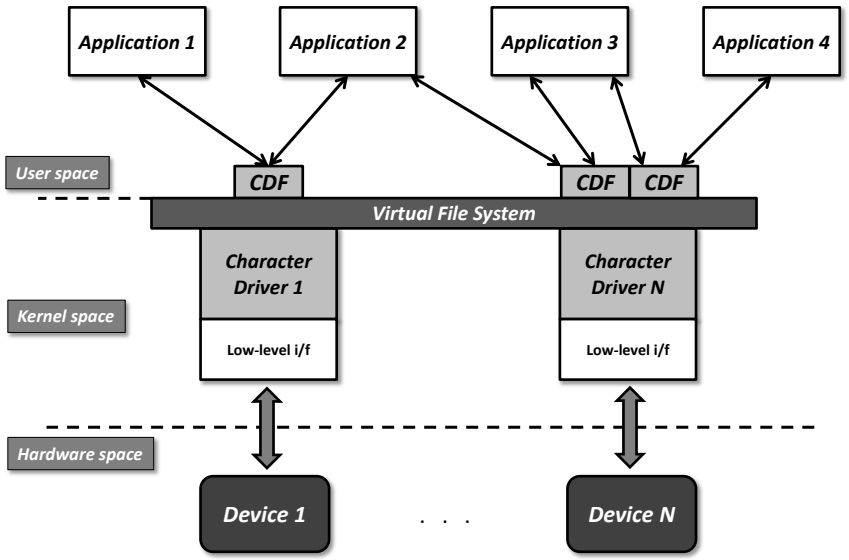


Figura 3.3: Visão geral de um *Character Device Driver* [15]

A ligação entre a aplicação e o *Character Device File* é baseada no próprio nome do *Character Device File*. Por outro lado, a conexão entre o *Character Device File* e o *Character Device Driver* é baseada no número do *Dharacter Device File*, o *device file number*, mais conhecido pelo par `<major, minor>`. A título de exemplo, executando o comando `$ls -l` no diretório `/dev`, podemos observar algo parecido com o seguinte:

```
crw------- 1 root root 5, 1 Jul 16 console
brw-rw---- 1 root disk 3, 0 Oct 6 hda
brw-rw---- 1 root disk 3, 1 Oct 6 hda1
crw------- 1 root tty  4, 1 Oct 6 tty1
crw------- 1 root tty  4, 2 Oct 6 tty2
```

O primeiro caráter indica o tipo de driver: `"c"` indica um *Character Device Driver* e `"b"` um *Block Device Driver*. Os dois números antes da data da última modificação, separados por uma vírgula são, respetivamente, o `major number` e o `minor number`.
O `major number` identifica o controlador associado ao dispositivo. Por exemplo, os dispositivos `/dev/tty1` e `/dev/tty2` são ambos geridos pelo *Character Device Driver* identificado com o número `4`. O `minor number` é usado pelo *Kernel* para identificar exatamente qual o dispositivo referenciado. Estes números são representados pelo tipo `dev_t` (definido em `linux/types.h`) no *Kernel*. Para obtê-los, poderão ser utilizadas as seguintes macros, definidas em `linux/kdev_t.h`:

- `MAJOR(dev_t dev)` - extrai o major number do dev
- `MINOR(dev_t dev)` - extrai o minor number do dev
- `MKDEV(int major, int minor)` - cria um dev apartir do major number e do minor number

Uma das primeiras tarefas a ser executada pelo *Character Device Driver* é a obtenção destes números. Para isso são utilizadas duas funções:

```
int register_chrdev_region(dev_t first, unsigned int cnt, char *name);

int alloc_chrdev_region(dev_t *first, unsigned int firstminor,
unsigned int cnt, char *name);
```

A primeira função regista `ctn` *device numbers*, começando pelo `first`, com o nome `name`. A segunda função aloca dinamicamente um `major number` livre e regista `cnt` *device file numbers*, começando por `<major alocado, firstminor>` e com o nome `name`. Quando não é mais necessário, um *device file number* deve ser libertado utilizando a seguinte função:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Seguidamente, é necessário interligar as operações do *device driver* a estes números. Para isso é utilizada a estrutura *file_operations*.
Cada ficheiro aberto é representado internamente por uma estrutura (*file_structure*) e tem a si associado uma série de funções e atributos, incluindo um campo denominado de *f_op*, que é um ponteiro para a estrutura *file_operations*. O *Virtual File System* (VFS) fica então responsável por fazer a interligação entre as operações no *user-space* e as operações no *kernel-space*.
Passando agora ao esqueleto de um módulo, este possui, no seu estado mais básico, um construtor e um destrutor. O construtor, ou função de inicialização, é responsável por inicializar o dispositivo e inserir o controlador (*device driver*) na estrutura do *Kernel*. A estrutura de inicialização de um módulo assemelha-se ao seguinte:

```
static int __init init_function(void)
{
    /* Initialization code here */
}
module_init(init_function);
```

Não é obrigatório, mas é boa prática declarar a função de inicialização como *static* pois é suposto esta não ser acessível fora do ficheiro em que está a ser declarada. O símbolo `__init` indica ao *Kernel* que esta função apenas será utilizada no instante da inicialização. Isto faz com que, após o módulo ser carregado, a função de inicialização é libertada, deixando livre o espaço em memória que ocupava. A macro `module_init()`, definida no *header* `module.h`, é obrigatória existir. Esta indica qual é a função de inicialização que deve ser chamada quando o módulo está a ser carregado.
Por outro lado existe uma estrutura de remoção do módulo, destrutor, que é responsável por remover as interfaces e restaurar os recursos para o ponto anterior à inserção do módulo. Esta estrutura é definida como:

```
static void __exit exit_function(void)
{
    /* Exit code here */
}
module_exit(exit_function);
```

O símbolo `__exit` indica que esta função é chamada apenas ao remover o módulo. Apenas pode ser chamada quando queremos remover um módulo dinâmico ou quando desligamos o sistema.

3.5 Configuração e compilação do *Kernel*

O primeiro passo a realizar antes de proceder à configuração e posterior compilação do *Kernel* é obter uma versão do mesmo. Existem inúmeros locais onde este pode ser obtido, tais como a sua página oficial, *http://www.kernel.org*, ou através do fabricante da plataforma de desenvolvimento, que muitas vezes disponibiliza todo o código fonte e também os executáveis testados. Esta última opção acaba por ter vantagens quando o tempo de desenvolvimento tem um prazo apertado. A Tabela 3.1 apresenta um exemplo do diretório raíz do *Kernel*. Estes diretórios contêm inúmeros ficheiros de código, configuração e *makefiles*[8] que serão utilizados nas fases de configuração e compilação do *Kernel*.

[8]Ficheiro utilizado para organizar código de compilação.

/arch	/firmware	/kernel	/scripts
/block	/fs	lib/	/security
/crypto	/include	mm/	/sound
/Documentation	init/	/net	/usr
/drivers	/ipc	/samples	/virt

Tabela 3.1: Excerto do diretório raiz do *Kernel*. [14]

Um desses ficheiros, que por defeito está ocultado, é o `dot-config`. A sua estrutura é simplista, contendo apenas uma coleção de opções (separadas em linhas) que posteriormente darão origem a um ficheiro de definições[9], o `autoconf.h`.

Listagem 3.1: Excerto de um ficheiro *.config* [14]

```
...
# USB support
#
CONFIG_USB=m
# CONFIG_USB_DEBUG is not set
# Miscellaneous USB options
#
CONFIG_USB_DEVICEFS=y
# CONFIG_USB_BANDWIDTH is not set
# CONFIG_USB_DYNAMIC_MINORS is not set
# USB Host Controller Drivers
#
CONFIG_USB_EHCI_HCD=m
# CONFIG_USB_EHCI_SPLIT_ISO is not set
# CONFIG_USB_EHCI_ROOT_HUB_TT is not set
CONFIG_USB_OHCI_HCD=m
CONFIG_USB_UHCI_HCD=m
...
```

Na listagem 3.1, que apresenta parte de um ficheiro `.config`, são apresentadas as 3 opções que podem ser definidas na configuração do *Kernel*. A primeira opção, `CONFIG_USB=m`, faz com que o subsistema USB seja compilado como um módulo dinâmico, ou seja, não fará parte da *imagem* do *Kernel* e será inserido mais tarde, após este estar em execução. Caso a opção fosse `=y`, o módulo USB seria compilado estaticamente, ou seja, faria parte da *imagem* do *Kernel*. Caso seja pretendido que um módulo não faça parte do *Kernel*, a linha correspondente é comentada com o caráter `#`[10]. De notar que nem sempre que a opção `=y` é definida

[9] Na linguagem C, denomina-se de *C-Header*.

[10] O texto `"is not set"` é escrito por questões de clarificação.

estamos perante um módulo. Por exemplo, a opção `CONFIG_USB_DEVICEFS=y` permite que outras opções possam também ser configuradas.

Para configurar o *Kernel* para a arquitetura *ARM* por exemplo, executa-se o comando[11]

```
$ make menuconfig ARCH=arm
```

Ao executar este comando, uma janela como a ilustrada na Figura 3.4 irá aparecer. A partir deste ponto, é possível então configurar devidamente o *Kernel*.

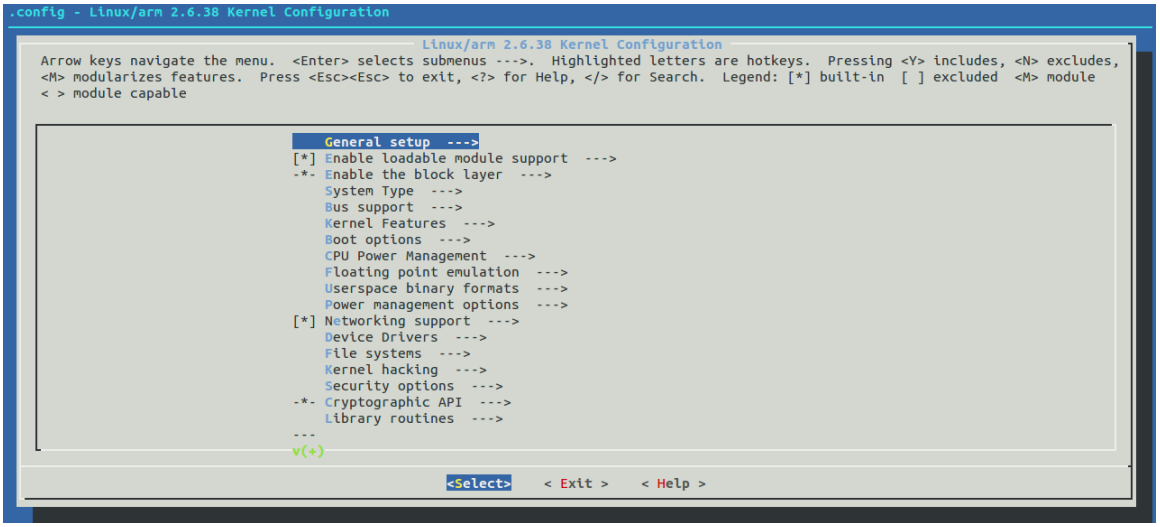


Figura 3.4: Menu de configuração do *Kernel*.

As opções de configuração presentes nesta janela estão definidas num ficheiro denominado de `Kconfig`. Este ficheiro situa-se num subdiretório do diretório `arch/`, cujo nome foi passado como parâmetro em `ARCH=<arquitetura>`. No caso anterior, o diretório seria `arch/arm/`. Para exemplo, a Listagem 3.2 apresenta o conteúdo de um ficheiro `kconfig` para a arquitetura *MIPS*.

Listagem 3.2: Excerto de um ficheiro *kconfig* [16]

```
config MIPS
bool
default y
select HAVE_IDE
select HAVE_OPROFILE
select HAVE_ARCH_KGDB
```

[11] Além do `menuconfig`, podemos utilizar outros editores, tais como o `xconfig` ou o `gconfig`. Todos eles irão produzir o mesmo resultado no ficheiro `.config`.

```
# Horrible source of confusion. Die, die, die ...
select EMBEDDED
select RTC_LIB

mainmenu "Linux/MIPS Kernel Configuration"

menu "Machine selection"

(many, many lines clipped)

source "net/Kconfig"
source "drivers/Kconfig"
source "fs/Kconfig"
source "arch/mips/Kconfig.debug"
source "security/Kconfig"
source "crypto/Kconfig"
source "lib/Kconfig"
```

O processo de configuração passa pela utilização de um editor que ao ser invocado irá numa primeira fase ler o ficheiro `.config` existente, aplicar as definições do ficheiro `kconfig` (encontrado no diretório da arquitetura passada como parâmetro) e finalmente irá mostrar todo o menu de opções. Assim que a configuração é terminada, as alterações são novamente guardadas no ficheiro `.config` [16]. Estando o *Kernel* configurado, o passo seguinte é a sua compilação. O resultado desta operação é uma *imagem* comprimida do *Kernel* denominada de `zImage`[12] e que pode ser encontrado no diretório `/arch/<arquitetura>/boot`. O comando utilizado para proceder então à compilação do *Kernel* é (para o caso de uma arquitetura *ARM*):

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- zImage
```

O primeiro argumento a passar é a arquitetura para o qual a compilação será efetuada. Depois indicamos o *cross-compiler* a ser utilizado e o tipo de *imagem comprimida* que queremos que resulte da compilação, neste caso a `zImage`. Na fase final da compilação, serão mostradas as seguintes mensagens,

```
Kernel:   arch/arm/boot/Image is ready
SHIPPED   arch/arm/boot/compressed/lib1funcs.S
AS        arch/arm/boot/compressed/lib1funcs.o
LD        arch/arm/boot/compressed/vmlinux
OBJCOPY   arch/arm/boot/zImage
Kernel:   arch/arm/boot/zImage is ready
```

[12]Esta *imagem* é a mais utilizada em sistemas embutidos. No entanto, o resultado da compilação pode originar outras, como a `bzImage` que é utilizada em sistemas *X86*.

indicando que o *Kernel* foi corretamente compilado e o ficheiro que pretendíamos (`zImage`) se encontra no diretório `arch/arm/boot/zImage`.

3.6 Sistema de Ficheiros Raiz

O *Sistema de Ficheiros Raiz* (*Root File System*) refere-se ao sistema de ficheiros que é montado no topo da hierarquia de um sistema de ficheiros, denominado por `"/"`. É o primeiro sistema de ficheiros a ser *montado* e, posteriormente, outros sistemas de ficheiros podem ser *montados* na mesma hierarquia. O *Linux* espera encontrar, em determinados diretórios, programas e utilitários utilizados para arrancar o sistema, inicializar serviços (ligações de rede e a *consola*), carregar os controladores de dispositivos e *montar* outros sistemas de ficheiros. Daí a importância da existência do *Sistema de Ficheiros Raiz*, assim como a sua organização interna.
Por forma a normalizar a criação dos sistemas de ficheiros, foi criado o *File-System Hierarchy Standard* (FHS), que estabelece um conjunto de regras a seguir no desenvolvimento dos mesmos. A Tabela 3.2 apresenta os diretórios que devem fazer parte do diretório raiz `"/"`:

Diretório	**Conteúdo**
/bin	Binários essenciais ao utilizador
/boot	Ficheiros estáticos utilizados pelo *bootloader*
/dev	Dispositivos e outros ficheiros especiais
/etc	Ficheiros de configuração e inicialização do sistema
/home	Diretório base dos utilizadores
/lib	Bibliotecas de código (biblioteca C) e módulos do *Kernel*
/media	Diretório onde são *montados* os dispositivos removíveis
/mnt	Diretório onde são *montados* sistemas de ficheiros temporários
/opt	Pacotes de software adicionais
/proc	Sistema de ficheiros virtual utilizado pelo *Kernel*
/root	Diretório base do utilizador raiz
/sbin	Executáveis do administrador de sistema
/sys	Sistema de ficheiros virtual para informações e controlo do sistema
/tmp	Ficheiros temporários
/usr	Programas do utilizador
/var	Ficheiros temporários de configuração e registos de dados

Tabela 3.2: Sistema de ficheiros raiz segundo o *FHS*. [16]

Uma vez que os sistemas embutidos têm limitação no armazenamento, alguns diretórios recomendados pelo *FHS* podem ser removidos e assim obter um sistema de ficheiros mais simplificado, mas igualmente funcional. Preencher a *árvore* de diretórios com os ficheiros de configuração/inicialização e executáveis necessários é uma tarefa árdua. Como já referido, o *Kernel* necessita de determinados ficheiros e programas para que possa configurar corretamente o sistema, montar o sistema de ficheiros e inicializar determinados serviços. Uma ferramenta que pode ser utilizada para obter tais ficheiros é o *Busybox* (`www.busybox.net/`). Denominado de *"The Swiss Army Knife of Embedded Linux"*, o *Busybox* é uma substituição compacta e eficiente das tradicionais aplicações e utilitários que encontramos num sistema *Linux*, como o `ls`, `cat`, `cp`, `dir`, `dmesg`, `kill`, `mount`, `unmount`, `ifconfig`, `netstat`, `route`, entre muitas outras. Esta aplicação serve de ponto de partida para a construção de um sistema de ficheiros.

3.6.1 Busybox

O *Busybox* (`www.busybox.com`) é uma ferramenta altamente modular, que pode ser configurada de acordo com as necessidades do projeto para o qual vai ser utilizada, por forma a fornecer um conjunto de utilidades tradicionalmente encontradas nos sistemas *Linux*. Apresenta um menu de configuração muito idêntico ao utilizado para configurar o *Kernel* (Figura 5.12).

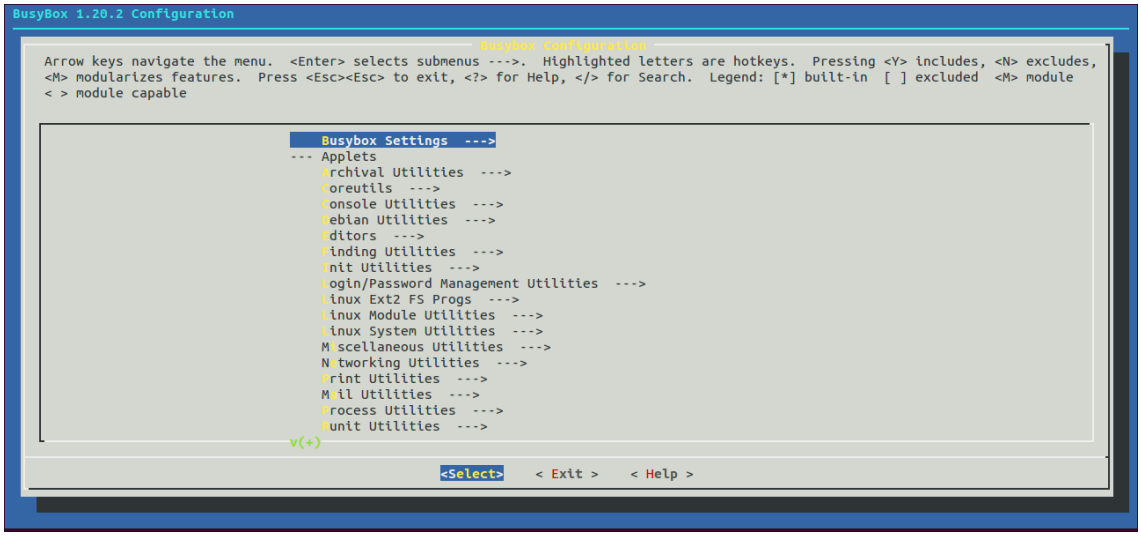


Figura 3.5: Menu de configuração do *Busybox*

A Figura 3.6 apresenta um sistema de ficheiros minimalista, desenvolvido com a utilização do *Busybox*. Com este pequeno sistema de ficheiros, é possível ter um sistema totalmente funcional, com todos os comandos que foram previamente habilitados na configuração do *Busybox*. Fazem parte da sua estrutura 5 diretórios e 8 ficheiros[13]. O diretório `bin/` aloja o

[13]Na realidade serão 12 ficheiros, pois os diretórios são um tipo especial de ficheiro para o *Linux*.

ficheiro executável do `busybox` e um atalho, denominado de `sh`, que aponta para o `Busybox`. O diretório `dev/` contem apenas um nó de dispositivo, a `consola`, que servirá para fazer interface com um utilizador. O ficheiro `rcS` é o *script* de inicialização que o *Busybox* utiliza no arranque. No último diretório, `lib/`, temos duas bibliotecas que contêm as diferentes funções do *C*, tal como a `printf()`, que muitos programas utilizam. Essas bibliotecas são a *glibc* (*libc-2.3.2.so*) e a *Linux Dynamic Loader* (*ld-2.3.2.so*) [14].

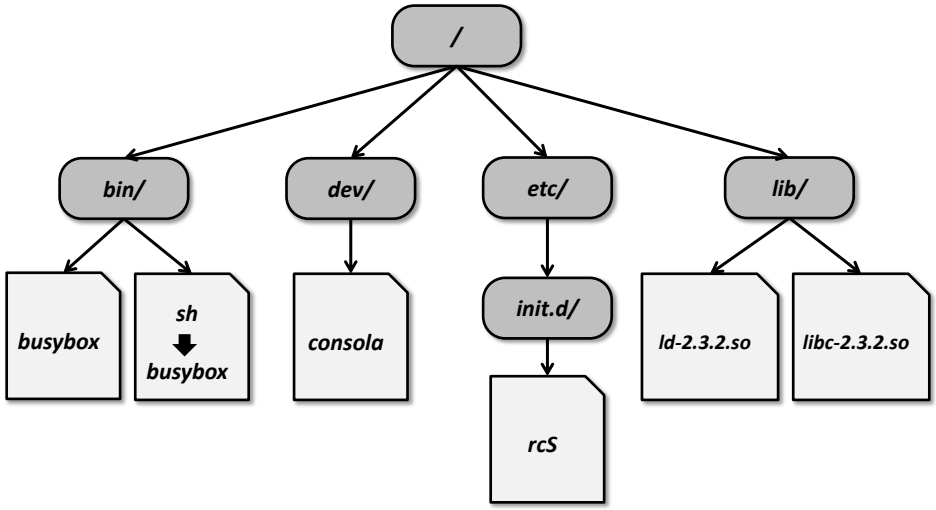


Figura 3.6: Sistema de ficheiros minimalista.

A Listagem 3.3 apresenta as últimas linhas de código (presentes no ficheiro `/init/main.c`) executadas pelo processo de inicialização do *Kernel*. Pelo menos uma das instruções, `run_init_process()` tem que funcionar e caso não tenha sucesso, será chamada a função `panic()` que parará a execução do *Kernel* e o sistema não funcionará.

Listagem 3.3: Passos finais da inicialização do *Kernel* [16]

