Universidade de Aveiro
2012
Departamento de Eletrónica, Telecomunicações e Informática

Rui Tiago Ferreira de Pina

CarDroid Locator – Localizador de viaturas com interface Android

Universidade de Aveiro
2012
Departamento de Eletrónica, Telecomunicações e Informática

Rui Tiago Ferreira de Pina

CarDroid Locator – Localizador de viaturas com interface Android

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 a orientação científica do Professor Doutor Rui Manuel Escadas Ramos Martins, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Júri

Presidente	Professor Doutor Manuel Bernardo Salvador Cunha Professor Auxiliar da Universidade de Aveiro
Vogal – Arguente Principal	Professor Doutor José Augusto Almeida Pinheiro de Carvalho Professor Adjunto do Departamento de Eletróncia da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Bragança
Vogal - Orientador	Professor Doutor Rui Manuel Escadas Ramos Martins Professor Auxiliar da Universidade de Aveiro

Agradecimentos

Gostaria de agradecer ao professor Rui Escadas pelo apoio prestado na sua orientação e pela oferta da possibilidade de efetuar um projeto de realização pessoal.
De igual forma aos colegas de curso que me ajudaram quando precisei, o que levou ao culminar desta etapa e consequente concretização deste curso.
Aos meus amigos de longa data, Renato e Mané que me acompanharam durante a vida e sem a sua amizade não seria possível o meu percurso.
A toda a minha família em especial ao meu irmão Miguel, à minha tia Alda, à minha avó Salette e ao meu avô Ferreira que sempre estiveram comigo.
Aos meus pais, dedico este trabalho, por todo o esforço e apoio que me proporcionaram para poder concluir esta etapa.
Por fim, à minha namorada, Ana Patrícia que sempre esteve ao meu lado com o seu apoio e amor. Pode aparecer em último mas porque esse lugar é normalmente guardado para as pessoas mais especiais.
Obrigado a todos.

A maior recompensa ao nosso trabalho
é a concretização pessoal.

Palavras-chave

Alarme automóvel, *Android*, *Car-jacking*, *Car Locator*, *GSM*, *GPS*.

Resumo

O aumento da criminalidade relacionada com o automóvel quer por assalto convencional quer por *car-jacking*, levou ao aparecimento de empresas que providenciam serviços de localização de viaturas. Estes serviços são baseados em dispositivos com tecnologias *GSM* e *GPS* e estão, frequentemente, relacionados com servidores, sendo estes de propriedade das empresas que prestam os serviços. Assim, todos os dados recolhidos pelos dispositivos, são armazenados por terceiros.

Existe ainda um fenómeno mundial que é os *Smartphones*. Dentro destes, o sistema operativo *Android* destaca-se dos demais, por ser de acesso livre e *open-source*.
Analisando o presente e o passado deste fenómeno, conclui-se que o futuro passa por cada ser humano utilizar diariamente um *Smartphone* para todo o tipo de atividades, quer sejam lúdicas, profissionais ou de comodidade.

Aproveitando este fenómeno, pretende-se colmatar a falha que é o fornecimento de dados a terceiros, dos serviços prestados por empresas de localização de viaturas. Assim, pretende-se desenvolver uma aplicação sob o sistema operativo *Android*, de modo aos serviços estarem apenas disponíveis pelos proprietários das viaturas. Pretende-se que a aplicação seja de fácil interação com o utilizador, para lhe proporcionar, além da segurança dos seus veículos, uma agradável interação com os mesmos.
Como não podia deixar de ser, é também pretensão deste projeto, o desenvolvimento do dispositivo que interage diretamente com a viatura, usando para isso as tecnologias *GSM* e *GPS*.

O presente documento é composto por uma introdução que explica o contexto em que se situa. São descritas as ferramentas usadas para a elaboração do mesmo assim como o trabalho realizado, quer da parte da aplicação quer do dispositivo. Para concluir, são enumeradas as conclusões obtidas pela elaboração do projeto assim como algumas considerações a ter em trabalhos futuros.

Keywords Car Alarm, *Android*, *Car-jacking*, *Car Locator*, *GSM*, *GPS*.

Abstract

The increase in crime related to the automobile by conventional assault or by car-jacking, led to the emergence of companies that provide services for locating vehicles. These services are based on devices with GSM and GPS technologies and are often related to servers, which are owned by the companies who provide the services. Thus, all data collected by the devices are stored by third parties.

There is also a global phenomenon that is the smart phones. Within these, the Android operating system stands out from the others, because it is free and open-source.
Analyzing the present and past of this phenomenon, it is concluded that the future is for every human being to use a smart phones daily for all kinds of activities, whether recreational, professional or convenience.

Taking advantage of this phenomenon, it is intended to fill the gaps in the provision of data to third parties for services provided by companies' vehicles location. Thus, we intend to develop an application on the Android operating system, so the services are only available for owners of the vehicles. It is intended that the application is easy to interact with the user, to provide, beyond the safety of their vehicles, a pleasant interaction with them. How could it be, is also pretense of this project, the development of the device that interacts directly with the car, using it for GPS and GSM technologies.

This document consists of an introduction explaining the context in which it is located. We describe the tools used for the development of the project as well as the work done, both on the part of the application or device. Finally, the conclusions obtained are listed by project design as well as some considerations to take into future work.

Índice

Índice de Figuras

Índice de Tabelas

Lista de Siglas e Acrónimos

^{0}C	- Grau Celcius
2G	- 2nd Generation
3GPP	- 3rd Generation Partnership Project
ADT	- Android Development Toolkit
A-GPS	- Assisted Global Positioning System
API	- Application Programming Interface
apk	- Android Application Package File
APN	- Access Point Name
App	- Application
AT Commands	- Attention Commands
AVD	- Android Virtual Device
CAN	- Controller Area Network
CDL	- CarDroid Locator
CEPT	- Conference of European Posts and Telegraphs
CPU	- Central Processing Unit
CTS	- Clear To Send
DCE	- Data Communication Equipment
DCS	- Digital Cellular Service
DETI	- Departamento de Eletrónica, Telecomunicações e Informática
DMA	- Direct Memory Access
DNS	- Domain Name System
DRAM	- Dynamic Random-Access Memory
DTE	- Data Terminal Equipment
EGSM	- Extended Global System for Mobile Communications
EN	- English
FOTA	- Firmware Over-The-Air

FTP	- File Transfer Protocol
GB	- Gigabyte
GHz	- Gigahertz
GPIO	- General Purpose Input/Output
GPRS	- General Packet Radio Service
GPS	- Global Positioning System
GSM	- Global System for Mobile Communications
GUI	- Graphical User Interface
hdpi	- High Dots Per Inch
HTTP	- Hypertext Transfer Protocol
I^2C	- Inter Integrated Circuit
ICCID	- Integrated Circuit Card Identification
id	- Identification
IDE	- Integrated Development Environment
JDK	- Java Development Kit
kbps	- Kilobits por segundo
Km/h	- Quilómetro por hora
ldpi	- Low Dots Per Inch
LED	- Light Emitting Diode
LIN	- Local Interconnect Network
M2M	- Machine-To-Machine
mdpi	- Medium Dots Per Inch
MHz	- Megahertz
NMEA $GGA	- NMEA GPS Fix Data
NMEA $GLL	- NMEA Geographical Latitude and Longitude
NMEA $GSA	- NMEA GPS Receiver Operating Mode
NMEA $GSV	- NMEA Satellites in View
NMEA $RMC	- NMEA Recommended Minimum Specific GPS Data
NMEA $VTG	- NMEA Vector Track and Speed Over Ground

NMEA $ZDA	- NMEA Date and Time
NMEA	- National Marine Electronics Association
PCS	- Personal Communication Service
PDP	- Packet Data Protocol
PNG	- Portable Network Graphics
PSD	- Packet Switched Data
PSP	- Polícia de Segurança Pública
PT	- Português
RAM	- Random Access Memory
RS-232	- Recommended Standard-232
RS-485	- Recommended Standard-485
RTS	- Request To Send
SDK	- Software Development Kit
SIM	- Subscriber Identity Module
SMS	- Short Message Service
SMTP	- Simple Mail Transfer Protocol
SO Android	- Sistema Operativo Android
SPI	- Serial Peripheral Interface
TCP/IP	- Transmission Control Protocol / Internet Protocol
TTFF	- Time To First Fix
UA	- Universidade de Aveiro
UART	- Universal Asynchronous Receiver/Transmitter
UMTS	- Universal Mobile Telecommunications System
USB	- Universal Serial Bus
USB-OTG	- USB On-The-Go
XML	- Extensible Markup Language

1. Introdução

1.1. Motivação e Enquadramento

Nos dias que correm e muito devido à crise económica que se vive, os assaltos são uma realidade que ninguém pode negligenciar. Um dos assaltos mais frequentes é o denominado de ***CarJacking*** que consiste na tentativa ou consumação do roubo da viatura, em que esta é retirada à vítima com uso de força ou ameaça [1]. Segundo o *Relatório Anual de Segurança Interna*, da PSP, foram registados em média, por dia, 60 assaltos a carros, apenas em Lisboa e Porto, em 2011 [2].

Outro fenómeno da atualidade é o sistema operativo ***Android***. O sistema operativo para dispositivos móveis da *Google*, lançado em 2008, registou no fim de 2011 a marca de 400 mil aplicativos disponíveis na loja virtual da *Google* para *downloads* (*Play Store*). Outro número impressionante é as 700 mil ativações diárias que se registam na plataforma *Android* [3].

Juntando estes dois fenómenos atuais, surge a ideia de desenvolver uma aplicação sobre a plataforma *Android*, para combater o *CarJacking e roubo de viaturas*.

A ideia da presente dissertação nasce também da deficiente oferta dada pelos construtores automóveis na área da segurança e alarme. Um dos grandes problemas na segurança automóvel é estes possuírem alarmes em que apenas despertam uma buzina ou uma sirene. Isto de nada impede o automóvel de poder ser assaltado ou mesmo roubado, sem que o proprietário tenha conhecimento, no momento do assalto. Outra deficiência existente é o fato de o proprietário poder perder as chaves da viatura, ou estas terem um qualquer outro problema e com isso ficar impossibilitado de usar a mesma.

Neste sentido, pretende-se desenvolver um sistema que ultrapasse todos estes problemas, usando como interface entre proprietário-veículo a plataforma *Android*.

1.2. Objetivos

É objetivo desta dissertação **desenvolver um dispositivo** eletrónico, controlado remotamente através da tecnologia *GSM*, capaz de ativar vários atuadores, dispor de vários sensores e outros dispositivos e também capaz de obter as coordenadas *GPS*, em tempo real, do local onde se encontra a viatura.

Para isto é usado um módulo *GSM*/*GPS*, um micro controlador para processamento, vários atuadores para acionar alguns dispositivos como o fechar e abrir vidros e portas, o controlo da ignição, o controlo do alarme, entre outros, microfone e câmara para possível vídeo chamada, cartão de memória para gravar dados e um cartão *SIM*, para estabelecer a comunicação entre o dispositivo e o proprietário, através de um telemóvel, por exemplo.

Para ter acesso a toda a informação recolhida pelo dispositivo e ter algum controlo sobre o mesmo, será **desenvolvida uma aplicação** para plataforma *Android*. A aplicação permitirá ao proprietário ter o total controlo sobre as funcionalidades do dispositivo assim como ter acesso a toda a informação disponível, como o local onde se encontra, o estado do alarme, portas e vidros, temperatura, som e imagem do interior da viatura, este último através de uma vídeo chamada.

É também objetivo da presente dissertação usar um **sistema de confidencialidade**, entre dispositivo e aplicação, para garantir a privacidade e segurança total de todo o sistema.

1.3.Antecedentes

O projeto aqui apresentado vem no seguimento de um projeto desenvolvido por alunos de Engenharia Eletrónica e Telecomunicações da Universidade de Aveiro [4], na área de localização de automóveis através de *GPS* [5].

Este projeto consiste num telemóvel convencional ligado a um circuito e sensores, capaz de interpretar *SMS* recebidas e posterior atuação de várias funcionalidades como cortar sistema elétrico do carro ou ativar sirene do alarme. Permite também receber dados de sensores, colocados na viatura, e posteriormente envio de SMS com coordenadas de GPS, para um número de telemóvel pré-definido.

1.4.Estrutura da Dissertação

Esta dissertação encontra-se dividida em seis capítulos, estruturados da seguinte forma:

- **Capítulo 1 – Introdução**: Neste capítulo é apresentado o enquadramento em que o projeto se insere, assim como os objetivos a serem cumpridos.
- **Capítulo 2 – Estado da Arte**: Neste capítulo são descritas as tecnologias usadas para a elaboração do projeto e também alguns produtos semelhantes ao proposto, presentes no mercado.
- **Capítulo 3 - Android**: O terceiro capítulo aborda o Sistema Operativo *Android* e descreve as versões existentes, a sua arquitetura assim como alguns tópicos relativos ao seu funcionamento e programação. São descritas também, algumas ferramentas necessárias ao desenvolvimento de aplicações *Android*.
- **Capítulo 4 - Software – Aplicação CarDroid Locator**: No capítulo quarto, é descrito o *software* criado para a aplicação móvel *Android*.
- **Capítulo 5 - Hardware**: No quinto capítulo, é apresentado o material usado assim como a descrição do dispositivo desenvolvido. É feita ainda uma breve descrição das ferramentas de desenvolvimento usadas e o código desenvolvido para proceder à gestão do dispositivo.

- **Capítulo 6 - Conclusões e Considerações Finais**: Neste capítulo apresentam-se as conclusões do trabalho efetuado, bem como algumas sugestões e considerações a ter em trabalhos futuros.

2. Estado da Arte

Neste capítulo, serão descritas sumariamente as várias tecnologias, usadas para a realização do projeto, assim como alguns produtos semelhantes ao proposto, presentes no mercado. O SO *Android*, pela sua importância neste projeto é deixado para o próximo capítulo.

2.1.Tecnologias

Esta subsecção diz respeito às várias tecnologias utilizadas, para o desenvolvimento da presente dissertação. São elas o *GSM*, *GPRS* e *GPS*.

2.1.1. GSM

GSM ou Sistema Global para comunicações Móveis, é uma tecnologia para comunicações móveis de segunda geração (*2G*), a primeira tecnologia de comunicações móveis digital. Esta tecnologia surge de uma convenção (*CEPT*), realizada em 1982, onde foi formado um grupo denominado de "*Group Special Mobile*" donde surge o acrónimo *GSM*, com o objetivo de estudar e desenvolver um sistema que obedecesse a alguns padrões, uma vez que à época existiam diversos sistemas estudados e desenvolvidos com o grave problema da incompatibilidade entre si. Assim, foi criado um sistema que obedecia aos critérios de boa qualidade de voz, eficiência espetral, terminais que se pudessem considerar "móveis", isto é, pequenos e que fossem de baixo custo, assim como outros critérios como suporte para *roaming* entre muitos outros [6].

O *GSM* opera nas frequências de 900MHz e 1800MHz, na Europa, operando noutras frequências (850MHz e 1900MHz) noutras partes do Globo.

A transferência de dados é feita por comutação de circuitos, onde é estabelecida uma conexão desde a origem ao destino. Isto implica que as operadoras das redes *GSM* cobrem o serviço por tempo de uso.

O *GSM* suporta serviços de chamadas de voz e transferência de dados a uma taxa não superior a 9.6kbps [7].

É um padrão desenvolvido pela *3GPP* e é "*Open-Source*".

2.1.2. GPRS

Serviço de Rádio de Pacote Geral é uma tecnologia criada para ser usada no *GSM*, para melhorar as taxas de transferência de dados. É conhecida como a tecnologia entre a segunda e terceira geração, isto é, 2.5G e permite a transferência de dados por comutação de pacotes, ao invés da anterior comutação por circuitos usada no *GSM*. Assim, os recursos da rede são

atribuídos apenas quando se pretende uma transferência de dados e não durante toda a ligação. Isto permite que vários utilizadores do serviço possam partilhar os mesmos recursos, o que levou à possibilidade de aumentar as taxas de transferência. Com isto, o pagamento de transferências de dados passou a ser pela quantidade de dados transferidos e não pelo tempo de utilização do serviço.

Esta, permite taxas de transferência de dados até 40kbps e possibilita o uso de serviços *online* como *Web*, *FTP*, *email*, mensagens multimédia, entre outros [8].

Entretanto surgiram tecnologias mais avançadas, com muito maior taxa de transmissão de dados (ex.: UMTS, 4G) mas que não são utilizadas para a aplicação que aqui se apresenta. De notar que tal teve o efeito positivo de baixar os custos dos serviços GPRS.

2.1.3. GPS

Sistema de Posicionamento Global é um sistema de navegação por satélite, desenvolvido e operado pelo Departamento de Defesa dos Estados Unidos da América. Este sistema permite determinar a posição, velocidade, fuso horário, entre outros dados de um equipamento recetor de *GPS*, em qualquer parte do Globo, em qualquer hora do dia e sob quaisquer condições climatéricas.

Um aparelho recetor obtém os dados de *GPS* sempre que consiga estar no campo de visão de pelo menos quatro satélites, estando no presente, uma constelação de trinta satélites em órbita. Isto permite a um recetor receber dados de mais satélites, o que resulta numa maior precisão e também permite que em qualquer ponto do Globo, estejam 4 satélites no seu campo de visão.

Atualmente, a precisão de dispositivos comuns centra-se em um metro, existindo equipamentos mais específicos, para uso em trabalhos de cartografia, por exemplo, em que a precisão é maior.

Cada satélite *GPS* transmite um sinal rádio na frequência de 1575.42MHz. Existem outros sinais a serem transmitidos noutras frequências, como o caso da frequência 1227.60MHz que é apenas para uso militar, chegando a precisões centimétricas.

O princípio de funcionamento para a obtenção da posição atual de um recetor, consiste na receção do sinal de quatro satélites GPS (mínimo). Uma vez que cada satélite possui um relógio atómico (relógio de extrema precisão) e o recetor possui também um relógio estável, o recetor calcula a distância a que está do satélite pelo intervalo de tempo em que o sinal foi emitido e recebido. Uma vez que o sinal recebido fornece a posição do satélite e hora de emissão do sinal, após algum processamento de informação, o recetor é capaz de se posicionar no Globo Terrestre [9]. Teoricamente seriam precisos 3 satélites para determinar as coordenadas x,y,z de um dado lugar, mas o 4º satélite é preciso para calcular o Δt entre o relógio do recetor e o dos satélites.

2.2.Produtos no mercado

Atualmente existem alguns produtos comerciais, semelhantes ao proposto na presente dissertação, dos quais se destacam o ***Car Locator*** da empresa *Inosat* [10] e o ***SmartStart*** da *Viper* [11].

2.2.1. Inosat Car Locator

A *Inosat* é uma empresa Portuguesa, fundada em 2000, que detém o marco de **líder Ibérica** em segurança automóvel.

O seu produto *Car Locator* consiste num dispositivo a ser aplicado em viaturas, que fornece alguns dados e a capacidade de imobilizar a viatura através de *SMS*.

O *Car Locator, através de um módulo de GPS,* permite ao utilizador saber o local onde se encontra a sua viatura. Para isso, o utilizador terá que aceder a um *site* usando *login* e *password*, através de um *browser*.

O *Car Locator* permite também o envio de *SMS* para um número pré-estabelecido, através de um módulo *GSM*, contendo vários alarmes. Alguns desses alarmes são:

- Estado da ignição da viatura;
- Estado (ligado/desligado) da bateria da viatura;
- Um alarme sempre que a viatura ultrapasse o limite de velocidade pré-estabelecido;
- Um alarme avisando que a viatura saiu de um perímetro pré-estabelecido.

Assim, o proprietário tem acesso a vários indicadores da sua viatura, tendo como única opção a imobilização da sua viatura, através de uma *SMS* ou através da sua área de cliente do *site*.

Este equipamento conta com a tecnologia *A-GPS*, e tem um preço de 449€. [12]

2.2.2. Viper SmartStart

A *Viper*, fundada em 1986, é uma empresa Americana sediada na Califórnia, **líder da indústria** de segurança de veículos e a primeira do Mundo a desenvolver circuitos integrados, totalmente personalizados para controlo e segurança de veículos.

O seu produto *Viper SmartStart* consiste numa aplicação para *Smartphones* capaz de ativar vários dispositivos de um veículo, desde que este esteja equipado com um módulo (*Viper SmartStart Module*).

Esta aplicação permite controlar vários dispositivos da viatura, como:

- Trancar portas e ativar o alarme;
- Destrancar portas e desativar o alarme;
- Ligar/desligar a ignição;
- Abrir porta da mala da viatura;

- Ativar sirene do alarme em caso de pânico.

É possível também obter a localização da viatura, através de dois métodos distintos.

Um método é através de *GPS*. Para isso, a viatura tem que estar equipada com um módulo de *GPS* (*SmartStart GPS*) e assim pode ser vista a localização da viatura através da aplicação. Outro método é através de guardar a posição da viatura, quando se abandona esta. Este método consiste em guardar a posição da viatura quando se tranca e liga o alarme da viatura. Assim, quando se pretender voltar à viatura, é dada a indicação, através de uma seta indicadora, da posição da viatura. Esta funcionalidade é chamada de *SmartPark*.

A aplicação conta com outras funcionalidades como controlar várias viaturas em simultâneo, controlar as portas/portões da casa/garagem ou usar um cronómetro quando se usa parquímetros e assim avisar o utilizador de quanto tempo lhe resta de estacionamento.

Este produto não está disponível na Europa e tem um custo de 299$US (*Viper SmartStart Module*) e 399$US (*Viper SmartStart GPS Module*). A aplicação é gratuita e encontra-se nas lojas *online* de aplicações para *Smartphones* como *Play Store*, *App Store* ou *BlackBerry App World*. [13]

3. Android

No presente capítulo serão descritas as versões existentes da plataforma *Android*, a sua arquitetura, o ambiente de desenvolvimento usado para a criação da aplicação bem como todo o suporte usado para o desenvolvimento da aplicação móvel.

3.1.Versões

Quanto a versões do sistema operativo *Android*, estas vão sendo atualizadas regularmente, de acordo com a evolução das várias tecnologias, adjacentes aos dispositivos móveis.

Segundo o *site* oficial para programadores de aplicações para o sistema *Android*, as versões existentes são [14]:

Versão	Nome	API
1.0	*Base*	1
1.1	*Base 1.1*	2
1.5	*Cupcake*	3
1.6	*Donut*	4
2.0	*Eclair*	5
2.0.1	*Eclair 0.1*	6
2.1.x	*Eclair MR1*	7
2.2.x	*Froyo*	8
2.3 2.3.1 2.3.2	*Gingerbread*	9
2.3.3 2.3.4	*Gingerbread MR1*	10
3.0.x	*Honeycomb*	11
3.1.x	*Honeycomb MR1*	12
3.2	*Honeycomb MR2*	13
4.0 4.0.1 4.0.2	*Ice Cream Sandwich*	14
4.0.3 4.0.4	*Ice Cream Sandwich MR1*	15
4.1 4.1.1	*Jelly Bean*	16

Tabela 3.1 - Versões e níveis API suportados pela plataforma *Android.* (*Google - Android Developers*)

Desde a primeira versão (1.0 - *Base*) disponível em Outubro de 2008, até à mais recente versão (4.1 *Jelly Bean*) disponível a Julho de 2012, muitos foram os melhoramentos e vários têm sido também, de versão para versão. De destacar um novo recurso, na última versão (*Jelly Bean*), dedicado a interfaces de grandes dimensões, como televisores. Este recurso tenta oferecer ao utilizador, uma nova experiência de utilização do sistema *Android*, para utilizadores que se

encontrem “longe” do dispositivo e assim a interação com o sistema não ser o convencional toque no monitor ou ecrã [15].

Embora os melhoramentos tenham sido muitos, dados oficiais da *Google*, indicam que a versão mais usada por todos os utilizadores de dispositivos *Android* é a *Gingerbread*, com mais de 50%, como mostra a Figura 3.1.