```
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
```

A utilização do atalho *bin/sh*, presente no sistema de ficheiros da Figura 3.6, é agora compreensível: faz com que o *Busybox* seja o primeiro processo a ser executado pelo *Kernel*. Após ser lançada a sua execução, o *Busybox* procura pelo ficheiro de inicialização `rcS`. Estando a inicialização completa, é apresentada a mensagem `"Please press Enter to activate this console"` na consola principal do sistema.

3.7 *Bootloader*

O *Bootloader* é o primeiro programa a ser executado num *Sistema Embutido Linux*. A sua principal tarefa é inicializar o processador (e, se necessário, um conjunto de outros periféricos), carregar o *Kernel* na memória RAM e passar-lhe o controlo de todo o sistema. Atualmente, os *bootloaders* também são capazes de fazer a gestão de memórias *Flash* (criar *segmentos*[14] e escrever dados nos mesmos, tais como um *sistema de ficheiros* ou uma *imagem* do *Kernel*.).
Existem muitos *bootloaders* para sistemas *Linux*, sendo que alguns deles têm dependência da arquitetura do processador. São exemplos o *RedBoot* (`http://sourceware.org/redboot/`), o *U-Boot* (`http://www.denx.de/wiki/U-Boot`) (ambos multi-arquitetura) e o *Yamon* (`http://www.mips.com/products/system-software/yamon/`) que é usado na arquitetura *MIPS*. Outra opção é utilizar o *bootloader* que é disponibilizado pelo fabricante da plataforma de desenvolvimento.
A Figura 3.7 ilustra os passos desde a primeira vez em que o sistema é ligado até ao primeiro *processo* a ser executado.

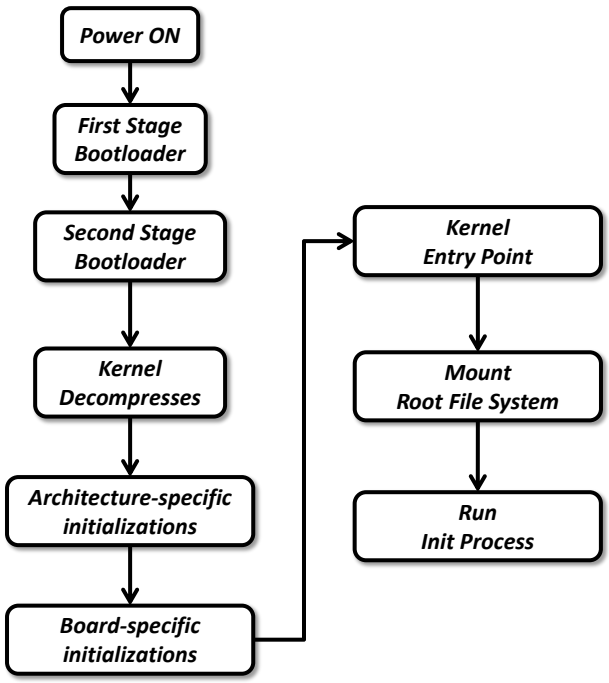


Figura 3.7: Inicialização do *Linux*. [16]

Quando o sistema inicia, o processador lê o conteúdo de uma posição de memória definida pelo fabricante (este passo é normalmente chamado de *first-stage bootloader*) e salta a sua

[14] Áreas de armazenamento da memória *Flash*

execução para essa localização[15]. O código agora a ser executado é o *bootloader* propriamente dito (denominado de *second-stage bootloader*). Neste ponto, após as configurações e inicializações iniciais, o *bootloader* copia a *imagem* do *Kernel* (que está armazenada numa seção da memória *Flash*) para a memória RAM e informa ao processador (através do *instruction pointer*) o endereço do primeiro executável do *Kernel*. A partir deste momento, o *Kernel* passa a controlar todo o sistema e o *bootloader*, assim como todas as configurações/inicializações feitas por este, deixam de existir. Ao adquirir o controlo, o *Kernel* executa as novas inicializações/configurações do sistema até atingir o *kernel entry point*, que é código independente da arquitetura. De seguida, é *montado* o sistema de ficheiros e executado o programa /init, que é o primeiro programa que o *Kernel* tenta executar.

Este capítulo apresentou, embora de forma superficial, os sistemas *Linux*. O conhecimento e compreensão de muitos dos conceitos aqui abordados são de extrema importância no desenvolvimento do trabalho desta Dissertação, pois um dos objetivos da mesma é que o sistema a desenvolver seja capaz de executar um sistema operativo baseado em *Unix*.

[15]O conteúdo lido é a posição de memória onde se encontra a primeira instrução do *bootloader*.

Capítulo 4

Plataforma de desenvolvimento

Neste capítulo será apresentado o conjunto de ferramentas utilizadas nesta Dissertação. Em primeiro lugar, é apresentada a plataforma *FriendlyArm Tiny6410*, que servirá de base a todo o projeto da IHM e seguidamente serão apresentadas as ferramentas de programação e compilação utilizadas no desenvolvimento das diferentes peças de *software/firmware*.

4.1 FriendlyArm Tiny6410

A plataforma de desenvolvimento *FriendlyArm Tiny6410* (Figura 4.1) é uma solução de alto desempenho, baseada no microcontrolador *S3C6410X da Samsumg* [17].

Figura 4.1: Plataforma de desenvolvimento Tiny6410

Integra tanto *hardware*, como um pacote completo de *software* (testado e funcional), o que faz com que seja uma referência no desenvolvimento de sistemas embutidos para aplicações industriais, multimédia ou apenas para fins educacionais. Suporta, por exemplo, o *Linux 2.6.38*, *Android 2.3* e *Windows CE 6.0*. É constituída por dois módulos: o módulo de processamento *Tiny6410 Core*, que contém o microcontrolador, a memória RAM, a memória *Flash*, e o módulo de periféricos *Tiny6410 Motherboard*.
A utilização desta plataforma no pré-desenvolvimento do sistema requerido, deveu-se ao facto de que esta encontrava-se disponível na empresa *Micro I/O* e possuía os requisitos e as caraterísticas necessárias para ser utilizada como tal.

4.1.1 Tiny6410 Core

Este módulo é o núcleo central de todo o sistema. Combina o microprocessador *ARM11 S3C6410X* da *Samsung* com a memória RAM, memória *Flash* e toda a eletrónica de regulação da tensão numa pequena placa com 64 x 50 mm de área (Figura 4.2).

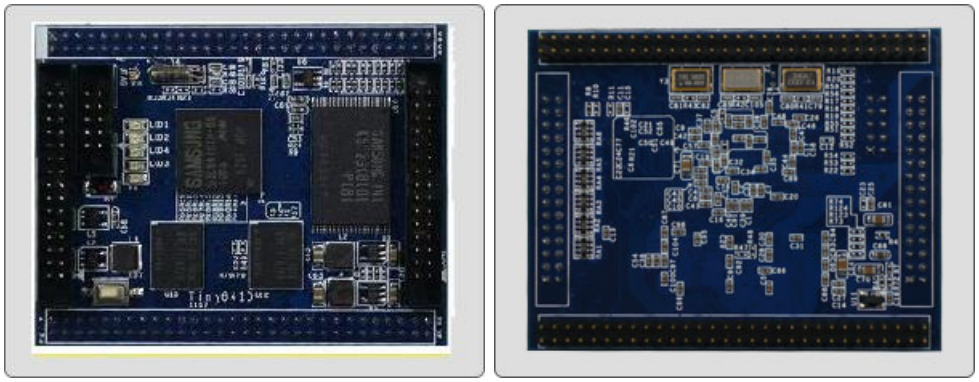


Figura 4.2: *Tiny6410 Core*

O *S3C6410X* é um microcontrolador de baixo custo, baixo consumo e inigualável desempenho gráfico [17], desenvolvido especialmente para sistemas que têm baterias como fonte de energia. Possui uma arquitetura RISC de 16/32-bits e inclui diversos periféricos e aceleradores de hardware, o que o torna especialmente útil em aplicações com processamento de áudio e vídeo. Em relação à memória, este módulo contém até 256 MBs de memória RAM e 2 GBs de memória *Flash* de tecnologia *NAND*.

A Tabela 4.1 resume as características gerais do *Tiny6410 Core Board*.

CPU	Samsung S3C6410A, run at 533Mhz ARM1176JZF-S, up to 667Mhz
RAM	256 DDR RAM
Flash	2GB Nand Flash
Interface	4 x User Leds
	10 pin 2.0mm space Jtag connector
	Reset button on board
Conector	2 x 60 pin 2.0mm space DIP connector
	2 x 30 pin 2.0mm space GPIO connector
Power	Supply Voltage from 2.0V to 6V
Size	64 x 50 x 12mm (L x W x H)
OS Support	Windows CE 6
	Linux 2.6.38
	Android 2.3
	Ubuntu 9.10

Tabela 4.1: Caraterísticas do *Tiny6410 Core Board*.

4.1.2 Tiny6410 Motherboard

Esta placa (Figura 4.3) combina um conjunto de interfaces que permite o rápido desenvolvimento de aplicações.

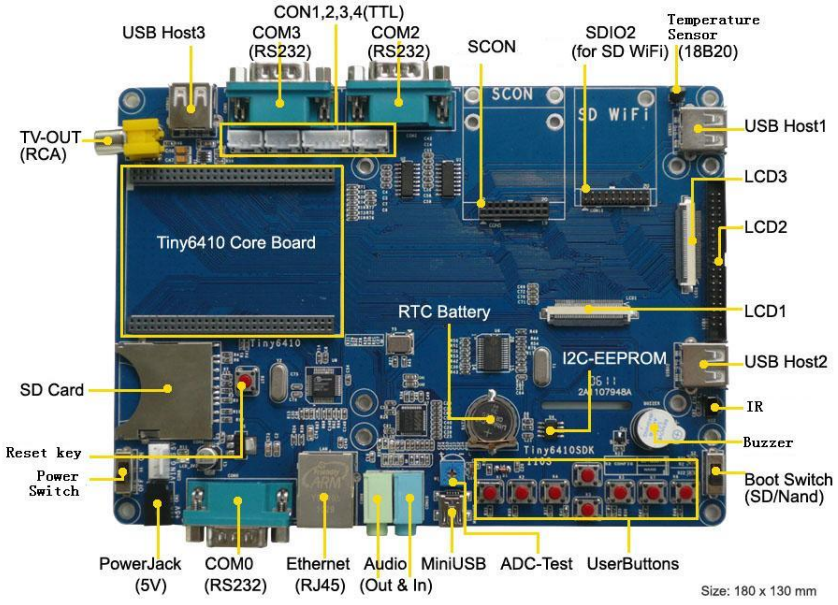


Figura 4.3: *Tiny6410 Motherboard*

Possui interface para ecrã com *touchscreen*, *Ethernet*, *RS-232*, *USB*, *áudio*, *cartão SD*, entre outros. Podem ser utilizados ecrãs nos tamanhos pré definidos de 3.5, 4.5 e 7 polegadas. Juntamente com a plataforma *FriendlyArm Tiny6410*, é fornecido um *Software Development Kit* (SDK) que permite, de uma maneira mais eficiente, o desenvolvimento de novas aplicações. No Anexo A estão descritos os passos iniciais para a utilização destas ferramentas.

4.2 Software de desenvolvimento

4.2.1 *Cross Development Toolchain*

Uma *toolchain* consiste num conjunto de ferramentas de programação que são utilizadas para criar um ou vários ficheiros executáveis numa determinada arquitetura de processador. Normalmente estas ferramentas são usadas em cadeia (como o próprio nome indica), ou seja, o resultado de saída de uma torna-se na entrada da outra e assim sucessivamente. Tipicamente uma *toolchain* contém um editor de texto para editar o código fonte, um compilador, um *linker*, as bibliotecas que fornecem um conjunto de funções, macros e/ou outro código auxiliar e um depurador de código. O termo *cross* indica que do processo de compilação, resulta um ficheiro executável numa arquitetura diferente daquela em que foi originalmente compilado.

Nesta Dissertação foi utilizada a *cross toolchain* disponibilizada pelo fabricante *FriendlyArm*, a `arm-none-linux-gnueabi-`. A sua instalação está descrita no Anexo A.

4.2.2 NetBeans

O NetBeans (`https://netbeans.org/`) é um ambiente de desenvolvimento integrado (*Integrated Development Environment* (IDE)) gratuito e de código aberto para desenvolvedores de software nas linguagens *Java*, *C*, *C++*, *PHP*, entre outras. Projetado de forma reutilizável, visa simplificar o desenvolvimento e aumentar a produtividade, pois reúne numa única aplicação as funcionalidades de escrita, compilação e depuração de código.
Desenvolvido na linguagem *Java*, é independente da plataforma e pode ser executado em qualquer sistema operativo que suporte a máquina virtual do java (*Java Virtual Machine* (JVM)).
Como compilador associado a este IDE, foi utilizado o *cross-compiler* `arm-none-linux-gnueabi-gcc`, fornecido pelo fabricante *FriendlyArm*.

4.2.3 Qt Creator

O *Qt Creator* é um ambiente de desenvolvimento integrado (IDE) que faz parte do conjunto de ferramentas (SDK) do *Qt* (`www.qt-project.org`). Baseado na linguagem *C++*, o *Qt* é uma aplicação multi-plataforma capaz de ser executada em *Windows*, *Linux* ou *Mac OS* e é considerada uma das aplicações mais utilizadas de desenvolvimento gráfico em *Linux*. O *Qt Creator* inclui um editor e integra o *Qt Designer*, possibilitando assim o desenvolvimento de

aplicações gráficas a partir de *widgets* próprios. Resumindo, esta aplicação possibilita a escrita de código, a sua depuração, compilação, controlo de versões, consulta de documentação, entre outras características que são de esperar num IDE moderno. Para o processo de compilação, o *Qt* utiliza duas ferramentas: o *QMake*, que é um gerador automático de *Makefiles* e o *cross-compiler* `arm-none-linux-gnueabi-g++`, que faz parte do conjunto de compiladores fornecido pelo fabricante *FriendlyArm*.

4.2.4 MPLAB X

O *MPLAB X*(`http://www.microchip.com/mplabx/`) é um IDE gratuito, disponibilizado pela *Microchip*, utilizado no desenvolvimento de aplicações para as diferentes famílias de microcontroladores e controladores de sinais digitais, comercializados pela mesma. A sua implementação é baseada no IDE da *Oracle*, o *NetBeans*, o que torna a sua utilização muito fácil e intuitiva, especialmente para que já está familiarizado com o *NetBeans*. Como compilador, existem diferentes possibilidades, dependendo da arquitetura interna do microcontrolador que queremos utilizar. Nesta Dissertação, foi utilizado um microcontrolador da família *dsPIC33* [18] e para tal, o compilador correspondente utilizado foi o *MPLAB C30*.

4.2.5 Eagle

O *EAGLE* (*Easily Applicable Graphical Layout Editor*) é uma ferramenta de desenho eletrónico que contém um editor de esquemáticos, utilizado no desenho de diagramas eletrónicos, um editor de desenho de *Printed Circuit Board* (PCB), um auto-router, que permite interligar todos os componentes do projeto, de acordo com o esquemático, um módulo *Computer-Aided Manufacturing* (CAM) que gera ficheiros de produção das PCBs (por exemplo, ficheiros GERBER ou ficheiros de *drill*) e *Bill of Material* (BOM), que tal como o nome indica, contém a lista de todos os componentes do projeto.

Capítulo 5

Implementação

Neste capítulo serão descritos os passos da implementação de todo o projeto da *Interface Homem-Máquina*. Em primeiro lugar será discutida a implementação do *Hardware*, seguido da discussão da implementação do *Firmware* e posteriormente serão apresentados os testes realizados para verificação da robustez e coerência de todo o sistema. Finalmente será discutida a implementação de um ambiente gráfico. A Figura 5.1 apresenta os atores do sistema geral de aplicação da IHM. O utilizador, através da IHM, tem acesso ao controlador, podendo consultar ou alterar as diferentes variáveis de controlo.

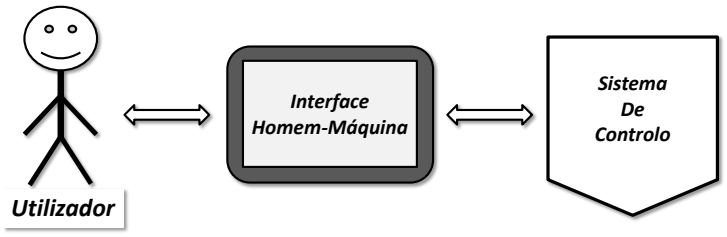


Figura 5.1: Atores do sistema

Um cenário possível e que será objeto de desenvolvimento nesta Dissertação, é a possibilidade do utilizador poder controlar a temperatura de uma sala de estar, através do acionamento de diferentes bombas e válvulas de água pré-aquecida por painéis solares e também poder visualizar a evolução da temperatura no tanque da água ou no painel solar.
De acordo com os objetivos apresentados no Capítulo 1 e após um estudo detalhado da plataforma de desenvolvimento *FriendlyArm*, foi elaborado um diagrama de blocos que ilustra, de uma forma genérica, os diferentes módulos que irão fazer parte do produto final (Figura 5.2). O *Tiny6410 Core* destaca-se como núcleo central do sistema. Como referido anteriormente, o desenvolvimento de um sistema embutido parte, por vezes, de soluções já existentes no mercado. Esta abordagem vem reduzir o tempo e o custo de desenvolvimento, pois a solução encontrada (*Tiny6410 Core*) é uma solução fiável, testada e funcional.

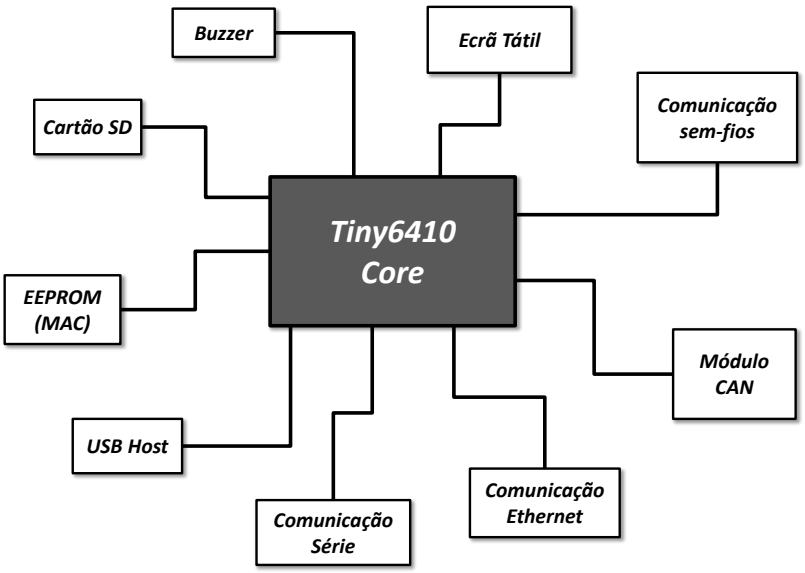


Figura 5.2: Diagrama de blocos do sistema

5.1 Hardware

A Figura 5.3 apresenta o diagrama de blocos geral do sistema com mais detalhe, onde estão identificadas as diferentes soluções de interface para cada módulo. No que se segue, serão discutidas, individualmente, cada uma dessas soluções.

5.1.1 Ecrã tátil

A utilização de um ecrã tátil num projeto desta natureza torna-se indispensável, pois muitas das aplicações práticas desta IHM serão sem recurso a teclados ou ratos, sendo esta a única forma do utilizador interagir com o sistema, em termos de envio de informação. Existem duas tecnologias possíveis de serem utilizadas na interface tátil: *resistiva* e *capacitiva*.
A interface resistiva é constituída por duas camadas separadas por uma pequena abertura preenchida de ar. A camada superior (aquela que o utilizador irá pressionar) é normalmente feita de um plástico flexível, revestido na parte inferior por um material condutor (*Indium Tin Oxide* (ITO)). A camada inferior é um vidro ou plástico duro, também revestido, na parte superior, por ITO. Quando pressionada, a camada superior entra em contato com a camada inferior e ambos os revestimentos de ITO tocam-se, alterando a resistência elétrica. É a medição desta resistência que determina a posição do toque.
A interface capacitiva consiste em duas camadas separadas de vidro, revestidas de igual modo com ITO. Uma vez que o corpo humano é condutor de cargas elétricas, ao tocar num ecrã

capacitivo, o campo eletrostático irá variar e a continua monitorização dessa variação determinará a posição do toque.

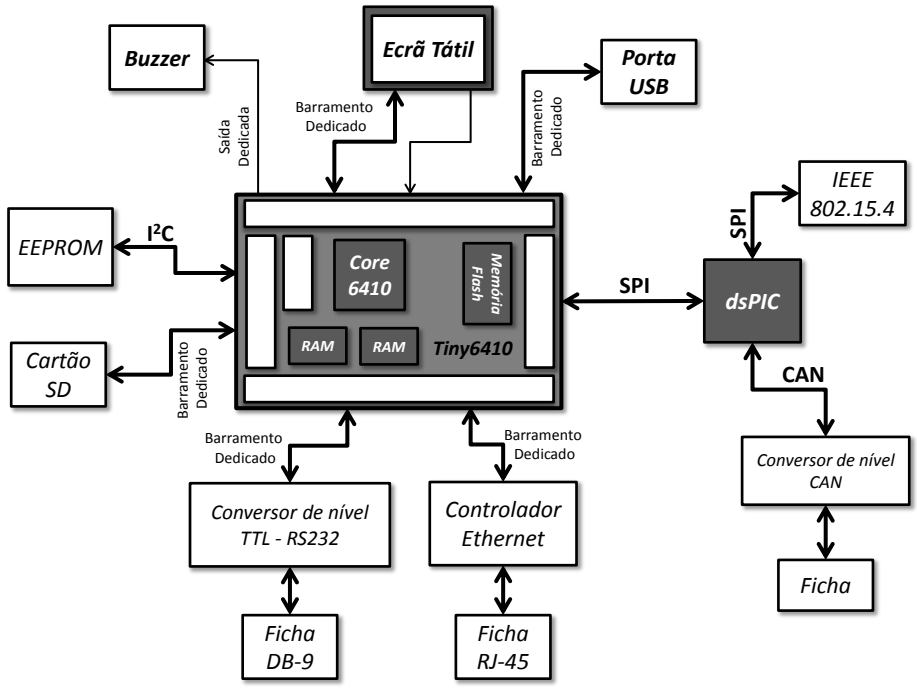


Figura 5.3: Arquitetura geral do sistema

Apesar de não suportar multi-toque, foi optado neste projeto a utilização da tecnologia *resistiva*, pois apresenta uma maior resistência aos ambientes em que vai operar (poeiras e humidade), tem um menor custo de produção e pode ser utilizada pelos dedos, mesmo com luvas, ou por uma caneta. Posto isto, foi utilizado o *LCD S70*, da *FriendlyArm*, com 7 polegadas de diagonal, uma resolução de 800x480 pixels e uma interface tátil resistiva. Este foi ligado ao controlador de imagem do microprocessador incluído no *Tiny6410 Core* (*Samsung S3C6410X*) através de um barramento dedicado e disponível num conjunto de pinos presentes no *Tiny6410 Core*. A parte tátil deste display foi diretamente ligada a um porto de entrada do microprocessador (pino *TS*, do *Tiny6410 Core*), sendo controlado por um *device driver* desenvolvido pela *FriendlyArm*. No entanto, estão também disponíveis 4 pinos (`TSXM`, `TSXP`, `TSYM` e `TSYP`) ligados diretamente às ADCs internas do microcontrolador, permitindo assim a utilização de outros sistemas táteis, com diferentes controladores.

5.1.2 *USB Host*

Uma porta USB em modo *Host* permite a ligação de diferentes dispositivos, tais como, discos rígidos externos, leitores de cartões, teclados ou ratos. Embora o seu uso seja muito pontual, a inclusão de uma porta USB *Host* neste projeto poderá trazer algumas facilidades no que

respeita a depuração de erros ou manutenção do sistema.
O controlador *USB Host* do *Samsung S3C6410X* é capaz de suportar duas portas compatíveis com a Revisão 1.1 do USB e está acessível através dos pinos *USB_DN* e *USB_DP* do *Tiny6410Core*.

5.1.3 UART e RS-232

A comunicação série é indispensável num sistema embutido, especialmente num *Sistema Embutido Linux*. Através deste tipo de comunicação é possível, por exemplo, aceder a um terminal ou consola e ter acesso ao sistema de ficheiros ou aos tradicionais comandos *UNIX*[1]. Neste projeto são utilizadas duas portas série, sendo a primeira utilizada por defeito para a consola do *Linux* e a segunda está disponível para ligação ao controlador *UNISOL*.
Os dados enviados pelo microcontrolador, através da *UART* estão no padrão elétrico *Transistor-Transistor Logic* (TTL), ou seja, o nível lógico `'0'` é representado por sinais com uma tensão entre os *0 volts* e os *0.8 volts* e o nível lógico `'1'` é representado por sinais com uma tensão entre os *2 volts* e *Vcc*[2]. Estes sinais são utilizados na comunicação entre diferentes microcontroladores ou outros dispositivos, mas com limitação na distância entre os mesmos.
Para que seja possível a comunicação a maiores distâncias, é necessário converter os níveis *TTL* para níveis *RS-232*. Neste caso, os sinais válidos são, tipicamente, de *-15 volts* a *-3 volts* para o nível lógico '1' e de *3 volts* a *5 volts* para o nível lógico '0'. Para fazer a conversão dos níveis *TTL* para níveis *RS-232*, é necessário a utilização de um conversor (Figura 5.4).
O conversor utilizado foi o *MAX3233E* da *Maxim* [19], com dois canais de conversão. Este conversor inclui no seu interior os condensadores utilizados no *charge-pump*[3], o que facilita a implementação tanto nível de espaço como a nível de número de componentes.

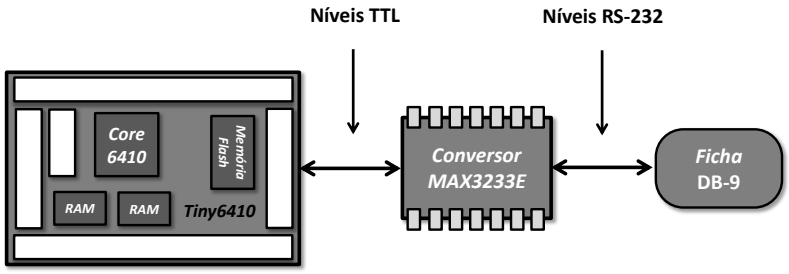


Figura 5.4: Diagrama da porta série do sistema

[1] De acordo com a instalação feita.
[2] Vcc corresponde à tensão de alimentação do microcontrolador. Tipicamente é de *3.3 volts* ou *5 volts*.
[3] O circuito de *charge-pump* é responsável por gerar as tensões de alimentação necessárias para a imposição dos níveis do *RS-232*.

5.1.4 Ethernet

A inclusão de uma porta *Ethernet* neste sistema, amplia o número de aplicações a dar ao mesmo: faz com que seja possível ligar-se a uma rede *TCP/IP* ou ligar-se a um *Controlador Lógico Programável* (CLP), correndo um protocolo como o *ModBus*.
Foi incluída uma porta *Ethernet* neste projeto com vista a aceder ao *Sistema de Ficheiros*, durante a fase de desenvolvimento, através do protocolo *telnet*, sendo este um método de trabalho mais rápido e eficaz, comparativamente à utilização da comunicação série RS-232.
A plataforma *FriendlyArm Tiny6410* utiliza o controlador *DM9000AEP* da *Davicom*. Como este controlador está descontinuado, foi utilizado neste projeto um substituto, o controlador *DM9000BEP* [20]. Este controlador foi escolhido pois, após um estudo do mesmo, foi verificado que este é compatível com o anterior a nível de ligações elétricas. Quanto ao *firmware*, o *device driver* do *Linux* responsável pela gestão da família de controladores *DM9000* está preparado para o reconhecer automaticamente, não sendo necessário qualquer desenvolvimento adicional de código.

5.1.5 Buzzer

Esta interface é provavelmente a mais simples do sistema. Devido à sua capacidade sonora, foi optada pela sua utilização essencialmente para casos de alertas ou para depuração. A sua implementação apenas utiliza uma das saídas do *Tiny6410 Core* e a sua ativação/desativação é controlada por software, através de um *device driver* desenvolvido especialmente para o efeito.

5.1.6 Cartão SD

Um *bootloader* proprietário da *FriendlyArm* vem instalado por defeito na memória *Flash* do *Tiny6410 Core*. Este *bootloader* permite arrancar[4] o sistema através de um cartão SD ou transferir o sistema operativo e/ou o sistema de ficheiros do cartão SD para a memória *Flash*. Assim, a utilização de um cartão SD torna o desenvolvimento de um sistema embutido mais rápido e eficaz. O cartão SD pode também ser utilizado como meio de armazenamento adicional, como por exemplo, de ficheiros de *log*.
Foi optado neste projeto a utilização de um cartão microSD, por questões de espaço e de preço, pois tem-se popularizado devido à utilização em dispositivos móveis. A sua implementação é de todo idêntica à utilizada na plataforma *FriendlyArm Tiny6410*.

[4]De notar que é preciso "informar"o microprocessador qual a região de memória que é utilizada para fazer o arranque do sistema.

5.1.7 EEPROM

Uma possível industrialização deste projeto justificou a utilização de uma *Electrically-Erasable Programmable Read-Only Memory* (EEPROM). A sua principal aplicação é armazenar numa posição específica o *MAC Address*, que é um número físico único associado à interface de comunicação *Ethernet*. E para garantir a sua unicidade, existem no mercado memórias que já o trazem gravado internamente. Neste projeto foi utilizada a memória *24AA02E48* da *Microchip* [21]. Esta memória, que comunica através do protocolo série *Inter-Integrated Circuit* (I2C), possui *2 Kbits* de espaço de armazenamento, que pode também ser utilizada, especialmente para casos de armazenamento de dados críticos.

5.1.8 Microcontrolador dsPIC

O *Kernel 2.6.38* utilizado neste projeto suporta a interface *CAN*, tratando o dispositivo controlador (por exemplo, o *MCP2515* da *Microchip*) como um dispositivo de rede, de maneira muito similar a uma interface *Ethernet*. Do mesmo modo, suporta a interface de comunicação *IEEE 802.15.4* e possui *device driver* para o *MRF24J40* (utilizado neste projeto e introduzido mais adiante). No entanto, ser o *Kernel* responsável por fazer a gestão destes 2 módulos era uma solução não testada, o que seria um risco desenvolver toda a eletrónica especificamente para a mesma. Foi então ponderada uma outra solução, mais fiável, testada e funcional. Assim, para fazer de interface entre o *Tiny6410 Core* e os módulos de comunicação *IEEE 802.15.4* e CAN, foi utilizado um microcontrolador adicional, o *dsPIC33FJ64MC506* da *Microchip*. Esta solução justifica-se pelo facto de existir um conjunto de rotinas e bibliotecas de software implementadas em microcontroladores da mesma família, na empresa onde foi realizada esta Dissertação, a *Micro I/O* (`htt://www.microio.pt`). Com isto, é reaproveitado código já desenvolvido e poupado tempo de desenvolvimento.
A comunicação entre o *Tiny6410 Core* e o *dsPIC* é feita através do protocolo SPI, um protocolo amplamente utilizado na comunicação entre circuitos integrados. Em alternativa, poderia ter sido utilizado o protocolo I2C, igualmente popular. Apesar do SPI utilizar quatro linhas de comunicação (podendo apenas ser três, no caso em que apenas comunicam dois dispositivos) mais uma linha adicional por cada novo dispositivo adicionado (denominada de *chip select*) em comparação com o I2C que utiliza apenas duas linhas, este protocolo revelou-se adequado para este projeto, uma vez que o número de linhas disponíveis era suficiente, para além de ser mais simples de implementar e atingir uma velocidade de comunicação superior (até 10 *Megabits por segundo* (Mbps), em comparação com o I2C que no máximo atinge 3.5 Mbps) [22].

Comunicação 802.15.4

A comunicação sem fios escolhida para este projeto é baseada na norma *IEEE 802.15.4* e vem implementada num módulo da *Microchip*, o *MRF24J40MA* [23]. Este módulo caracteriza-se

pela sua antena integrada, baixo custo e baixo consumo. Das bibliotecas referidas anteriormente, existem disponíveis um conjunto de abstrações para a configuração e utilização deste módulo. Outro motivo que justifica a escolha desta tecnologia vem do facto de que está a ser amplamente utilizada em dispositivos para automação residencial, como sensores de movimento ou atuadores de estores e iluminação.

Figura 5.5: Módulo *MRF24J40MA*.

Comunicação CAN

O *dsPIC33FJ64MC506* possui um módulo CAN capaz de implementar os protocolos *CAN A/B* especificados pela *Bosch* [24]. Para que seja possível a utilização deste protocolo é necessária a interligação entre o controlador interno do *dsPIC33FJ64MC506* e um conversor de nível capaz de fazer a interface com o meio físico. O conversor de nível escolhido foi o *MCP2551* da *Microchip*.

5.1.9 Fonte de alimentação

Apesar de não estar presente na arquitetura geral do sistema, a fonte da alimentação de um sistema é um elemento importante: é ela que irá alimentar todos os circuitos. Mas para tal, é necessário cumprir uma série de requisitos, como as tensões e a corrente máxima que deve fornecer. A definição destes requisitos foi realizada no final do desenvolvimento, pois só com a aplicação final é que foi possível aferir o exato consumo elétrico da *Interface Homem-Máquina*, pois este dependia da carga de processamento do *Central Processing Unit* (CPU) e do funcionamento dos restantes periféricos durante a execução da aplicação.

5.1.10 Printed Circuit Board

Escolhidos todos os componentes a utilizar neste projeto, foi desenhado o esquema elétrico da *Interface Homem-Máquina* na ferramenta de desenho *Eagle*, que pode ser encontrado no Anexo B. A placa de circuito impresso (*Printed Circuit Board*) foi também desenhada utilizando uma ferramenta própria para o efeito, disponibilizada igualmente no *Eagle*. Adicionalmente, foi utilizada uma ferramenta gratuita, *POVRAY* (`www.povray.org`), capaz de

traduzir a informação do *Eagle* para obter uma imagem em três dimensões. Esta imagem (Figura 5.6) fornece uma perceção do produto final diferente daquela que obtemos com o *Eagle* e revelou-se muito útil na obtenção do resultado final[5].

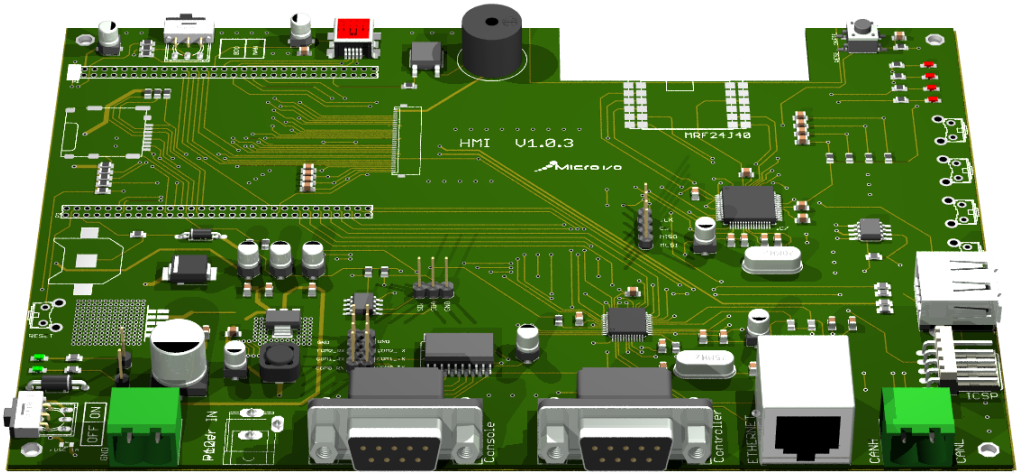


Figura 5.6: Projeção 3D da placa final.

A disposição dos componentes na placa está feita por forma a possibilitar uma montagem mais fácil e simples na vertical (por exemplo, numa parede).

Todas as entradas que requerem a utilização de cabos, como a entrada de alimentação, *Ethernet*, *CAN* e *RS-232*, encontram-se na parte inferior da placa, o facilita a ligação dos cabos e evita que estes mesmos quebrem, pois se ficassem na parte superior, tinham tendência a dobrar, devido à ação da gravidade.

Na zona inferior esquerda estão concentrados todos os componentes respetivos à alimentação (reguladores e condensadores de filtragem). Na parte superior, estão montados os suportes para o *Tiny6410 Core* e por baixo deste, está o *socket* para o cartão *microSD*. A ficha de ligação do cabo do ecrã está colocada na periferia do *Tiny640Core*, tornado o desenho das pistas menos complexo. O módulo *MRF24J40MA* está montado no canto superior direito da placa e devido a recomendações do fabricante, foi aberta uma área de 6 cm2 em torno da antena, por forma a evitar interferências na comunicação. Tanto o conversor *MAX3233E* como o controlador *DM9000BEP* foram colocados o mais próximo possível das respectivas fichas, mais uma vez, para simplicidade no desenho das pistas. No espaço restante, foi colocado o microcontrolador.

A Figuras 5.7, 5.8 e 5.9 apresentam o resultado final do projeto desenvolvido.

[5]De notar que esta ferramenta não possui modelos de todos os componentes utilizados o projeto.

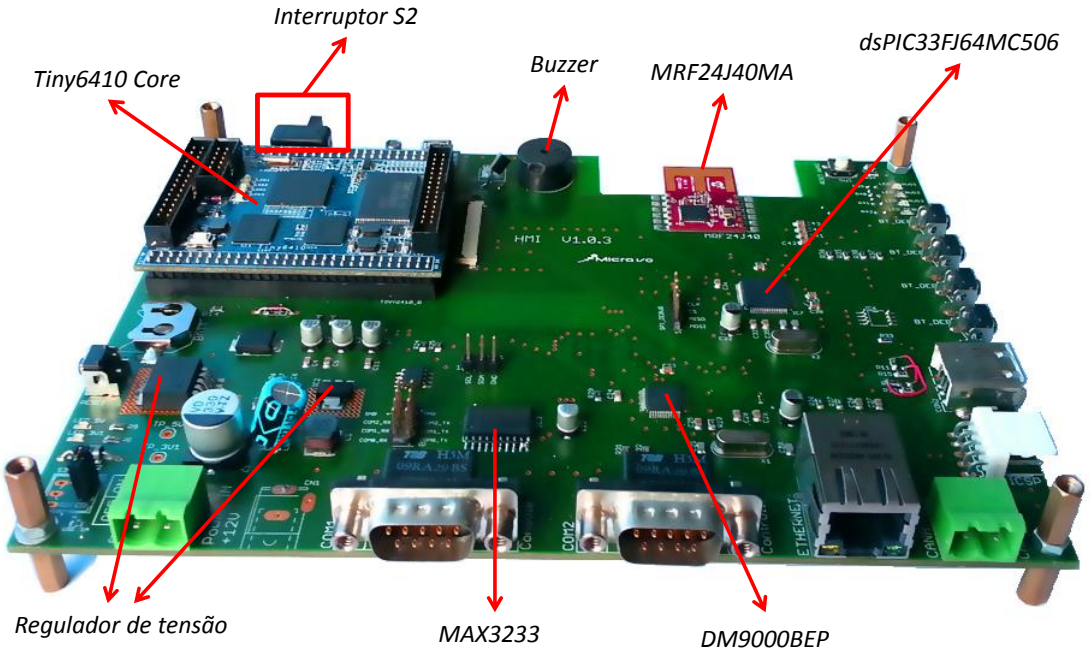


Figura 5.7: Placa final com *Tiny6410 Core*

Figura 5.8: Placa final com ecrã tátil

Figura 5.9: Vistas laterais da placa final

5.2 Firmware

Nesta secção será descrito o processo de desenvolvimento e implementação de todas as peças de software deste projeto.

5.2.1 Linux Kernel

Neste projeto foi utilizado o *Kernel 2.6.38* disponibilizado pelo fabricante *FriendlyArm*. Esta escolha justifica-se pelo facto de tornar o desenvolvimento mais rápido, uma vez que a sua compilação já se encontra testada e funcional. Adicionalmente, foram feitas algumas modificações na configuração do mesmo, que vêm de acordo com a nova plataforma de hardware onde irá ser executado.

Device Drivers

Apesar do *Kernel* oferecer uma gama vasta de *device drivers* para uma grande gama de dispositivos, foi necessário, devido à particularidade do projeto, desenvolver 2 novos controladores, um para os *leds* e outro para o *buzzer*. Foi também atualizado o controlador do módulo de comunicação *SPI*, que, nesta versão do *Kernel*, continha com alguns erros.

LEDs Device Driver

Este *device driver* é responsável pelos *leds* presentes no *Tiny6410 Core*. Permite aceder individualmente a cada um dos *leds*, ao contrário do *device driver* desenvolvido pela *FriendlyArm*, onde o acesso aos *leds* era feito em grupo. A Figura 5.10 ilustra o diagrama de inicialização deste *device driver*.

Para que seja possível utilizar este *device driver* na configuração do *Kernel*, o ficheiro de código foi colocado no diretório `drivers/char/` com o nome `HMI6410_leds.c` e foram modificados os ficheiros `kconfig` e `Makefile` presentes no mesmo diretório.

No ficheiro `kconfig` foram adicionadas as seguintes linhas:

```
config HMI6410_BOARD_LEDS
tristate "LED support for HMI6410 Board"
depends on CPU_S3C6410
default y
help
  This option enables support for leds connected to GPIO lines
  on HMI6410 Board.
```

Finalmente, foi adicionada ao ficheiro `Makefile` a linha que faz a correspondência entre a opção de escolha deste módulo e o ficheiro a compilar:

```
obj-$(CONFIG_HMI6410_BOARD_LEDS) += HMI6410_leds.o
```

Após a inicialização de todo o sistema Linux, os diferentes *leds* do sistema estarão acessíveis no diretório `/dev` através dos *nós* `ledsCore0`, `ledsCore1`, `ledsCore2` e `ledsCore3`.

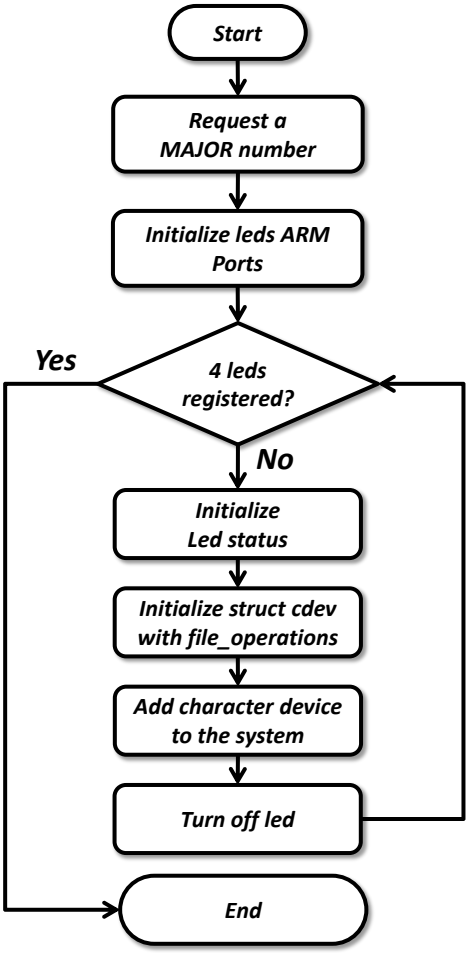


Figura 5.10: Diagrama de inicialização do *device driver* dos *leds*.

Buzzer Device Driver

Este *device driver* é responsável pela interface com um *buzzer*. O seu funcionamento é muito idêntico ao dos *leds*, variando apenas o *porto* de ligação e o número de dispositivos, que neste caso é de apenas um. O *buzzer* estará acessível no diretório `/dev/` através do *nó* `buzzer0`.

De igual modo, o ficheiro de código foi colocado no diretório `drivers/char/` com o nome `HMI6410_buzzer.c` e foram modificados os ficheiros `kconfig` e `Makefile` presentes no mesmo diretório.

No ficheiro `kconfig` foram adicionadas as seguintes linhas:

```
config HMI6410_BUZZER
tristate "Buzzer support for HMI6410 Board"
depends on CPU_S3C6410
default y
help
  This option enables support for buzzer connected to GPIO lines
  on HMI6410 Board.
```

Finalmente, foi adicionada no ficheiro `Makefile` a linha que faz a correspondência entre a opção de escolha deste módulo e o ficheiro a compilar:

```
obj-$(CONFIG_HMI6410_BUZZER) += HMI6410_buzzer.o
```

SPI

Apesar do esforço constante das equipas de desenvolvimento do *Kernel*, a versão do *Kernel* utilizada neste projeto possuía um erro na implementação do *device driver* do módulo *SPI*, procedendo-se à sua correção.

Configuração

No diretório raiz `Linux-2.6.38` existe um conjunto de ficheiros de configuração disponibilizado pelo fabricante *FriendlyArm*. Cada um destes ficheiros contém uma série de opções pré-definidas, variando algumas delas de acordo com o ecrã a utilizar no projeto. Neste projeto, utilizou-se um ecrã com 7 polegadas de diagonal e o ficheiro correspondente utilizado foi o `config_mini6410_s70`. Para o utilizar no processo de configuração, renomeou-se o mesmo para `.config` através do comando:

```
$cp config_mini6410_s70 .config
```

O processo de configuração foi desencadeado através do comando:

```
$make ARCH=arm xconfig
```

Na Figura 5.11 está ilustrada a janela de configuração do *Linux*, com as opções dos *device drivers* dos *leds* e do *buzzer* selecionadas. Estas opções são aquelas descritas anteriormente na subsecção dos *device drivers*. De notar que a escolha destas opções faz com que estes módulos sejam estáticos, ou sejam, farão parte da *imagem binária* final do *Kernel*.

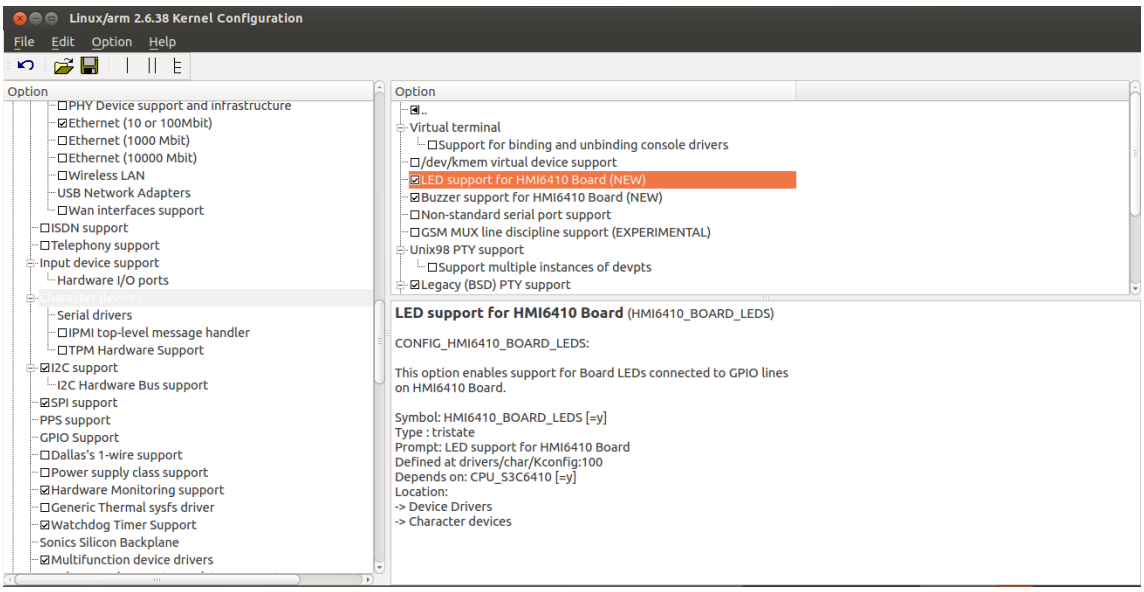


Figura 5.11: Janela de configuração do *Linux*

Compilação

Após guardar as configurações feitas anteriormente, iniciou-se o processo de compilação através do comando:

```
$make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- zImage
```

Neste momento serão mostradas, na consola, inúmeras mensagens de compilação. As seguintes mensagens serão mostradas após a conclusão do processo de compilação:

```
Kernel: arch/arm/boot/Image is ready
  SHIPPED arch/arm/boot/compressed/lib1funcs.S
  AS      arch/arm/boot/compressed/lib1funcs.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
```

A *imagem* do *Kernel* a ser "gravada"na memória *Flash* do sistema pode ser encontrada no diretório `arch/arm/boot/zImage`.

5.2.2 Sistema de ficheiros

Neste ponto será discutida a implementação do *Sistema de Ficheiros Raíz*. Este passo envolve escolher os diferentes ficheiros e juntá-los por forma a que todo o sistema seja coerente e funcional.

Busybox

Foi utilizada a ferramenta *Busybox*, introduzida anteriormente, no desenvolvimento do *sistema de ficheiros.*
O primeiro passo foi descarregar o *Busybox 1.20.2* (disponível em `http://www.busybox.net/`) e criar o diretório `busybox-1.20.2`. De seguida procedeu-se à configuração do mesmo, acedida através do comando

```
$make ARCH=arm menuconfig
```

A única alteração na configuração do *Busybox* foi a seleção da opção `Build Busybox as a static binary`, que inclui todas as bibliotecas necessárias na *imagem* final do *Busybox*, o que resulta numa operação mais rápida do que no caso em que as bibliotecas fossem partilhadas, apesar de resultar num ficheiro de maior dimensão.

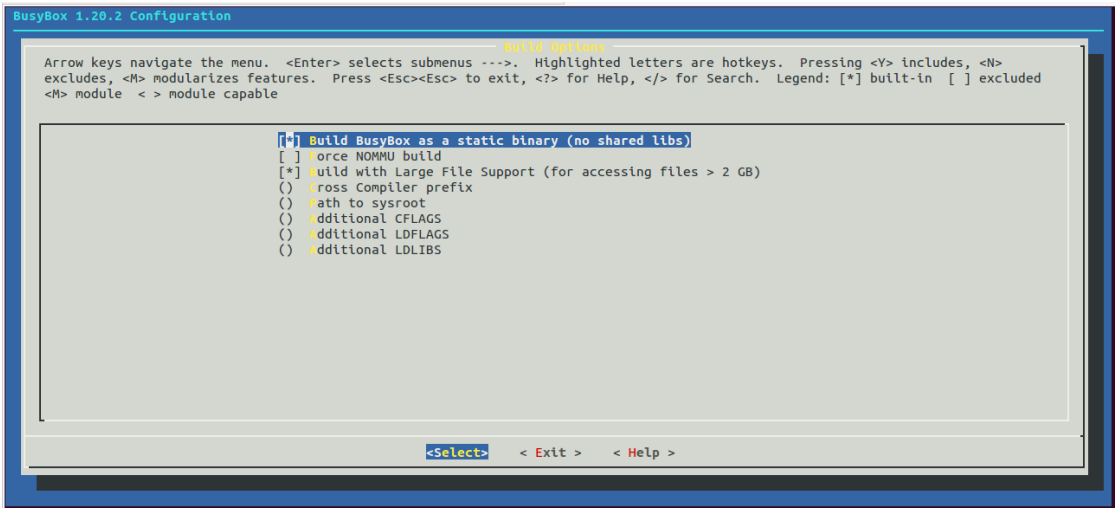


Figura 5.12: Menu de configuração do *Busybox* com a opção `Build Busybox as a static binary` selecionada.

Após a configuração, o processo de compilação foi desencadeado através do comando

```
$make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- install
```

Resultante do processo de compilação, um novo diretório foi criado, o diretório `_install`. Este diretório é o ponto de partida para o sistema de ficheiros. A sua estrutura ainda é muito reduzida, contendo apenas três diretórios (`/bin`, `/sbin` e `/usr`) e o ficheiro `linuxrc`.
A construção do restante do sistema de ficheiros foi efetuada seguindo as recomendações do FHS e reutilizando algumas das partes constituintes do *sistema de ficheiros* disponibilizado pelo fabricante *FriendlyArm*. De uma forma genérica, foram copiadas as *bibliotecas* necessárias (encontradas no diretório `lib/` da *toolchain*) para o diretório `/lib` do sistema de ficheiros, criados os *scripts* ou ficheiros de inicialização utilizados no arranque do sistema, criados todos os diretórios restantes necessários para o funcionamento do sistema, como por exemplo os diretórios `/proc` ou `/temp` e adicionados dois novos *nós* de dispositivo, o `/dev/null` e o `/dev/console`, necessários por todos os sistemas *Linux*. A Figura 5.13 apresenta o resultado final do *sistema de ficheiros*, onde apenas são mostrados os diretórios constituintes do mesmo.

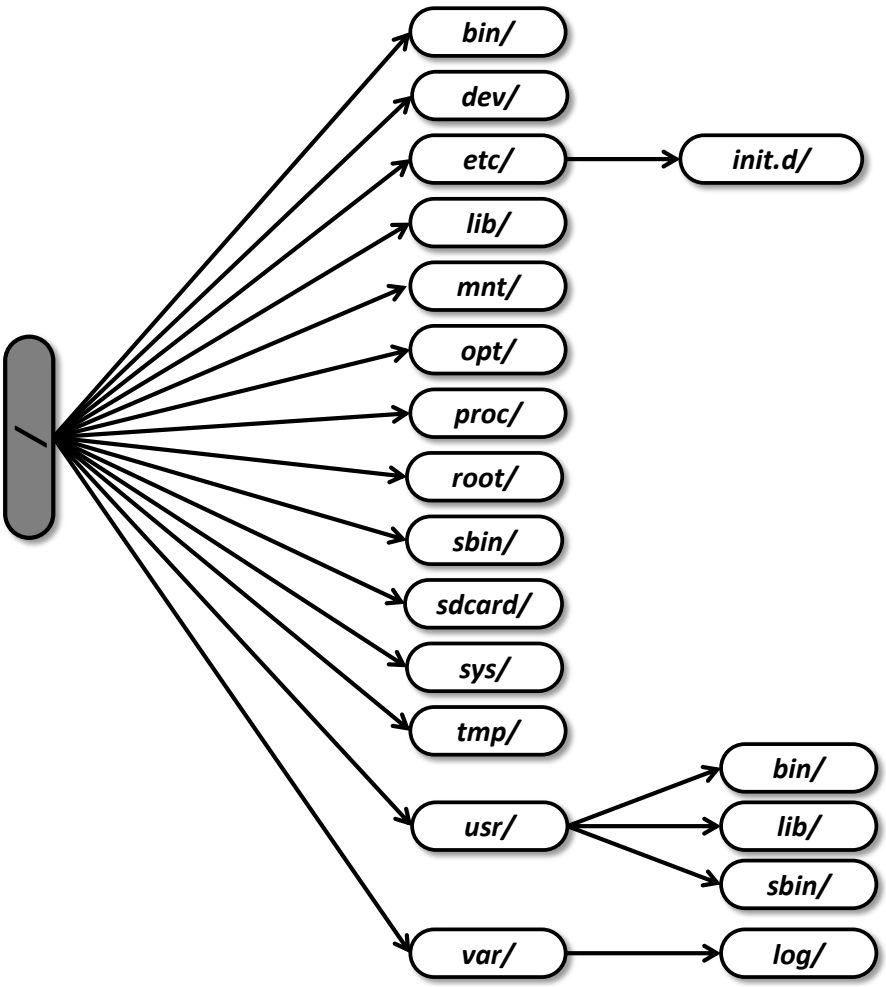


Figura 5.13: Sistema de ficheiros final.

Como referido anteriormente, no decorrer do processo de inicialização, o *Busybox* executa comandos do *script* presente em `/etc/init.d/rcS`. Este *script* foi desenvolvido por forma a que no arranque do sistema, sejam executados os seguintes passos: primeiro, são montados diferentes sistemas de ficheiros, como por exemplo, o *proc file system*, montado no diretório `/proc` e que é utilizado por muitas aplicações para obterem informações do sistema e o *sysfs file system*, montado no diretório `/sys`. Depois, o diretório `/dev` é preenchido com todos os *nós* de dispositivos que foram criados no processo de arranque do sistema. É adicionado um comando que informa o *Kernel* a executar o comando `/sbin/mdev` toda a vez que seja adicionado ou removido um dispositivo do sistema, fazendo com que o correspondente *nó* do dispositivo seja, respetivamente, inserido ou removido. De seguida, é lançado um *logger* do sistema (`syslogd`), com o objetivo de capturar problemas no arranque do sistema ou mensagens de erro do *Kernel* e definida a hora do sistema através do comando `/sbin/hwclock -s`. Antes de iniciar a *shell* do *Linux*, é lançado o *inetd*, responsável por fazer a gestão de serviços de *internet* nas diferentes interfaces de rede instaladas. O último comando presente no *script* (`busybox sh`) inicia o *shell* e apresenta a mensagem "`Please press Enter to activate this console.`"na consola. De notar que poderia ser iniciado outro programa em vez do *shell*.
Após a conclusão do sistema de ficheiros, o diretório `_install/` foi convertido para uma *imagem* do sistema de ficheiros *Unsorted Block Image File System* (UBIFS), através do comando `$ mkubimage-mlc2 _install/ rootfs.ubi`, que posteriormente será "gravada"na memória *Flash* do *Tiny6410 Core*.

5.2.3 dsPIC Firmware

Sempre que é necessário trocar informação entre o *Tiny6410 Core* (dispositivo *Master* no barramento *SPI*) e o microcontrolador (*Slave*), a aplicação desenvolvida do lado do *Master* encapsula uma mensagem e envia-a, através de uma escrita no *Character Device File* correspondente ao barramento *SPI*. O *firmware* desenvolvido para o microcontrolador tem como principal objetivo fazer o encaminhamento dessa mensagem para o respetivo periférico: um segundo barramento *SPI*, responsável por comunicar com o módulo *IEEE 802.15.4*, ou o módulo *CAN*. Este encaminhamento é baseado no primeiro carácter recebido: caso seja o carácter `'0'`, a mensagem é reencaminhada para o módulo *IEEE 802.15.4*. Caso seja o carácter `'1'`, é reencaminhada para o módulo *CAN*. Para atingir este objetivo, foi utilizado o controlador *Direct Memory Access* (DMA).
O controlador DMA é um subsistema do microcontrolador que facilita a transferência de dados entre alguns periféricos e a memória RAM, independentemente, sem a intervenção do CPU. Sem o DMA, o CPU ao fazer uma leitura/escrita na memória, ficaria totalmente ocupado e indisponível para realizar outras tarefas. Com o DMA, o CPU inicia a transferência e executa outras tarefas, esperando uma interrupção a sinalizar o fim da escrita/leitura. A

Figura 5.14 apresenta o controlador DMA num sistema computacional.

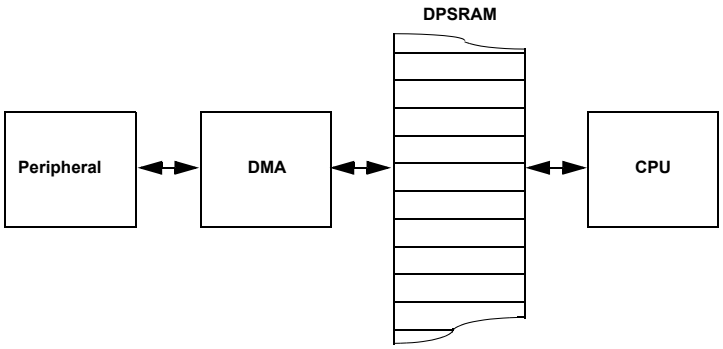


Figura 5.14: Controlador de DMA [25]

Neste projeto, o controlador DMA foi configurado para operar em modo *Ping-Pong* (Figura 5.15). Neste modo, são utilizados dois *buffers* de dados e, enquanto o processador processa os dados de um, o outro está a ser preenchido pelo DMA. O processador é informado pelo controlador DMA, através de *interrupção*, quando um dos *buffers* está cheio e pode ser processado.

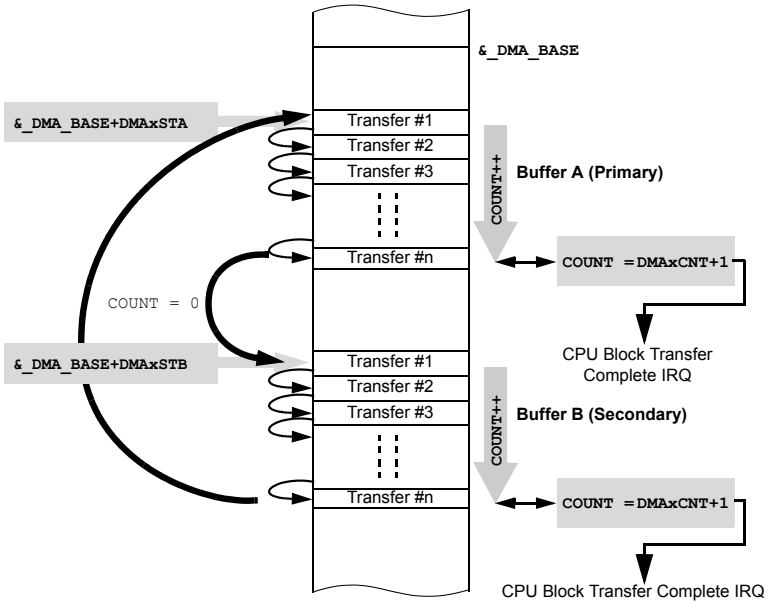


Figura 5.15: Modo *Ping-Pong* [25]

O restante *firmware* utilizado, nomeadamente o de interface com o módulo de comunicação *IEEE 802.15.4* e CAN, foi implementado através das bibliotecas existentes na empresa *Micro I/O*.

5.3 Testes do sistema

Nas secções seguintes, serão apresentados os testes efetuados ao sistema, por forma a verificar o seu correto funcionamento.

5.3.1 Instalação do *Kernel* e Sistema de Ficheiros

Foi formatado um cartão *microSD* como descrito no Anexo A e foram colocados, nos respetivos diretórios, os ficheiros resultantes da compilação do *Kernel* e do sistema de ficheiros (descritos anteriormente). O ficheiro `FriendlyArm.ini` foi apagado e definido como:

```
#This line cannot be removed. by FriendlyARM(www.arm9.net)
CheckOneButtons=No
Action=install
OS=Linux
VerifyNandWrite=No
StatusType =  LED
#################### Linux ######################
Linux-BootLoader = superboot-6410.bin
Linux-Kernel = Linux/zImage
########## RUN
#Linux-CommandLine = root=/dev/mtdblock2 rootfstype=yaffs2 init=/linuxrc
console=ttySAC0,115200
#Linux-RootFs-RunImage = Linux/rootfs.ext3
########## INSTALL
Linux-CommandLine = root=ubi0:FriendlyARM-root ubi.mtd=2 rootfstype=ubifs
init=/linuxrc console=ttySAC0,115200
Linux-RootFs-InstallImage = Linux/rootfs.ubi
```

O cartão foi então inserido no respectivo *socket* da IHM e o interruptor `S2`[6] foi posicionado para `"1 - SD BOOT"` (Ver Figura 5.7). Ao ligar o sistema, o *led 4* começou a piscar, indicando que a instalação foi iniciada. Segundos depois, os *leds* começaram a piscar sequencialmente indicando uma instalação completa.
Foi então reiniciado o sistema, reposicionando antes o interruptor `S2` para a posição `"2 - NAND"`. No final do arranque do sistema, um conjunto de mensagens de *estado* foi mostrado no ecrã sendo a última delas a esperada mensagem `"Please press Enter to activate this console"` (Figura 5.16).

[6]Este interruptor seleciona o método de arranque do sistema

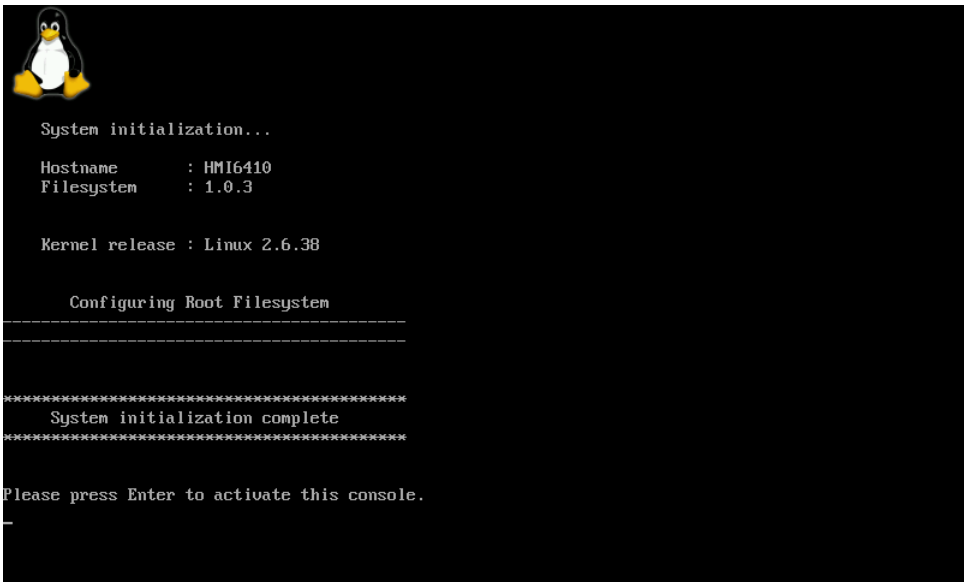


Figura 5.16: Captura do ecrã inicial do sistema

5.3.2 Dispositivos

No ficheiro `FriendlyArm.ini` foi passado o parâmetro `console=ttySAC0,115200`, indicando ao *Kernel* que a consola estará disponível na porta série `ttySAC0` com um *baudrate* igual a `115200 bps`. Isto significa que é possível aceder ao sistema de ficheiros através desta porta série. Abaixo é apesentado o conteúdo do diretório `/dev` que contém todos os *nós* dos dispositivos instalados no arranque do *Kernel*.

```
/dev # ls
backlight           ptyp2               tty24               tty6
buzzer0             ptyp3               tty25               tty60
console             ptyp4               tty26               tty61
cpu_dma_latency     ptyp5               tty27               tty62
device              ptyp6               tty28               tty63
fb0                 ptyp7               tty29               tty7
fb1                 ptyp8               tty3                tty8
fb2                 ptyp9               tty30               tty9
fb3                 ptypa               tty31               ttySAC0
full                ptypb               tty32               ttySAC1
i2c                 ptypc               tty33               ttySAC2
input               ptypd               tty34               ttySAC3
kmsg                ptype               tty35               ttyp0
ledsCore0           ptypf               tty36               ttyp1
ledsCore1           random              tty37               ttyp2
```