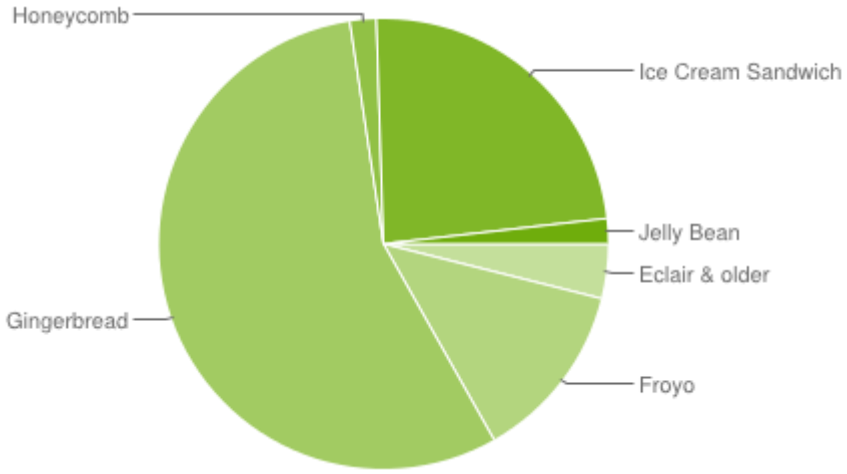

Figura 3.1 - Estudo da *Google* referente às versões que acederam ao *Google Play* nos 14 últimos dias de Setembro de 2012. (*Google - Android Developers*)

A versão *Gingerbread* conta com 55.8%, a *Ice Cream Sandwich* com 23.7%, a *Froyo* com 12.9% e a versão *Jelly Bean* com 1.8%.

Da Figura 3.2 é possível ver que a versão *Gingerbread* mantém-se na liderança, com uma percentagem constante e com grande diferença para as restantes, ao longo do tempo.

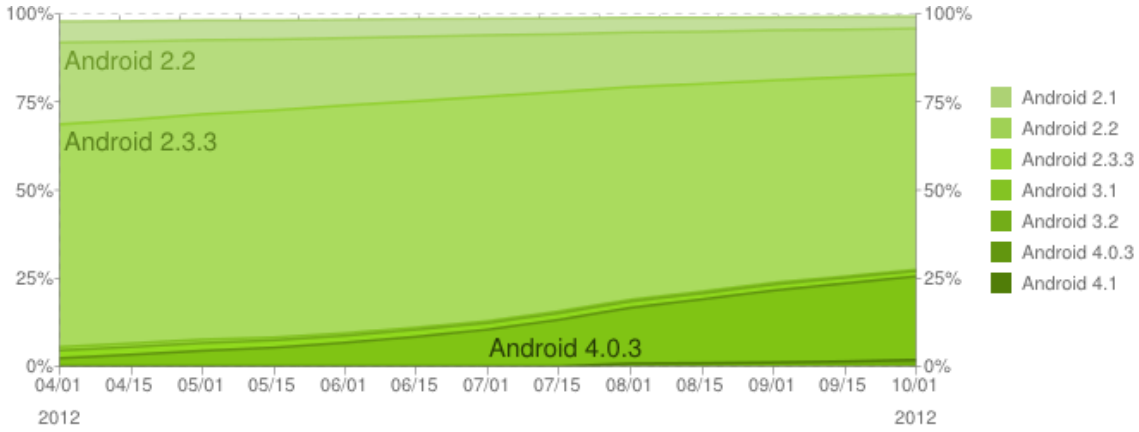

Figura 3.2 - Estudo da *Google* referente às versões que acederam ao *Google Play* entre Abril e Setembro de 2012. (*Google - Android Developers*)

Da Figura 3.2, pode ser visto que a versão *Gingerbread* tem mantido a sua cota de mercado, a versão *Ice Cream Sandwich* tem vindo progressivamente a aumentar a sua cota, estando disponível desde Dezembro de 2011. A versão *Jelly Bean* tem crescido razoavelmente, mas ainda se encontra disponível no mercado à relativamente pouco tempo e ainda não se podem tirar grandes conclusões. As restantes versões têm vindo a perder utilizadores.

Pode concluir-se assim, que a versão "melhor" para começar a desenvolver aplicações é a 2.3.3 (*Gingerbread*), migrando depois para as restantes versões.

3.2.Ambiente de desenvolvimento

No que toca ao ambiente de desenvolvimento, usou-se o *Eclipse* que é um *IDE* (*Integrated Development Environment*) e serve para desenvolver aplicações em *Java*. É o *IDE* aconselhado pela *Google* para desenvolver aplicações para *Android*, uma vez que a *Google* fornece a ferramenta *ADT* (*Android Development Toolkit*).

Tanto o *eclipse* como o *ADT* são livres e "*open source*", o que torna apelativo o seu uso.

São usadas ainda outras ferramentas como o *JDK* (*Java Development Kit*) e *SDK* (*Software Development Kit*) para o desenvolvimento de aplicações para *Android*.

Para uma melhor clarificação do que é e para que serve cada ferramenta, é descrito de seguida a ordem cronológica da instalação das ferramentas assim como uma breve descrição das mesmas.

- Instalação da ferramenta ***JDK***. É uma ferramenta necessária para escrever e correr aplicações escritas em *Java*, que é o caso do *Eclipse*.
- Instalação do ***Eclipse***. É o programa onde é escrita a aplicação em linguagem *Java*.
- Instalação do ***SDK***. É um conjunto de ferramentas de desenvolvimento de *software* que permite a criação de aplicações para várias plataformas ou sistemas.
- Instalação do ***ADT***. É um *plugin* para o *Eclipse* que fornece um conjunto de ferramentas que são integradas com o *IDE Eclipse* [16]. Oferece acesso a diversos recursos que ajudam a desenvolver aplicações *Android* como o acesso ao *GUI* (*Graphical User Interface*), documentação para os "*API levels*" (*Application Programming Interface*) do *Android* ou também editores específicos para criar ficheiros *XML* (e*Xtensible Markup Language*).
- Criação de um ***AVD*** (*Android Virtual Device*). É um emulador que permite criar um dispositivo *Android* virtual, sendo possível escolher várias configurações como a versão *Android* que se pretende, memória que usa ou a dimensão do ecrã, entre outras.

Devido à lentidão do computador usado [17], quando este corria o emulador, foi usado diretamente um dispositivo Android [18] para correr a aplicação e ir fazendo o *debugging* da aplicação ao longo da criação da mesma. Foi também razão para isto haver certos aspetos que o emulador não conseguia reproduzir como a vibração, o som ou o real funcionamento em ambiente "hostil" como o caso do GPS em interiores, entre outros aspetos.

3.3.Suporte ao desenvolvimento

Como suporte ao desenvolvimento, além da disciplina de Computação Móvel frequentada no 1º semestre de 2011 como cadeira de opção, a base de informação foi o próprio *site* oficial do *Android* http://developer.android.com, onde contém toda a documentação necessária e também um fórum de perguntas e respostas *http://stackoverflow.com* onde existe praticamente informação para todas as dúvidas que podem surgir ao longo da criação de uma aplicação.

Foi também usado o *site* *http://www.iconfinder.com* onde se podem encontrar vários *ícones* para uso em aplicações, assim como um site da *Google* *http://android-ui-utils.googlecode.com/hg/asset-studio/dist/index.html* que serve para transformar imagens em *ícones* como *ícones* para menus, notificações, abas, etc. Este último tem uma particularidade que se prende na própria estrutura das aplicações em *Android*, que é o fato dos *ícones* usados nos vários campos, como menus, terem "regras" pré estabelecidas pela *Google*. Essas regras ou recomendações são os tamanhos dos *ícones* para as várias densidades possíveis de ecrãs, as cores usadas para os diferentes estados dos botões ou mesmo os estilos usados para as várias versões do *SO Android*.

Foi usado o *site* *http://soundjax.com* para se obter alguns sons, como alarmes e ainda o *site* *http://www.colorhunter.com* para obter os códigos das cores usadas na aplicação, uma vez que as cores são usadas como códigos como por exemplo o branco "#FFFFFF".

3.4.Arquitetura

A estrutura da arquitetura *Android* pode ser dividida em 5 partes, como se pode ver na Figura 3.3.

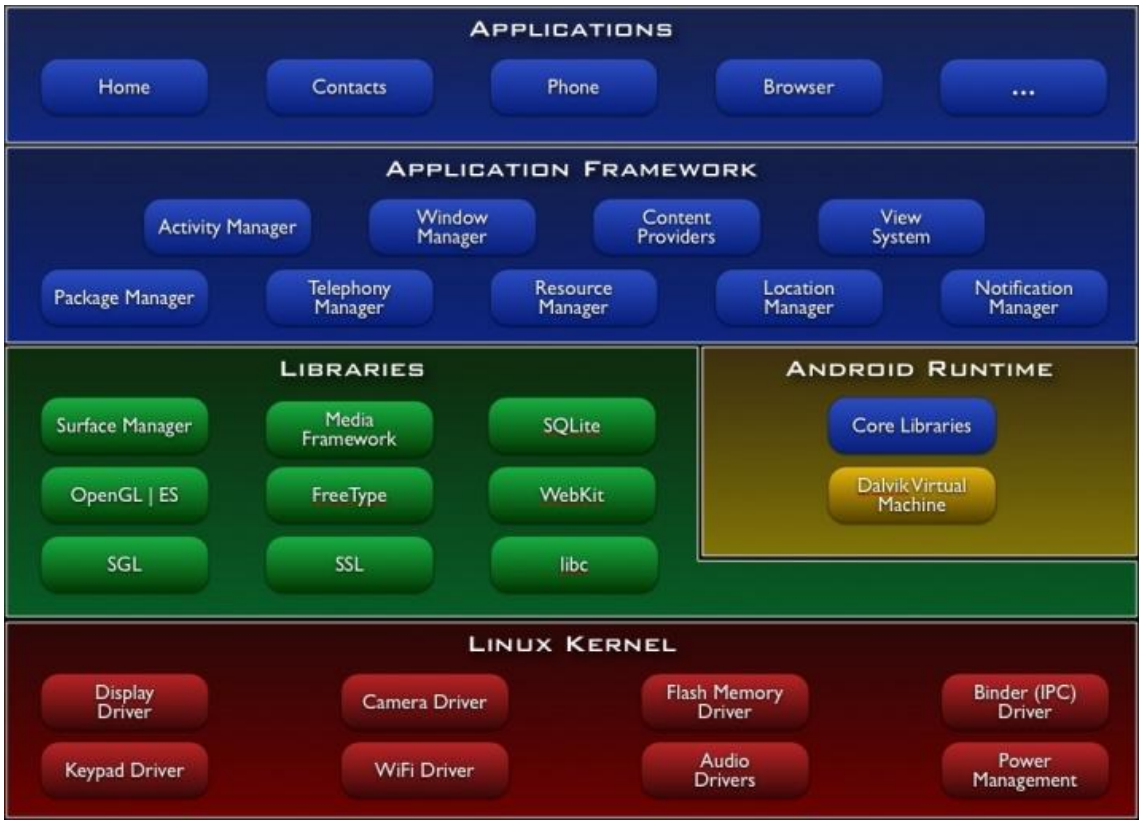

Figura 3.3 - Arquitetura da plataforma *Android.* (*Google - Android Developers*)

Começando pela base, tem-se a ***Linux Kernel***, versão 2.6. Esta, além de permitir uma abstração entre as camadas de *hardware* e *software*, é também responsável pelos serviços centrais do sistema tais como segurança, gestão de memória e de processos, *drivers* para os diversos tipos de *hardware*, comunicação em rede, entre outros.

Depois tem-se a camada ***Android Runtime***, onde se encontra um conjunto de bibliotecas que fornece a maioria das funcionalidades disponíveis nas bibliotecas de linguagem *Java*. Tem-se também a máquina virtual *Dalvik*. Esta é uma máquina virtual otimizada para dispositivos móveis permitindo que um dispositivo possa executar várias máquinas virtuais, simultaneamente, de forma eficiente. Cada aplicação corre no seu próprio processo, utilizando uma instância própria da máquina virtual *Dalvik*.

De seguida tem-se a camada ***Libraries*** que inclui um conjunto de bibliotecas *Android,* escritas em C/C++ usadas por diversos componentes do sistema. Estas são expostas aos programadores de *Android* através da camada seguinte, *Application Framework*. De destacar a biblioteca *Media*, responsável pelo tratamento de imagens *PNG*, por exemplo, e a biblioteca *SQLite*, responsável por uma base de dados leve e muito poderosa.

A camada ***Application Framework*** oferece aos programadores as *APIs* usadas na criação das aplicações nativas do *Android*. Assim, os programadores têm ao seu dispor um vasto leque de funcionalidades, o que torna a programação *Android* tão apelativa e que origina uma grande quantidade de aplicações disponibilizadas na *Play Store*. É ainda possível usar as funcionalidades

que um programador criou, se este as tornar públicas, de aplicação para aplicação, o que torna uma plataforma de desenvolvimento aberta e constantemente alargada.

Por fim, na camada superior ***Applications***, encontram-se as aplicações nativas, escritas em *Java*, tais como calendário, mapas, *browser*, gestor de contatos, programa de trocas de mensagens escritas, entre muitas outras. [19]

3.5.Desenvolvimento

Como referido anteriormente, as aplicações *Android* são escritas em linguagem *Java*.

Uma aplicação *Android* é composta por vários componentes como atividades, serviços, fornecedores de serviços, recetores de *broadcast*, ficheiro manifesto entre outros.

Quando uma aplicação é criada e compilada, esta é empacotada, juntamente com outros recursos utilizados pela aplicação, num pacote com o sufixo *.apk*.

Um pacote *.apk* contém um arquivo manifesto (*AndroidManifest.xml*) onde são declarados todos os componentes da aplicação assim como as bibliotecas usadas, permissões, requisitos da aplicação, versão e configurações de *hardware* necessárias.

O *SO Android* é um sistema *Linux* de vários-utilizadores, onde cada aplicação é um utilizador diferente. Para cada aplicação, é atribuído um identificador de utilizador de *Linux* único. O sistema define as permissões para todos os arquivos, de modo a que apenas a aplicação que tenha o seu identificador único correspondido a esse arquivo, o possa aceder. Desta maneira, cada aplicação acessa apenas aos componentes que realmente necessita para fazer o seu trabalho e não mais que esses. Usando este princípio (*principle of least privilege*), é criado um ambiente muito seguro onde as aplicações não possam interferir ou aceder a partes do sistema que não estejam habilitadas para isso.

Como referido anteriormente, cada aplicação corre num único processo de *Linux*, uma vez que cada processo tem a sua máquina virtual *Dalvik*. Assim, cada aplicação corre isoladamente de outras aplicações. O *SO Android* começa um processo quando um componente da aplicação necessita de ser executado. Quando o processo já não for mais necessário, cabe também ao *SO Android* terminar o mesmo. De notar que o *SO Android* pode terminar um processo quando a memória utilizada estiver próxima do limite máximo. [20]

De modo a que o *SO Android* possa realizar a gestão das aplicações, estas possuem três estados: Ativo, Parado e Pausado. As transições entre estes estados são realizadas através de sete métodos, como demonstra o ciclo de vida de uma atividade na Figura 3.4.

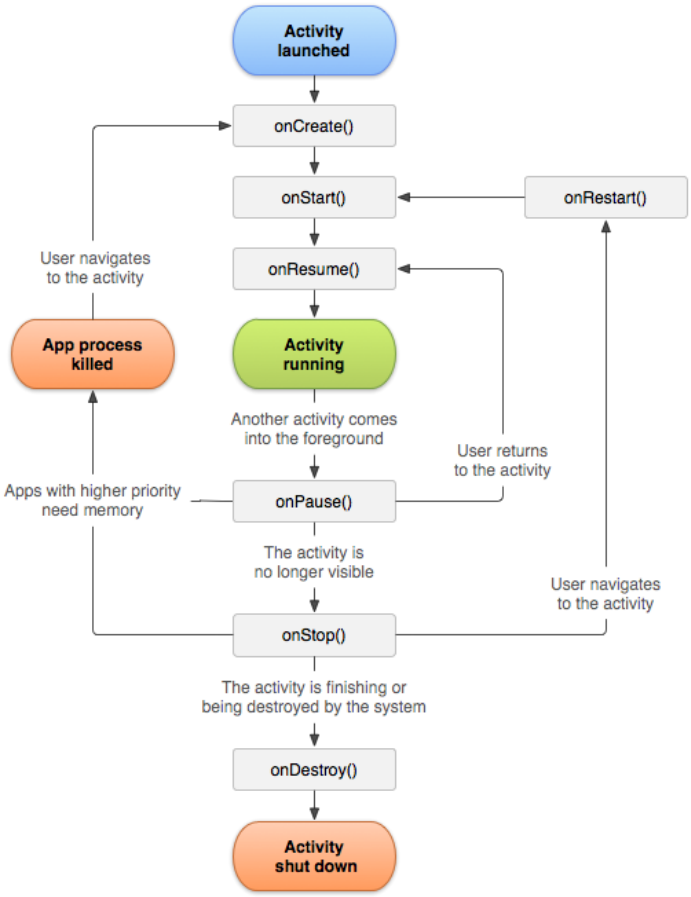

Figura 3.4 - Ciclo de vida de uma atividade. (*Google - Android Developers*)

Uma atividade, geralmente, corresponde a um ecrã. Cada atividade pode iniciar uma outra atividade, ou várias. Sempre que uma atividade é iniciada, a atividade anterior é parada, com o método *onStop()*, ficando esta preservada na "*back stack*". Assim, sempre que o utilizador pressionar o botão "*back*", a atividade atual é destruída, através do método *onDestroy()* e é resumida a atividade que estava na "*back stack*" através do método *onResume()*.

A atividade está visível para o utilizador apenas entre os métodos *onStart()* e *onStop()*, embora possa existir outra atividade "à frente" desta, estre os estados *onPause()* e *onStop()*, parcialmente transparente ou não cobrir a totalidade do ecrã. Assim que a atividade entra no estado "*stopped*", está é colocada em *background* e não é vista pelo utilizador. [21]

Ao contrário de uma atividade, um serviço não possui interface visual ou ecrã. Este componente corre em segundo plano e é usado para executar operações de longa duração.

Como se pode ver na Figura 3.5, o ciclo de vida de um serviço é diferente do ciclo de vida de uma atividade. Assim, todo o tempo de vida de um serviço ocorre durante o método *onCreate()* e *onDestroy()*. Um serviço começa quando é invocado o método *startService()*, através de um *Intent* e termina quando é invocado o método *stopService()*. [22]

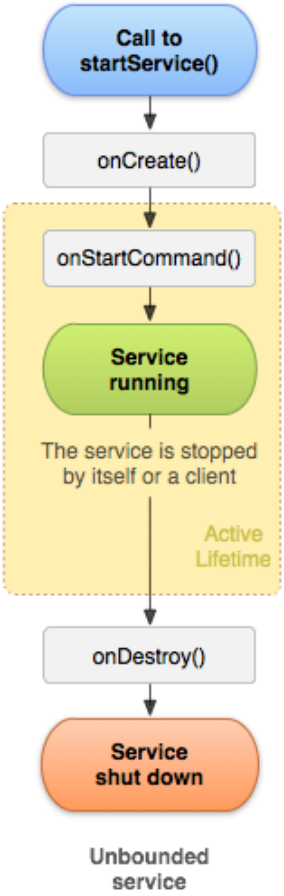

Figura 3.5 - Ciclo de vida de um serviço. (*Google - Android Developers*)

Outro componente de uma aplicação é o *Broadcast Receiver*. Este componente é usado essencialmente para responder a eventos. Destina-se a executar uma quantidade mínima de trabalho, como por exemplo iniciar um serviço. Ao contrário das atividades, este não possui uma interface visual, mas pode por exemplo criar uma notificação na barra de estado, onde seja alertado o utilizador para algum evento que tenha surgido.

Quanto aos recursos da aplicação, estes são declarados no ficheiro *R.java*, que é gerado no momento da compilação da aplicação. Estes recursos são por exemplo imagens, sons, ficheiros de *layout*, entre outros e encontram-se em subpastas da pasta *res*.

De modo a permitir uma maior abrangência na divulgação de uma aplicação, é possível disponibilizar diferentes recursos consoante o idioma, configuração do dispositivo como resolução de ecrã, entre outros. Cabe ao sistema escolher as subpastas, consoante as configurações do dispositivo.

4. Software – Aplicação CarDroid Locator

O *software* prende-se numa aplicação para dispositivos móveis, usando a plataforma *Android*.

A aplicação é uma de duas partes que compõem o produto que se pretende desenvolver, na presente dissertação. Assim, a aplicação em conjunto com o dispositivo formam um só fim. Esta comunica com o dipositivo através de SMS, onde troca comandos, que serão descodificados para despoletar as ações pretendidas. A aplicação pretende oferecer ao utilizador uma interação intuitiva com as suas viaturas. Para isso é composta por 3 partes. A parte principal, onde oferece todas as ações a tomar com a viatura como o controlo do alarme, dos periféricos assim como a exibição de alguns dados da viatura. Uma parte relativa às configurações da aplicação, onde o utilizador pode configurar a aplicação da maneira que pretenda. E uma terceira parte, onde o utilizador pode armazenar todas as suas viaturas de modo a poder selecionar a que pretenda controlar.

Neste capítulo será apresentado o fluxograma da aplicação e serão enumerados todos os ficheiros criados. Os recursos usados na aplicação serão descritos em anexo. Será feita também, com algum detalhe, a explicação do funcionamento da aplicação.

Em termos gerais, a aplicação apresenta o seguinte aspeto:

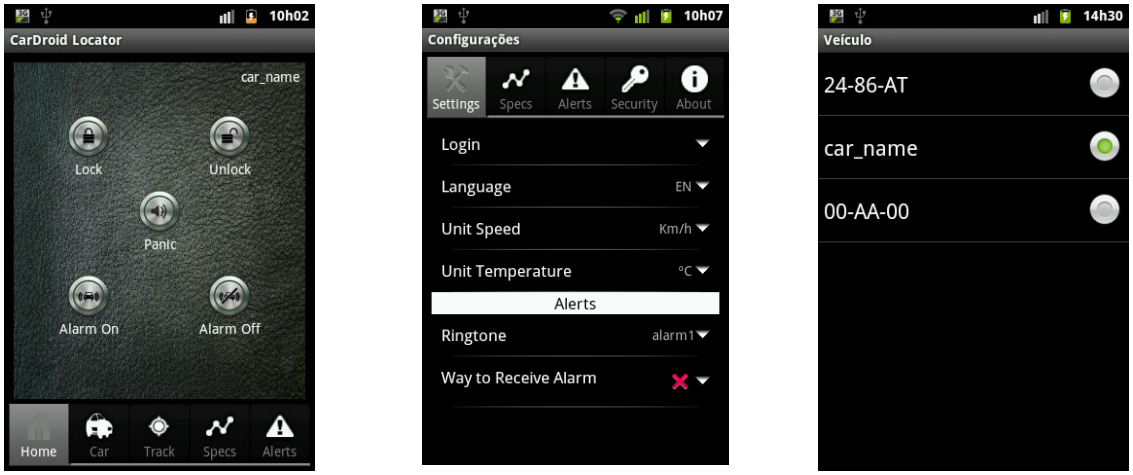

Figura 4.1 – Layout inicial à esquerda, layout inicial do menu de configurações ao centro e layout de menu de veículos à direita.

A aplicação resulta de três menus, sendo eles "*Home*", "*Configurations*" e "*Car (switch)*".

Na Figura 4.1, na imagem da esquerda é apresentado o ecrã inicial da aplicação, inserido no menu "*Home*". Como pode ser visto, existem cinco ecrãs disponíveis no menu "*Home*", sendo todos eles dedicados a ações a tomar com a viatura.

Na imagem do centro é apresentado o ecrã inicial do menu "*Configurations*". Este menu contém cinco ecrãs, estando todos eles relacionados com configurações da aplicação.

Por último, na imagem da direita é apresentada a lista de veículos inseridos na aplicação, que representa o único ecrã do menu "*Car (switch)*".