```
ledsCore2           rtc                 tty38               ttyp3
ledsCore3           rtc0                tty39               ttyp4
log                 sdcard              tty4                ttyp5
loop                spidev0.0           tty40               ttyp6
mem                 spidev1.0           tty41               ttyp7
mmcblk0             touchscreen         tty42               ttyp8
mtd0                tty                 tty43               ttyp9
mtd0ro              tty0                tty44               ttypa
mtd1                tty1                tty45               ttypb
mtd1ro              tty10               tty46               ttypc
mtd2                tty11               tty47               ttypd
mtd2ro              tty12               tty48               ttype
mtdblock0           tty13               tty49               ttypf
mtdblock1           tty14               tty5                ubi0
mtdblock2           tty15               tty50               ubi0_0
network_latency     tty16               tty51               ubi_ctrl
network_throughput  tty17               tty52               urandom
null                tty18               tty53               usbdev1.1
ppp                 tty19               tty54               vcs
psaux               tty2                tty55               vcs1
ptmx                tty20               tty56               vcsa
pts                 tty21               tty57               vcsa1
ptyp0               tty22               tty58               watchdog
ptyp1               tty23               tty59               zero
```

De realçar a presença dos *nós* de dispositivo `ttySAC0` e `ttySAC1`, correspondentes às duas portas série utilizadas neste sistema, dos *nós* `ledsCore0`, `ledsCore1`, `ledsCore2` e `ledsCore3`, correspondentes aos 4 *leds* do *Tiny6410 Core*, do *nó* `buzzer0` correspondente ao *Buzzer* e dos *nós* `spidev0.0` e `spidev1.0` correspondentes ao barramento SPI. Estes últimos foram alvo de modificação e/ou desenvolvimento nesta Dissertação e, consequentemente, foram submetidos a testes.

Leds

Foi desenvolvida uma aplicação na linguagem `C` que permite interagir com os *leds* do *Tiny6410 Core* por forma a testar o seu funcionamento.
A Figura 5.17 apresenta o menu dessa aplicação. Por defeito, o controlo é feito no *nó* `/dev/ledsCore0`, podendo ser alterado executando a opção `3`. É possível ligar ou desligar cada um dos *leds* individualmente ou em conjunto. Todos os *leds* foram testados com sucesso.

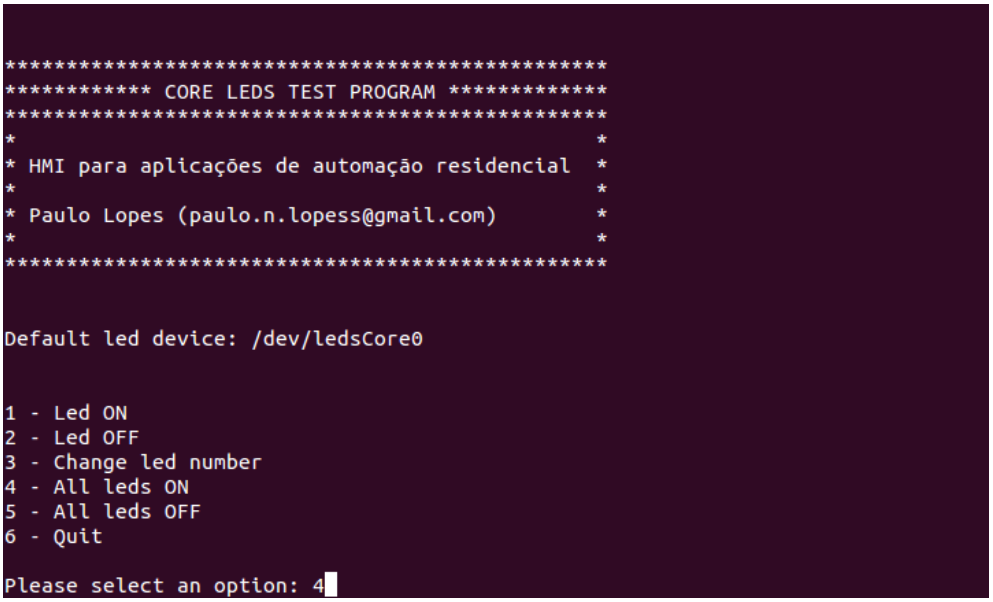


Figura 5.17: Aplicação de teste para os *leds* do *Tiny6410 Core*

Buzzer

Foi desenvolvida uma aplicação na linguagem `C` que permite interagir com o *Buzzer* por forma a testar o seu funcionamento. A Figura 5.18 apresenta o menu dessa aplicação. O seu funcionamento é muito simples, onde é possível apenas ligar ou desligar o *Buzzer*, sendo o mesmo testado com sucesso.

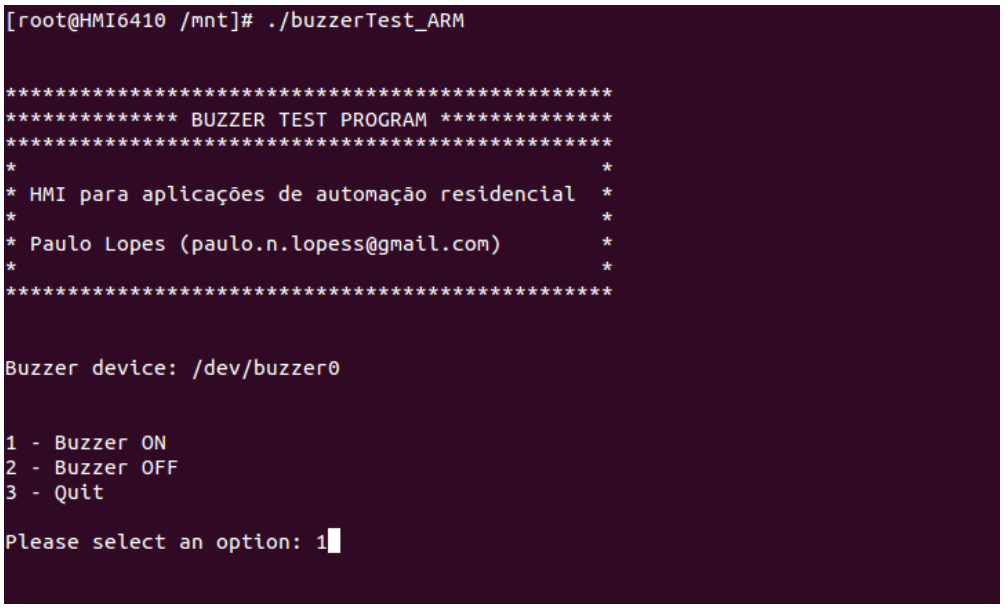


Figura 5.18: Aplicação de teste para buzzer

SPI

A Figura 5.19 apresenta o resultado do teste ao barramento SPI. Foi utilizado um osciloscópio de 4 canais, com analizador do protocolo SPI, onde cada um dos canais foi ligado aos 4 pontos de teste: `SPI_MISO`, `SPI_MOSI` `SPI_CLK` e `SPI_CS`.
A *vermelho* temos representado o sinal *Chip Select* (`SPI_CS`), a *azul* o sinal de *relógio* (`SPI_CLK`) e a *rosa* o sinal de *dados* (`SPI_MOSI`).
Através do comando `$echo abcd > /dev/spidev1.0`, foi enviado o conjunto de caracteres `'abcd'` seguido do caráter especial `ASCII`, *New Line*. Em hexadecimal, estes caracteres são representados pela sequência `0x61 0x62 0x63 0x64 0x0A`, precisamente a mesma sequência capturada pelo osciloscópio, confirmando o correto funcionamento do barramento SPI.

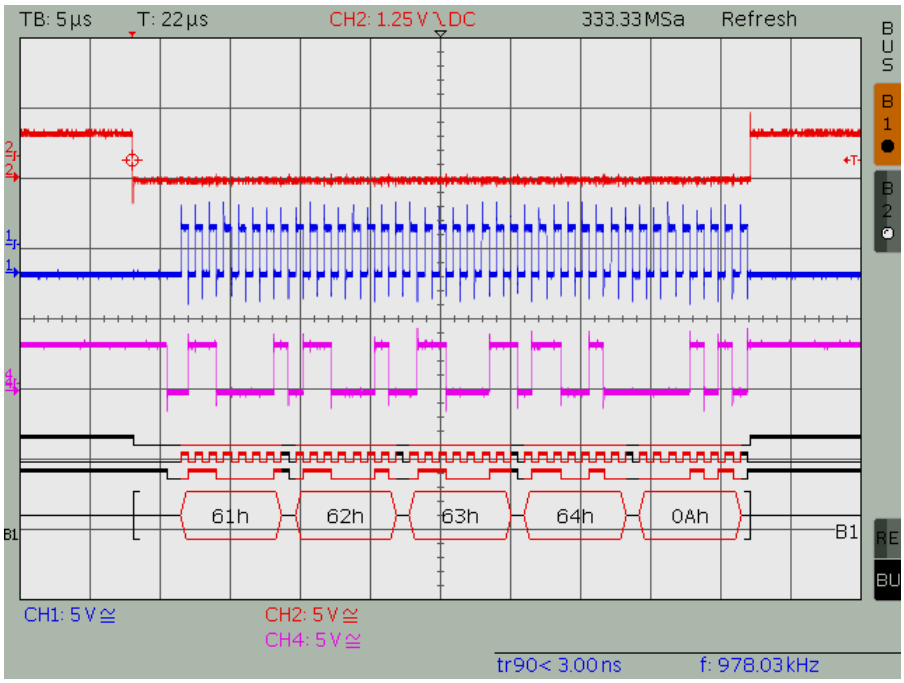


Figura 5.19: Captura efetuada pelo osciloscópio do barramento SPI

MRF24J40MA

O módulo de comunicação *MRF24J40MA* foi testado após o sucesso do teste do barramento *SPI*. Para isso, foi enviada a sequência de caráteres `"12345678"` através do *Tiny6410 Core* para o barramento (utilizando o comando `$echo -n 012345678 > /dev/spidev1.0`). Esta mensagem foi recebida e processada pelo microcontrolador e reencaminhada para o módulo *MRF24J40MA* (o primeiro carácter recebido foi o `'0'`). A trama emitida para o meio, foi observada através da ferramenta *Zena* da *Microchip* (Figura 5.20). O campo *Payload* contém a sequência de números enviada, representada em *ASCII* pela sequência `0x30 0x31 0x32`

`0x33 0x34 0x35 0x36 0x37 0x38`.

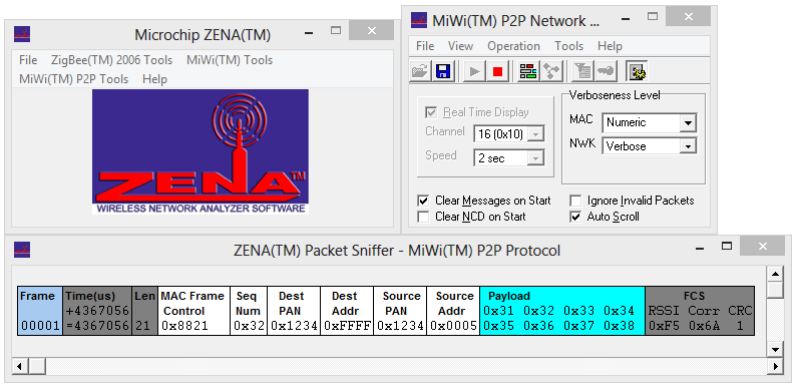


Figura 5.20: Captura efetuada pelo *Zena*

Ethernet

Para testar a interface *Ethernet* foi executado o comando `ping` num terminal remoto. Um dos ficheiros presentes no diretório `/etc`, o `eth0-config`, chamado pelo *script* de inicialização `init.d/rcS`, contém as definições da configuração da rede. O *IP* atribuído ao sistema foi o `192.168.1.230`. Foi executado então o comando `$ping 192.168.1.230` e foram retornadas as seguintes linhas presentes na Figura 5.21, indicando que a interface *Ethernet* está acessível e funcional.

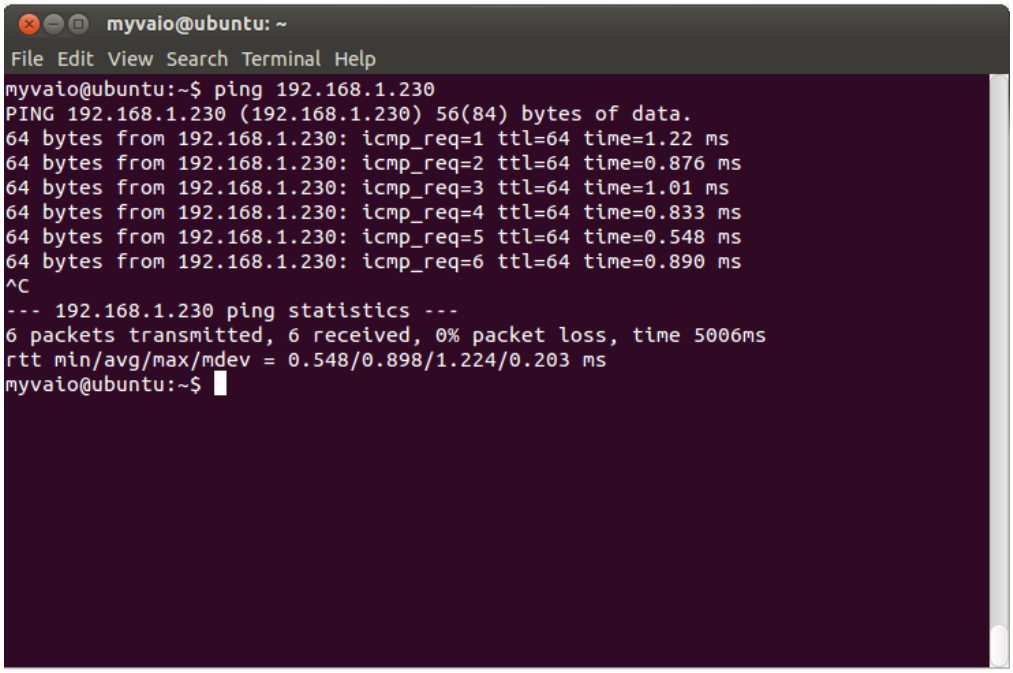


Figura 5.21: Resultado do comando `$ping 192.168.1.230`

5.4 Interface gráfica

As soluções gráficas em *Linux embutido* são baseadas no conceito de *frame buffer*, que é uma camada de abstração entre os controladores gráficos e as aplicações ou bibliotecas. Uma dessas bibliotecas é o *Qt*, que é uma *framework* multiplataforma utilizada no desenvolvimento de aplicações gráficas.
O primeiro passo foi descarregar as bibliotecas do *Qt*, que podem ser encontradas em `http://get.qt.nokia.com/qt/source/qt-everywhere-opensource-src-4.8.1.tar.gz` para a plataforma *ARM*. Para configurar e preparar as mesmas para a compilação, foi executado o seguinte comando:

```
$ ./configure -opensource -confirm-license -embedded arm -xplatform
qws/arm-none-linux-gnueabi-g++ -little-endian -qt-zlib -qt-libtiff
-qt-libpng -qt-libmng -qt-libjpeg -prefix /usr/local/qt-arm
```

Esta configuração irá preparar o *Qt* para ser compilado para um sistema *ARM little-endian*, cujo *cross-compiler* tem o prefixo `arm-none-linux-gnueabi-`. Para compilar e instalar as bibliotecas foram executados os comandos `$make` e `$sudo make install`. O passo seguinte foi copiar todas as bibliotecas compiladas para o sistema de ficheiros da IHM, para a localização `/usr/local/Qt`.
Para testar se todo o processo foi concluído com êxito, foi executada a aplicação `/usr/local/Qt/examples/qws/framebuffer/framebuffer`. Esta aplicação irá mostrar no ecrã três quadrados: um vermelho, um verde e um azul (Figura 5.22).

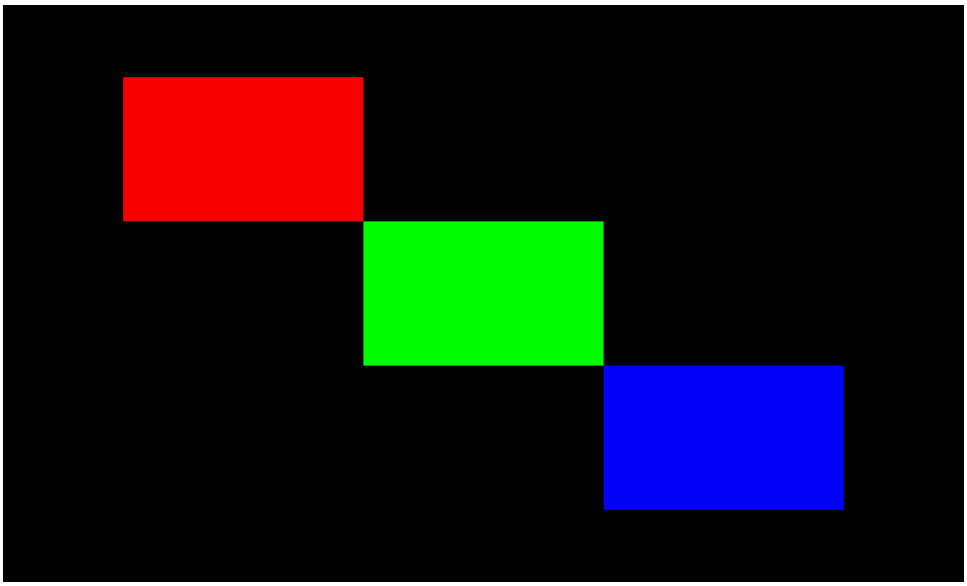


Figura 5.22: Teste à instalação do *Qt*

As bibliotecas compiladas já incluem uma biblioteca para fazer de interface com o *touchscreen*. Para configurar a interface de *touchscreen*, foi criado o ficheiro `ts.conf` no diretório `/etc` com as seguintes linhas:

```
module_raw input
module pthres pmin=1
module variance delta=30
module dejitter delta=10000
module linear
```

e adicionadas ao *script* de inicialização `/etc/init.d/rcS` os comandos seguites:

```
QTDIR=/usr/local/Qt
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/Qt/lib
export TSLIB_CALIBFILE=/etc/pointercal
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_PLUGINDIR=/usr/lib/ts
export TSLIB_CONSOLEDEVICE=none
export TSLIB_FBDEVICE=/dev/fb0
export TSLIB_TSDEVICE=/dev/touchscreen
export TSLIB_TSEVENTTYPE=INPUT
export QWS_SIZE=800x480
export QWS_DISPLAY=LinuxFb:mmWidth=175:mmHeight=175
```

Finalmente, o passo seguinte foi calibrar o *touchscreen* através de uma aplicação disponibilizada, lançada através do comando `$ts_calibrate` (Figura 5.23).

Figura 5.23: Aplicação para calibração do *touchscreen*

Com isto é finalizado todo o processo de desenvolvimento da *Interface Homem-Máquina*, encontrando-se a mesma testada e funcional. Assim, qualquer desenvolvimento adicional, centrar-se-á na camada das aplicações do utilizador, como será o caso do projeto *UNISOL*, introduzido no capítulo seguinte.

Capítulo 6

O projeto *UNISOL*

6.1 Enquadramento

Apoiado pelo *Quadro de Referência Estratégico Nacional* (QREN), o projeto *UNISOL* tem por objetivo desenvolver um conjunto integrado de atividades de *Investigação e Desenvolvimento* que permitam conceber um sistema universal e inovador, autónomo e inteligente, de gestão e acumulação de energia solar térmica que pode utilizar praticamente qualquer coletor solar existente no mercado.

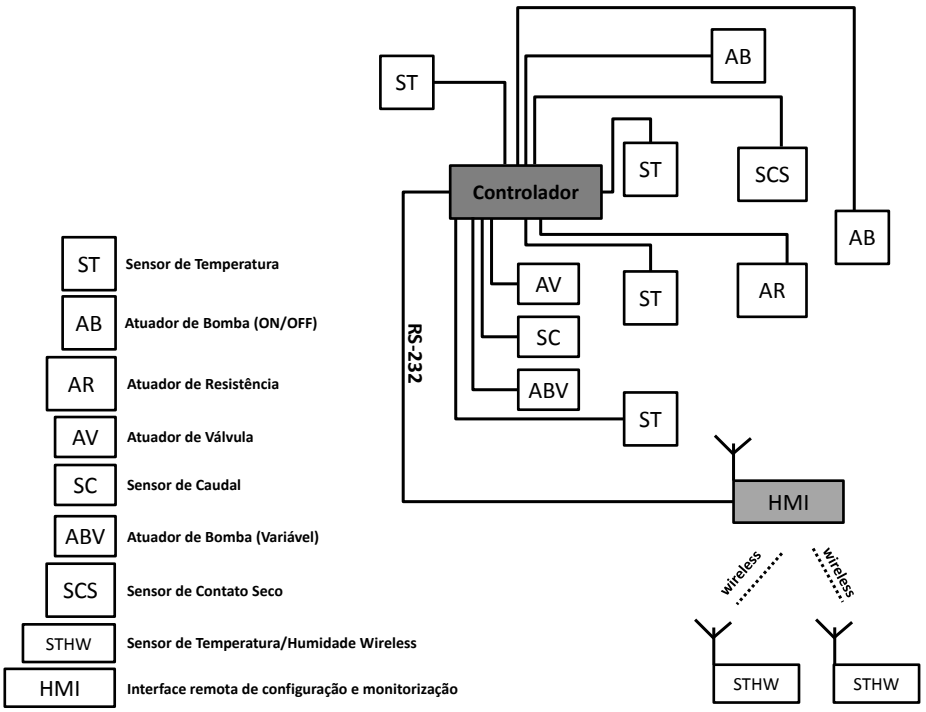


Figura 6.1: Arquitetura concetual de sensores e atuadores do sistema

Este sistema destina-se ao pré-aquecimento de águas sanitárias e/ou aquecimento ambiente. O projeto apresenta princípios de universalidade e de integração interior e exterior nos edifícios, princípios estes resultantes da aplicação do sistema registado num *Modelo de Utilidade Português*. Apresenta também um circuito permutador de duplo sentido, já objeto de um pedido de *Patente Portuguesa* e um sistema de controladores modulares autónomos e integráveis. Este projeto foi incluído na *Task 39* da `SHC - "Solar Heating and Cooling Programme"` da *Agência Internacional da Energia* como demonstrador da utilização de materiais poliméricos em sistemas solares térmicos. A Figura 6.1 apresenta a arquitetura concetual de sensores e atuadores do sistema para o pré-aquecimento de águas sanitárias e aquecimento ambiente do projeto *UNISOL*.

6.2 Simulador

O trabalho desta Dissertação foi desenvolvido em paralelo com o projeto *UNISOL*, o que originou a necessidade de desenvolver uma aplicação que simulasse um sistema de acordo com os requisitos do mesmo. A Figura 6.2 apresenta o conjunto de subsistemas utilizados na simulação.

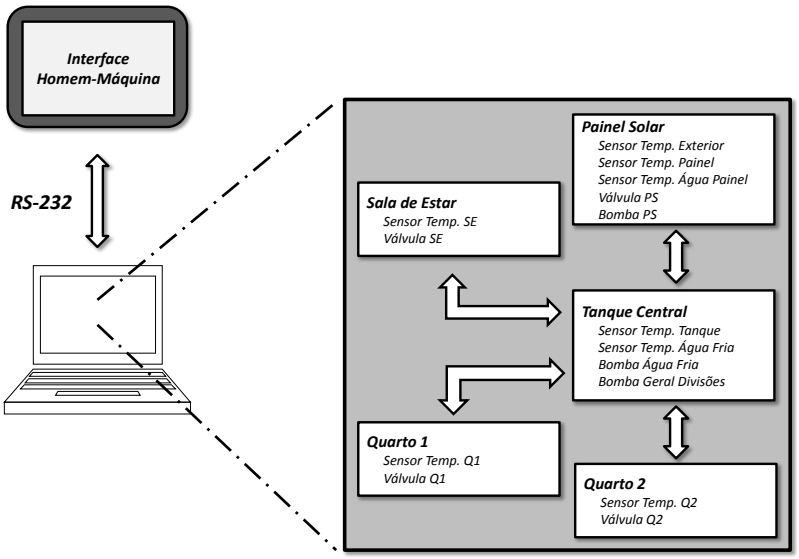


Figura 6.2: Subsistemas da simulação.

6.2.1 Placa controladora (Simulação PC)

Foi desenvolvida uma aplicação gráfica, utilizando o IDE *NetBeans*, na linguagem *Java*, que simula o conjunto de subsistemas apresentados na Figura 6.2. O objetivo desta aplicação é fornecer à *Interface Homem-Máquina* os valores dos diversos sensores de temperatura ou o es-

tado das válvulas e das bombas de água. Estas variáveis são enviadas sempre que solicitadas, ou seja, funciona num mecanismo de *pergunta/resposta*. A Figura 6.3 apresenta a interface gráfica da aplicação de simulação.

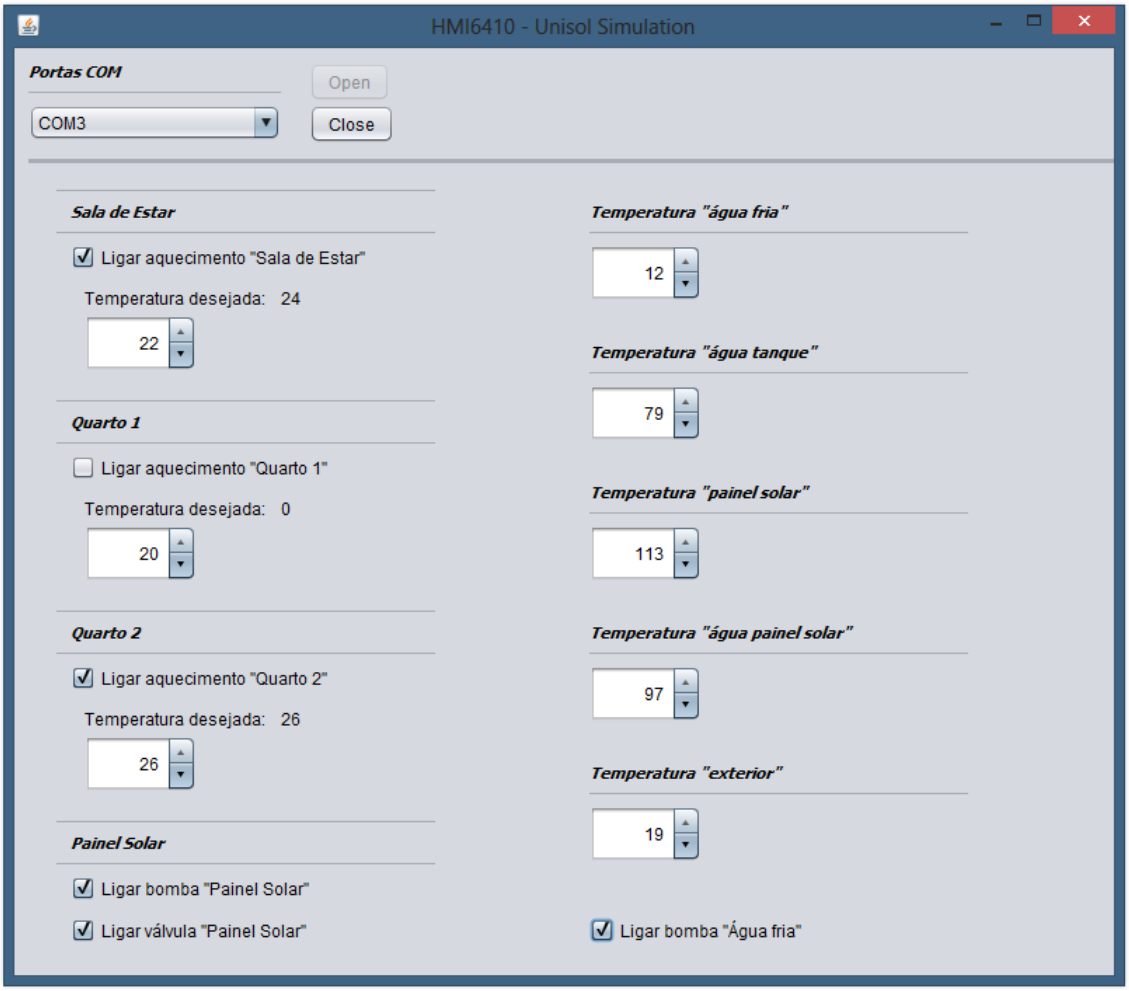


Figura 6.3: Interface gráfica da aplicação de simulação.

6.2.2 Aplicação da *Interface Homem-Máquina*

Foi desenvolvida uma interface gráfica amigável capaz de visualizar o valor dos diversos sensores de temperatura e o estado das válvulas e das bombas de água. Esta interface possui cinco janelas: a primeira janela (Figura 6.4) apresenta o tanque central de água, a segunda apresenta o painel solar (Figura 6.5) e as restantes três (por exemplo, a Figura 6.6 apresenta a divisão *Sala de Estar*) correspondem a uma divisão diferente da casa.

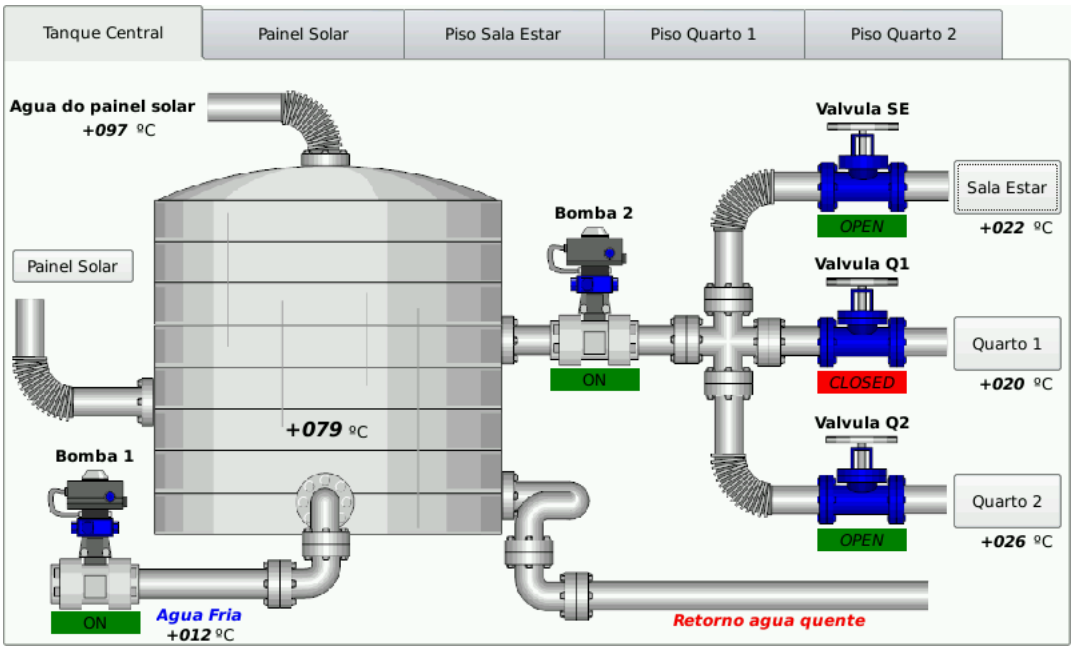


Figura 6.4: Separador *Tanque Central* da Interface Homem-Máquina

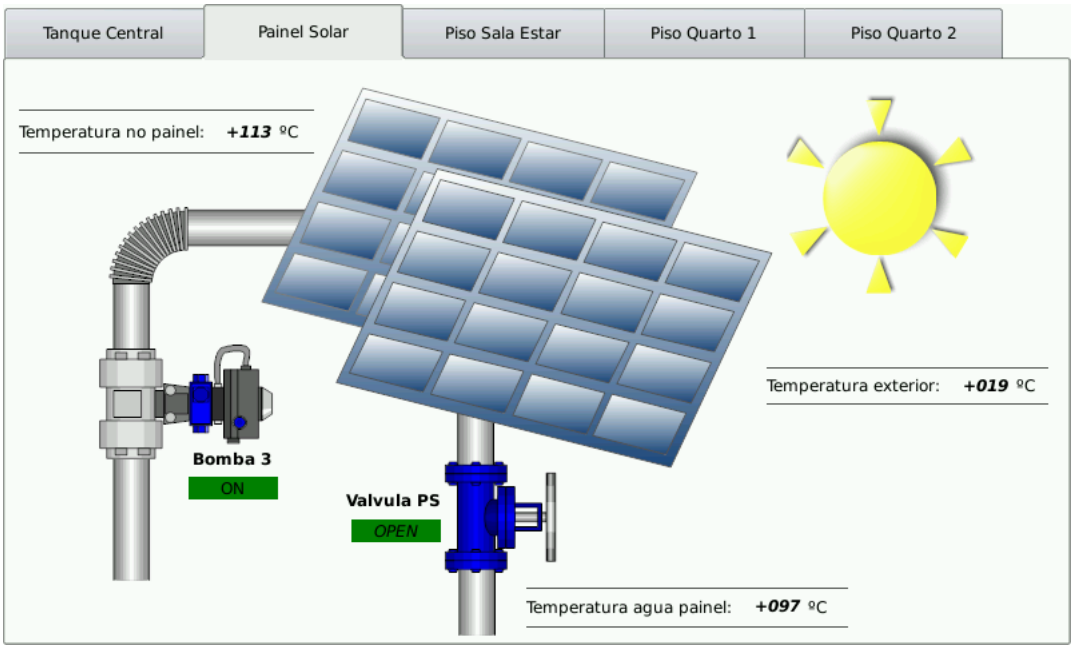


Figura 6.5: Separador *Painel Solar* da Interface Homem-Máquina

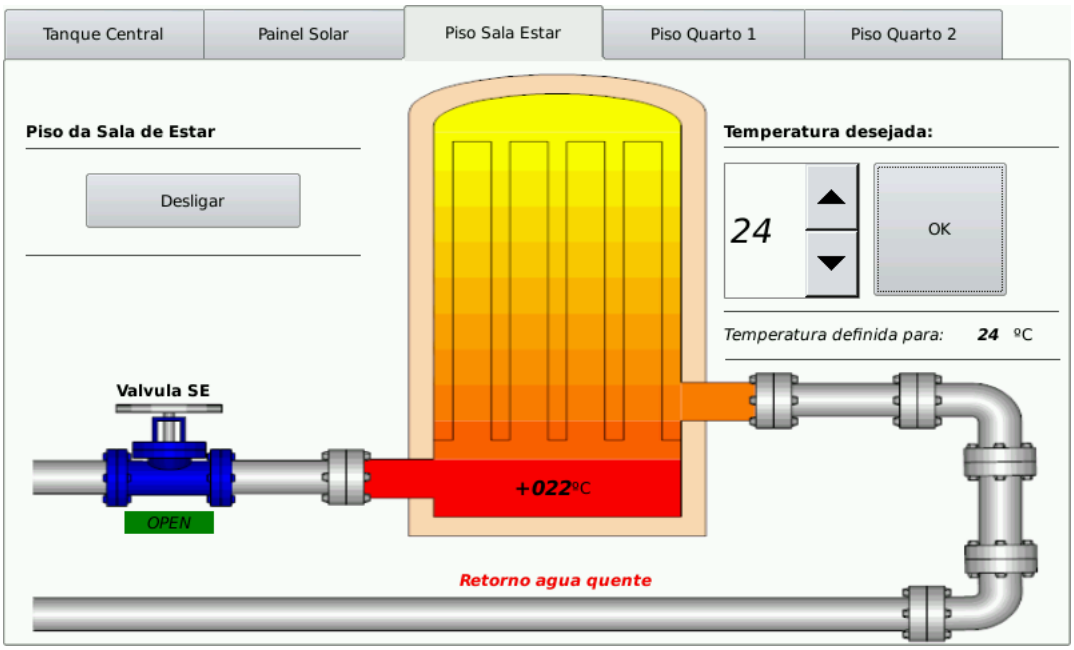


Figura 6.6: Separador *Sala de Estar* da Interface Homem-Máquina

Podemos observar que os valores da temperatura ou o estado das bombas/válvulas de água estão de acordo com a aplicação de simulação (Figura 6.3).

O utilizador além de visualizar o estado destas variáveis, pode também definir dois parâmetros do sistema, nomeadamente, ligar/desligar o aquecimento numa divisão ou a temperatura desejada, caso pretenda o aquecimento ligado. Pela Figura 6.6, observa-se que o sistema de aquecimento está ligado (o botão respetivo apresenta o texto `"Desligar"`. Caso estivesse desligado, apresentaria o texto `"Ligar"`) e a temperatura foi definida para `24 °C`. O piso apresenta `22 °C` de temperatura, pois o sistema ainda não atingiu a temperatura definida.

Estrutura da aplicação

Esta aplicação foi desenvolvida em *Qt*. Um dos mecanismos distintos do *Qt* é o uso dos *Signals and Slots*, que são utilizados para comunicar entre objetos. Sempre que ocorre um evento em particular, é *emitido* um *Signal* e é chamado, em resposta, o *Slot* que estiver *ligado* a esse *Signal*. Um determinado *Signal* pode causar a execução de vários *Slots* e um *Slot*, em particular, pode ser executado devido à emissão de diferentes *Signals*.

A Figura 6.7 apresenta o diagrama de *Signals and Slots* desta aplicação, assim como as principais *Threads* da mesma.

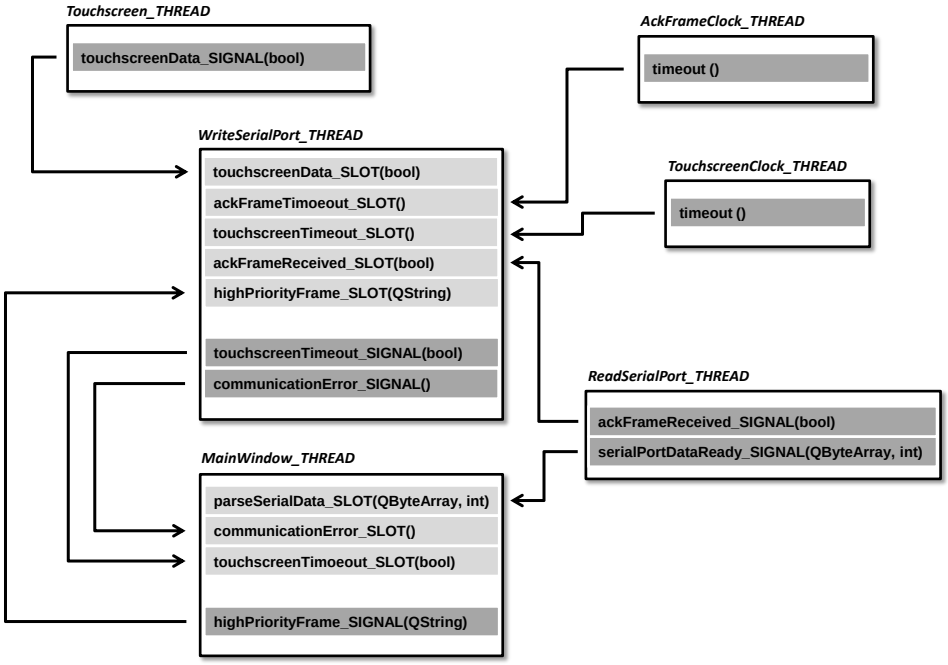


Figura 6.7: Diagrama de *Signals and Slots* da aplicação gráfica.

Sendo utilizada uma estrutura de *pergunta/resposta* como mecanismo de comunicação, esta aplicação inicia-se enviando tramas com comandos pré-definidos (introduzidos na próxima secção), esperando pela correta resposta da aplicação de simulação. Se tal acontecer, será enviada a trama seguinte e assim sucessivamente. Caso contrário, será sempre reenviada a mesma trama até o sistema decidir que houve um erro na comunicação.

A *WriteSerialPort_Thread* é responsável por fazer a gestão do envio das tramas. Sempre que envia uma trama, inicia um relógio (que corre na *AckFrameClock_Thread*) e aguarda o *Signal* `ackFrameReceived_Signal(bool)` para decidir se envia a próxima trama ou reenvia a mesma. Este *Signal* será emitido pela *ReadSerialPort_Thread* ao ser recebida uma trama de resposta e após ser realizado um conjunto de operações para determinar se esta está consistente e de acordo com as regras do protocolo de comunicação. Caso esteja de acordo, o parâmetro `bool` será enviado como *verdadeiro* (*True*), o que causará o envio da trama seguinte. Caso contrário, será enviado a *falso* (*False*) e a trama anterior será reenviada até que o relógio emita o *Signal* `timeout`. Neste ponto, o sistema decide que existem problemas no canal de comunicação e a *WriteSerialPort_Thread* emite o *Signal* `communicationError_Signal()`. Outro *Signal* emitido pela *ReadSerialPort_Thread*, no caso de ter recebido uma trama correta, é o `serialPortDataReady_Signal(QByteArray, int)`.

Tendo em consideração este contexto, os *Signals* `communicationError_Signal()` e `serialPortDataRead_Signal(QByteArray, int)` irão disparar a execução de dois *Slots* definidos na *MainWindow_Thread*: o *Slot* `parseSerialData_Slot(QByteArray, int)` e o *Slot*

`communicationError_SLOT(QByteArray,int)`. O primeiro analisará a trama recebida e atualizará os *widgets* da aplicação gráfica de acordo com os valores recebidos. O segundo desativará a janela gráfica principal e exibirá uma janela gráfica secundária com uma mensagem de erro a informar que existe um problema na comunicação. Os *Signals* e *Slots* associados ao *touchscreen* foram implementados para fazer uma gestão energética do *backligth*, ou seja, caso o utilizador não esteja a utilizar o sistema (carregando no *touchscreen*), este desliga-se após um período previamente pré-definido.
O mecanismo de comunicação deste sistema está continuamente e de forma cíclica a enviar um conjunto de tramas pré-definidas (introduzidas na secção seguinte) em memória, denominadas *tramas de prioridade normal*. No entanto, é necessário, por vezes, enviar tramas que requerem tratamento imediato (*tramas de prioridade alta*). Estas tramas são despoletadas pela interação do utilizador com o sistema, nomeadamente nos momentos em que pretende ligar/desligar o sistema ou definir uma temperatura. A *MainWindow_Thread* emite assim o *Signal* `highPriorityFrame_Signal(QString)`, informando a *WriteSerialPort_Thread* da existência de uma trama prioritária para envio.

6.2.3 Protocolo de comunicação

Foi estabelecido um conjunto de regras na comunicação entre a *Interface Homem-Máquina* e o sistema computacional onde está a ser executada a aplicação de simulação do sistema *UNISOL*. Estas regras resumem-se essencialmente numa trama de dados, que se distingue por um campo de *comando* que identifica qual a ação a ser tomada. A Figura 6.8 apresenta o formato geral da trama utilizada no protocolo de comunicação.

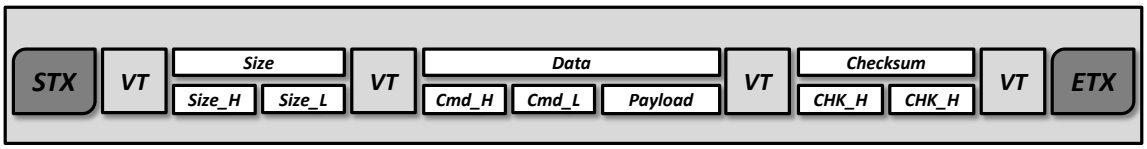


Figura 6.8: Formato geral da trama do protocolo de comunicação.

Todas as tramas são iniciadas pelo caráter especial *ascii* `STX` (*Start of Text*), seguido do campo *Size*, representado pelos *bytes* `Size_H` e `Size_L`, que indica o tamanho, em *bytes*, do campo *Data*. Este é composto pelo *comando*, representado pelos *bytes* `Cmd_H` e `Cmd_L`, e pelo *payload*. O *payload* não tem tamanho fixo e transporta dados específicos de cada *comando*. O campo *Checksum*, representado pelos *bytes* `CHK_H` e `CHK_L`, é formado pela função lógica *Ou-exclusivo* (`XOR`) de todos os *bytes* do campo *Data*. Finalmente a trama termina com o caráter especial *ascii* `ETX` (*End of Text*). Entre cada campo da trama, foi inserido o caráter especial *ascii* `VT` (*Vertical Tab*), para facilitar o processamento da trama. A Tabela 6.1 apresenta os comandos definidos para o protocolo de comunicação.

Comando	Descrição
SS - *System Status*	Verificar se o aquecimento de cada divisão está ligado/desligado
PS - *Pump Status*	Verificar o estado das bombas de água
VS - *Valve Status*	Verificar o estado das válvulas de água
GT - *Get Temperatures*	Adquirir o valor das diferentes temperaturas
ST - *Set Temperature*	Definir a temperatura do sistema de aquecimento de uma determinada divisão da casa
RS - *Room Set*	Ligar/Desligar o sistema de aquecimento de uma determinada divisão da casa

Tabela 6.1: Comandos utilizados no protocolo de comunicação.

Esta simulação deu uma ideia geral do funcionamento da *Interface Homem-Máquina* como parte de um sistema de controlo. Após a sua conclusão, o sistema real de controlo do projeto *UNISOL* entrou em fase de testes, onde esta mesma IHM foi integrada e passou a recolher informação de alguns dos sensores, bombas e válvulas instalados. Para tal, foi desenvolvida uma nova aplicação gráfica de teste utilizando o mesmo protocolo de comunicação anteriormente descrito. A Figura 6.9 apresenta a interface gráfica desenvolvida[1].

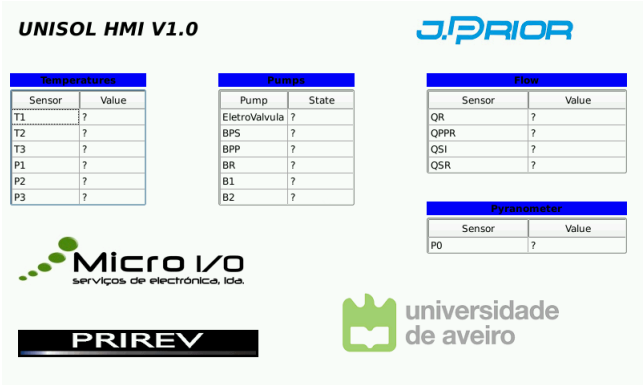


Figura 6.9: Formato geral da trama do protocolo de comunicação.

Neste momento todo o sistema encontra-se em fase de testes, funcional, revelando-se robusto e fiável na sua implementação.

[1]De notar que não está presente informação, pois no momento da captura da imagem, a IHM não estava ligada ao sistema controlador.

Capítulo 7

Conclusões e Trabalho Futuro

Este capítulo finaliza esta Dissertação e são apresentadas, numa primeira secção, as principais conclusões a retirar do trabalho efetuado. Na segunda secção, são discutidos alguns pontos de trabalho futuro.

7.1 Conclusões

O trabalho desenvolvido nesta Dissertação focou-se no desenvolvimento de uma plataforma computacional, capaz de suportar um sistema operativo baseado em *Unix*, que fará de interface entre o utilizador e o sistema de controlo.

Numa primeira fase, foi efetuado o estudo de uma plataforma de desenvolvimento com caraterísticas semelhantes (*FriendlyArm Tiny6410*) ao pretendido, executando um sistema operativo baseado em *Unix*, o *Linux 2.6.38*. Este estudo foi uma mais valia na compreensão do funcionamento de um *Sistema Embutido Linux*. Com ele, foram retiradas conclusões quer a nível de *hardware*, como por exemplo, que interfaces utilizar e a sua integração no sistema, quer a nível de *software*, no que tocou a desenvolvimento de *device drivers* específicos para os módulos adicionais implementados (*Buzzer* e *Leds*), entre outras.

Foram especificados todos os requisitos do sistema e da sua arquitetura e desenvolvida uma *Printed Circuit Board* com todos os periféricos necessários para satisfazer os mesmos, capaz de integrar o *Tiny6410 Core*. Esta nova plataforma possui toda a versatilidade necessária para servir de ponto de partida para um sistema computacional.

Todo o *software* necessário foi devidamente instalado (*Linux Kernel*, *Sistema de Ficheiros* e *aplicação gráfica*), ajustado e testado, estando a funcionar de acordo com o pretendido.

Uma aplicação de simulação foi desenvolvida por forma a testar o conceito proposto no projeto

UNISOL e provou que é possível aceder aos parâmetros de controle de um sistema remoto, tais como valores de temperatura ou informação de outro tipo de sensores. Posto isto, foi desenvolvida uma aplicação de teste especificamente para o projeto *UNISOL* que se encontra neste momento instalada e em fase de testes num demonstrador desenvolvido para o efeito.

7.2 Trabalho Futuro

A *Interface Homem-Máquina* desenvolvida nesta Dissertação apresenta inúmeras possibilidades de trabalho futuro, pois é uma plataforma extremamente versátil.

Firmware

Neste projeto foi utilizado o *Kernel 2.6.38* disponibilizado pelo fabricante *FriendlyArm*, sendo feitas as adaptações necessárias consoante o *hardware* utilizado. Como trabalho futuro, poderiam ser testadas outras versões mais recentes do *Kernel*, ou migrar para o *Windows CE* ou mesmo para o emergente e sólido *Android*.

O *Bootloader* utilizado neste projeto é proprietário, limitando a utilização do *Tiny6410 Core*, pois está desenvolvido com alguma particularidade para a plataforma *FriendlyArm*. Poderiam então ser implementados outros *Bootloaders*, como por exemplo o *U-Boot* ou o *RedBoot*.

Tecnologia *touchscreen*

Este projeto utiliza a tecnologia *resistiva*. No entanto poderá ser interessante, noutras aplicações, dotá-lo com tecnologia *capacitiva*. Para tal será necessário desenvolver novos *device drivers* e instalar novas bibliotecas de calibração e utilização.

Comunicação

Esta plataforma possui 4 interfaces de comunicação: *RS-232*, *Ethernet*, *CAN* e *IEEE 802.15.4*. Inúmeros protocolos já desenvolvidos, em diversas aplicações, poderão ser implementados nas mesmas e testar o seu comportamento. De futuro, seriam interessante implementar o protocolo de comunicação para aplicações de automação *ModBus* e testar a comunicação entre a *Interface Homem-Máquina* desenvolvida e um *Controlador Lógico Programável*.

Uma outra abordagem será a remoção do microcontrolador que faz de interface entre o *Tiny6410 Core* e os módulos de comunicação *IEEE 802.15.4* e *CAN*, passando a ser o *Kernel* a fazer a gestão dos mesmos.

Bibliografia

[1] José Roberto Muratori and Paulo Henrique Dal Bó. *Automação residencial: histórico, definições e conceitos*. [online]. Avaliabe: `http://www.osetoreletrico.com.br/web/documentos/fasciculos/Ed62_fasc_automacao_capI.pdf`. Accessed: July, 2013.

[2] Gradiant. *Advanced interfaces for domotic systems*. [online]. Avaliabe: `http://www.gradiant.org/en/news/news/258-interfaces-avanzadas-para-sistemas-domoticos.html`. Accessed: July, 2013.

[3] Schneider Electric. *Diálogo Homem-Máquina*. [online]. Avaliabe: `http://www.schneiderelectric.pt/documents/product-services/training/doctecnico_hmi.pdf`. Accessed: July, 2013.

[4] Texas Instruments. *RS-422 and RS-485 Standards Overview and System Configurations*. [online]. Avaliabe: `http://www.ti.com/lit/an/slla070d/slla070d.pdf`. Accessed: July, 2013.

[5] Analog Devices. *Controller Area Network (CAN) Implementation Guide*. [online]. Avaliabe: `http://www.analog.com/static/imported-files/application_notes/AN-1123.pdf`. Accessed: July, 2013.

[6] Texas Instruments. *The RS-485 Design Guide*. [online]. Avaliabe: `http://www.ti.com/lit/an/slla272b/slla272b.pdf`. Accessed: July, 2013.

[7] mBus. *An Overview of Controller Area Network (CAN) Technology*. [online]. Avaliabe: `http://www.parallax.com/dl/docs/prod/comm/cantechovew.pdf`. Accessed: July, 2013.

[8] ZigBee Alliance. *ZigBee Specification Overview*. [online]. Avaliabe: `http://www.zigbee.org/Specifications/ZigBee/Overview.aspx`. Accessed: July, 2013.

[9] Institute of Electrical and Electronics Engineers. *802.15.4-2003: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*. New York, IEEE Press. October 1, 2003.

[10] E. Callaway, P. Gorday, and L. Hester. *Home Networking with IEEE 802.15.4: A Developing Standard for Low-Rate Wireless Personal Area Networks*. IEEE Communications Magazine, August, 2002.

[11] Paolo Baronti, Prashant Pillai, Vince W.C. Choo, Stefano Chessa, Alberto Gotta, and Y. Fun Hu. *Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards*. ScienceDirect, December, 2006.

[12] Karim Yaghmoura, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems*. second edition. O'Reilly Media, August, 2008.

[13] Richard Stallman. *Linux e o projeto GNU*. [online]. Avaliabe: `http://www.gnu.org/gnu/linux-and-gnu.html`. Accessed: March 17, 2013.

[14] Cristopher Hallinan. *Embedded Linux Primer: A Practical Real-World Approach*. second edition. Prentice Hall, November, 2010.

[15] Linux For You. *Device Drivers, Part 4: Linux Character Drivers*. [online]. Avaliabe: `http://www.linuxforu.com/2011/02/linux-character-drivers/`. Accessed: March 17, 2013.

[16] Genne Sally. *Pro Linux Embedded Systems*. Apress, December, 2009.

[17] Samsung. *Samsung S3C6410 Mobile Processor*. [online]. Avaliabe: `http://www.samsung.com/global/business/semiconductor/file/media/s3c6410_datasheet_200804-0.pdf`. Accessed: November 11, 2013.

[18] Microchip. *dsPIC33FJ64MC506A*. [online]. Avaliabe: `http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en541809`. Accessed: November 11, 2013.

[19] Maxim Integrated. *MAX3233E/MAX3235E*. [online]. Avaliabe: `http://www.maximintegrated.com/datasheet/index.mvp/id/2008`. Accessed: November 11, 2013.

[20] Davicom. *DM9000BEP Product Brief*. [online]. Avaliabe: `http://www.davicom.com.tw/userfile/24247/DM9000BEPProductBrief_v1.0.pdf`. Accessed: November 11, 2013.

[21] Microchip. *24AA02E48 EEPROM*. [online]. Avaliabe: `http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en538609`. Accessed: November 11, 2013.

[22] D. A. Godse and A. P. Godse. *Microprocessor, Microcontroller and Applications*. Technical Publications, January, 2008.

[23] Microchip. *MRF24J40MA*. [online]. Avaliabe: `http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en535967`. Accessed: November 11, 2013.

[24] Robert Bosch. *CAN Specification Version 2.0*. [online]. Avaliabe: `http://esd.cs.ucr.edu/webres/can20.pdf`. Accessed: November 11, 2013.

[25] Microchip. *Section 22. Direct Memory Access (DMA)*. [online]. Avaliabe: `http://ww1.microchip.com/downloads/en/DeviceDoc/70348C.pdf`. Accessed: November 11, 2013.

Anexo A

Procedimentos adicionais

Instalação da *toolchain*

Uma toolchain consiste num conjunto de ferramentas de programação que são usadas para produzir ficheiros executáveis numa determinada arquitetura, diferente daquela onde ocorreu a compilação.

Normalmente estas ferramentas são usadas em cadeia, onde o output de uma torna-se no input da outra. Tipicamente uma *toolchain* contém um editor de texto para editar o código fonte, um compilador, um *linker*, um conjunto de *bibliotecas* que fornecem um grupo de funções, macros e/ou outro código auxiliar e um debugger.

Juntamente com a placa de desenvolvimento *FriendlyArm Tiny6410* vêm incluídos dois DVDs que fornecem um conjunto de ferramentas de apoio ao desenvolvimento de software. Uma dessas ferramentas fornecidas é a *cross-compiler toolchain* que pretendemos instalar.

Para instalar a *cross-compiler toolchain* devem ser executados os seguintes passos:

Passo 1: Copiar o ficheiro `DVDA:/linux/arm-linux-gcc-4.5.1-v6-vfp-0101103.tgz` para uma pasta temporária (por exemplo, `/home/tmp/`)

Passo 2: Ir para a pasta `/home/tmp/` através do comando `$cd home/tmp/`

Passo 3: Executar `$sudo tar xvzf arm-linux-gcc-4.5.1-v6-vfp-20101103.tgz -C /`

Passo 4: Editar o ficheiro `~/.bashrc` adicionando a linha `PATH=\$PATH:/opt/FriendlyARM/toolschain/4.5.1/bin` no final

Passo 5: Iniciar de novo a sessão (*logout + login*) ou reiniciar o `~/.bashrc` executando o comando `$source ~/.bachrc`

Neste momento, todos os ficheiros da *cross-compiler toolchain* foram extraídos para a pasta `/opt/FriendlyARM/toolschain/4.5.1`. A instalação foi feita com sucesso se, ao executar o comando `$arm-none-linux-gnueabi-gcc -v`, aparecer uma mensagem idêntica à ilustrada na Figura A.1.

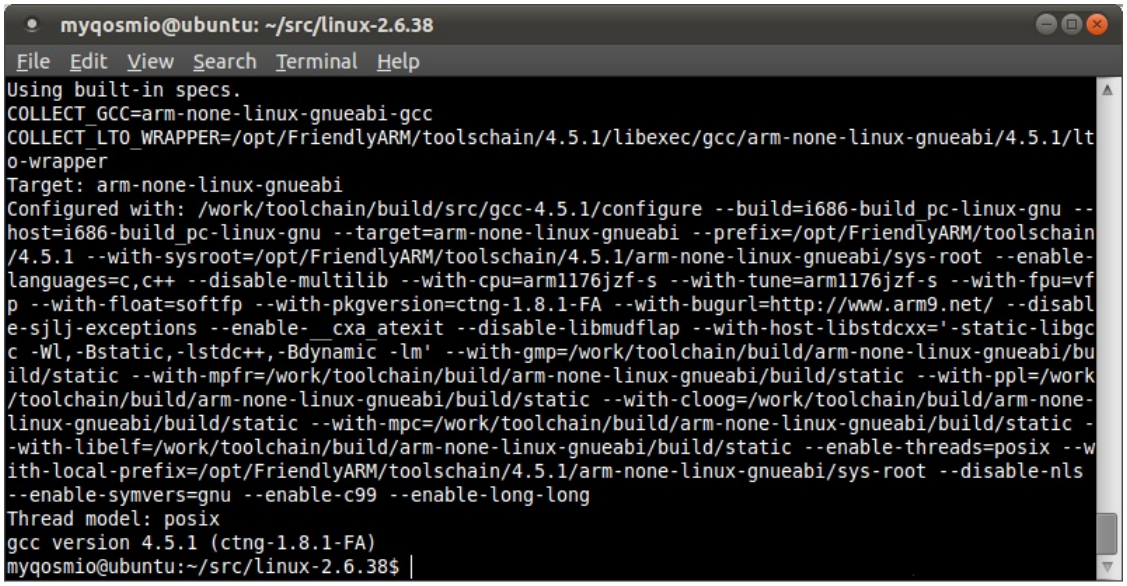


Figura A.1: Processo de instalação da *toolchain*.

Configuração e compilação do *Kernel*

Neste ponto serão abordados os passos utilizados na configuração e compilação da versão `2.6.38` do *Kernel*.

Passo 1: Extrair o ficheiro `DVDA:/linux-2.6.38-20111205.tgz` para `home/$USER/src`. (Para facilitar, renomear o diretório extraída `linux-2.6.38-20111205` para `linux-2.6.38`).

Passo 2: Na consola, executar `$cd home/$USER/src/linux-2.6.38` (diretório de trabalho)

Podemos observar que neste diretório existem um conjunto de ficheiros denominados de `config_mini6410_XXX`, onde `XXX` é substituído pelo correspondente nome, dependendo do ecrã que é pretendido utilizar.