4.1.Fluxograma

Os fluxogramas seguintes representam a sequência operacional possível, através dos vários *layouts* existentes na aplicação. Deste modo, pretende-se dar uma ideia geral de como se pode interagir com os vários *layouts* da aplicação.

A aplicação pode ser dividida em 3 partes, de um ponto de vista visual.

A primeira, prende-se na aba inicial (*Tab1*) onde contém 5 *layouts* distintos (*home*, *car*, *track*, *specs* e *alerts*). Dentro de cada um destes *layouts* existe um menu (*menu1*) que contém três campos (*Conf*, *CarChoose* e *Exit*). O *layout track* contém o *menu4* que contém 6 campos, 3 deles são os mesmos que em *menu1* e outros 3 (*SavePosition*, *DeleteMarker* e *Layers*) prendem-se em opções relativas ao próprio *layout track*.

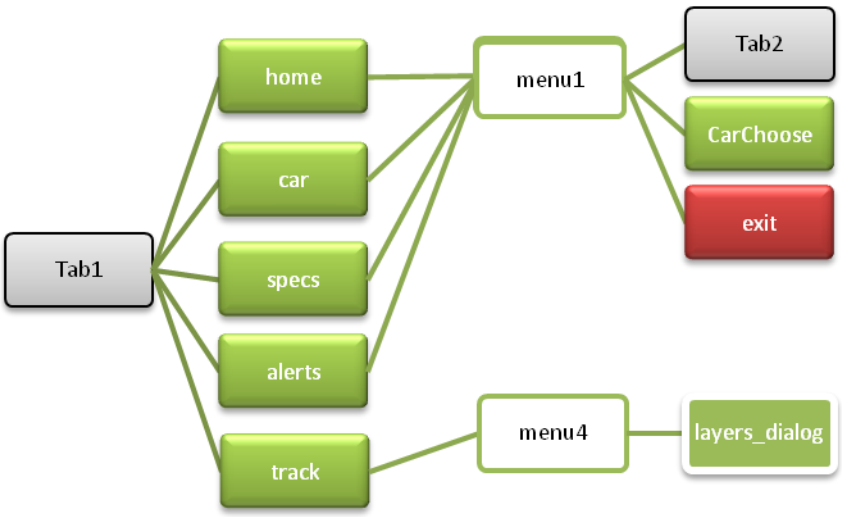

Figura 4.2 - Fluxograma da aba principal.

A segunda prende-se na segunda aba (*Tab2*) que diz respeito às configurações da aplicação. Esta pode ser acedida através do *menu1*, anteriormente explicado. Dentro desta aba existem 5 *layouts* distintos, todos eles de configurações. São eles *confconf*, *confspecs*, *confalerts*, *confsecurity* e *confabout*. Em todos eles existe um menu (*menu2*) que contém a opção *Home*, que retorna a aplicação para a primeira aba, para o *layout home*.

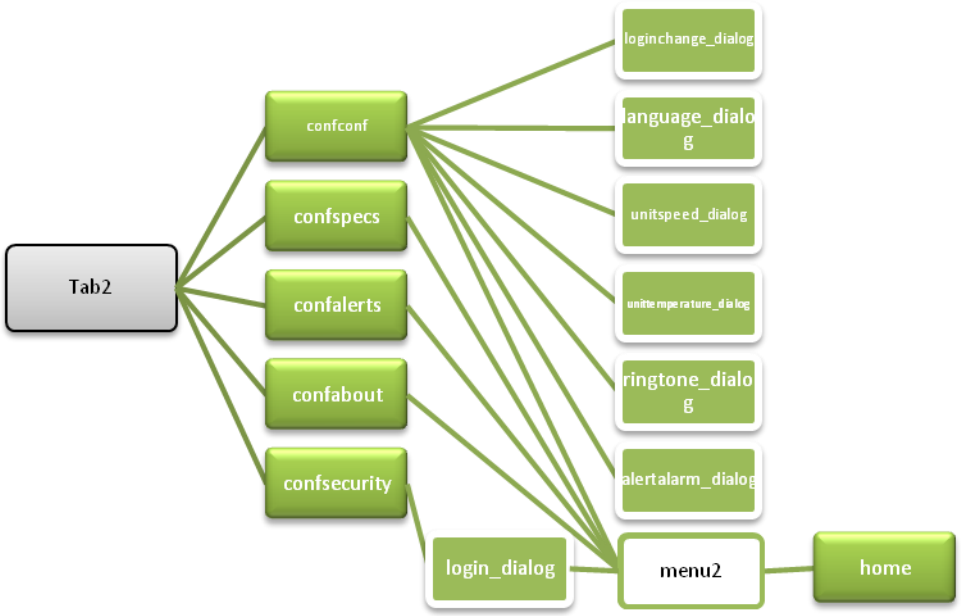

Figura 4.3 - Fluxograma da aba de configurações.

Por fim tem-se a parte *CarChoose*, que se prende numa lista das viaturas inseridas, na base de dados da aplicação, pelo utilizador. Neste *layout* existe também um menu (*menu3*) onde tem as opções *Home* e *AddCar*. A primeira para retornar à primeira aba e a segunda para adicionar novas viaturas, à base de dados da aplicação.

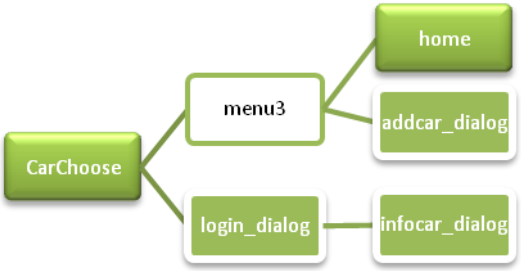

Figura 4.4 - Fluxograma de CarChoose.

Além dos *layouts* anteriormente referidos, existem ainda outro tipo de *layouts* que são *Dialogs*. Estes *layouts* são listas de opções, informações ou campos editáveis, relativamente ao *layout* principal e são apresentados por cima do *layout* principal, não usando todo o ecrã. São eles *layers_dialog*, disponível em *menu4* que contém uma lista de 2 opções. *Login_dialog*, disponível em *CarChoose,* se for pressionado continuamente um item da lista e é usado para inserir um *login* e uma *password* para ficar acessível o *Dialog inforcar_dialog*, que contém informação relativa ao

item (viatura) selecionado. Este *login_dialog* pode ser encontrado também no *layout security*. Aqui tem o objetivo de permitir ao utilizador usar o *layout*, se inserir o *login* e *password* corretamente. Existem ainda os *Dialogs* presentes em *confconf*. São eles *loginchange_dialog*, onde é possível alterar a *password*, *language_dialog* para alterar o idioma da aplicação, *unitspeed_dialog* e *unittemperature_dialog* para escolher as unidades de velocidade e temperatura, respetivamente, *ringtone_dialog* para escolher o som dos alertas e por fim *alertalarm_dialog* para escolher a maneira de se receber o alerta de alarme da viatura.

Fica assim explicado, de uma maneira geral, os vários *layouts* existentes e a possível sequência entre eles. Posteriormente será explicado com algum detalhe, as várias atividades existentes, assim como a relação entre as atividades e os *layouts* e toda a estrutura existente como base de dados, mapas, recursos, etc., da aplicação.

4.2.Pastas e Ficheiros

Nesta seção são apresentadas as pastas e os ficheiros criados, no desenvolvimento da aplicação, à exceção do ficheiro *CarDroidLocator.apk* e *R.java* que foram gerados pelo próprio ambiente de desenvolvimento, já discutido em 3.5. Aqueles que têm a terminação *.png* são imagens, a terminação *.mp3* são sons, os ficheiros com a terminologia *.java* são ficheiros de código fonte onde é desenvolvido realmente o código para as várias atividades, serviços, base de dados, etc., assim como toda a interação entre ambos. Todos os outros são pastas.

Os ficheiros com a terminologia *.xml* são os *layouts* ou *screens*, que são a parte visível para o utilizador, à exceção dos ficheiros *strings.xml* e *AndroidManifest.xml*.

A árvore de ficheiros é a que se segue:

- *src*
  - *com.ControlDroid.CarDroidLocator*
    - *AddCar.java*
    - *AddItemOverlay.java*
    - *AlertReceiver.java*
    - *Alerts.java*
    - *Car.java*
    - *CarChoose.java*
    - *CDLService.java*
    - *ConfAbout.java*
    - *ConfAlerts.java*
    - *ConfConf.java*
    - *ConfSecurity.java*
    - *ConfSpecs.java*
    - *Home.java*

Login.java

SendSMS.java

Specs.java

StartCDLServiceAtBoot.java

Tab1.java

Tab2.java

Track.java

- *com.ControlDroid.CarDroidLocator.DataBase*

  AlertsDBAdapter.java

  CarDroidLocatorDBHelper.java

  CarsDBAdapter.java

- *gen*
  - *com.ControlDroid.CarDroidLocator*

    R.java
- *bin*
  - *CarDroidLocator.apk*
  - *…*
- *res*
  - *drawable-hdpi*

    ic_android.png

    ic_button_alarmoff_normal.png

    ic_button_alarmoff_pressed.png

    ic_button_alarmoff.xml

    ic_button_alarmon_normal.png

    ic_button_alarmon_pressed.png

    ic_button_alarmon.xml

    ic_button_camera_normal.png

    ic_button_camera_pressed.png

    ic_button_camera.xml

    ic_button_engineoff_normal.png

    ic_button_engineoff_pressed.png

    ic_button_engineoff.xml

    ic_button_engineon_normal.png

    ic_button_engineon_pressed.png

    ic_button_engineon.xml

    ic_button_lock_normal.png

    ic_button_lock_pressed.png

    ic_button_lock.xml

- *ic_button_panic_normal.png*
- *ic_button_panic_pressed.png*
- *ic_button_panic.xml*
- *ic_button_trunk_normal.png*
- *ic_button_trunk_pressed.png*
- *ic_button_trunk.xml*
- *ic_button_unlock_normal.png*
- *ic_button_unlock_pressed.png*
- *ic_button_unlock.xml*
- *ic_button_windowdown_normal.png*
- *ic_button_windowdown_pressed.png*
- *ic_button_windowdown.xml*
- *ic_button_windowup_normal.png*
- *ic_button_windowup_pressed.png*
- *ic_button_windowup.xml*
- *ic_cdl.png*
- *ic_marker_blue.png*
- *ic_marker_red.png*
- *ic_menu_car.png*
- *ic_menu_config.png*
- *ic_menu_deletepos.png*
- *ic_menu_exit.png*
- *ic_menu_home.png*
- *ic_menu_layers.png*
- *ic_menu_newcar.png*
- *ic_menu_savepos.png*
- *ic_tab_about_selected.png*
- *ic_tab_about_unselected.png*
- *ic_tab_about.xml*
- *ic_tab_alerts_selected.png*
- *ic_tab_alerts_unselected.png*
- *ic_tab_alerts.xml*
- *ic_tab_car_selected.png*
- *ic_tab_car_unselected.png*
- *ic_tab_car.xml*
- *ic_tab_config_selected.png*
- *ic_tab_config_unselected.png*
- *ic_tab_config.xml*

ic_tab_home_selected.png
ic_tab_home_unselected.png
ic_tab_home.xml
ic_tab_security_selected.png
ic_tab_security_unselected.png
ic_tab_security.xml
ic_tab_specs_selected.png
ic_tab_specs_unselected.png
ic_tab_specs.xml
ic_tab_track_selected.png
ic_tab_track_unselected.png
ic_tab_track.xml
ic_ua.png
screen_car2.png
screen_leather.png
shape_alertdialog.xml

- *drawable-ldpi*

  …(mesmos ficheiros que em *drawable-hdpi*, com baixa resolução)…

- *drawable-mdpi*

  …(mesmos ficheiros que em *drawable-hdpi*, com média resolução)…

- *layout*

  addcar_dialog.xml
  alertalarm_dialog.xml
  car.xml
  confabout.xml
  confalerts.xml
  confconf.xml
  confsecurity.xml
  confspecs.xml
  deletealert_dialog.xml
  home.xml
  infocar_dialog.xml
  language_dialog.xml
  layers_dialog.xml
  login_dialog.xml
  loginchange_dialog.xml
  ringtone_dialog.xml
  specs.xml
  tab1.xml

tab2.xml

track.xml

unitspeed_dialog.xml

unittemperature_dialog.xml

- *layout-land*

  home.xml

- *menu*

  menu1.xml

  menu2.xml

  menu3.xml

  menu4.xml

- *raw*

  alarm1.mp3

  alarm2.mp3

  alarm3.mp3

  alarm4.mp3

  setalarmon.mp3

- *values*

  strings.xml

- *values-pt*

  strings.xml

AndroidManifest.xml

4.3.Desenvolvimento

Será inicialmente explicado o primeiro *layout* visível da aplicação, ao qual seguir-se-ão os restantes e suas atividades, pela ordem que se encontram, em termos visuais, na aplicação.

Para melhor compreensão, os nomes dos ficheiros usados com a primeira letra maiúscula dizem respeito a código fonte (*.java*) e os nomes apenas com letras minúsculas dizem respeito a ficheiros de *layout* (*.xml*). Além de ser para melhor compreensão, também é imposto pelo ambiente de desenvolvimento que os nomes dos ficheiros *.xml* não possam ter letras maiúsculas. Outro aspeto para melhor compreensão é o fato de o *layout* usado numa atividade, tenha o mesmo nome que esta, apenas com letras minúsculas, como por exemplo o *layout home.xml* corresponde ao *layout* usado na atividade *Home.java*.

Inicialmente tem-se a primeira aba ***tab1.xml***. É um ficheiro especial que será usado para a aba principal. Este usa um objeto "<*TabHost*>" que contém dois filhos. Um com etiquetas (<*TabWidget*>) que serve para os vários campos da aba onde se pode colocar o nome e a imagem do *layout* correspondente e outro (<*FrameLayout*>) que exibe o conteúdo correspondente.

Para usar este *layout* especial, tem-se o ficheiro ***Tab1.java*** que tem a extensão *TabActivity*. Para fazer uso do *layout* usa-se o código *setContentView(R.layout.tab1)*. Esta aba contém cinco campos (*Home, Car, Track, Specs* e *Alerts*). Para cada campo é usado um *Intent* que chamará a atividade que se pretende e o respetivo código para se colocar o nome e a imagem do campo. A Figura 4.5 mostra a aba principal da aplicação.

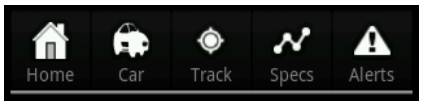

Figura 4.5 - L*ayout tab1.xml*.

De seguida tem-se o *layout* ***home.xml***, que diz respeito ao primeiro ecrã presente na aplicação. Este contém cinco botões e o respetivo nome e contém um campo, no canto superior direito, onde será mostrado o nome/matrícula da viatura selecionada. De fundo tem uma imagem que não é mais que uma fotografia tirada ao tablier de um automóvel.

Para obter os botões e os respetivos nomes na disposição em que se encontram, recorreu-se a <*RelativeLayout*> e <*LinearLayout*> que são objetos, que contêm filhos, e servem para enquadrar os seus filhos no ecrã. Os seus filhos, neste caso, são <*Button*> e <*TextView*> que são também objetos e como o nome indica, são para criar botões e textos. As imagens e os textos usados serão abordados mais à frente.

No caso particular da atividade *Home.java*, existem dois *layouts* distintos que podem ser usados. Ambos têm o mesmo nome mas encontram-se em subpastas diferentes. Um na subpasta *layout* e outro na subpasta *layout-land*. A subpasta *layout* serve para se colocarem os *layouts* que se pretendam usar com o dispositivo em modo *portrait*, ou seja na vertical, enquanto a subpasta *layout-land* serve para se colocar os *layouts* que se pretendam usar em modo *landscape*, ou seja na horizontal. Assim, o *SO Android* define qual o *layout* que será apresentado ao utilizador, recorrendo ao giroscópio do dispositivo para saber em que orientação este se encontra. A Figura 4.6, à esquerda mostra o *layout home.xml* na subpasta *layout* e à direita o *layout home.xml* na subpasta *layout-land*.

De notar que as figuras apresentadas são *printscreens* da aplicação em funcionamento, logo apresentam a barra de tarefas na parte superior e neste caso a *Tab1*, na parte inferior.

Figura 4.6 - L*ayout home.xml* em modo *portrait* à esquerda e *landscape* à direita.

O ficheiro ***Home.java*** tem extensão *Activity*. Esta é uma atividade normal e usa uma grande quantidade de recursos. Inicialmente usa uma *SharedPreferences* que é uma interface que permite guardar dados, ou seja, permite guardar um valor relacionado com um nome. Assim, tem-se acesso a esse valor em qualquer atividade da aplicação e permite guardar esses dados de forma permanente, mesmo que a aplicação seja terminada ou o dispositivo seja reiniciado. A *SharedPreferences* usada inicialmente tem o nome de "*Language Preferences*" e serve para guardar o idioma selecionado pelo utilizador. Assim, como esta atividade é sempre iniciada, quando se inicia a aplicação, impõe-se sempre que o idioma da aplicação seja o selecionado pelo utilizador.

Depois é usado o código relativo à função de cada botão. Este código faz o interface com as funções vibração e som quando se pressiona o botão. Contém ainda código para invocar a atividade de envio de mensagens de texto onde é enviado um identificador, para a atividade de envio de mensagens saber quem a invocou e assim desempenhar a sua função dependente de quem a invocou.

Tem também uma função *login()* que invoca o *layout login_dialog.xml* para se inserir um *login* e uma *password*, validar e proceder a sua ação. Este *layout* de *login* é usado quando se pressiona um botão e este tenha a preferência de pedir a *password* para desempenhar o seu papel. Para isso existe uma outra *SharedPreferences* de nome "*Security Preferences*" que tem campos para os vários botões de várias atividades, entre outros, para a aplicação saber se o utilizador pretende que seja confirmado a ação do botão com *password* ou sem *password*.

Depois tem uma função *onActivityResult()* que serve para avisar o utilizador, através de uma mensagem escrita no ecrã, do resultado da ação da atividade de envio de mensagens escritas. Assim, se for confirmado o envio da *SMS* para a viatura, o utilizador é informado dessa ação.

Tem ainda uma função *onCreateOptionsMenu()* para inserir um menu, na parte inferior do ecrã, se o utilizador usar o botão físico de menu. Como já explicado em 4.1 o menu é composto por três campos (*Config*, *CarChoose* e *Exit*). Cada campo, quando selecionado, invoca a atividade correspondente. No caso de *Exit*, a aplicação é terminada e é mostrada uma mensagem, ao utilizador, de que a viatura continua protegida.

É ainda usada a base de dados da aplicação e uma outra *SharedPreferences* de nome "*Choose Preferences*" que tem informação relativa à viatura selecionada. Esta informação permite colocar o nome/matrícula da viatura no ecrã, para o utilizador saber a que viatura corresponde as ações que tomar, ao pressionar os botões.

São finalmente usadas as funções *onResume()*, *onPause() e onDestroy()* onde é fechada ou aberta a base de dados, para permitir um correto funcionamento de toda a aplicação.

De seguida tem-se o *layout* ***car.xml*** apresentado na Figura 4.7. À semelhança do *layout home.xml* este usa <*LinearLayout*> e <*RelativeLayout*> onde são inseridos botões. Usa também um <*TextView*> para mostrar o nome/matrícula da viatura e um fundo, que representa uma viatura e que foi criado num programa de edição de imagens. Este *layout* contém oito botões que têm a função de desligar ou ligar a ignição da viatura selecionada, iniciar uma videochamada para visualizar o interior do veículo em tempo real, fechar ou abrir as janelas e ainda abrir a mala da viatura.

Figura 4.7 - L*ayout car.xml*.

Para usar o *layout car.xml* tem-se a atividade ***Car.java*** que tem como extensão *Activity*. Esta atividade é muito semelhante à atividade *Home.java*. Usa as mesmas *SharedPreferences* mas outros campos, o código da ação de cada botão é semelhante, à exceção que agora não tem som ao pressionar os botões, usa na mesma uma função de *login()* para o mesmo efeito, as funções *onActivityResult()*, *onCreateOptionsMenu()*, *onResume()*, *onPause()* e *onDestroy()* são também usadas para o mesmo efeito que em *Home.java*.

Nesta atividade é usado apenas um *layout*, quer o dispositivo se encontre em *portrait* quer em *landscape*. Uma vez que não existe um *layout* com o mesmo nome, na subpasta *layout-land*, o *SO Android* roda o *layout* existente e apresenta-o em *landscape*. Neste caso particular, não se pretende que o *layout* seja rodado ao rodar o dispositivo, porque se assim fosse, a imagem de fundo ficaria distorcida e os botões não coincidiriam com o local apropriado. Assim usa-se o código *setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT)* para garantir que o *layout* não seja rodado.

Depois tem-se o *layout* ***track.xml*** que corresponde à vista do *Google Maps*. Neste ficheiro, usa-se o objeto <*com.google.android.maps.MapView*> para se usar os mapas da *Google*. Dentro deste objeto usa-se o campo *android:apiKey="…my maps API Key…"* onde será colocado uma *API Key* pessoal. Para se obter uma *API Key*, pode ser consultada a página https://developers.google.com/maps/documentation/android/mapkey?hl=pt-PT onde explica o procedimento para obtenção de um *API Key* para os mapas da *Google*.

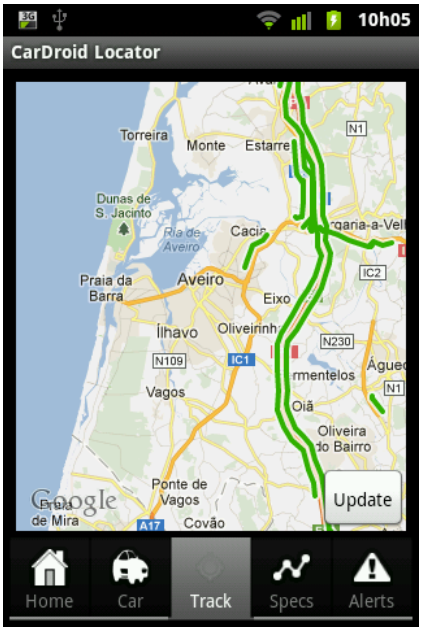

Figura 4.8 - *Layout track.xml*.

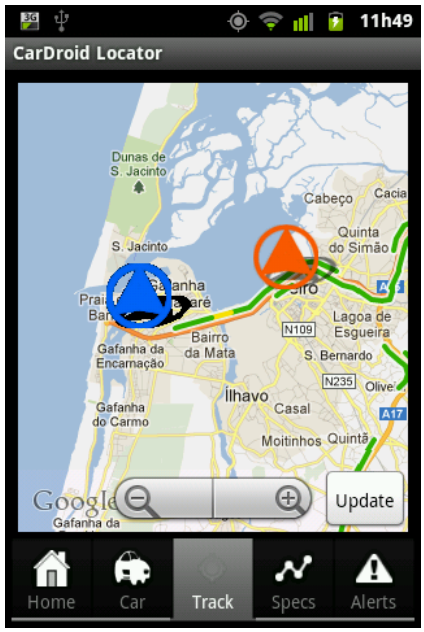

Figura 4.9 - *Layout track.xml* com *ItemOverlays*.

Além dos mapas, é usado um botão no canto inferior direito do ecrã, como pode ser visto na Figura 4.8.

Quanto ao ficheiro ***Track.java***, este tem a extensão *MapActivity*. Inicialmente é imposto ao dispositivo para se manter com o ecrã ligado. Para isso usa-se a classe *PowerManager.WakeLock* com a *flag* *SCREEN_DIM_WAKE_LOCK* que impõe ao dispositivo que fique com o ecrã parcialmente escurecido e com a iluminação do teclado desligada. É aqui escolhida esta opção e não ficar com o ecrã sem iluminação ou com a iluminação a 100%, uma vez que este *screen* pode ser usado durante algum tempo, pelo utilizador e assim reduzir o consumo de bateria do dispositivo. A opção de manter o ecrã parcialmente ligado, é desligada na função *onDestroy()*,

para garantir que nas restantes atividades da aplicação, a iluminação possa ser desligada pelo dispositivo.

Depois são impostos ao ecrã os botões de *zoom*, para que o utilizador possa fazer "*zoom in"* ou "*zoom out"* ao mapa. De seguida, são iniciados dois *ItemOverlays*. Um *ItemOverlay* é, neste caso, uma imagem que será sobreposta ao mapa. Para isso, foi criada uma outra atividade com o nome ***AddItemOverlay.java*** que usa a extensão *ItemizedOverlay<OverlayItem>*. Esta atividade é uma lista de *OverlayItems* e usa as funções *AddItemOverlay()* que serve para ajustar os limites da imagem, *addItem()* que serve para adicionar um objeto especifico à lista, *createItem()* que retorna o elemento pretendido, *size()* que retorna o número de elementos da lista e *clearOverlay()* que remove todos os elementos da lista.

De seguida usa-se a classe *LocationManager* que fornece acesso aos serviços de localização do sistema. No presente caso, pretende-se usar o serviço de *GPS*. Para isso impõe-se uma condição, para verificar se o serviço de *GPS* já se encontra ligado. Se não se encontrar ligado, a aplicação mostra uma mensagem de aviso e redireciona o utilizador para a janela de definições do dispositivo, que contém as definições de localização e assim, o utilizador pode ligar o *GPS* de uma forma mais cómoda e simples.

É depois criada uma função para controlar o botão "*Update*". Esta função usa a vibração e recorre, novamente, à *SharedPreferences* "*Security Preferences*" para usar o *layout login_dialog*, se for caso disso. Além disso, recorre também à atividade *SendSMS.java* onde envia uma mensagem escrita, para a viatura selecionada, como já foi descrito anteriormente.

Depois é imposta uma condição onde verifica se já existe um *OverlayItem* relativo a uma viatura, recorrendo a uma nova *SharedPreferences* com o nome "*Track Preferences*". Se já existir, é colocado o *OverlayItem*, por cima do mapa, recorrendo à função *createCarMarker()*, que será descrita à frente.

De seguida é feita uma nova condição, desta vez para verificar se a atividade (*Track.java*) foi iniciada pelo utilizador, através da interação com a aplicação ou iniciada através do serviço. É imposta esta condição porque no momento que o utilizador usa o botão "*Update*", este envia uma mensagem escrita para a viatura a pedir as coordenadas da mesma e esta retorna uma nova mensagem escrita para o dispositivo, com as coordenadas da viatura. Esta mensagem escrita é recebida pela aplicação através de um serviço, que será descrito posteriormente, e o serviço inicia novamente a atividade *Track.java*. Os dados recebidos são guardados na *SharedPreferences* "*Track Preferences*" e é criado um novo *ItemOverlay*, com a posição atual da viatura.

A atividade, usa ainda as funções *GeoUpdateHandler()*, *createMyMarker()*, *createCarMarker()*, *zoomCalc()*, *onCreateOptionsMenu()*, *onOptionsItemSelected()*, *isRouteDisplayed()* e *isLocationDisplayed()*. Estas duas últimas fazem parte da classe *MapActivity* e servem, como o nome indica, para mostrar as ruas e para informar o sistema que serão usados dados de localização.

A função *GeoUpdateHandler()* é usada sempre que a posição do dispositivo for alterada e recorre ao *LocationListener*. Assim, é constantemente atualizada a posição do dispositivo e

marcada no ecrã, através da função *createMyMarker()*. A função *createMyMarker()* e *createCarMarker()* servem para colocar os *ItemOverlays* no mapa, na localização pretendida. De salientar que o *ItemOverlay* referente à viatura recorre a uma imagem vermelha e o *ItemOverlay* referente à posição do dispositivo recorre a uma imagem azul, como pode ser visto na Figura 4.9, criadas para o efeito e serão explicadas posteriormente. Além disto, estas duas funções recorrem a uma outra função *zoomCalc()*. Esta função tem o objetivo de mostrar a parte do mapa que realmente interessa ao utilizador. Isto é, são feitas condições para verificar se existe o *ItemOverlay* referente à viatura, o *ItemOverlay* referente à posição do dispositivo ou se existem ambos. No primeiro caso, centra o mapa no local onde se encontra o veículo e coloca um *zoom* para se poder ver algumas ruas à sua volta. No segundo caso, faz o mesmo que no primeiro centrando o mapa na posição atual do dispositivo. O *zoom* aqui usado tem o valor 19 e foi escolhido de forma a se poderem ver algumas ruas à volta do *ItemOverlay*. No último caso, mostra o mapa onde estão inseridas as posições dos dois *ItemOvrerlays*, com uma pequena margem a mais, para ambos os *ItemOverlays* ficarem visíveis pelo utilizador. Como é possível que o utilizador esteja em movimento e assim a posição do dispositivo esteja sempre a variar, é sempre recalculada a sua posição e é feito o *zoom* correspondente, para mostrar sempre as duas posições, estando o utilizador a aproximar-se da viatura ou a afastar-se da mesma.

Por fim tem-se as funções *onCreateOptionsMenu()* que serve para criar o *menu4*, e a função *onOptionsItemSelected()* para introduzir código relativamente à ação a realizar por cada um dos seis itens do *menu4*. Dos seis itens já referidos em 4.1, de salientar os *itens SavePosition* e *DeleteMarker*. O primeiro serve para guardar a posição atual do dispositivo, como sendo a posição da viatura. Isto é interessante para por exemplo, quando o utilizador estaciona a sua viatura num parque de estacionamento, marca a posição no momento que a estaciona e no local e depois, quando estiver longe desta, poder ligar a aplicação no *layout track* e assim saber onde se encontra a sua viatura, relativamente à posição onde ele mesmo se encontra e poder assim seguir de encontro à sua viatura. Além desta situação pode ser usado também para uma outra situação qualquer que pretenda guardar o local onde se encontra no momento e poder lá voltar posteriormente, sem ter que recorrer ao *hardware* necessário, colocado na viatura.

O item *DeleteMarker* serve para eliminar o *ItemOverlay* relativo à viatura e poder usar este *layout* como uma qualquer outra aplicação de navegação, onde se encontra o mapa sempre centrado na posição atual do dispositivo.

Segue-se o *layout* ***specs.xml*** que mostra a telemetria, da viatura selecionada e pode ser visto na Figura 4.10.

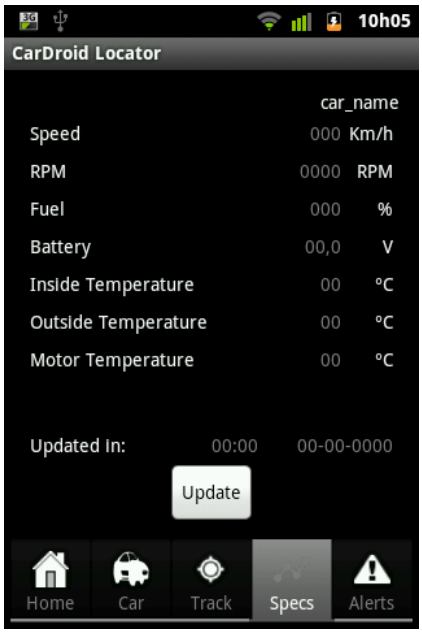

Figura 4.10 - *Layout specs.xml*.

Como nos *layouts home.xml* e *car.xml*, este também usa um <*TextView*> para o nome/matrícula da viatura, no canto superior direito. Depois usa o objeto <*ScrollView*> que permite deslocar os vários itens presentes "dentro" do objeto *ScrollView*. Estes itens são considerados filhos do objeto *ScrollView* e usam os objetos <*Relative Layout*> e <*TextView*>. O primeiro para enquadrar todo o item no ecrã e o segundo para as várias frases ou palavras a usar. Como pode ser visto na Figura 4.10, cada item tem três *TextView*. Um para o nome, outro para o valor e outro para a unidade. Existem ao todo sete itens. Depois tem um novo *RelativeLayout* com mais três *TextView* para colocar a hora e data da atualização dos outros itens. Por fim tem um botão, recorrendo ao objeto <*Button*>.

Para usar o *layout specs.xml* tem-se a atividade ***Specs.java*** que tem como extensão *Activity*. Esta atividade tem como objetivo mostrar a telemetria da viatura selecionada, no momento que o utilizador use o botão "*Update*". Para isso, o código relativo ao botão é semelhante ao usado pelos outros botões, das outras atividades e começa com a condição à *SharedPreferences* "*Security Preferences*" para verificar se é necessário a introdução de *password* para desempenhar a função do botão. Se sim mostra o *Dialog login_dialog* para introduzir a *password*. Se a *password* for bem sucedida, usa a classe *SendSMS* com um *Intent*, para enviar uma mensagem escrita, para a viatura selecionada, a pedir a telemetria atual.

Depois do código referente ao botão, é feita uma condição para ver se a atividade foi iniciada pelo utilizador ou através do serviço, como foi explicado na atividade anterior. Se foi iniciado pelo serviço, atualiza os campos com os valores recebidos.

A atividade usa ainda as funções *onCreateOptionsMenu()* para uso do *menu1*, *onResume()*, *onPause()* e *onDestroy()*. As duas últimas para fechar a base de dados, como nos casos das atividades anteriores. A função *onResume()* tem como objetivo colocar o nome/matrícula da viatura, através da base de dados, como já foi referido anteriormente e tem ainda como objetivo

colocar apenas os itens que o utilizador pretenda, no ecrã. A escolha dos itens é feita na atividade *ConfSpecs.java* e será explicado mais à frente. Para a colocação ou não colocação dos itens, recorre-se a uma nova *SharedPreferences* com o nome "*Specs Preferences*" onde tem campos booleanos e indicam quais os itens a mostrar. Para isso usa-se o código *rlSpeed.setVisibility(View.VISIBLE)* e *rlSpeed.setVisibility(View.GONE)* para mostrar e não mostrar, respetivamente. *rlSpeed* diz respeito ao *RelativeLayout* do item "*Speed"*.

Para terminar a primeira aba, tem-se o *screen* dos alertas. Aqui não é usado um ficheiro para *layout* uma vez que a atividade ***Alerts.java*** tem como extensão *ListActivity*. Pretende-se aqui mostrar apenas uma lista de itens, como mostra a Figura 4.11 e com a classe *ListActivity* não é necessário um ficheiro de *layout*.

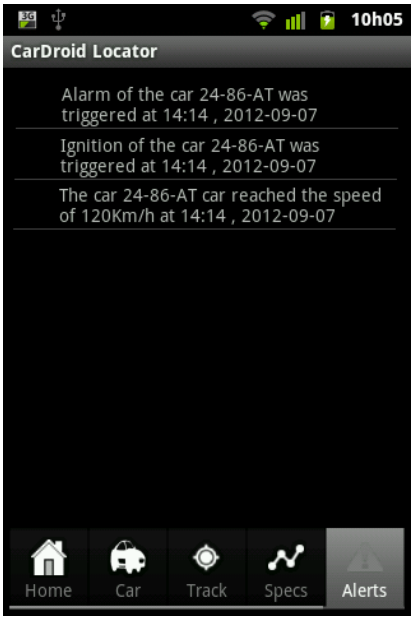

Figura 4.11 - *Screen* de alertas com alguns alertas na lista.

Uma vez que o *screen* dos alertas serve para mostrar os alertas, esta atividade vai ser usada pelo serviço, ao receber uma mensagem escrita, quando despertar algum alerta na viatura. Para isso, no início da atividade, faz-se uma condição para ver se a atividade foi iniciada pelo serviço. Se sim, faz novas condições para saber que tipo de alerta foi despoletado. Dentro destas condições é criada uma frase, que vai ser apresentada na lista, com os dados recebidos na mensagem escrita. Além disso, é chamada uma função *alarme()* em que é passada essa frase. Se a atividade não for chamada pelo serviço, usa apenas a função *fillData()*. De notar que a função *fillData()* é usada depois da condição de verificar se foi o serviço que iniciou a atividade e é usada também, mesmo que tenha sido o serviço a iniciar a atividade.

A função *fillData()* tem como objetivo, preencher a lista com os alertas existentes na base de dados. A base de dados é criada quando existir o primeiro alerta e é atualizada sempre que houver um novo alerta. Isto é feito dentro da função *alarme()*. Esta função tem como objetivo receber a frase correspondente ao alerta, inserir esse alerta na base de dados e despertar um alarme. O alarme é conseguido através da classe *AlarmManager*. Esta classe tem como função

acordar o dispositivo, se este se encontrar em modo *sleep* e iniciar uma atividade, passada com um *Intent*. Aqui foi criada uma nova atividade para realizar as funções pretendidas, para avisar o utilizador do alerta. A atividade criada tem o nome de ***AlertReceiver.java*** e tem como extensão a classe *BroadcastReceiver*. Esta atividade começa com o uso da classe *NotificationManager* que tem como função notificar o utilizador de um evento que tenha ocorrido, seja através de um som, de luzes ou de outro tipo de aviso. Logo depois, é usada a classe *Notification* que representa o modo de persistência que vai ser usado, para avisar o utilizador. Aqui vai ser usada uma notificação, recorrendo também à barra de notificações. Os modos de aviso usados são a vibração, através do campo *notification.defaults* com a *flag* *DEFAULT_VIBRATE*, luzes, através do *LED* presente na parte frontal dos dispositivos e som através de um toque específico, escolhido pelo utilizador na atividade *ConfConf.java*. De notar que foi usada a *flag* *FLAG_INSISTENT* para que os avisos se mantenham até que o utilizador use a barra de notificações para poder terminar os avisos.

Assim, pode-se resumir que quando ocorrer um alerta na viatura, o módulo instalado na viatura envie uma mensagem escrita para o dispositivo. A aplicação, através do serviço, recebe a mensagem escrita e interpreta o seu conteúdo. A aplicação é iniciada no *screen* dos alertas, é apresentado o tipo de alerta ocorrido e o utilizador é avisado, através de vibração, de um som e de uma notificação, que ocorreu um alerta na viatura.

Além de tudo isto, é ainda feita uma condição à *SharedPreferences* "*Alert Alarm Preferences*" para saber que tipo de ação deve ser tomado, se eventualmente o alerta for do alarme. Estas ações podem ser, efetuar uma chamada de retorno para a viatura ou uma vídeo chamada, para que o utilizador possa ter som ou som e imagem do interior da viatura para poder saber o que despertou o alarme. Esta ação é escolhida pelo utilizador no *sreen Settings* da atividade *ConfConf.java*.

Retomando à atividade *Alerts.java*, existem ainda as funções *onListItemClick()*, *onCreateOptionsMenu()*, *onResume()*, *onPause()* e *onDestroy()*. A função *onCreateOptionsMenu()* para usar o *menu1*, as funções *onResume()*, *onPause()* e *onDestroy()* para abrirem e fecharem a base de dados, respetivamente, como nas atividades anteriores e a função *onListItemClick()* para a ação de se clicar num item da lista. A ação aqui pretendida é apenas para dar ao utilizador, a opção de eliminar o item da lista. Para isso recorre-se ao *layout* ***deletealert_dialog.xml***, usado como *Dialog*, como pode ser visto na Figura 4.12. O trabalho aqui realizado pela função é o de apagar o elemento da base de dados e novamente chamar a função *fillData()* para preencher novamente a lista, atualizada.

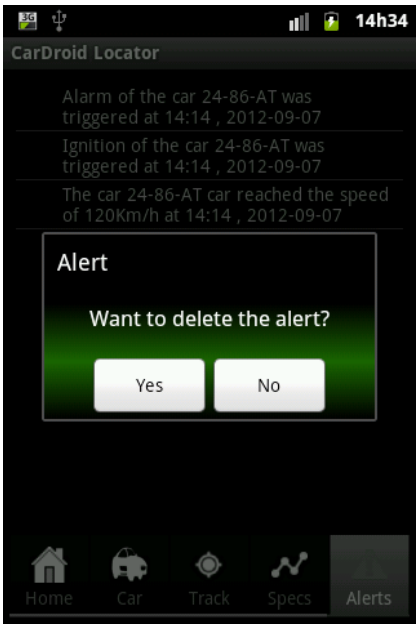

Figura 4.12 - *Layout deletealert_dialog.xml*.

Passando agora para a segunda aba, das configurações, esta é semelhante à primeira aba. O nome do *layout* é ***tab2.xml*** e a atividade que o usa tem o nome ***Tab2.java***. A diferença para a primeira é que esta é apresentada na parte superior do ecrã e os campos são outros. Aqui os campos são *Settings*, *Specs*, *Alerts*, *Security* e *About*. Cada um corresponde a uma atividade e serão descritas de seguida.

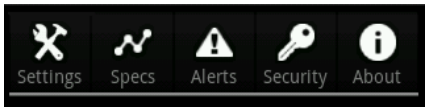

Figura 4.13 - *Layout tab2.xml*.

Inicialmente tem-se o *layout* ***confconf.xml*** para os *Settings* da aplicação, como mostra a Figura 4.14.

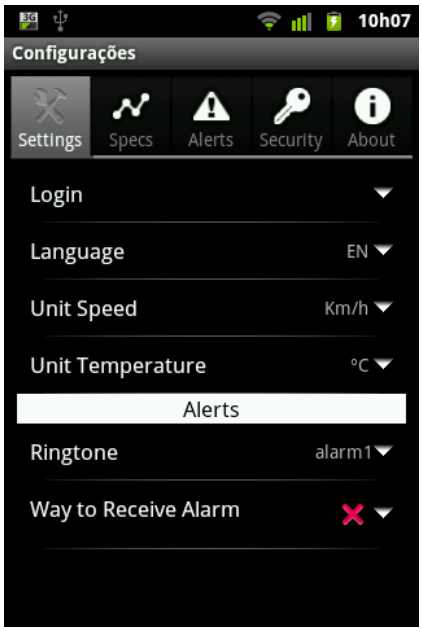

Figura 4.14 - *Layout confconf.xml*.

Aqui é usado o objeto <*ScrollView*> para todo o *layout*. Dentro deste, são usados <*RelativeLayout*> *com* <*TextView*> e <*ImageView*>, este último para usar imagens. Os *TextView* são usados para colocar o nome ou frase apresentados, assim como as unidades. As *ImageView* são usadas para colocar uma seta em cada *RelativeLayout*. É também usada para o símbolo do item "*Way to Receive Alarm*". Além disso é usada também para colocar as linhas, para servir de separador entre os itens. Um exemplo é o código que se segue, para colocar as linhas: *android:src="@android:drawable/divider_horizontal_dim_dark"*.

Para fazer uso do *layout confconf.xml* criou-se a atividade ***ConfConf.java*** que tem como extensão *Activity*. Esta atividade cria seis *SharedPreferences* com os nomes "*Login Preferences*", "*Language Preferences*", "*Unit Speed Preferences*", "*Unit Temperature Preferences*", "*Ringtone Preferences*" e "*Alert Alarm Preferences*". Algumas delas já foram referenciadas anteriormente, nas várias atividades que as usam.

Inicialmente, a atividade recorre a uma condição à *SharedPreferences* "*Security Preferences*" para verificar se o utilizador pretende inserir a *password* ao iniciar esta mesma atividade. Esta escolha é feita na atividade *ConfSecurity.java* e será discutida à frente. Se a condição for para inserir a *password*, será iniciada a atividade *Login.java*, através de um *Intent*. É aqui usada a solução de recorrer a uma nova atividade para apresentar o *layout login_dialog.xml*, em vez de usar um *Dialog* como nas atividades anteriores, para um correto funcionamento da aplicação, uma vez que aqui se trata de uma atividade e não de um campo da atividade.

A atividade ***Login.java*** usa o *layout* ***login_dialog.xml*** como um *Dialog*, como pode ser visto na Figura 4.15. O seu código prende-se apenas em verificar se o texto introduzido nos campos "*Login*" e "*Passwrod*" foram corretamente preenchidos, comparando com o texto guardado na *SharedPreferences* "*Login Preferences*" e também colocar uma *flag* booleana (*flag* guardada na *SharedPreferences* "*Login Preferences*") a verdadeiro, se for bem-sucedido. A atividade termina e

retorna à atividade *ConfConf.java*. Aqui, como esta atividade não tinha sido terminada, por se ter iniciado uma outra atividade através de *Intent* e não se ter invocado a função *onDestroy()*, a atividade é iniciada, invocando a função *onRestart()*. Nesta função é inserido código com a condição de ver se a *flag* é verdadeira ou falsa. Se for verdadeira, mostra o *layout* da atividade *ConfConf.java*, se for falsa termina a atividade, retornando para o layout inicial da aplicação.

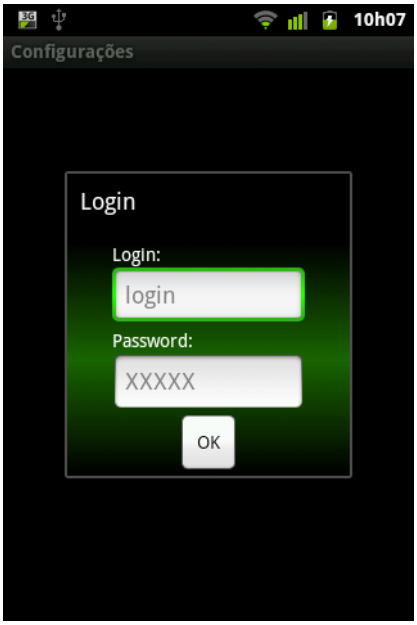

Figura 4.15 -*Layout login_dialog.xml*.

Continuando, na parte inicial da atividade *ConfConf.java*, são inseridos valores, nos campos das *SharedPreferences*, aqui criadas. Todos estes trechos de código são iniciados com uma condição para verificar se já existe algum valor nos campos. Este código será apenas executado na primeira utilização da atividade. Primeiro cria-se um *login* e uma *password* que têm o valor "*12345*" para ambos os campos. Fica assim criada o *login* e *password* por defeito. Depois, quanto à *SharedPreferences* do idioma, coloca-se apenas os valores do idioma como pode ser visto o valor "*EN*" na linha de *Language* na Figura 4.14. Não é aqui escolhido o idioma, por ser feito na atividade *Home.java*, na primeira utilização da aplicação. Seguidamente é colocado o valor "*Km/h*" no campo da *SharedPreferences* das unidades de velocidade. É feito o mesmo para a *SharedPreferences* das unidades de temperatura, agora com o valor "^{0}C". Depois é escolhido o "*alarme1*" como toque predefinido para os alertas e é colocado o seu valor na linha correspondente, como pode ser visto na Figura 4.14. Finalmente é escolhido o valor "*none*" para a maneira como é recebido o alerta de alarme e é colocada a imagem, correspondente ao valor, na linha "*Way to receive alarm*".

Estando todos os valores iniciados e com um valor por defeito, é usado o código referente à escolha de cada linha de opções. Como cada linha é relativa a um *RelativeLayout*, foi escolhida aqui a opção de usar o *RelativeLayout* como "botão", isto é, foi usada a função *setOnClickListener()* para cada *RelativeLayout*, código que geralmente se usa para os botões e

tem como função ser invocado sempre que se clique no objeto a si associado, neste caso o *RelativeLayout*.

Inicialmente tem-se o código relativo à primeira linha, *Login*. Aqui será apresentado o *layout* ***loginchange_dialog.xml***, como *Dialog*, para o utilizador alterar o *login* e *password*. O *layout* é apresentado na Figura 4.16. Este *layout* é constituído por quatro <*TextView*>, quatro <*EditText*> e um <*Button*>. Os campos *EditText* servem para inserir texto, que será usado na parte de código. Aqui serão comparados os valores inseridos nos campos "*OldLogin*" e "*OldPassword*" com os valores presentes na *SharedPreferences* "*Login Preferences*". Se coincidirem, estes valores são substituídos pelos valores presentes nos campos "*NewLogin*" e "*NewPassword*".