```
config_mini6410_x35 - for Sony 3.5"LCD, 240x320
config_mini6410_w35 - for TFT landscape 3.5"LCD, 320x240
config_mini6410_n43 - for NEC4.3"LCD, 480x272
config_mini6410_l80 - for Sharp 8" (or compatible models)LCD 640x480
config_mini6410_a70 - for 7" true color screen, 800x480
config_mini6410_vga1024x768 - for VGA module, 1024x768
config_mini6410_vga800x600 - for VGA module, 800x600
config_mini6410_vga640x480 - for VGA module, 640x480
config_mini6410_ezvga800x600 - for simple VGA module, 800x600
```

Estes ficheiro darão origem ao já referido ficheiro `.config`.

Passo 3: Renomear o ficheiro `config_mini6410_h43` para `.config` executando o comando `$cp -b config_mini6410_n43 .config`

Passo 4: Executar `$make ARCH=arm xconfig`

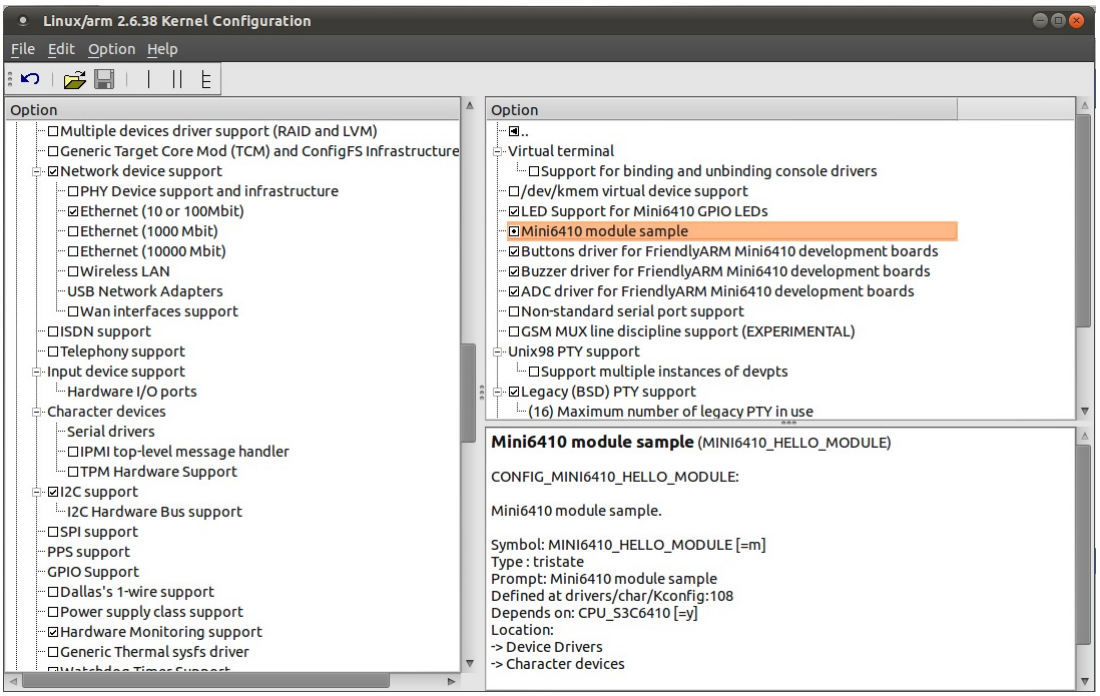


Figura A.2: Menu de configuração do *Kernel*

Na Figura A.2 está ilustrado o menu de configuração do *Kernel*. Serão utilizadas as opções definidas por defeito.

Passo 5: Executar `$make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-zImage`

Este último passo começa a fase de compilação do *Kernel*. O ficheiro resultante de todo o processo de compilação (`zImage`) pode ser encontrado no diretório `/home/$USER/src/linux-2.6.38/arch/arm/boot`.

Instalação através do cartão SD

Neste ponto serão descritos os passos de instalação de todo o software (*Kernel*, sistema de ficheiros e ambiente gráfico) na plataforma *FriendlyArm Tiny6410*[1].

Passo 1: Inserir um cartão *SD* no leitor de cartões do PC

Passo 2: Executar o programa `SD-Flasher.exe` localizado em `DVDA:/tools`

Passo 3: Em `"Image File to Fuse..."` selecione o ficheiro `superboot-6410.bin` localizado em `DVDB:/images`

Passo 4: Clique no botão `Scan` e selecione o cartão *SD* a utilzar e de seguida clique no botão `Scan`

Passo 5: No explorador do *Windows*, abra o diretório raiz do cartão *SD* e crie um diretório com o nome `images`

Passo 6: Crie um ficheiro com o nome `FriendlyArm.ini` no diretório `/images` e crie um subdiretório com o nome **Linux/**

Passo 7: Edite o ficheiro `FriendlyArm.ini` com:

```
#This line cannot be removed. by FriendlyARM(www.arm9.net)
CheckOneButton=No
Action=install
OS= Linux
VerifyNandWrite=No
StatusType = Beeper| LED
#################### Linux ####################
Linux-BootLoader = Linux/u-boot_nand-ram256.bin
```

[1]Estes passos devem se realizados num sistema operativo *Windows*