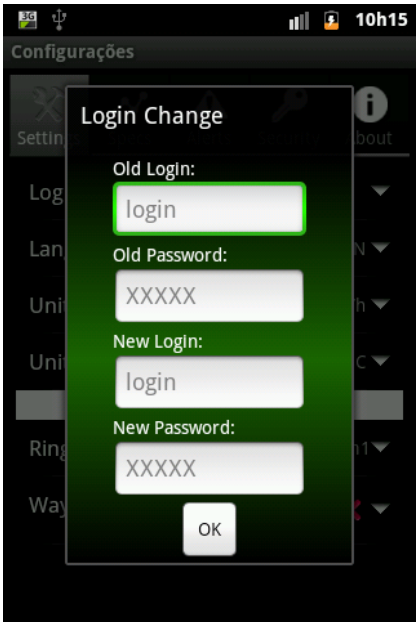

Figura 4.16 - *Layout loginchange_dialog.xml*.

Depois segue-se o código relativo ao idioma. De igual forma ao campo anterior, criou-se aqui o *layout* ***language_dialog.xml***, para ser escolhido o idioma que o utilizador pretenda. Os idiomas possíveis são apenas Inglês e Português, uma vez que a aplicação se encontra em fase de protótipo e não se ganharia muito, nesta fase, fazer uso extensivo de idiomas diferentes.

O *layout* é apresentado na Figura 4.17 e recorre a dois <*LinearLayout*> onde cada um contém um <*TextView*> e um <*RadioButton*>. O *RadioButton* é usado quando se pretende um campo que corresponda a dois estados diferentes, assinalado ou não assinalado.

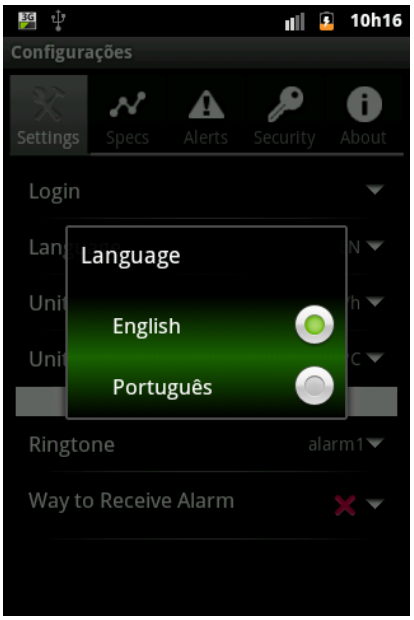

Figura 4.17 - *Layout language_dialog.xml*.

O código utilizado recorre, uma vez mais, à função *setOnClickListener()* para cada campo, onde será atualizado o campo "*LANGUAGE*" da *SharedPreferences* "*Language Preferences*". Além disso, será utilizada a classe *Resources*, para acessar aos recursos da aplicação, a classe *DisplayMetrics*, para acessar à informação relativa ao ecrã do dispositivo e a classe *Configurations* para obter informação relativa às configurações do dispositivo, neste caso o campo "*Locale*" que representa o idioma ou o País usado pelo dispositivo. Como a versão do *SO Android* utilizada (V2.3.3) não contém o idioma Português, teve que ser criado o idioma.

De seguida tem-se o código referente aos campos das unidades de velocidade e de temperatura. É conseguido da mesma maneira que os anteriores, recorrendo aos *layouts* ***unitspeed_dialog.xml*** e ***unittemperature_dialog.xml***, respetivamente. Nestes dois campos usou-se apenas uma opção, em cada um, uma vez mais por se tratar de um protótipo e não ser, para já, necessário alongar muito as opções.

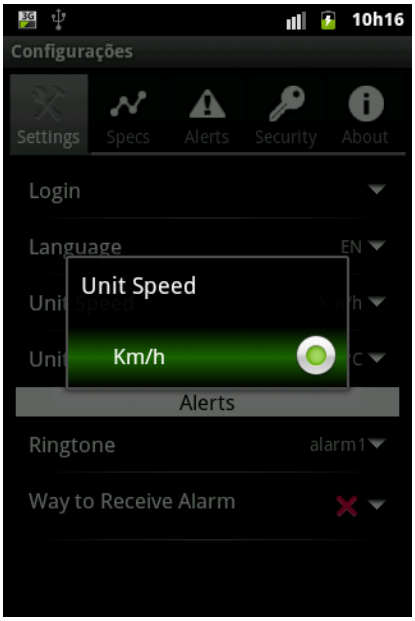

Figura 4.19 - *Layout unitspeed_dialog.xml*.

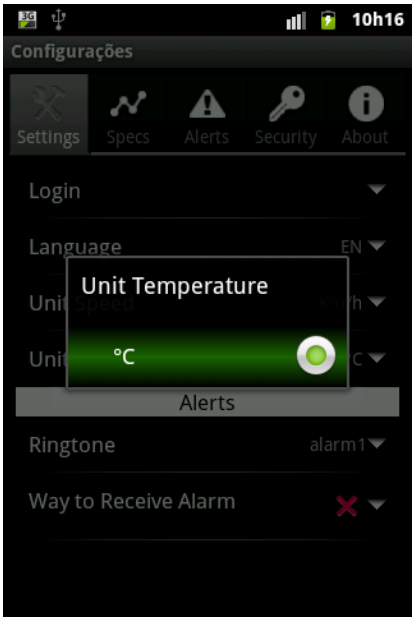

Figura 4.18 - *Layout unittemperature.xml*.

Depois tem-se o código relativo ao toque usado para os alertas. O *layout* desenvolvido tem o nome ***ringtone_dialog.xml*** e apresenta-se na Figura 4.20.

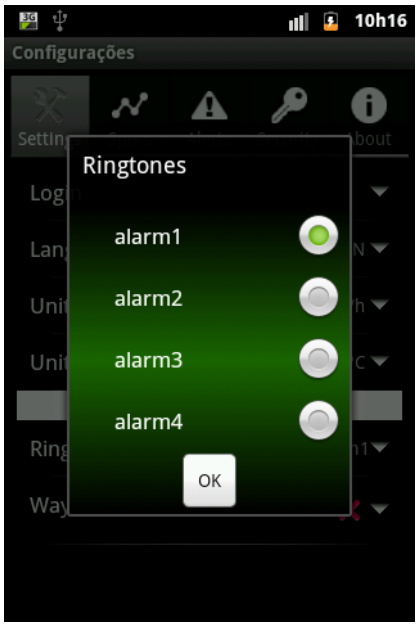

Figura 4.20 - *Layout ringtone_dialog.xml*.

O *layout* apresenta quatro *LinearLayout*, onde cada um contém um *TextView* e um *RadioButton*. Contém ainda um *Button*, que será utilizado para escolher o toque. De notar que no presente *layout* usou-se um botão para confirmar, ao contrário dos anteriores, por aqui ser necessário clicar nos vários campos para poder ouvir o seu toque e não selecionar o toque ao clicar no seu campo.

Quanto ao código, recorre-se à função *setOnclickListener()* para utilizar o campo do toque e depois recorre-se novamente à mesma função para executar o código relativo a clicar em cada

toque disponível. A função de clicar num toque tem como objetivo reproduzir o som respetivo. Para isso usa-se a classe *MediaPlayer* para reproduzir os sons, com o código que se segue:

```
mPlayer = MediaPlayer.create(ConfConf.this, Uri.parse("android.resource:
        //com.ControlDroid.CarDroidLocator/" +R.raw.alarm1));
mPlayer.start();
```

Depois, dentro da função relativa ao botão "*OK*", tem-se o código para guardar na *SharedPreferences* "*Ringtone Preferences*" o valor do som escolhido.

Segue-se o campo "*Way to Receive* Alarm". O *layout* criado para o efeito tem o nome ***alertalarm_dialog.xml*** e é apresentado na Figura 4.21. Recorre a três *LinearLayout* com um *TextView* e um *RadioButton*, cada.

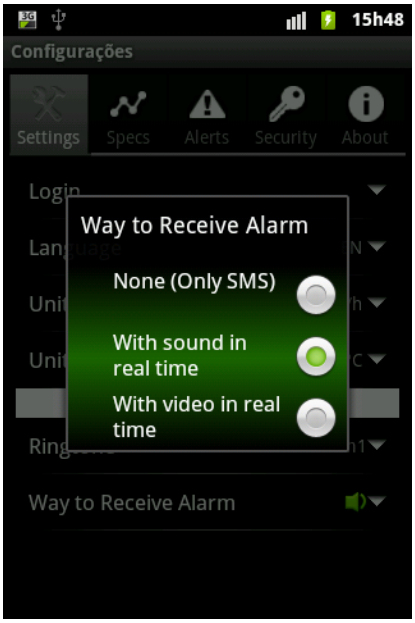

Figura 4.21 - *Layout alertalarm_dialog.xml*.

Quanto ao código, tem o objetivo de atualizar a *SharedPreferences* "*Alert Alarm Preferences*" e de colocar a imagem respetiva à escolha, feita pelo utilizador.

Por fim, a atividade *ConfConf.java* tem ainda as funções *onCreateOptionsMenu()* e *onOptionsItemSelected()* para fazerem uso do *menu2*, como descrito nas atividades anteriores.

Segue-se a atividade ***ConfSpecs.java***. Esta recorre ao *layout* ***confspecs.xml*** e é apresentado na Figura 4.22. O *layout* usa os objetos <*ScrollView*> para todo o seu conteúdo, onde contém *RelativeLayout* com o *campo* <*CheckBox*>. O objeto *CheckBox* é semelhante ao *RadioButton*, com um aspeto gráfico diferente. São usados sete campos com uma <*ImageView*> a servir de separador. A imagem usada é uma linha de divisão, já usada no *layout confconf.xml*.

Relativamente ao código, este usa a extensão *Activity* e cria a *SharedPreferences* "*Specs Preferences*" com sete campos booleanos. Estes campos dizem respeito a cada opção e servem para guardar o seu estado, consoante a escolha do utilizador. É usado, também nesta atividade, o código relativo a pedir *password* ao iniciar a atividade. O processo é igual ao usado na atividade

ConfConf.java, chamando a atividade *Login.java* e verificando a *flag* da *SharedPreferences* "*Login* Preferences" em *onRestart()*.

Posteriormente é usado um trecho de código, para cada opção. O código recorre à função *setOnClickListener()* para verificar qual o estado do *CheckBox* e atualiza o campo respetivo da *SharedPreferences* "*Specs Preferences*".

A atividade usa também o código para apresentar o *menu2*, como descrito em atividades anteriores.

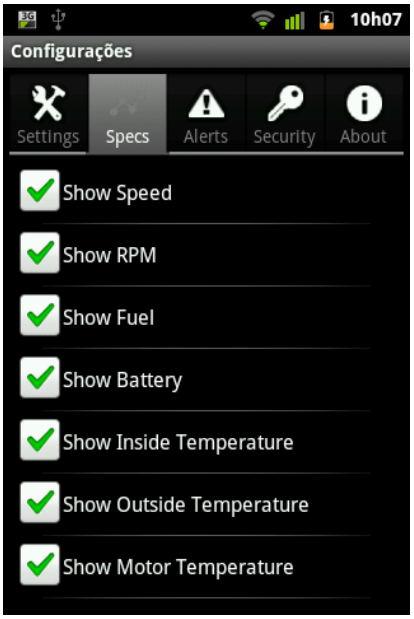

Figura 4.22 - *Layout confspecs.xml*.

Continuando na aba das opções, tem-se a atividade *ConfAlerts.java*, recorrendo ao *layout* ***confalerts.xml***.

Como pode ser visto na Figura 4.23, contém três opções, com um *TextView* para o texto e um *ToggleButton* para o estado da opção. O *ToggleButton* é semelhante ao *CheckBox* e ao *RadioButton*, apresentando um aspeto gráfico diferente. Foi escolhido este objeto por se tratar de opções que estarão ligadas ou desligadas, consoante a escolha do utilizador, e não *CheckBox,* por exemplo, por este objeto dizer respeito mais a outro tipo de opções. Também aqui, foi usada uma linha separadora entre cada opção para dar mais qualidade visual à aplicação.

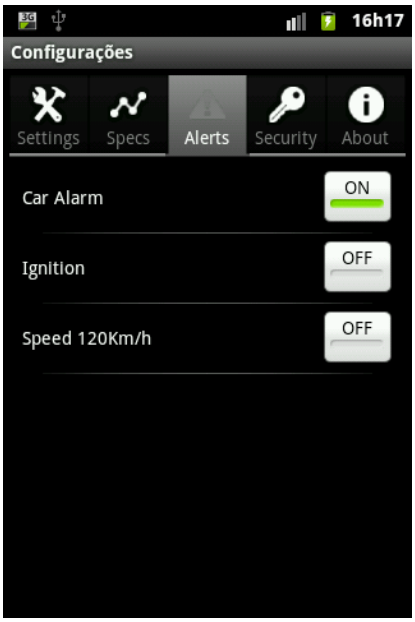

Figura 4.23 - *Layout confalerts.xml*.

Quanto ao código referente à atividade ***ConfAlerts.java***, é muito semelhante ao usado na atividade anterior. Usa a atividade *Login.java* para pedir o *login* e *password* ao iniciar a atividade. Usa uma *SharedPreferences* com o nome "*Alert Preferences*" para guardar os valores das opções. O código relativo a cada opção tem a mesma função, tratando-se aqui de *ToggleButton* e não *CheckBox*. E usa também as funções *onCreateOptionsMenu()* e *onOptionsItemSelected()* para apresentar o *menu2*.

Depois tem-se a atividade ***ConfSecurity.java*** que usa o *layout* ***confsecurity.xml***.

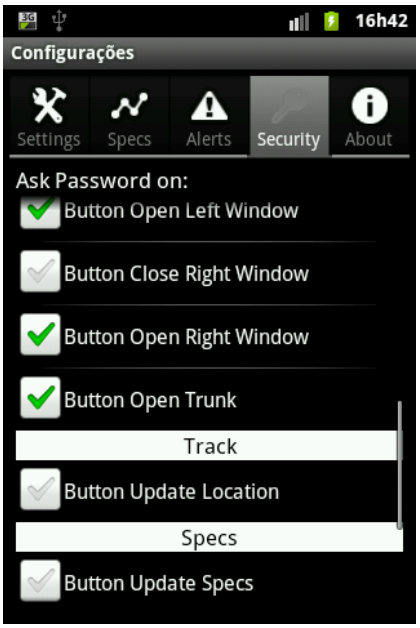

Figura 4.24 - *Layout confsecurity.xml*.

A Figura 4.24 não permite visualizar todas as opções do *layout*, por estas serem muitas. Pode ser visto, do lado direito no botão de *scroll*, que as opções mostradas são uma pequena parte das

existentes.

Este *layout* recorre inicialmente a um *RelativeLayout* com um *TextView* com a frase "*Ask Password on:*". Depois usa o objeto <*ScrollView*> para poder deslocar as várias opções para cima ou para baixo, sem que a frase inicial seja omitida do ecrã. Cada opção é composta por um *CheckBox* com a frase correspondente. Fica assim visto que este *layout* é relativo às ações que requerem *password*. São elas, os cinco botões presentes no *layout home.xml*, oito botões presentes no *layout car.xml*, o botão "*Update*" do *layout track.xml*, o botão "*Update*" do *layout specs.xml* e as três atividades (Settings, *Specs* e *Alerts*) da aba das configurações.

Passando agora para a descrição do código da atividade, esta usa a extensão *Activity* e é iniciada com a invocação da atividade *Login.java*, para pedir *login* e *password* ao iniciar a atividade. Aqui, à semelhança das atividades anteriores, inicia a atividade *Login.java* para apresentar o *layout login_dialog.xml*, para o utilizador inserir o *login* e a *password*. É depois feita a condição à *flag* da *SharedPreferences* "*Login Preferences*", em *onRestart()*, para verificar se a *password* foi bem sucedida. Mas ao contrário das atividades anteriores, aqui é sempre pedido o *login* e *password* ao iniciar a atividade e não segundo a preferência do utilizador. Para esta atividade não existe a opção de não pedir *login* e *password*. Foi imposta esta obrigação por se tratar da atividade que permite utilizar todos os botões livremente, sem pedir *password*, e poder existir alguma falta de segurança, no caso de alguém ter acesso indevido ao dispositivo que contém a aplicação. Ou seja, foi imposta esta necessidade, de pedir *password* nesta atividade, para aumentar o nível de segurança da aplicação.

Posteriormente é criada uma nova *SharedPreferences* com o nome "*Security Preferences*" onde serão guardados, em campos booleanos, as preferências do utilizador relativamente aos campos onde é necessário inserir *password* para ter acesso à sua ação.

Depois, é usado um trecho de código semelhante, para cada opção, onde verifica o estado do *CheckBox* e atualiza o campo respetivo da *SharedPreferences*.

Por fim, são usadas as funções *onCreateOptionsMenu()* e *onOptionsItemSelected()* para apresentar o *menu2*.

Finalmente, tem-se a atividade ***ConfAbout.java***, recorrendo ao *layout* ***confabout.xml***.

Figura 4.25 - *Layout confabout.xml*.

O *layout* apresentado na Figura 4.25 recorre aos objetos *TextView* e *ImageView*, para apresentar as frases e as imagens.

Quanto ao código, este tem como extensão *Activity*, apresenta apenas o *layout* (*setContentView(R.layout.confabout)*) e usa as funções respetivas para apresentar o *menu2*.

Ficam assim descritas, todas as atividades presentes nas duas abas da aplicação.

De seguida, serão descritos os *layouts* dos quatro menus existentes.

Inicialmente tem-se o *menu1*, que pode ser acedido em *Home*, *Car*, *Specs* e *Alerts* e o seu *layout* é apresentado na Figura 4.26, com o nome ***menu1.xml***.

Figura 4.26 - *Layout menu1.xml*.

Este *layout* usa o objeto <*menu*> que contém três objetos <*item*>. O objeto *menu* para indicar que se trata de um *menu* e os objetos item para criar os itens. Cada item é composto por três campos (*id*, *title* e *icon*). O campo *id* serve para dar um nome de identificação ao item, para posteriormente, quando for usado em ficheiros de código, seja identificado. O campo *title* para dar um nome ao item e *icon* para usar uma imagem para o item. As imagens utilizadas serão abordadas posteriormente.

O item *Config* serve para reencaminhar o utilizador para a aba das configurações, o item *Car(switch)* serve para reencaminhar o utilizador para a lista de veículos, que será abordada à frente e o item *Exit* serve para sair da aplicação. De notar que ao usar este último item, a aplicação fica apta a receber os alertas que possam surgir.

Segue-se o *menu2*, com o *layout* ***menu2.xml***. Este *menu* é acedido em todas as *screens* da aba das configurações. O *menu* contém apenas um item, que redireciona a aplicação para o ecrã inicial.

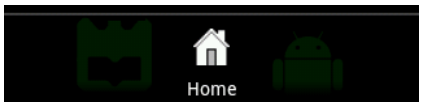

Figura 4.27 - *Layout menu2.xml*.

Depois tem-se o *menu3*, usando o *layout* ***menu3.xml***.

Figura 4.28 - *Layout menu3.xml*.

Este *menu* está disponível na atividade *CarChoose.java*, isto é, na lista de viaturas. Contém dois itens, um para redirecionar o utilizador para o ecrã inicial e outro para o *layout addcar_dialog.xml*, que permite criar e adicionar à base de dados, uma nova viatura. Na Figura 4.29 é apresentado o *layout* ***addcar_dialog.xml***.

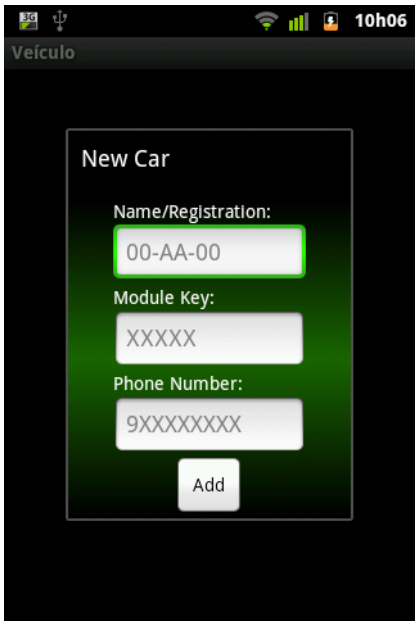

Figura 4.29 - *Layout addcar_dialog.xml*.

Este *layout* permite inserir uma nova viatura com o nome ou matrícula, o código do módulo instalado na viatura e o número do cartão *SIM* presente no módulo. Para isso recorre aos objetos *TextView*, *EditText* e a um *Button* "*Add*", para confirmação.

Por fim, tem-se o *menu4*, com o *layout* ***menu4.xml***.

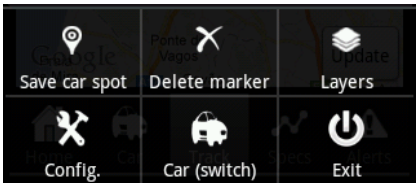

Figura 4.30 - *Layout menu4.xml*.

Este *menu* é acedido no ecrã *Track*, da aba principal, e contém seis itens. Três deles são os mesmos que o *menu1*. Tem ainda um item para poder guardar a posição atual, como sendo a posição da viatura, um item para eliminar a posição da viatura e um último item para escolher a vista do mapa, como já foi mencionado na descrição da atividade *Track.java*. Este último, recorre ao *layout* ***layers_dialog.xml***, como pode ser visto na Figura 4.31.

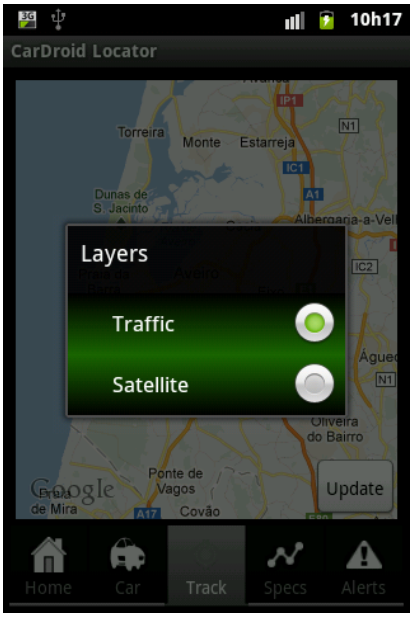

Figura 4.31 - *Layout layers_dialog.xml*.

Este *layout* usa os objetos *LinearLayout*, para enquadrar os *TextView* e os *RadioButton*, no ecrã.

Segue-se agora a descrição da atividade ***CarChoose.java***, que é acedida através do item *Car(switch)* do *menu1* e serve para mostrar a lista de viaturas presentes na base de dados. Esta atividade não recorre a nenhum *layout*, uma vez que se trata de uma lista e usa a extensão *ListActivity*.

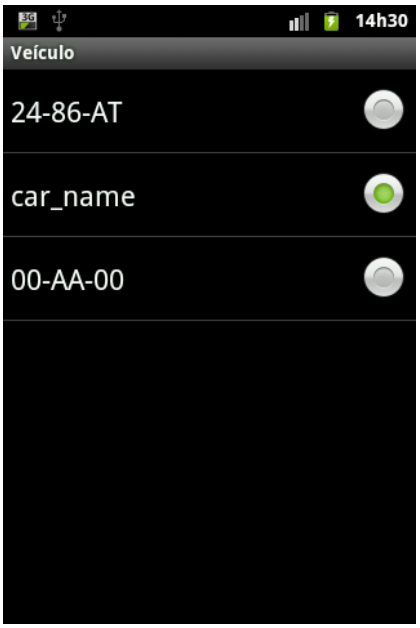

Figura 4.32 - *Screen* da lista de viaturas, com algumas viaturas inseridas.

Inicialmente é criada a lista, através da classe *ListView* e é imposta a condição de ser uma lista de uma escolha apenas, através do campo *setChoiceMode* da classe *ListView*.

Depois é usada a função *setOnItemLongClickListener()*, que é invocada sempre que se pressionar um item da lista durante breves instantes e serve para apresentar a informação, relativamente à viatura do item selecionado. Dentro desta função, inicialmente é usado o *layout login_dialog.xml* para o utilizador inserir o *login* e *password*, para poder visualizar a informação relativa à viatura. É aqui, novamente imposto a confirmação desta ação, através de *password*, uma vez que se trata de informação que não pode ser acedida por terceiros, de forma ilícita, no caso destes terem acesso indevido ao dispositivo que contém a aplicação. O acesso à informação, do código da viatura e do número do cartão *SIM*, poderia dar acesso à viatura em questão.

Ao ser verificado o *login* e *password*, é apresentado o *layout* ***inforcar_dialog.xml***, como mostra a Figura 4.33. Este *layout* recorre, uma vez mais, aos objetos *TextView* e a dois *Button*.

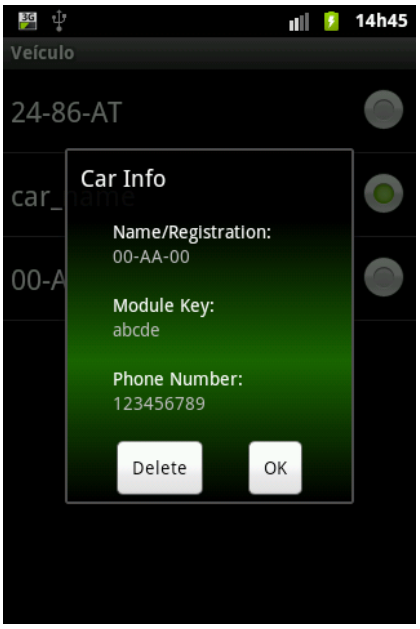

Figura 4.33 - *Layout infocar_dialog.xml*.

Aqui, é usada a base de dados, relativa às viaturas, para aceder à informação relativa à viatura e mostrar ao utilizador. Além disso, o botão "*Delete*" serve para eliminar a viatura selecionada. Para isso, foi criada uma função para o botão "*Delete*", com o nome *delete()*. Esta função começa por obter o "*id*" da viatura, que é um número que representa a viatura, na base de dados. Depois obtém a posição da viatura, na lista. Depois é eliminada a viatura da base de dados, e é verificado se a viatura eliminada era a mesma viatura que estava selecionada como a pretendida. Se sim, atualiza a *SharedPreferences* "*Choose Preferences*", que tem como objetivo guardar o *id* relativo à base de dados e a posição da viatura na lista, para que não haja nenhuma viatura escolhida. Depois, ainda dentro da função *delete()*, usa-se a função *filldata()* para preencher a lista, com a base de dados atualizada. Finalmente verifica se a viatura eliminada era a última da base de dados e se assim for, termina o serviço, uma vez que a não haver viaturas na base de dados, não serão recebidos alertas pela aplicação e assim a não ser necessário o serviço.

Esta atividade tem ainda a função *onListItemClick()* que serve para quando se clicar num item, com um simples toque, marcar a viatura relacionada com o *item* como escolhida e é aqui atualizada a *SharedPreferences* "*Choose Preferences*" para se ter acesso, em toda a aplicação, à viatura escolhida.

A atividade conta ainda com as funções para apresentar o *menu3* e as funções *onResume()*, *onPause()* e *onDestroy()* para inserir o código relativo a abrir ou fechar a base de dados, para um correto funcionamento da aplicação.

Para terminar a descrição das atividades que usam *layouts*, tem-se a atividade ***AddCar.java***, que usa o *layout addcar_dialog.xml*, já descrito anteriormente, que pode ser acedido através do item "*New*" do *menu3*. Esta atividade tem o objetivo de inserir novas viaturas na base de dados e usa como extensão a classe *Activity*.

Inicialmente, a atividade apresenta o *layout addcar_dialog.xml* como *Dialog*, para o utilizador

preencher os campos necessários à criação de uma nova viatura. Depois é usada uma função para a ação do botão "*Add*". O código aqui presente diz respeito à verificação dos campos preenchidos e à gravação dos dados, na base de dados. São feitas condições para ver se os campos foram devidamente preenchidos. Essas condições são a verificação de existirem caracteres para o nome/matrícula da viatura, se o código do módulo é composto por 5 carateres e se o número do cartão *SIM* é composto por 9 dígitos. Se forem verificadas todas as condições, é invocada a função *save()* que tem o objetivo de guardar os dados. Esta função, além de guardar uma nova viatura na base de dados, inicia ainda o serviço, para garantir que o serviço está sempre ligado, quando existirem viaturas na base de dados. Finalmente, a atividade usa ainda as funções *onPause()*, *onResume e onDestroy()* para garantir um correto funcionamento da base de dados, como descrito em atividades anteriores.

De seguida, são descritos os ficheiros de código que dizem respeito à base de dados.

A base de dados usada é uma base de dados *SQLite*, que é a base de dados recomendada em aplicações *Android*. Esta base de dados pode ser vista como várias tabelas, onde cada tabela diz respeito a um tipo de objeto a guardar e em que cada tabela pode ser vista como um conjunto de linhas e colunas. As colunas são impostas ao criar a tabela e cada uma diz respeito a um tipo de dados sobre o objeto a guardar. As linhas são criadas à medida que o utilizador insere novos objetos à respetiva tabela da base de dados. Resumindo, uma base de dados *SQLite* pode ser vista como um conjunto de tabelas compostas por um número fixo de colunas e um número indefinido de linhas, onde o utilizador vai aumentando ou diminuindo o número de linhas, de cada tabela, ao inserir ou remover objetos à base de dados.

Inicialmente é criado um ficheiro de código, com o nome ***CarDroidLocatorDBHelper.java*** que usa como extensão a classe *SQLiteOpenHelper* que serve para auxiliar a manipulação da base de dados e garantir uma simples utilização da mesma. Este ficheiro começa com a criação do nome da base de dados "*CarDroidLocatorDataBase*", a versão e as tabelas que serão usadas. As tabelas usadas serão duas, com os nomes "*Cars*" e "*Alerts*" e serão posteriormente criados ficheiros responsáveis pela gestão de cada tabela.

Ainda em *CarDroidLocatorDBHelper.java*, é usada a função *onCreate()* onde são iniciadas as tabelas, na primeira utilização da base de dados. É também usada a função *onUpgrade()*, para ser possível atualizar a própria base de dados, no futuro, se for caso disso.

Quanto às tabelas, criaram-se dois ficheiros com os nomes *CarsDBAdapter.java* e *AlertsDBAdapter.java*. Cada ficheiro diz respeito a uma tabela.

Na tabela "*Cars*", o ficheiro ***CarsDBAdpater.java*** usa as funções específicas às tabelas de uma base de dados *SQLite*. Começa portanto com a função *open()*, que serve para abrir ou criar, se for a primeira utilização, a tabela. Depois usa a função *close()* que serve para fechar a tabela. É depois criada a função *createCars()* que faz uso do método *insert()* que serve para inserir uma nova linha à tabela e dar um número *id* a cada linha e faz uso, também, da função *createContentValues()*, criada para registar um conjunto de dados que formará uma linha da tabela, recorrendo à classe *ContentValues*. Aqui, os dados usados são três que dizem respeito ao

número de colunas de cada objeto. São eles o nome ("*name*"), o código ("*modkey*") e o número ("*number*"). É depois criada a função *updateCars()* que usa o método *update*, para atualizar uma linha da tabela. Usa ainda a função *deleteCars()* que recorre ao método *delete*, que serve para eliminar uma linha da tabela. Usa ainda as funções *fetchCars()* e *fetchAllCars()* que usam o método *query*. Este método permite fornecer os dados relativos a uma linha, usando apenas o "*id*" correspondente à linha. A função *fetchAllCars()* fornece um *Cursor* que aponta para imediatamente antes da primeira linha presente na tabela. Um *Cursor* é uma classe que permite ter acesso às linhas das tabelas.

Quanto à tabela dos alertas, o ficheiro criado para o efeito tem o nome ***AlertsDBAdapter.java*** e é semelhante ao ficheiro descrito anteriormente.

Esta tabela contém apenas uma coluna "*alert*" e diz respeito ao alerta a ser gravado. As funções usadas são *open()*, *close()*, *createAlerts()*, *deleteAlerts()*, *fetchAlerts()*, *fetchAllAlerts()* e *createContentValues()*. Cada função é semelhante à usada na tabela anterior. De notar que na presente tabela não se usa a método *update*, uma vez que não faz sentido atualizar um alerta que foi recebido.

Ficam assim descritos todos os ficheiros que dizem respeito à base de dados.

Para terminar a descrição dos ficheiros de código, serão descritos os ficheiros *CDLService.java*, *StartCDLServiceAtBoot.java* e *SendSMS.java*.

Começando pelo ficheiro ***CDLService.java***, que diz respeito ao serviço da aplicação, este usa o componente *Service* como extensão. Como já foi descrito anteriormente, o serviço é iniciado na atividade *AddCar.java* aquando da criação de novas viaturas, através do comando *startService()*. Este comando invoca a função *onStartCommand()*, presente na atividade do serviço e tem como função, neste caso, registar a ação pretendida para usar o serviço. A ação pretendida é a de iniciar o serviço sempre que surja uma nova mensagem escrita no dispositivo. O código usado para isso é o seguinte:

```
filter.addAction("android.provider.Telephony.SMS_RECEIVED");
registerReceiver(receiver, filter);
```

Como se pretende usar o método *Receiver*, cria-se uma nova função, usando a classe *BroadcastReceiver* que é a classe que recebe *Intents*, enviados pelo método *sendBroadcast()*, que faz parte do sistema. Assim, sempre que existirem novos *Intents*, neste caso mensagens escritas, o serviço aqui criado será invocado e efetuará o trabalho, relativo ao código usado na função *BroadcastReceiver()*, mais propriamente num método que lhe pertence e que tem o nome *onReceive()*.

Dentro deste método, inicialmente são obtidos os dados relativos ao *Intent* recebido (mensagem de texto), nomeadamente o conteúdo da mensagem e o número do remetente. Depois é usada a tabela das viaturas da base de dados para verificar se o número do remetente está presente, ou seja, se o número que enviou a *SMS* é de alguma viatura presente na base de dados. Se não for, não é feita nenhuma ação pelo serviço. Se for de uma viatura presente, é usado o método *abortBroadcast()* seguido de uma função (*doAlerts()*) criada para desenvolver a

ação pretendida. O método *abortBroadcast()* indica ao sistema que qualquer outro *BroadcastReceiver*, presente noutra aplicação, não receberá este *broadcast*. Assim, evita-se que as mensagens que dizem respeito à aplicação *CarDroidLocator* não sejam enviadas para a caixa de mensagens recebidas, e também evita que surja a notificação de nova mensagem recebida.

A função *doAlerts()* foi criada para analisar o conteúdo da mensagem recebida. Esta recebe o número do remetente, o conteúdo da mensagem e o número de identificação ("*id*") da viatura em causa. Assim, inicialmente são criadas as cinco frases possíveis de serem o conteúdo da mensagem. As cinco frases dizem respeito aos três alertas existentes na aplicação, à mensagem que contém a telemetria e à mensagem que contém a posição da viatura. As frases são compostas por três letras iniciais (*CDL*) que dizem respeito ao nome da aplicação e serve para garantir que se trata realmente de uma possível mensagem esperada. Depois tem 5 caracteres, que dizem respeito ao código do módulo da viatura, para garantir um nível de segurança maior. Depois tem uma palavra que descreve o tipo de mensagem, se é de alerta e qual, se é da posição da viatura ou se é da telemetria. No caso de alarme a mensagem contém ainda a hora e data. No caso dos outros dois alertas, a mensagem não contém mais nenhuma informação, no seu conteúdo. No caso de ser a posição da viatura, esta contém ainda a latitude e longitude da posição. No caso de ser de telemetria, esta contém toda a informação relativa à telemetria e também a hora e data da mesma.

Depois são feitas condições para ver que tipo de mensagem se trata. No caso de ser um alerta, é invocada a atividade *Alerts.java* e é-lhe passada a informação relativa ao alerta. No caso de se tratar da posição da viatura, é invocada a atividade *Track.java* e é-lhe passada a informação. No caso de ser a mensagem de telemetria, é invocada a atividade *Specs.java* e é-lhe então passada a informação necessária.

Existe ainda uma outra atividade que tem como único objetivo, iniciar o serviço quando o dispositivo for iniciado. Esta atividade tem o nome ***StartCDLServiceAtBoot.java*** e recorre à extensão *BroadcastReceiver*.

Esta atividade correrá em *background*, assim que o dispositivo for iniciado e por isso o utilizador não terá a perceção do seu funcionamento.

Inicialmente é acedida a base de dados, para verificar se existe alguma viatura presente. Se sim, inicia o serviço através do método *startService()*. Se não existirem viaturas, a atividade é terminada e não inicia o serviço.

Finalmente tem-se a atividade ***SendSMS.java*** que usa como extensão a classe *Activity* e tem como objetivo enviar mensagens escritas e também fazer chamadas telefónicas. Esta atividade é usada por várias outras atividades, nomeadamente aquelas que utilizam botões para desempenhar uma função na viatura ou sobre esta.

A atividade começa por usar uma condição para verificar se quem iniciou a atividade pretende que esta faça uma chamada ou envie uma *SMS*. Se for para fazer uma chamada, a atividade usa a função *call()*, criada para fazer uma chamada telefónica, onde lhe é passado o número para o qual deve ligar. O código usado para o efeito é o seguinte:

```
Intent i_call = new Intent(Intent.ACTION_CALL, Uri.parse("tel:" + phoneNumber));
startActivity(i_call);
```

Se for para enviar uma *SMS*, é utilizada a *SharedPreferences* "*Choose Preferences*", para se obter a viatura selecionada e posteriormente obtidos os dados da viatura, através da base de dados. São depois usados esses dados para criar a mensagem. Finalmente é chamada a função *sendSMS()*, criada para o envio de mensagens escritas. Esta função faz uso da classe *SmsManager* e do método *sendTextMessage()* como mostra o exemplo de código seguinte:

```
SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage(number, null, msg, null, null);
```

Por fim, a atividade faz uso das funções *onResume()*, *onPause()* e *onDestroy()* para colocar o código relativo a fechar e abrir a base de dados, para um correto funcionamento da mesma.

4.4.AndroidManifest.xml

Para terminar a descrição da aplicação, será descrito nesta secção o ficheiro *AndroidManifest.xml*.

Este ficheiro é responsável por declarar todos os componentes da aplicação, assim como as permissões, bibliotecas e requisitos da aplicação, ao sistema.

Inicialmente é declarado o pacote que contém a aplicação, como demonstra o código `package="com.ControlDroid.CarDroidLocator"`. Depois é declarado o *API Level*, usado pela aplicação, neste caso, como se trata da versão 2.3.3 do *SO Android*, corresponde ao *API Level* 10.

De seguida é usado o objeto <*application*> onde serão declaradas todas as atividades presentes na aplicação. A existir uma atividade que não seja aqui declarada, esta não fará parte da aplicação. Dentro do objeto <*application*>, inicialmente é declarada a imagem e o nome que se pretende para a aplicação. *Icon* e nome que será apresentado no dispositivo, referente à aplicação.

Depois são usados os objetos <*activity*>, que são usados para a declaração de cada atividade. Dentro de cada um, é declarado o nome da atividade, a etiqueta que se pretende mostrar ao utilizador, referente à atividade entre outras possíveis condições como o caso do tema usado pela atividade ou a orientação do ecrã, pretendido para a atividade.

Relativamente à atividade que deve ser a atividade de lançamento da aplicação, esta usa o objeto <*intente-filter*> que contém os objetos <*action*> e <*category*> que servem, neste caso, para declarar que será esta a atividade a primeira a ser lançada, ao iniciar a aplicação.

Além das atividades, existem objetos específicos para declarar os serviços ou os *Receivers*. No caso do serviço, existe o objeto <*service*> onde é declarado o ficheiro *CDLService.java*. Para os *Receivers*, na presente aplicação existem dois. Um que diz respeito ao ficheiro *AlertReceiver.java* e outro que diz respeito ao ficheiro *StartCDLServiceAtBoot.java*. Este último usa, além do objeto

<*receiver*>, os objetos <*intente-filter*> e <*action*> para expressar a vontade de ser lançado, no ato do dispositivo ser iniciado. O código usado é o seguinte:
<action android:name="android.intent.action.BOOT_COMPLETED"/>.

Existe ainda o objeto <*uses-library*> que diz respeito às livrarias requeridas pela atividade. Aqui foi usada apenas a livraria refente aos mapas, como mostra o código seguinte:
<uses-library android:name="com.google.android.maps"/>.
Por último, são usadas as permissões requeridas pela aplicação. As permissões requeridas são *INTERNET*, *SEND_SMS*, *RECEIVE_SMS*, *CALL_PHONE*, *VIBRATE*, *WAKE_LOCK*, *WRITE_SMS*, *READ_SMS*, *ACCESS_COARSE_LOCATION*, *RECEIVE_BOOT_COMPLETED* e *ACCES_FINE_LOCATION*. Um exemplo de código é o seguinte:
<uses-permissionandroid:name="android.permission.INTERNET"/>.

5. Hardware

O dispositivo desenvolvido tem como material utilizado o *Kit* de desenvolvimento da *Microchip* "*PIC32 Ethernet Starter Kit*" (ver Figura 5.1) que contém o microcontrolador utilizado para controlar todos os periféricos e fazer a gestão do dispositivo. Além deste, utiliza também uma outra placa de desenvolvimento da *Microchip* "*Machine-To-Machine (M2M) PICTail Daughter Board*" (ver Figura 5.2) que possui um módulo *GSM/GPRS* para comunicação e um módulo de *GPS* para obtenção de coordenadas.

É usada também uma *placa de circuito impresso*, para simular os periféricos de uma viatura, recorrendo a *LEDs*, para demonstração do funcionamento de todo o sistema.

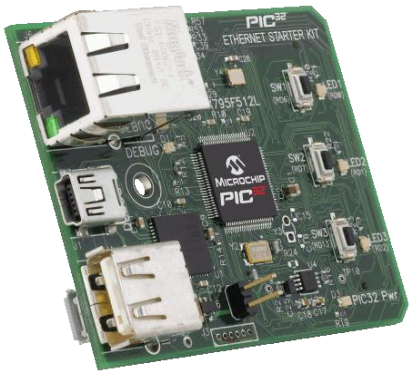

Figura 5.1 - *PIC32 Ethernet Starter Kit*.

Figura 5.2 - *Machine-To-Machine (M2M) PICTail Daughter Board*.

5.1.Material usado

O material escolhido, teve em conta a qualidade/preço, o *know-how* e os recursos pretendidos.

Uma vez que no presente curso existe muita interação com microcontroladores da *Microchip* (*PICs*), foi tido esse *know-how* em conta e optado por um microcontrolador da *Microchip*. Uma vez

que a *Microchip* disponibiliza uma placa de desenvolvimento baseada em *GSM* e *GPS*, como são os principais recursos que se pretendem para a presente dissertação, foram escolhidas as placas de desenvolvimento, atrás mencionadas.

De notar que existe uma funcionalidade que se pretendia implementar e que com o equipamento escolhido, tal não será possível, que é o caso da vídeo chamada. Para isso seria necessário um módulo de comunicação de terceira geração como o *UMTS*, por exemplo.

5.1.1. PIC32 Ethernet Starter Kit

O *kit* de desenvolvimento contém a placa de circuito impresso onde estão conectados todos os componentes. Os componentes são:

- Microcontrolador *PIC32MX795F512L*;
- Microcontrolador *PIC32MX440F512H USB* para fazer o "*on-board debugging*";
- Conector *USB mini-B* para *debugging*;
- Conector *USB micro-AB* para aplicações *USB OTG*;
- Conector *USB* tipo A para aplicações baseadas em anfitrião;
- Conector *Ethernet RJ-45* para aplicações baseadas em conexão de Internet;
- Conector *FX10A-120P/12-SV(71)* para conectar outras placas de desenvolvimento;
- *LED*s de indicação e aviso;
- Botões de pressão para eventuais necessidades;
- Cristal de 8 *MHz* e outro de 50 *MHz*;
- Regulador de tensão de 3.3V;
- *Ethernet Transceiver DP83848CVV/NOPB*;
- outros (resistências, condensadores, etc..). [23]

De todos os componentes oferecidos pela placa de desenvolvimento, destaca-se o microcontrolador *PIC32MX795F512L,* o conector USB para *debugging*, ou seja, para programar o microcontrolador e o conector *FX10A-120P/12-SV(71)* para se ter acesso a todos os pinos do microcontrolador.

5.1.1.1. PIC32MX795F512L

O *PIC32MX795F512L* é um microcontrolador de 32 bits, baseado em arquitetura *MIPS32*, capaz de funcionar a uma frequência de até 80*MHz*, com uma tensão de funcionamento entre 2.3V e 3.6V. Possui uma memória *RAM* de 32 *KB*, uma memória *Flash* de (512*KB* + 12*KB boot*) e 128*KB DRAM*.

Oferece ainda vários periféricos como 6 módulos *UART* capazes de suportar *RS-232*, *RS-485* e *LIN*, 4 módulos *SPI*, 5 módulos I^2C, 5 *Timers* de 16 bits, 5 interrupções externas, 8 canais *DMA*,

USB 2.0 e um controlador *USB-OTG*, um controlador 10/100*Mbps Ethernet* com canais *DMA* dedicados, 2 módulos *CAN* com canais *DMA* dedicados, 16 *ADCs* de 10 bits e ainda 85 pinos dedicados de *Input/Output* com velocidades de até 80*MHz* para alternar o seu estado. [24]

5.1.2. M2M PICtail Daughter Board

A placa de desenvolvimento *M2M PICtail Daughter Board* conta com vários componentes como:

- Módulo *GSM/GPRS LEON-G200* da *u-blox*;
- Módulo de *GPS NEO-6Q* da *u-blox*;
- Leitor de cartões *SIM C707-10M006-136-2*;
- Antena *SMD Hexa-Band* PA-25 *Flex* para comunicação *GSM/GPRS*;
- Antena *SGP.25D* para obtenção de dados de *GPS*;
- Conector *MCX50* para uma antena de *GPS* externa;
- outros. [25]

5.1.2.1. LEON-G200

LEON-G200 é o módulo *GSM*/*GPRS*, *quad-band* capaz de suportar as bandas *GSM* 850 *MHz*, *EGSM* 900 *MHz*, *DCS* 1800 *MHz* e *PCS* 1900 *MHz*. Permite taxas de até 85.6 *Kbps* em *down-link* e até 42.8 *Kbps* em *up*-link.

Oferece uma interface *UART*, uma interface *DDC* (compatível com I^2C) exclusiva para comunicação com módulos *GPS*, cinco pinos de *Input*/*Output* (*GPIO*), dois interfaces para áudio analógico e uma interface para áudio digital.

O módulo comunica com o *DTE*, neste caso o microcontrolador, através de comandos *AT* (de acordo com os standards *3GPP*: *TS 27.007* [26], *TS 27.005* [27],*TS 27.010* [28]), pela interface *UART*.

Tem um funcionamento a 3.8V (típico) com um consumo de energia de até 1.6mA em modo inativo, até 300mA em modo de chamada e de até 410mA em modo de transferência de dados, dependendo da frequência de utilização. De notar que a sua fonte de alimentação deve estar preparada para picos de corrente de até 2.5A (2A típico), necessários para "*burst*" de transferência de dados, como no caso do registo do módulo na rede.

O módulo providencia ainda pinos exclusivos para carregamento de bateria, se for usada, como por exemplo o pino *CHARGE_SENSE* conectado internamente a uma *ADC* capaz de medir a tensão de carregamento.

Disponibiliza ainda algumas funcionalidades como *CellLocate*, que permite obter a posição do módulo, através do sinal captado pelas antenas da rede, *FOTA* que permite atualizar o *firmware* através da rede da operadora, entre outras.

Permite também acesso a serviços de rede como *TCP/IP*, *FTP*, *HTTP*, *SMTP* e *DNS*. [29]

5.1.2.2. NEO-6Q

NEO-6Q é o módulo *GPS* da *u-blox* capaz de suportar *A-GPS* e "*AssistNow Autonomous*". *AssistNow Autonomous* é uma funcionalidade parecida ao *A-GPS* sem necessitar de uma ligação à rede. O seu princípio baseia-se em guardar os dados recolhidos pelo módulo e usá-los posteriormente para obter um melhor resultado da localização. Os dados recolhidos podem ser usados até 3 dias depois de terem sido recebidos.

O tipo de receção é de 50 canais, capaz de ostentar *TTFF* em menos de 1 segundo, usando um sistema de aquisição de dados com 2.000.000 de correlatores.