```
Linux-Kernel = Linux/zImage
Linux-CommandLine = root=ubi0:FriendlyARM-root ubi.mtd=2 rootfstype=ubifs
init=/linuxrc console=ttySAC0,115200
Linux-RootFs-InstallImage = Linux/rootfs_qtopia_qt4-mlc2.ubi
Linux-RootFs-RunImage = Linux/rootfs_qtopia_qt4.ext3
```

Passo 8: Copie os ficheiros `u-boot_nand-ram256.bin`, `rootfs_qtopia_qt4-mlc2.ubi` e `rootfs_qtopia_qt4.ext3` para o diretório `Linux`

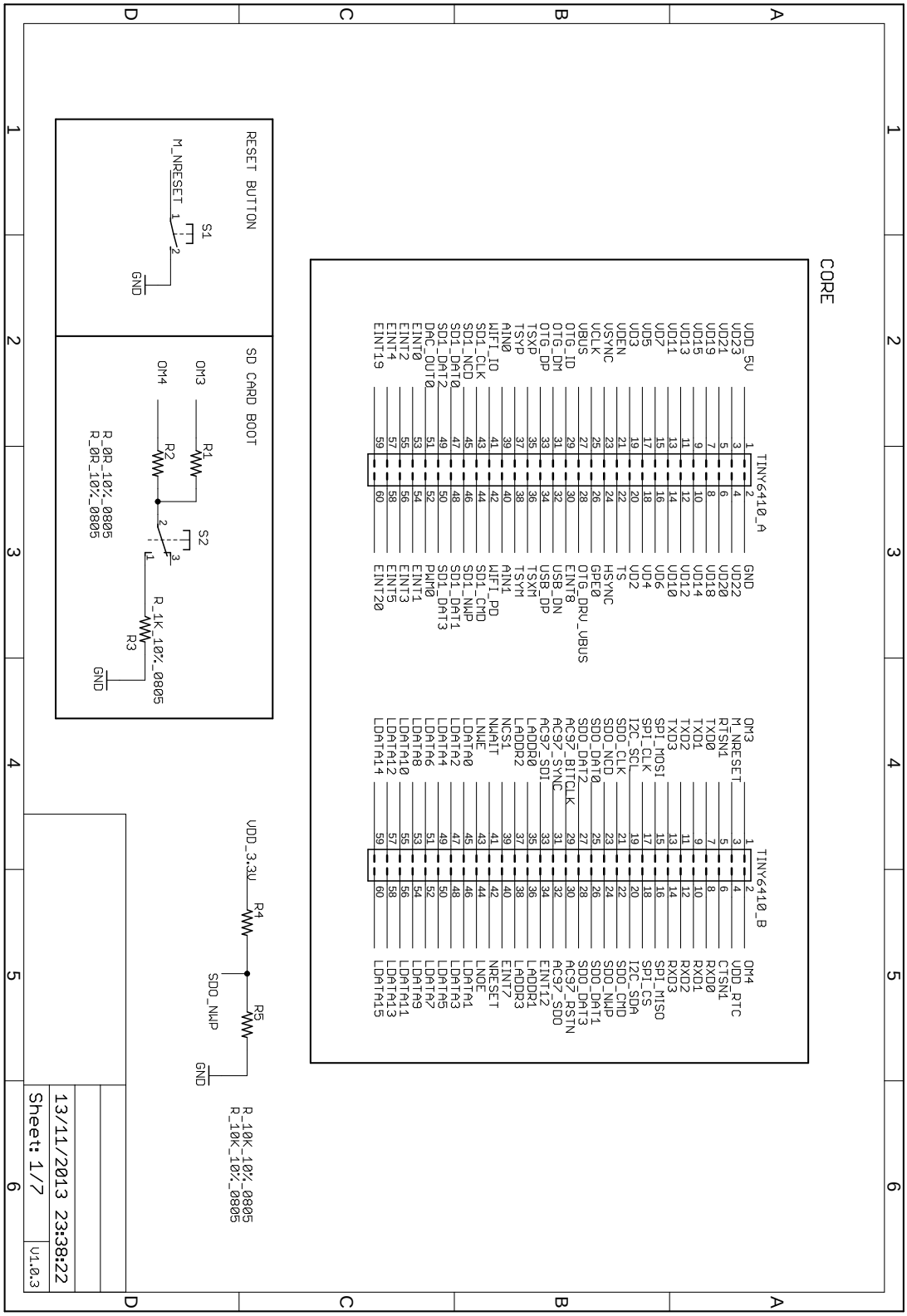
Passo 9: Posicione o interruptor `S2` para a posição `SDBOOT`, insira o cartão e ligue o sistema

Ao ligar o sistema ouvirá um *beep* e o *LED 4* do *Tiny6410 Core* começará a piscar, indicando que o sistema está a ser instalado. Caso isto não aconteça reveja os passos anteriores. No final da instalação, ouvirá dois *beeps* e todos os *leds* começarão a piscar sequencialmente. Trocar a posição do interruptor `S2` e reiniciar o sistema.

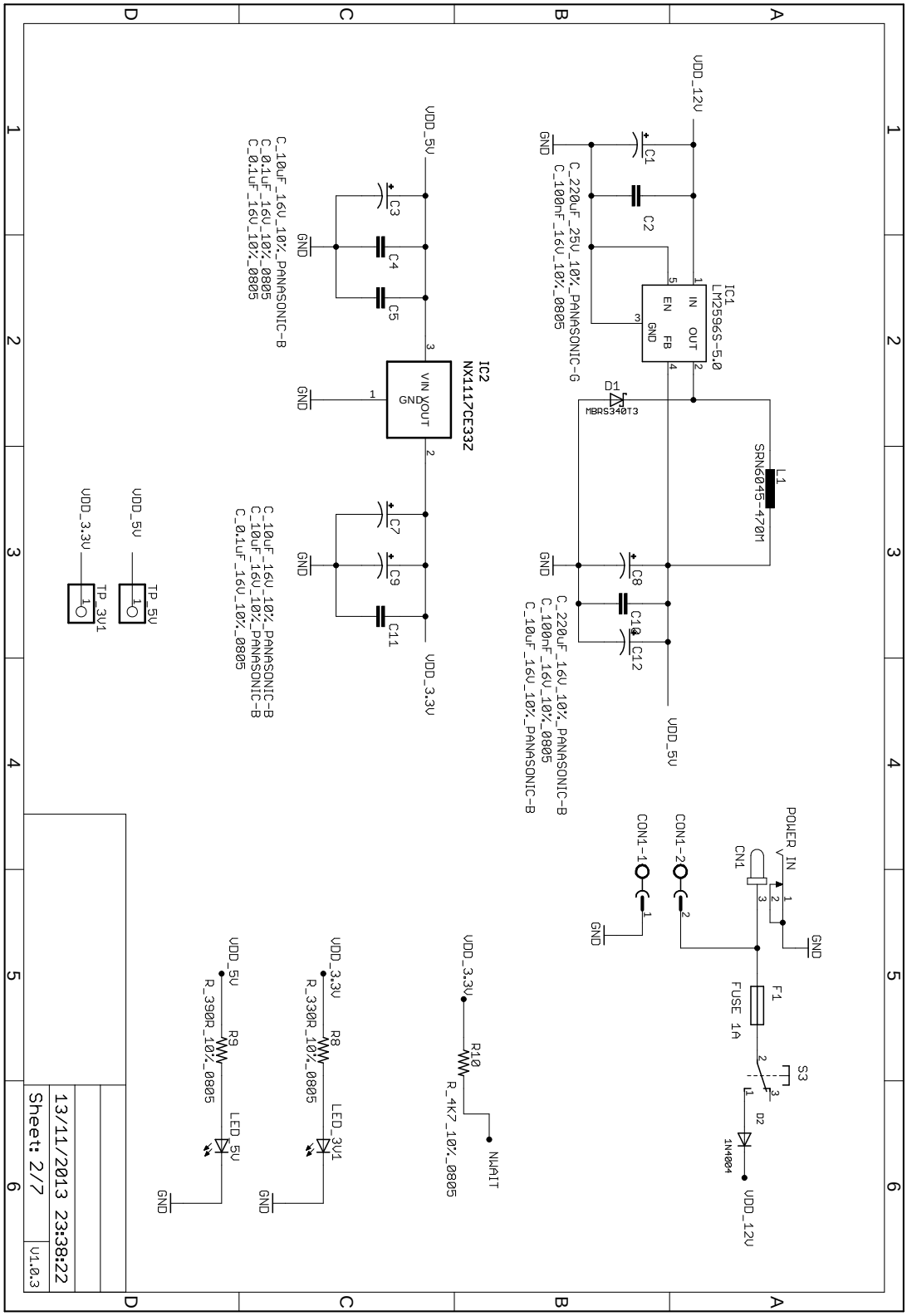
Anexo B

Esquema elétrico

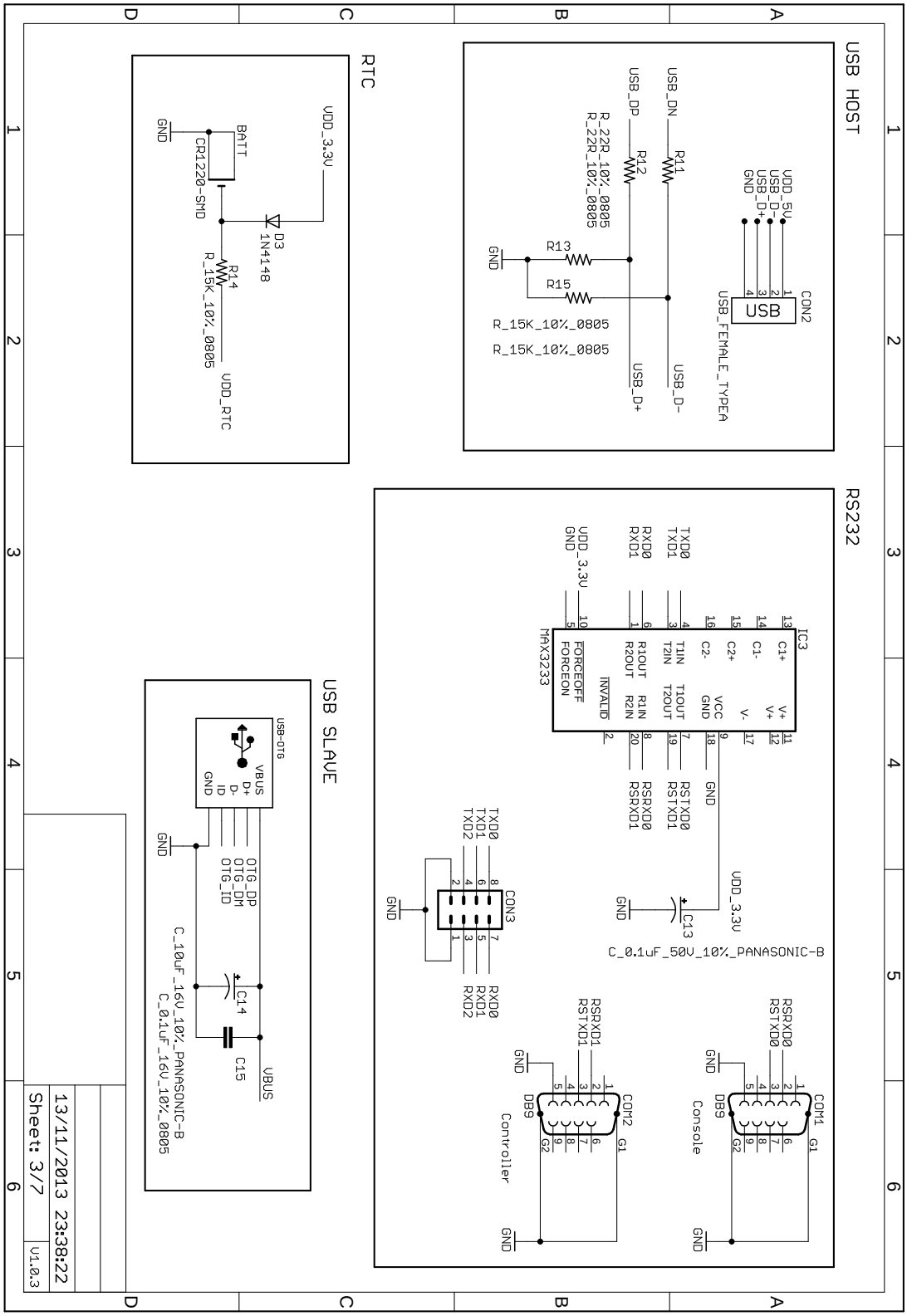
Folha 1


CORE
TINY6410_A
TINY6410_B
VDD_5V
VDD23
VDD21
VDD19
VDD15
VDD13
VDD11
VDD7
VDD5
VDD3
VDEN
VSYNC
VCLK
VBUS
OTG_ID
OTG_DM
OTG_DP
TSXP
TSYP
AIN0
WIFI_IO
SD1_CLK
SD1_NCD
SD1_DAT0
SD1_DAT2
DAC_OUT0
EINT0
EINT2
EINT4
EINT19
GND
VDD22
VDD20
VDD18
VDD14
VDD12
VDD10
VDD6
VDD4
VDD2
TS
HSYNC
GPE0
OTG_DRV_VBUS
EINT8
USB_DN
USB_DP
TSXM
TSYM
AIN1
WIFI_PD
SD1_CMD
SD1_NWP
SD1_DAT1
SD1_DAT3
PWM0
EINT1
EINT3
EINT5
EINT20
OM3
M_NRESET
RTSN1
TXD0
TXD1
TXD2
TXD3
SPI_MOSI
SPI_CLK
I2C_SCL
SDO_CLK
SDO_NCD
SDO_DAT0
SDO_DAT2
AC97_BITCLK
AC97_SYNC
AC97_SDI
LADDR0
LADDR2
NCS1
NWAIT
LNWE
LDATA0
LDATA2
LDATA4
LDATA6
LDATA8
LDATA10
LDATA12
LDATA14
OM4
VDD_RTC
CTSN1
RXD0
RXD1
RXD2
RXD3
SPI_MISO
SPI_CS
I2C_SDA
SDO_CMD
SDO_NWP
SDO_DAT1
SDO_DAT3
AC97_RSTN
AC97_SDO
EINT12
LADDR1
LADDR3
EINT7
NRESET
LNOE
LDATA1
LDATA3
LDATA5
LDATA7
LDATA9
LDATA11
LDATA13
LDATA15
RESET BUTTON
S1
M_NRESET
GND
SD CARD BOOT
OM3
OM4
R1
R2
S2
R_0R_10%_0805
R_0R_10%_0805
R_1K_10%_0805
R3
GND
VDD_3.3V
R4
R5
SDO_NWP
GND
R_10K_10%_0805
R_10K_10%_0805
13/11/2013 23:38:22
Sheet: 1/7
V1.0.3

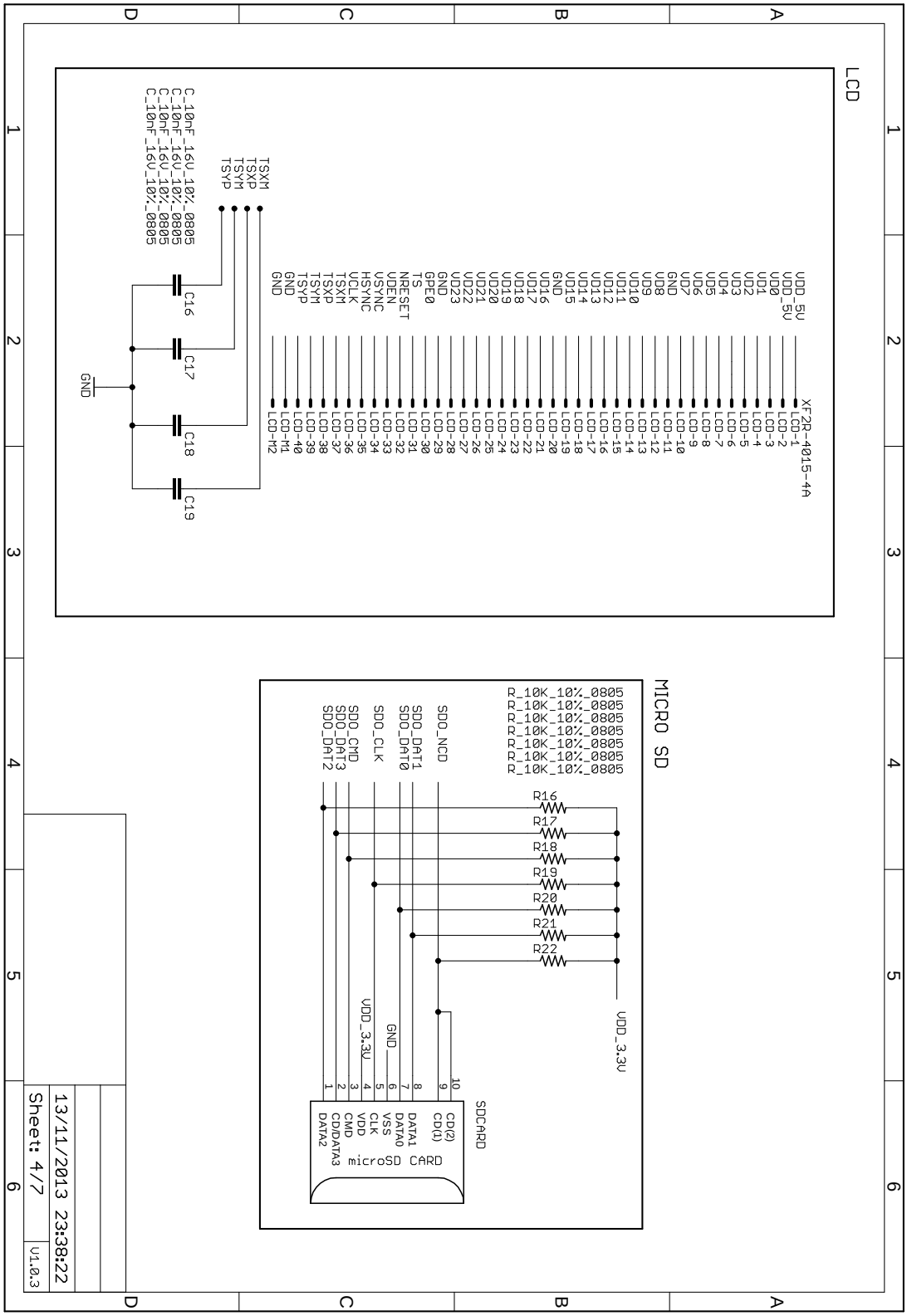
Folha 2


VDD_12V
C1
C2
C_220uF_25V_10%_PANASONIC-G
C_100nF_16V_10%_0805
GND
IC1
LM2596S-5.0
IN
OUT
EN
FB
GND
D1
MBRS340T3
L1
SRN6045-470M
C8
C10
C12
C_220uF_16V_10%_PANASONIC-B
C_100nF_16V_10%_0805
C_10uF_16V_10%_PANASONIC-B
VDD_5V
POWER IN
CN1
CON1-1
CON1-2
F1
FUSE 1A
S3
D2
1N4004
VDD_12V
VDD_5V
C3
C4
C5
C_10uF_16V_10%_PANASONIC-B
C_0.1uF_16V_10%_0805
IC2
NX1117CE33Z
VIN
VOUT
C7
C9
C11
C_10uF_16V_10%_PANASONIC-B
C_10uF_16V_10%_PANASONIC-B
C_0.1uF_16V_10%_0805
VDD_3.3V
TP_5V
TP_3V1
VDD_3.3V
R10
R_4K7_10%_0805
NAAIT
R8
R_330R_10%_0805
LED_3V1
R9
R_390R_10%_0805
LED_5V
Sheet: 2/7
13/11/2013 23:38:22
V1.0.3

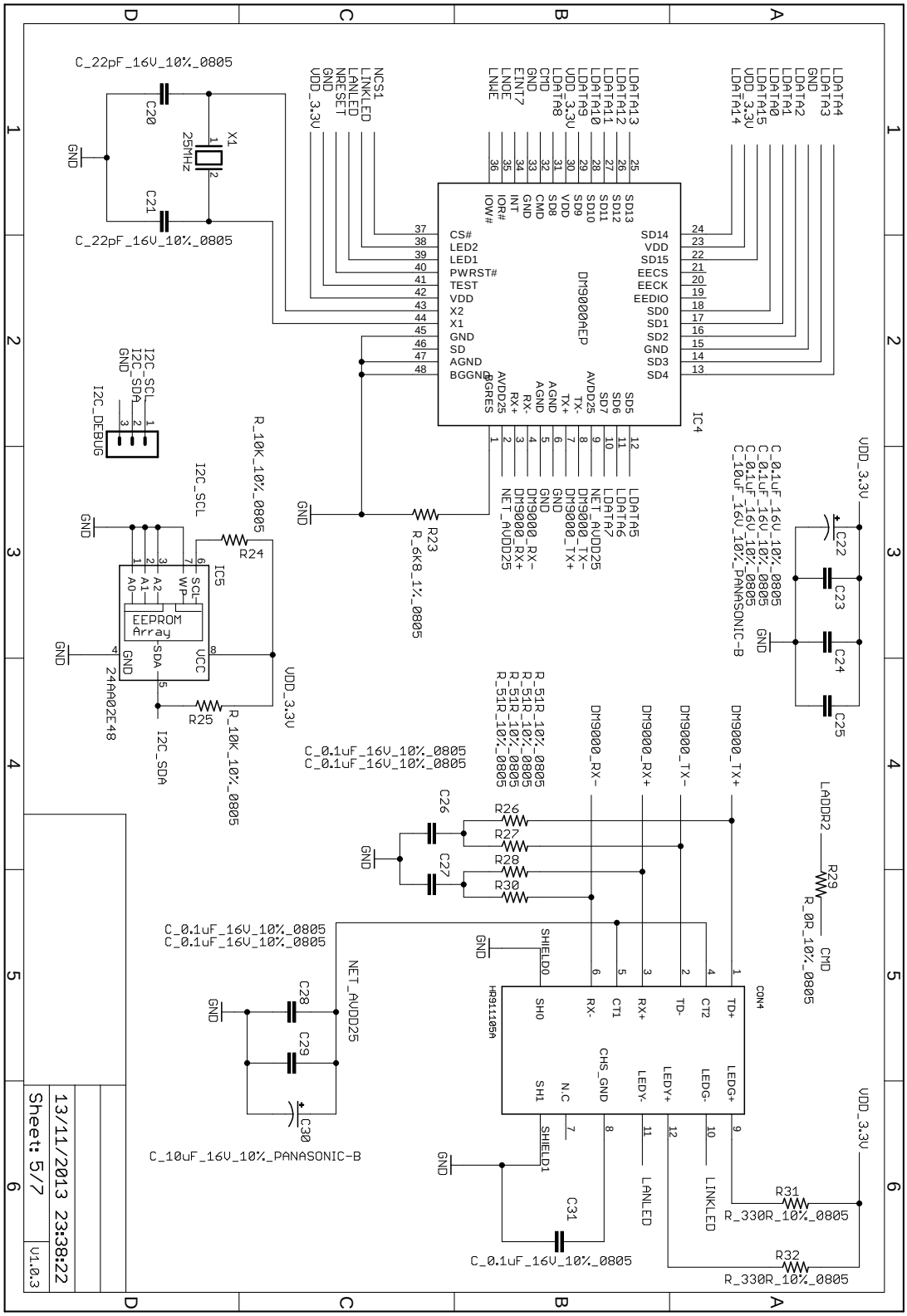
Folha 3


USB HOST
VDD_5V
USB_D-
USB_D+
GND
CON2
USB
USB_FEMALE_TYPEA
USB_DN
USB_DP
R11
R12
R_22R_10%_0805
R_22R_10%_0805
R13
R15
GND
R_15K_10%_0805
R_15K_10%_0805
USB_D-
USB_D+
RTC
VDD_3.3V
BATT
CR1220-SMD
GND
D3
1N4148
R14
R_15K_10%_0805
VDD_RTC
RS232
IC3
MAX3233
FORCEOFF
FORCEON
INVALID
C1+
C1-
C2+
C2-
V+
V-
VCC
GND
T1IN
T2IN
R1OUT
R2OUT
T1OUT
T2OUT
R1IN
R2IN
TXD0
TXD1
RXD0
RXD1
RSTXD0
RSTXD1
RSRXD0
RSRXD1
VDD_3.3V
GND
C13
C_0.1uF_50V_10%_PANASONIC-B
CON3
COM1
COM2
DB9
Console
Controller
G1
G2
USB SLAVE
USB-OTG
VBUS
D+
D-
ID
GND
OTG_DP
OTG_DM
OTG_ID
C14
C15
C_10uF_16V_10%_PANASONIC-B
C_0.1uF_16V_10%_0805
VBUS
13/11/2013 23:38:22
Sheet: 3/7
V1.0.3

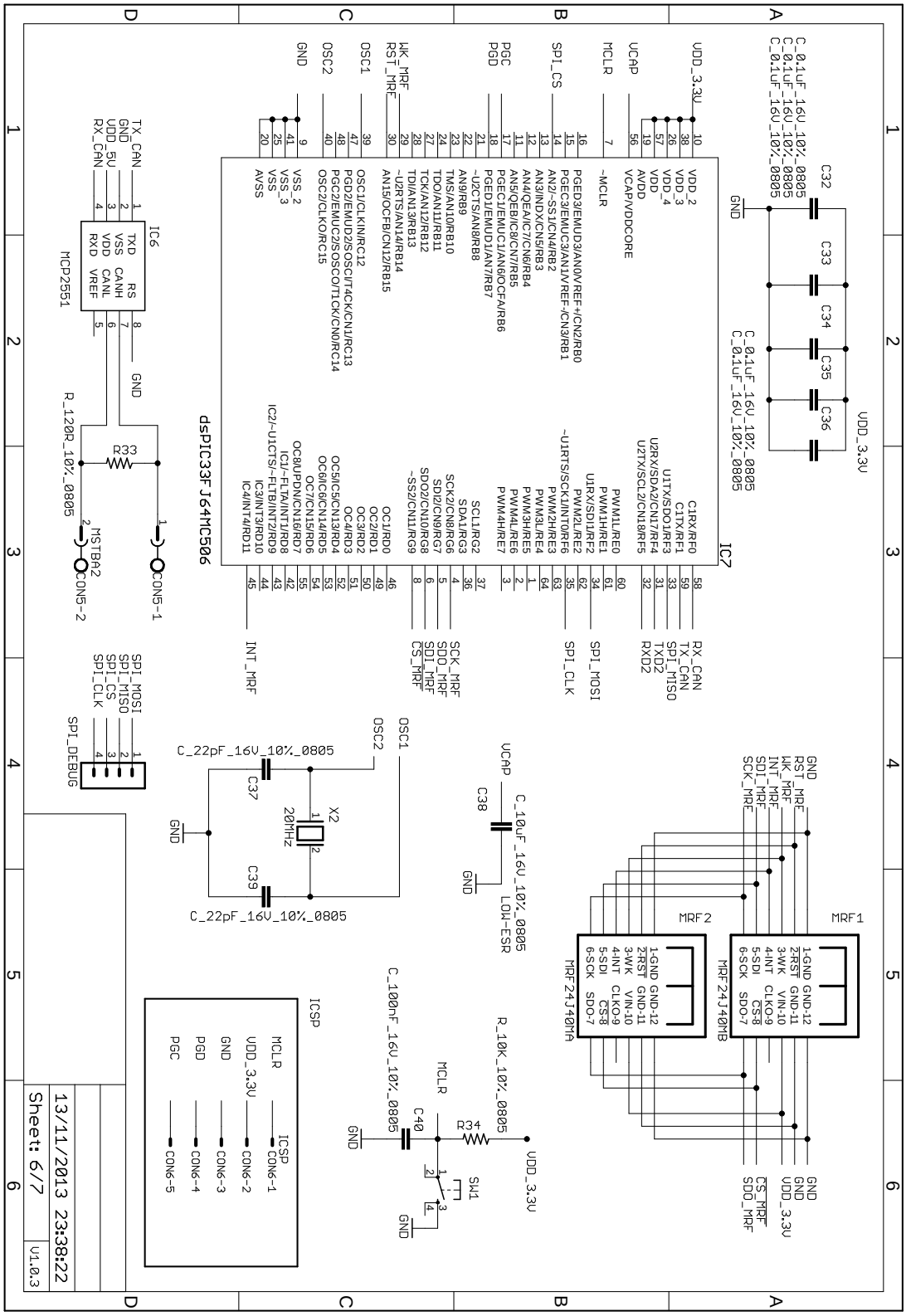
Folha 4


LCD
XF2R-4015-4A
MICRO SD
SDCARD
microSD CARD
VDD_3.3V
GND
C16
C17
C18
C19
R16
R17
R18
R19
R20
R21
R22
13/11/2013 23:38:22
Sheet: 4/7
V1.0.3

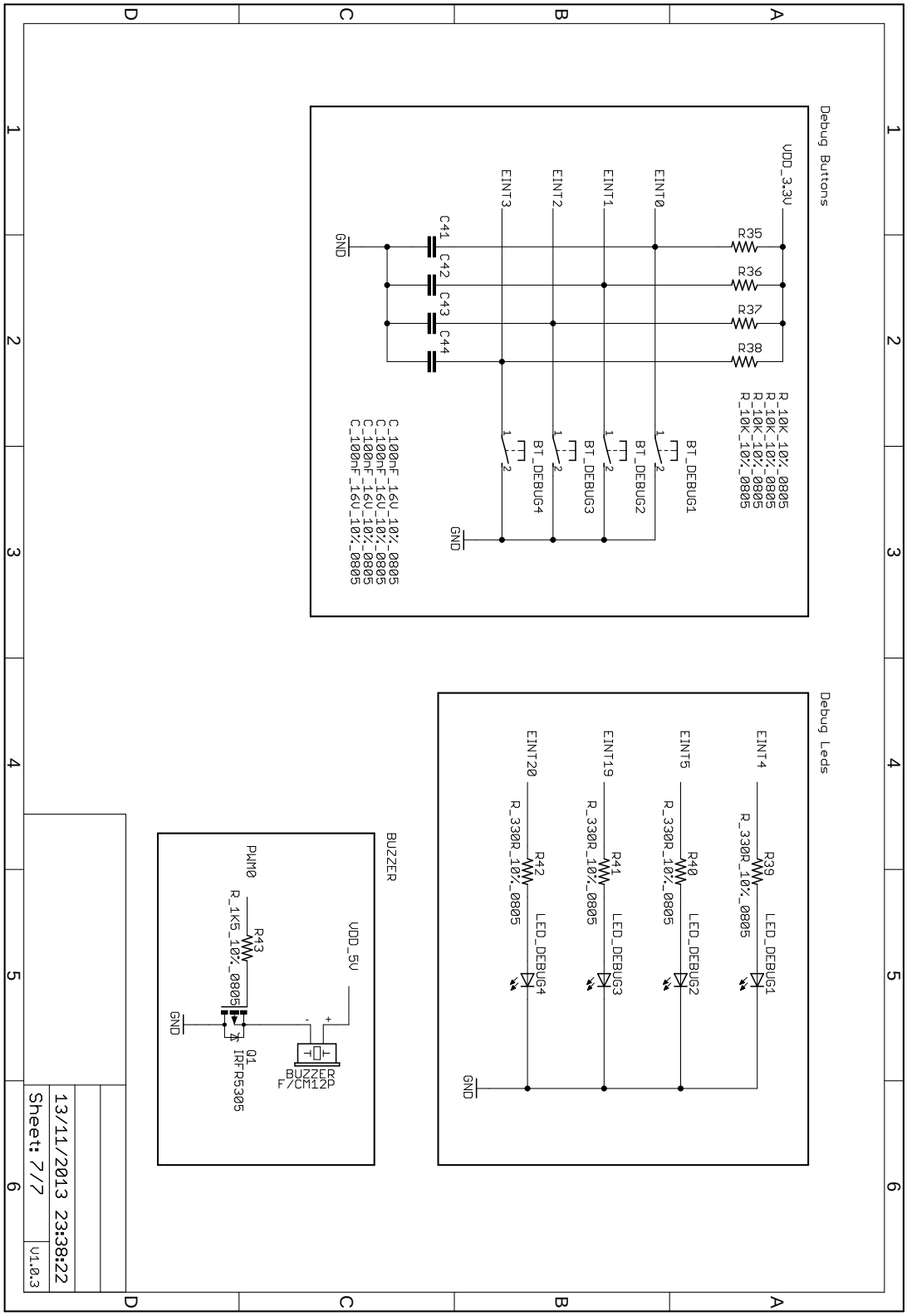
Folha 5


DM9000AEP
IC4
CON4
HR911105A
24AA02E48
EEPROM Array
C_22pF_16V_10%_0805
C_0.1uF_16V_10%_0805
C_10uF_16V_10%_PANASONIC-B
R_10K_10%_0805
R_6K8_1%_0805
R_51R_10%_0805
R_0R_10%_0805
R_330R_10%_0805
C_10uF_16V_10%_PANASONIC-B
C_0.1uF_16V_10%_0805
I2C_DEBUG
I2C_SCL
I2C_SDA
LADDR2
CMD
LINKLED
LANLED
VDD_3.3V
GND
Sheet: 5/7
13/11/2013 23:38:22
V1.0.3

Folha 6


dsPIC33FJ64MC506
IC7
MRF1
MRF2
MRF24J40MB
MRF24J40MA
IC6
MCP2551
ICSP
SPI_DEBUG
C_0.1uF_16V_10%_0805
C_22pF_16V_10%_0805
C_10uF_16V_10%_0805
C_100nF_16V_10%_0805
R_120R_10%_0805
R_10K_10%_0805
20MHz
LOW-ESR
13/11/2013 23:38:22
Sheet: 6/7
V1.0.3

Folha 7


Debug Buttons
VDD_3.3V
EINT0
EINT1
EINT2
EINT3
R35
R36
R37
R38
R_10K_10%_0805
R_10K_10%_0805
R_10K_10%_0805
R_10K_10%_0805
BT_DEBUG1
BT_DEBUG2
BT_DEBUG3
BT_DEBUG4
GND
C41
C42
C43
C44
C_100nF_16V_10%_0805
C_100nF_16V_10%_0805
C_100nF_16V_10%_0805
C_100nF_16V_10%_0805
Debug Leds
EINT4
EINT5
EINT19
EINT20
R39
R40
R41
R42
R_330R_10%_0805
R_330R_10%_0805
R_330R_10%_0805
R_330R_10%_0805
LED_DEBUG1
LED_DEBUG2
LED_DEBUG3
LED_DEBUG4
GND
BUZZER
PWM0
R43
R_1K5_10%_0805
Q1
IRFR5305
VDD_5V
BUZZER
F/CM12P
GND
13/11/2013 23:38:22
Sheet: 7/7
V1.0.3