Quanto à performance, tem um tempo de 26 segundos quer em modo "*Cold Start*" quer em modo "*Warm Start*" e tem um tempo de 1 segundo em modo "*Hot Start*".

Quanto à precisão, para uma posição horizontal é de 2.5m e para a velocidade é de 0.1m/s. Tem um limite de operacionalidade de até 4g, 500m/s de velocidade e 50.000m de altitude.

O seu funcionamento é de 3V (típico) e oferece interfaces como *UART*, *USB 2.0*, *SPI* e *DDC*. Este último para comunicar diretamente com um módulo de *GSM*. Oferece também um pino dedicado a interrupções externas para "acordar" o módulo.

O módulo *NEO-6Q* possui uma funcionalidade "*Automotive Dead Reckoning*" que permite aumentar a precisão da posição, em períodos de não receção de sinal ou degradação do mesmo. Esta funcionalidade assenta na receção de dados (digitais) de sensores externos como giroscópios, entre outros, capaz de "prever" a posição relativa em que se encontra.

Todas estas funcionalidades produzem um bom resultado, ideal para ser usado em aplicações veiculares. [30]

5.2.Diagrama de blocos

O seguinte diagrama de blocos representa os vários componentes presentes no dispositivo.

Do lado esquerdo, tem-se a alimentação, que deve ser 5V DC estabilizados. A alimentação é fornecida a todos os blocos principais.

Depois tem-se a unidade de processamento que se prende no microcontrolador *PIC32MX795F512L* da *Microchip*, que controla todos os outros componentes e faz a respetiva gestão do dispositivo. De notar que é o bloco de processamento que é responsável pela configuração do módulo *GSM* e *GPS*, através de comandos *AT*.

Segue-se o bloco *M2M* que tem como principais blocos, o módulo *GSM* e o módulo *GPS*. Através do módulo *GPS*, o dispositivo pode receber informação relativa à posição em que se encontra, que por sua vez pode ser usado pelo módulo *GSM* para enviar a informação recolhida, através da rede de telecomunicações.

Este último módulo tem como objetivo trocar informação com a aplicação *Android*, desenvolvida. Comunica também com o módulo de processamento, através da *UART*, em caso de receber instruções da aplicação *Android*. Essas instruções podem ser para realizar uma tarefa no último bloco (periféricos) ou podem ser a pedir uma informação relativa ao dispositivo, como por exemplo o estado de algum periférico ou a posição relativa do dispositivo.

O último bloco diz respeito a todos os periféricos que podem ser usados. Estes são ativados/desativados ou fornecem informação à unidade de processamento, através de portas de *input/output*.

Por fim, do lado direito, tem-se a antena externa de *GPS*, para poder obter melhores resultados, na captura dos dados, relativos à posição do dispositivo.

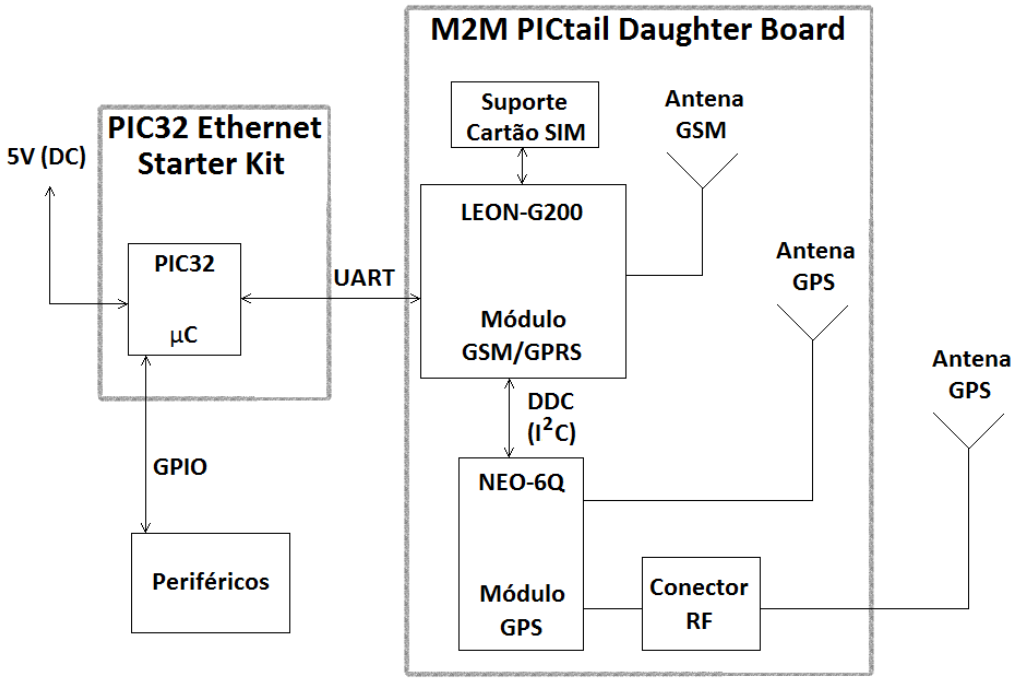

Figura 5.3 - Diagrama de blocos - *hardware*.

5.3.Equipamento desenvolvido

O equipamento desenvolvido consiste em duas partes. Uma que contém os componentes necessários para o dispositivo ser instalado numa viatura e outra que consiste num painel de demonstração, uma vez que não foi possível instalar o dispositivo numa viatura.

O dispositivo a ser instalado na viatura consiste nas placas de desenvolvimento *PIC32 Ethernet Starter Kit*, que contém o microcontrolador e *M2M PICTail Daughter Board* que contém o *modem GSM*, o módulo *GPS* e as respetivas antenas. Além destes, são usados alguns componentes para viabilizar todo o sistema, como um microfone, *LEDs* de aviso, entrada de alimentação e todas as ligações necessárias.

O painel de demonstração, criado para simular os periféricos de uma viatura, consiste num circuito onde foram instalados *LEDs*. Na frente do circuito foi colocado o desenho de uma viatura, de maneira aos *LEDs* coincidirem com os periféricos, para ser percetível o funcionamento do dispositivo. Além dos *LEDs*, existem sensores, também eles para simular os periféricos da viatura. São eles um botão de pressão para simular o sensor volumétrico ou sensor de movimento, usado para disparar o alarme e um *buzzer* para simular a sirene ou buzina.

O resultado final encontra-se na Figura 5.4.

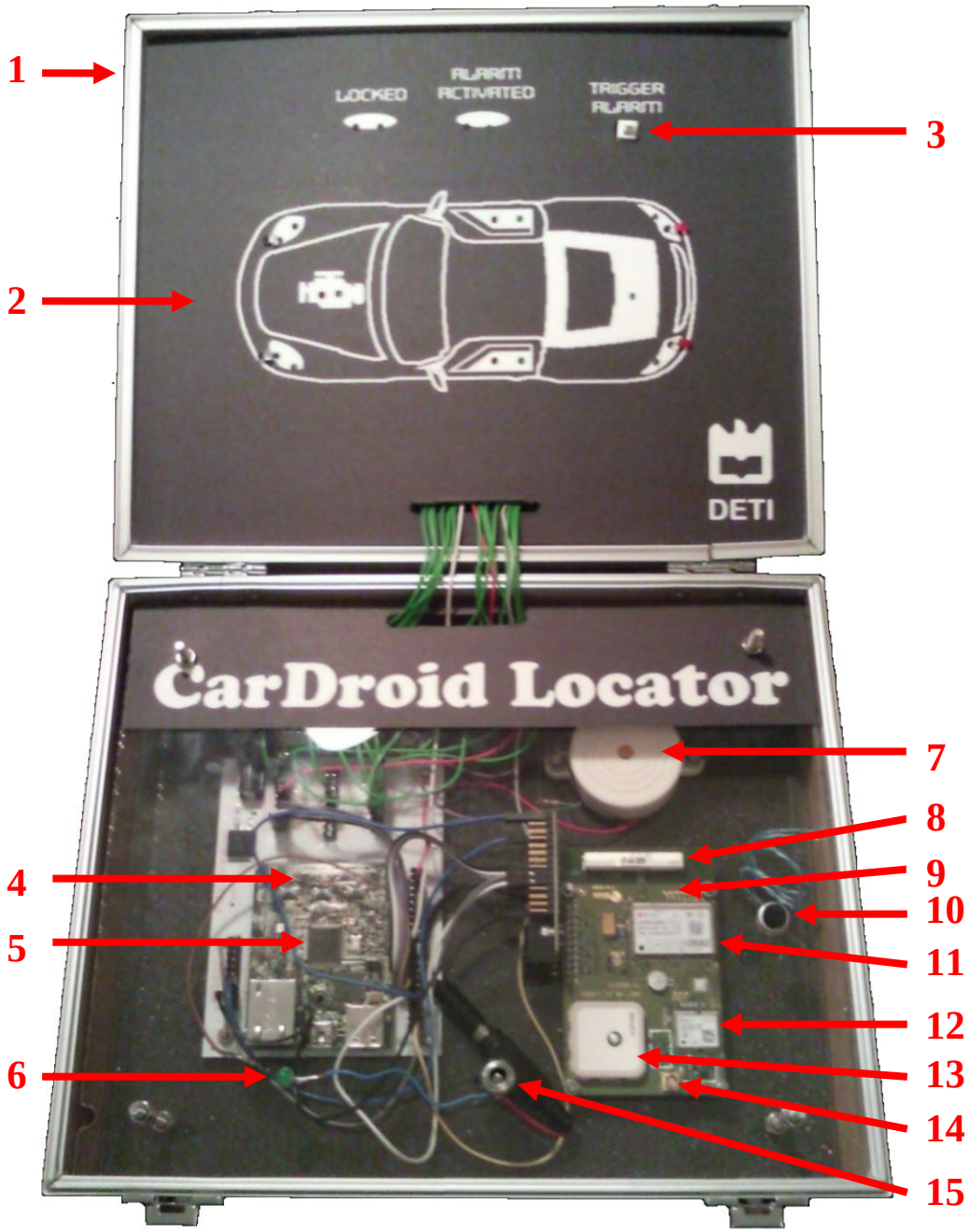

Figura 5.4 - Equipamento desenvolvido.

- 1 – Caixa metálica para acondicionar o equipamento;
- 2 – Painel de demonstração;
- 3 – Botão de simulação de sensor de alarme;
- 4 – *PIC32 Ethernet Starter Kit*;
- 5 – Microcontrolador *PIC32MX795F512L*;
- 6 – *LED* indicador de sistema a funcionar;
- 7 – *Buzzer* para simular buzina de alarme;
- 8 – Antena *GSM*;
- 9 – *M2M PICTail Daughter Board*;
- 10 – Microfone;
- 11 – Módulo *GSM* (*LEON-G200*);
- 12 – Modulo *GPS* (*NEO-6Q*);
- 13 – Antena *GPS* interna;
- 14 – Entrada para antena *GPS* externa;
- 15 – Entrada para alimentação.

Como pode ser visto na Figura 5.4, o equipamento está acondicionado numa mala. A parte inferior serve para acomodar o dispositivo a ser instalado numa viatura. Na parte superior encontra-se o painel de simulação.

O painel de simulação é ligado por fios, aos pinos *GPIO* respetivos, do microcontrolador.

A alimentação e a antena *GPS* externa, não se encontram na Figura 5.4 para esta poder ser clara.

5.4.Ferramentas - IDE e Compilador

O ambiente de desenvolvimento usado, para criar o código a ser instalado no microcontrolador, é o *MPLAB X* da *Microchip*.

MPLAB X é um programa de desenvolvimento, gráfico, baseado na plataforma *NetBeans*. É "*open-source*", portanto, livre e é usado para criar aplicações a serem instaladas em microcontroladores, disponibilizado pela *Microchip*.

Este programa permite escrever o código desenvolvido, diretamente no microcontrolador, oferecendo bibliotecas de todos os microcontroladores disponíveis (da *Microchip*). Inclui também uma solução integrada de *debugging*, além de outras ferramentas como simulador de *software*, gerenciamento de projetos suportando também muitas ferramentas de *hardware*.

O programa é compatível com o sistema operativo *Windows*, *MAC* e *Linux*, tendo sido usado, para o presente projeto, sobre *Windows*.

O *IDE* suporta também várias linguagens de programação, tendo sido usada a linguagem C. Para tal foi usado o compilador, indicado para o *IDE*, *MPLAB XC32 C Compiler*, responsável em traduzir o código fonte em código máquina, percetível pelo microcontrolador.

Este compilador é compatível com o microcontrolador usado, e oferece várias soluções para simplificar e melhorar a escrita de código de alto nível. Oferece, portanto, um editor de erros e possibilidade de usar *breakpoints* em todas as instruções de código, muito útil para perceber eventuais erros de programação. Oferece também uma ferramenta capaz de inspecionar e exibir todas as variáveis usadas e seus valores, em vários formatos, permitindo grande facilidade na perceção de todo o funcionamento do código e muito útil, também para corrigir ou evitar eventuais erros.

Assim como o *IDE*, também o compilador tem uma versão livre e é disponibilizado pela *Microchip*. [31]

5.5. Código

Na presente subsecção, será descrito o código desenvolvido para o microcontrolador, usado no dispositivo. Será, portanto, explicado todo o funcionamento do dispositivo criado, capaz de obedecer a comandos, transmitidos via *GSM*, através de *SMS* ou de sensores presentes (periféricos), através de impulsos transmitidos via "*wired*" aos pinos de *input* do microcontrolador.

De notar, que foram usadas bibliotecas da *Microchip* desenvolvidas diretamente para as placas de desenvolvimento usadas, tendo sido assim, usadas algumas funções já criadas.

As bibliotecas usadas são, num nível mais alto, ***libubx.c***, responsável pela configuração das várias funcionalidades de ambos os módulos. Depois num mesmo nível tem-se ***libgps.c***, responsável pela aquisição de dados *GPS* e respetivo processamento dos mesmos. Num nível mais baixo, encontra-se a biblioteca ***libacd.c***, responsável pelo tratamento das várias funções usadas nas bibliotecas anteriormente mencionadas, que tem como objetivo a escolha dos vários comandos *AT*, para representar as funções escolhidas. Por fim, num nível inferior tem-se as bibliotecas ***libcom.c*** e ***libp32.c***. A primeira responsável pela gestão das várias interfaces usadas, como a *UART* ou o canal *DMA*. É responsável também pela gestão das interrupções e da memória. A segunda está responsável pela inicialização dos vários componentes de ambas as placas, como por exemplo a escolha da frequência usada pelo microcontrolador, a configuração dos *LEDs* de aviso ou a configuração dos pinos de alimentação ou *reset*, das placas.

Para uma melhor compreensão do código, é representado na Figura 5.5 o fluxograma referente a este.

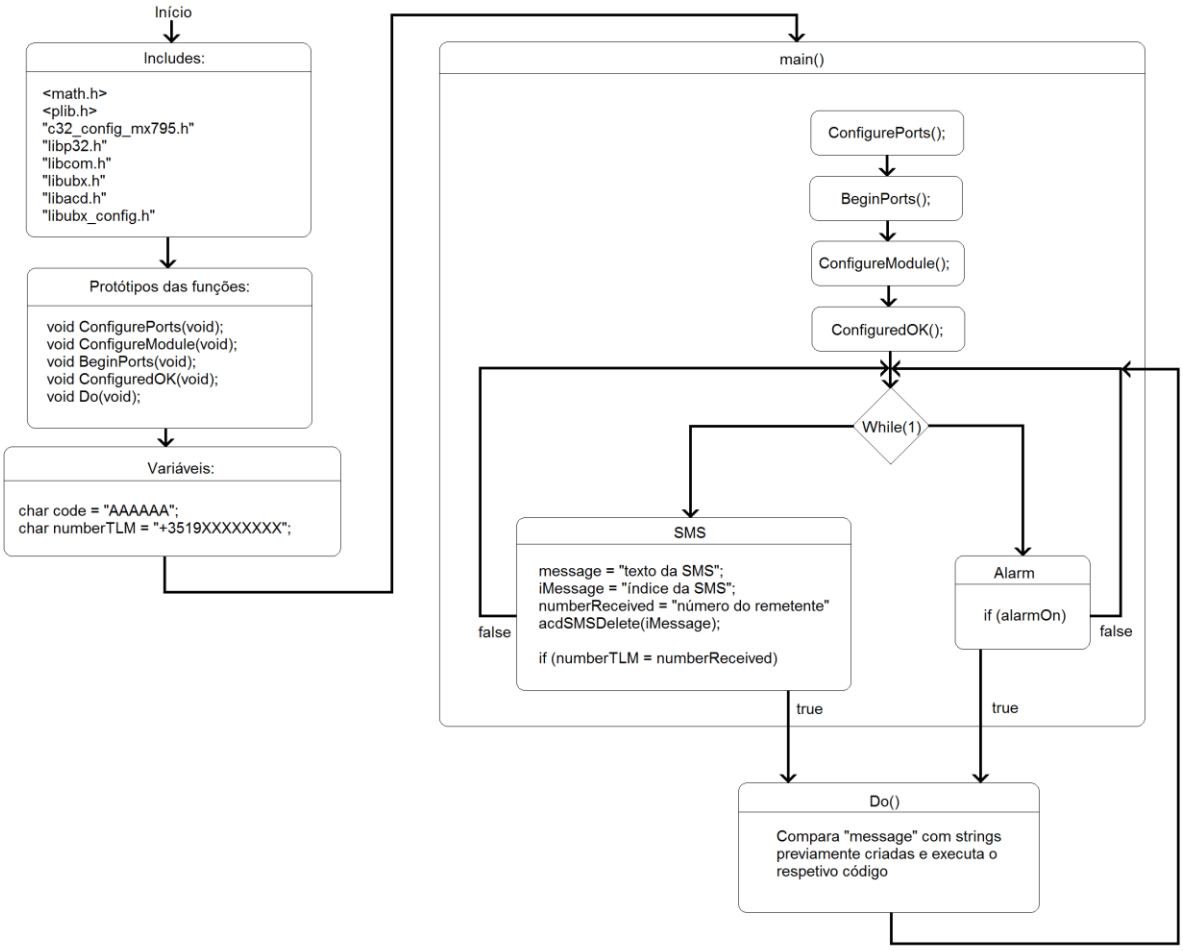

Figura 5.5- Fluxograma do código.

Do código criado, este começa com a atribuição do código do módulo (5 carateres) e do número de um cartão *SIM* (12 algarismos) a duas variáveis. O código e o número do cartão *SIM* serão comparados, sempre que seja recebida uma nova *SMS*.

Depois usa a função ***ConfigurePorts()***, criada para configurar os pinos que serão usados pelos periféricos existentes. Um exemplo da configuração de um pino é o seguinte:

```
PORTSetPinsDigitalOut(IOPORT_G, BIT_13);
#define CDL_LOCK IOPORT_G, BIT_13
#define CDLLockOn() PORTSetBits(CDL_LOCK), lock=1
#define CDLLockOff() PORTClearBits(CDL_LOCK), lock=0
```

Aqui é configurada a porta G13, que é destinada a trancar a viatura.

A próxima função a ser usada é ***BeginPorts()***, onde são impostos os estados iniciais de todos os periféricos. De todos eles destaca-se o alarme, que é inicialmente desligado e o fecho das portas que toma o estado de aberto.

De seguida é usada a função ***ConfigureModule()***, criada para fazer toda a configuração do módulo *GSM* e *GPS*. Esta função é iniciada com a invocação da biblioteca *libp32.c*, onde vão ser inicializadas as placas, isto é, a entrada em funcionamento quer do microcontrolador quer dos

restantes módulos existentes. De seguida é configurada a interface *UART*, assim como o canal *DMA*.

É depois configurada a rede *GSM*, com os seguintes comandos:

- AT – Testa o funcionamento da comunicação entre o módulo *GSM* e o microcontrolador;
- ATZ – Elimina qualquer configuração que possa estar em memória do módulo *GSM*;
- AT&K3 – Define o mecanismo de controlo de fluxo para que seja controlado pelo *DTE* (microcontrolador) através de *RTS/CTS*;
- ATE0 – Elimina o envio do eco, aos comandos enviados pelo microcontrolador, pelo módulo;
- AT+CGMI – Obtém o nome do fabricante, para garantir a boa configuração do módulo;
- AT+CGMM – Obtém o modelo do módulo, para garantir a boa configuração do módulo;
- AT+CMEE=2 – Reporta os erros ocorridos, através de mensagens escritas e não por códigos numéricos;
- AT+UPSV=0 – Desabilita o controlo de poupança de energia;
- AT+CCID – Obtém o *ICCID* (*Integrated Circuit Card ID*) do cartão *SIM*, para confirmar a presença de um cartão *SIM*;
- AT+CREG=0 – Garante que o módulo não está registado em nenhuma operadora;
- AT+UGPOIC=20,2,0 – Configura o pino 20 (responsável pelo GPIO1), como indicador de rede (2), a nível baixo (0). Este pino é ligado a um *LED* que indica se o módulo está registado em alguma rede de telecomunicações;
- AT+COPS=0 – Escolhe o modo automático para registar o módulo na rede;
- AT+COPS? – Verifica se já existe alguma rede, ao qual o módulo esteja registado. Se sim ativa uma *flag* para anunciar ao programa que o módulo *GSM* foi registado com sucesso.

Estando a rede *GSM* configurada, procede-se à configuração das mensagens escritas com os seguintes comandos:

- AT+CSMS=1 – Escolhe o serviço de mensagens a usar (referência nos documentos [32], [33] e [27]);
- AT+CPMS=”MT” – Indica que o local onde serão guardadas as *SMS* é na memória do módulo *GSM* e posteriormente (quando terminar a memória) no cartão *SIM*;
- AT+CMGF=1 – Indica que o formato das *SMS* é no modo texto;
- AT+CSDH=0 – Indica que não quer que sejam mostrados os parâmetros extra das *SMS*;
- AT+CNMI=2,1 – Define o modo de aviso na receção de novas *SMS*. O 2 representa que as *SMS* devem ser guardadas em *buffer*, quando o canal de transmissão estiver ocupado e o 1 representa que deve ser enviado ao *DTE* (microcontrolador) a memória e o índice de onde foi guardada a *SMS*.

De seguida, regista-se o módulo *GPS NEO-6Q*, com os comandos seguintes:

- AT+UGIND=0 – Desabilita indicações não solicitadas por parte do *GPS*;

- AT+UGPRF=0 – Desabilita o fluxo de informação recebida pelo *GPS* para outro local, como por exemplo para um ficheiro ou para um endereço *IP*;
- AT+UGGGA=1 – Habilita o tipo de informação que se pretende receber pelo *GPS*, aqui as mensagens *NMEA $GGA*;
- AT+UGGLL=1 – Habilita mensagens *NMEA $GLL*;
- AT+UGGSA=1 – Habilita mensagens *NMEA $GSA*;
- AT+UGGSV=1 – Habilita mensagens *NMEA $GSV*;
- AT+UGRMC=1 – Habilita mensagens *NMEA $RMC*;
- AT+UGVTG=1 – Habilita mensagens *NMEA $VTG*;
- AT+UGZDA=1 – Habilita mensagens *NMEA $ZDA*;
- AT+UGPS=1,4 – Liga o módulo *GPS* com o modo de auxílio "*AssitsNow Online*".

Depois é a vez de configurar a rede *GPRS*, com os seguintes comandos:

- AT+CGREG=0 – Desabilita a receção de informação, relativa à alteração de estado da rede *GPRS*;
- AT+UPSD=0,1,"internet.vodafone.pt" – Define o valor de *PSD*, aqui escolhido o *APN* da *Vodafone*;
- AT+UPSDA=0,1 – Regista o comando anterior na memória não volátil, do *modem*;
- AT+CGATT? – Verifica se o serviço *GPRS* está ativo;
- AT+UPSDA=0,3 – Ativa o *PDP*. Fica assim estabelecida a conexão entre o *modem* e o servidor da rede.

Estando o dispositivo conectado à rede, é usada a função ***ConfiguredOK()*** onde liga o *LED* de aviso de configuração bem-sucedida.

Fica assim terminada a parte inicial de código, passando agora para um ciclo infinito (***while(1)***), onde o microcontrolador fica "eternamente" à espera de receber uma tarefa para executar, quer seja através de um comando via *SMS*, quer de um periférico.

Como o dispositivo não está realmente instalado numa viatura, encontrando-se ainda numa placa de demonstração, como referido em 5.3, existe apenas um periférico que pode despoletar uma ação ao microcontrolador. Esse periférico é o alarme. Por isto, o microcontrolador espera receber uma de duas ações:

- Receber um impulso no porto configurado, quando recebe uma nova *SMS* (RB2);

- Receber um impulso no porto configurado para o alarme (RG6).

Quando é recebida uma nova *SMS*, é guardado o conteúdo da *SMS*, o número do remetente e o índice de memória onde foi guardada a *SMS*. Depois é apagada a *SMS* da memória e comparado o número do remetente com o número permitido. Se o número for permitido, procede à ação que deve ser executada, através da função *Do()*, se não for permitido não faz nada, continuando no ciclo infinito à espera de nova ação.

A função *Do()* será explicada à frente.

Quando é recebido o impulso referente ao sensor do alarme (porto RG6), inicialmente é analisada uma variável. A variável é "*alarmOn"* que indica se o alarme está ativo. Esta verificação é apenas por uma questão de bom funcionamento, uma vez que não fazia sentido o alarme ser despoletado sem estar ativo. Depois é criada uma mensagem que é guardada na variável "*message"* e por fim é invocada a função *Do()*.

Fica assim descrito todo o código presente na função ***main()***.

Passando à última função existente (***Do()***), esta é responsável por executar todas as ações pedidas, comparando a variável "*message"* com uma *string* previamente criada. A variável "*message*" contém o texto da *SMS* recebida ou a *string* "*alarmActivated*" caso tenha sido despoletado o alarme. A *string* previamente criada começa com "CDL" precedida do código do módulo, criado no início do programa, terminando com a ação a tomar, como por exemplo "CDLAAAAALOCK" onde o código é "AAAAA" e a ação é trancar a viatura ("LOCK").

Caso a variável "*message*" seja igual a "*alarmActivated*", será ativada a sirene do alarme, serão colocadas as luzes e os piscas a piscar e será recebida a hora e data atuais, usando para isso a hora do *GPS*. Depois é enviada uma *SMS* ao utilizador, da aplicação móvel, com um identificador que despertará o alarme na aplicação.

Se for para trancar a viatura, verifica primeiro se não está trancada, seguindo-se o desligar das luzes se estas se encontrarem ligadas, o desligar do motor se estiver em funcionamento, o fechar dos vidros se estes se encontrarem abertos e por fim tranca a viatura.

Se for para destrancar a viatura, verifica se está trancada, desliga o alarme se este se encontrar ativo, destranca a viatura e liga as luzes durante um tempo pré-estabelecido.

Se for para ativar o alarme, verifica se a viatura se encontra trancada e se sim apenas liga o alarme. Se não tiver trancada, faz o mesmo procedimento de trancar incluindo a ativação do alarme.

Se for para desligar o alarme, existem duas opções. A primeira se for meramente para desativar o alarme, onde é simplesmente desligado. A segunda se o alarme se encontrar em ação. Aqui desliga a buzina, desliga as luzes e por fim desliga o alarme.

Se a mensagem for de pânico, liga a sirene, as luzes e os piscas.

Se for para ligar o motor verifica se este se encontra desligado e se o estado "destrancado" se encontra ativo. Se sim, liga o motor e as luzes da viatura.

Se for para desligar o motor, verifica apenas se este se encontra ligado e se sim, desliga-o.

Para abrir ou fechar os vidros, quer do condutor quer do passageiro, é verificado se este se encontra aberto ou fechado e é procedida a ação solicitada. Aqui a ação é ativar o "sensor" (neste caso um *LED*) por um tempo pré-determinado onde é simulado o tempo de abrir ou fechar um vidro.

Se for para abrir a mala, é verificado se o alarme se encontra desligado. Se sim, ativa o *LED* durante um tempo pré-determinado (1 segundo), simulando o trinco da mala de uma viatura.

Se a mensagem for a pedir a localização, é obtida a localização atual, através do *GPS* e enviada uma nova *SMS* com a posição.

Se a mensagem for de telemetria, é recebida a informação do *GPS*. Como não existem sensores da viatura, a única informação possível de enviar é a velocidade, a hora e a data, recebidas pelo GPS.

Fica por desenvolver o código relativo ao envio de alerta de ignição, alerta de limite de velocidade, envio de alguns dados de telemetria, uma vez que não existem sensores para obter todos estes dados e o código de chamada e vídeo chamada para ser possível ouvir e ver o que se passa dentro da viatura. A parte da vídeo chamada é impossível com o módulo usado.

6. Conclusões e Considerações Finais

Apresentam-se aqui, as principais conclusões relativas ao trabalho efetuado. Expõem-se também, algumas sugestões de trabalho futuro, que poderão melhorar e dar continuidade ao projeto elaborado, na presente dissertação, assim como considerações finais relativas ao processo de elaboração de uma dissertação.

6.1. Conclusões

O foco da presente dissertação prendia-se em dois objetivos principais. Sendo o primeiro objetivo criar uma aplicação móvel para o sistema operativo *Android* e o segundo, criar um dispositivo, baseado em tecnologias *GSM* e *GPS*, capaz de ativar vários atuadores, controlado remotamente, com a interação do utilizador com a aplicação móvel.

Quanto ao primeiro objetivo, foi amplamente conseguido, tendo sido até superados os objetivos iniciais. Pretendia-se recorrer aos conhecimentos adquiridos numa das disciplinas do curso (Computação Móvel), para se criar uma aplicação simples, com as funcionalidades básicas do controlo remoto de uma viatura, com recurso a botões simples e menus básicos.

Todas as funcionalidades pretendidas foram realizadas, à exceção de uma, mais concretamente a vídeo chamada. Esta funcionalidade não foi produzida, devido à falta de capacidade para tal, do dispositivo disponível, usado para testes.

Quanto às restantes funcionalidades, a aplicação móvel está habilitada a interagir com o alarme do dispositivo, quer para ativar quer para desativar ou mesmo para o colocar em funcionamento, como é o caso da função “Pânico”. Está também habilitada a interagir com o fecho centralizado das portas do veículo, oferecendo a opção de trancar as portas, sem a ativação do alarme. Está apta também, a abrir e fechar as janelas laterais, a abrir a mala traseira ou até ligar e desligar a ignição do veículo.

Além das funcionalidades que se prendem em ativar/desativar atuadores, a aplicação móvel está também apta a receber dados de telemetria da viatura, mais concretamente a velocidade, as rotações do motor, a percentagem de combustível presente no tanque, o estado da bateria e as temperaturas do interior e exterior da viatura, assim como do motor da mesma.

Esta é obviamente ainda capaz de receber a localização do veículo e apresentá-la num mapa, enquanto apresenta também a posição em que se encontra o utilizador. Quanto às capacidades relacionadas com a posição, esta é ainda capaz de guardar o local em que se encontra o utilizador, como sendo o local da viatura. Função criada com o intuito de ser usada em parques de estacionamento.

Quanto aos alertas possíveis de serem recebidos, existem três. São eles o alarme, a entrada em funcionamento do motor, mais concretamente o estado da ignição e o limite de velocidade, quando ultrapassado.

Além das funcionalidades com direta interação com o veículo, existem outras, focadas no interface com o utilizador. São elas, a opção de adicionar o número de veículos desejado pelo utilizador, a possibilidade de alterar o idioma da aplicação (Português ou Inglês), a escolha do tipo de toque que se pretenda, ao receber um alerta ou o tipo de receção do alerta, sendo possível iniciar uma chamada telefónica para o veículo e ouvir o som do seu interior. É possível também escolher o tipo de alertas que se pretende receber e os campos que se pretendam visualizar, no ecrã de telemetria.

Além de todas as funcionalidades mencionadas, foi ainda tido em conta o aspeto de segurança, que na aplicação em causa, é de extrema importância e seria um erro grave, negligenciar este aspeto. Para isso, foram impostas duas medidas de segurança.

Uma que se prende na introdução de uma *password*. Esta pode ser escolhida pelo utilizador e obrigatória para usar os botões que despoletem uma ação no veículo. Opção que pode ser escolhida pelo utilizador para qualquer botão. Além disto, a *password* é obrigatória para alterar as preferências atrás mencionadas assim como para visualizar a informação, relativa a cada veículo, como o caso do código do módulo.

Outra que passa por criar um código único para cada dispositivo (no caso de virem a ser usados em massa). Assim, ao criar um novo veículo, terá que ser introduzido este código, como também o número do cartão *SIM*, presente no dispositivo. Com esta medida e com uma medida introduzida no código do microcontrolador (número do cartão *SIM* do utilizador da aplicação), responsável pela gestão do dispositivo, apenas o terminal que obtenha o cartão *SIM*, com o número introduzido no microcontrolador, possa comunicar com o dispositivo.

Com esta medida, torna-se praticamente impossível aceder ao dispositivo e consequentemente ao veículo, sem que se tenha conhecimento do código ou a obtenção do cartão *SIM*, associado ao dispositivo, mesmo em caso de furto do terminal que contenha a aplicação.

Com todas as funcionalidades e medidas de segurança mencionadas, pensa-se ter sido criada uma boa aplicação.

Quanto ao segundo objetivo, este ficou aquém dos objetivos impostos inicialmente.

Pretendia-se criar um protótipo, a ser montado numa viatura, que pudesse garantir todos os requisitos ou funcionalidades da aplicação.

Devido à receção tardia do *hardware* e também ao demasiado tempo empregue no primeiro objetivo, não foi possível passar à fase de prototipagem nem à fase de implementação do dispositivo, numa viatura.

Assim, ficou-se por uma placa de demonstração, recorrendo a *LEDs*, para a simulação das funcionalidades.

Contudo, grande parte da gestão do dispositivo foi elaborada. Foi cumprido o objetivo de se atuar em quase todos os componentes, simulados, de um veículo. São eles, o alarme, o fecho de portas, os vidros laterais e a porta traseira. Foi também conseguido o envio das coordenadas de *GPS*, quando solicitadas, assim como o envio do alerta de alarme, em caso de ser despoletado e o envio da velocidade, hora e data caso sejam pedidos os dados de telemetria.

Ficou por elaborar a gestão relativa à obtenção de dados dos restantes dois alertas (ignição e limite de velocidade) e dos restantes dados de telemetria. Também ficou em falta a inclusão de uma vídeo câmara e um microfone, para ser possível ver e ouvir o interior do veículo, em caso de roubo ou furto. Ficou também por cumprir o objetivo de incluir uma unidade de memória, para ser possível guardar dados como som ou imagem, do interior da viatura.

Embora não tenham sido alcançados todos os objetivos, pensa-se ter conseguido um bom produto final, na confluência de todo o trabalho exercido, para a obtenção de bons resultados.

6.2.Trabalho Futuro

Para sugestões de trabalho futuro, como todo o projeto assentou em dois objetivos distintos, com um objetivo final comum, serão apresentadas algumas ideias, quer para a aplicação móvel, quer para o dispositivo.

No que toca à aplicação móvel, poderiam ser acrescentadas as seguintes funcionalidades:

- Usar as capacidades de *Bluetooth* do terminal móvel, para comunicar com o dispositivo. Poderia ser dada a opção ao utilizador de que serviço pretendesse usar para comunicar com o veículo, *GSM* ou *Bluetooth*. No caso de ser *Bluetooth*, teria que estar ao alcance do veículo. Esta funcionalidade traria grandes melhorias, tanto para a aplicação como para o utilizador. Primeiro, porque o serviço de *SMS* poderá ser pago, ao contrário do de *Bluetooth*, depois pelo tempo de receção de uma *SMS* ser muito maior que a transferência de informação, via *Bluetooth* e por fim a funcionalidade de telemetria poderia ser contínua, isto é, o utilizador poderia usufruir das informações de telemetria continuamente e em tempo real, estando dentro ou próximo da viatura;
- Ter uma opção, em que o utilizador pudesse escolher entre uma viatura de três ou cinco portas. Aqui, seria apresentado um *layout* no ecrã "Viatura", com as caraterísticas do veículo, escolhidas pelo utilizador. Esta opção teria que ser agregada às informações de cada viatura (a ter mais que uma);
- Poderem ser adicionados vários números de cartões *SIM*, a cada dispositivo. Aqui teria de ser o utilizador com o cartão *SIM* registado no dispositivo, a poder inserir novos números de cartões *SIM* ao dispositivo. Uma funcionalidade favorável, quando se tem mais que um utilizador da mesma viatura;
- Controlo do ar condicionado da viatura. Opção significativa para dias de muito frio ou muito calor;
- Incluir a opção de corte de energia da viatura. Opção a usar no caso de furto da viatura e a consequente imobilização da mesma;
- Incluir um novo alerta quando existir um corte de corrente no dispositivo, ou seja, quando a bateria da viatura for cortada do módulo;

- Poder definir o limite de velocidade do alerta “Limite de velocidade ultrapassado”, ou mesmo, usar vários limites de velocidade;
- Por fim, mas não menos importante, desenvolver o código relativo à vídeo chamada, para ser possível visualizar o interior/exterior da viatura pretendida.

Quanto ao dispositivo, poderiam ser adicionadas as seguintes caraterísticas:

- Incluir uma ou mais vídeo câmaras, no interior ou exterior da viatura. Teria de ser possível a escolha da câmara, através da aplicação móvel;
- Incluir uma pequena bateria, de 3.8V, dentro do dispositivo. Esta seria útil, para quando houvesse um corte de energia, onde seria emitido um alerta para a aplicação. Este alerta poderia passar pela receção das coordenadas da viatura, na aplicação, com um intervalo de cinco minutos;
- Incluir uma unidade extra de memória, como um cartão *microSD*, onde poderia ser gravada a imagem ou som das vídeo câmaras e microfone. Poderia ser dada a opção ao utilizador, do processo de gravação, isto é, o utilizador poderia escolher se a gravação era feita apenas quando fosse despoletado o alarme, quando a viatura estivesse em andamento, qual a vídeo câmara a usar em cada situação, entre outras possíveis. Opções que poderiam ser usadas, para identificar possíveis roubos à viatura ou possíveis acidentes;
- Incluir mecanismo de segurança, no caso de ser pedido o ligar da ignição, através da aplicação. Mecanismo que pode passar por verificar o estado das mudanças (a serem manuais);
- Incluir mecanismo de segurança, no caso de ser pedido o desligar da ignição, através da aplicação. Mecanismo que pode passar por verificar a velocidade da viatura e executar o pedido, apenas quando esta baixar de uma determinada velocidade, por exemplo 10 Km/h;
- Ligar o dispositivo através de *CAN bus*, à viatura;
- Proceder à prototipagem do dispositivo.

6.3.Notas Finais

No final da realização desta dissertação, cheguei a algumas conclusões que considero relevantes para projetos futuros.

Primeiro e por ter sido um projeto proposto por mim, a escolha das ferramentas certas e a ideia da possível implementação de um produto como este no mercado, é essencial para a motivação de quem o realiza.

Depois a nível de metodologia, penso que o estabelecimento de metas e datas é de extrema importância para o sucesso. Apesar de ter elaborado um *Gantt Chart*, no início deste projeto, este não foi nada realista. As principais razões para o incumprimento das datas foram:

- Não foi tido em conta o tempo de pesquisa e aprendizagem, para desenvolver uma aplicação a este nível, com recurso a várias funcionalidades e com uma tentativa de

interface com o utilizador, um pouco agradável e algo elaborada a nível gráfico. Este foi o processo mais moroso e o principal para o incumprimento das datas, inicialmente estabelecidas;

- Não foi tido também em conta possíveis contratempos, como foi o caso da entrega do material para o desenvolvimento de *hardware*. É de referir que inicialmente tinha sido escolhido outro tipo de *hardware*, mas o fim da sua produção e a quebra de stock em vários sítios de venda, atrasou em muito, o desenvolvimento da segunda parte do projeto;

O estabelecimento de metas também não foi rigorosamente cumprido, uma vez que na aplicação móvel foram ultrapassadas as metas e incrementadas muitas funcionalidades que não faziam parte das metas iniciais. Embora no final se tenha obtido um resultado muito melhor, prejudicou nas metas estabelecidas, no que diz respeito ao *hardware*.

A nível pessoal, penso que poderia ter alcançado um resultado melhor, se tivesse sido mais crítico, em relação a alguns contratempos surgidos.

Para finalizar, foi uma experiência muito enriquecedora a vários níveis e espero poder ter a oportunidade de realizar mais projetos, no futuro, onde sinta uma satisfação e motivação pessoal para os realizar.

Bibliografia

[1] Policia de Segurança Publica :: Noticias :: Detalhe:. Disponível em http://www.psp.pt/Pages/Noticias/MostraNoticia.aspx?NoticiasID=24. Acedido: 28 Setembro 2012.

[2] Registados 200 mil crimes em Lisboa e no Porto em 2011 | CARJACKING. Disponível em http://carjacking.com.pt/registados-200-mil-crimes-em-lisboa-e-no-porto-em-2011. Acedido: 22 Fevereiro 2012.

[3] Android recebe 700 mil atualizações por dia. Disponível em http://www.mundodossmartphones.com/modelos/android-recebe-700-mil-atualizacoes-por-dia. Acedido: 15 Fevereiro 2012.

[4] Departamento de Electrónica, Telecomunicações e Informática › Página inicial. Disponível em http://www.det.ua.pt. Acedido em 16 Fevereiro 2012.

[5] Alarm GSM Car Locator. Disponível em http://carlocator.pt.vu. Acedido: 15 Fevereiro 2012.

[6] GSM | About Us. Disponível em http://supertrunfonet.tripod.com/trunfonticiadofuturo/id1.html. Acedido: 18 Fevereiro 2012.

[7] GSMA. Disponível em http://www.gsma.com. Acedido: 18 Fevereiro 2012.

[8] GSMA. Disponível em http://www.gsma.com. Acedido: 18 Fevereiro 2012.

[9] Global Positioning System - Wikipedia, the free encyclopedia. Disponível em http://en.wikipedia.org/wiki/Global_Positioning_System. Acedido: 18 Fevereiro 2012.

[10] Nº1 no Combate ao Carjacking | Localizador GPS | Localizador de Carros | Localizador de Viaturas | Inosat Car Locator. Disponível em http://www.inosat.pt/particulares/car-locator-1.aspx. Acedido: 16 Fevereiro 2012.

[11] Viper – SmartStart. Disponível em www.viper.com/SmartStart. Acedido: 28 Fevereiro 2012.

[12] Informação retirada do site http://www.inosat.pt. Acedido: 16 Fevereiro 2012.

[13] Informação retirada do site www.viper.com/SmartStart. Acedido: 28 Fevereiro 2012.

[14] <uses-sdk> | Android Developers:. Disponível em http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels. Acedido: 2 Outubro 2012.

[15] PackageManager | Android Developers: Disponível em http://developer.android.com/reference/android/content/pm/PackageManager.html#FEATURE_TELEVISION. Acedido: 2 Outubro 2012.

[16] Android Developer Tools | Android Developers: Disponível em http://developer.android.com/tools/help/adt.html#. Acedido: 3 Outubro 2012.

[17] Computador usado para desenvolvimento da aplicação: LG S1 Express Dual, processador Intel Centrino Duo T2400 1.83GHz, memória RAM 2,5GB, Windows 7 Professional SP1 32 bits.

[18] Telemóvel usado para correr e fazer *debugging* da aplicação: Huawei Ideos X3, CPU 600MHz, Android OS v2.3.3 (Gingerbread), display 3.2in (320 x 480 pixels), com A-GPS.

[19] App Framework | Android Developers: Disponível em http://developer.android.com/about/versions/index.html. Acedido: 3 Outubro 2012.

[20] Application Fundamentals | Android Developers: Disponível em http://developer.android.com/guide/components/fundamentals.html. Acedido: 8 Outubro 2012.

[21] Activities | Android Developers: Disponível em http://developer.android.com/guide/components/activities.html. Acedido: 8 Outubro 2012.

[22] Services | Android Developers: Disponível em http://developer.android.com/guide/components/services.html. Acedido: 8 Outubro 2012.

[23] PIC32 Ethernet Starter Kit. Disponível em http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2615&dDocName=en545713. Acedido 28 Novembro 2012.

[24] 32-bit PIC® Microcontrollers - Overview | Microchip Technology Inc. Disponível em http://www.microchip.com/pagehandler/en-us/family/32bit. Acedido: 28 Novembro 2012.

[25] Machine-to-Machine (M2M) PICtail Daughter Board. Disponível em http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en553141. Acedido: 28 Novembro 2012.

[26] 3GPP TS 27.007 - Technical Specification Group Core Network and Terminals; AT command set for User Equipment (UE).

[27] 3GPP TS 27.005 - Technical Specification Group Terminals; Use of Data Terminal Equipment - Data Circuit terminating Equipment (DTE-DCE) interface for Short Message Services (SMS) and Cell Broadcast Service (CBS).

[28] 3GPP TS 27.010 V3.4.0 - Terminal Equipment to User Equipment (TE-UE) multiplexer protocol (Release 1999).

[29] LEON-G100, G200 GSM/GPRS 2.5G modules. Disponível em http://www.u-blox.com/en/wireless-modules/gsm-gprs-modules/leon-gsm-module-family.html. Acedido: 28 Novembro 2012.

[30] NEO-6 series. Disponível em http://www.u-blox.com/en/gps-modules/pvt-modules/neo-6-family.html. Acedido: 28 Novembro 2012.

[31] MPLABX | Microchip Technology Inc. Disponível em http://www.microchip.com/pagehandler/en-us/family/mplabx. Acedido: 29 Novembro 2012.

[32] 3GPP TS 23.040 - Technical realization of Short Message Service (SMS).

[33] 3GPP TS 23.041 - Technical realization of Cell Broadcast Service (CBS).

Anexos

Anexo A – Recursos

Quanto aos recursos utilizados para criar a aplicação, tem-se as imagens, os sons, as palavras ou frases (*strings*), os *layouts* e os *menus*. Estes dois últimos, já foram escortinados na secção 4.3. Quanto aos outros, serão descritos nesta secção.

Quanto às **imagens**, existem três pastas, onde cada uma tem como função guardar as imagens com a respetiva densidade de ecrã. São elas *drawable-hdpi*, *drawable-mdpi* e *drawable-ldpi*, que dizem respeito às densidades elevada, média e baixa, respetivamente. Cabe ao sistema escolher a que melhor se enquadra com o dispositivo.

Assim, foram criadas imagens, sempre com as três densidades, com o apoio de uma ferramenta, referente ao *site* http://android-ui-utils.googlecode.com/hg/asset-studio/dist/index.html, onde este gera as imagens, usando uma qualquer imagem com uma qualquer resolução ou densidade.

Para uma melhor compreensão, devido a serem muitas imagens e ficheiros relacionados com as imagens, teve-se em conta o nome das imagens relativas a cada função. Para as imagens dos botões usou-se o prefixo "*ic_button_*", para os *menus* o prefixo "*ic_menu_*" e para as abas o prefixo "*ic_tab_*". Existem ainda algumas imagens que não se inserem nestes três grupos e têm o seu nome próprio.

Depois, relativamente aos botões, estes usam duas imagens diferentes para ambos os estados, quando pressionado ou quando não pressionado. Usou-se então o sufixo "*_normal*" e "*_pressed*", respetivamente. O mesmo acontece com as imagens das abas que podem ter dois estados, selecionado ou não selecionado. Usou-se por isso o sufixo "*_selected*" e "*_unselected*", respetivamente.

Existe ainda um ficheiro (*.xml*) relacionado a cada botão ou *item* de aba, que tem como objetivo mostrar ao sistema qual a imagem a usar em cada estado do botão ou item de aba.

Será aqui apresentado apenas um, estando os outros subentendidos de usarem o mesmo código mas para as respetivas imagens.

Por exemplo, o ficheiro *ic_button_lock.xml* usa o objeto <*selector*> que contém dois objetos <*item*>. Cada objeto <*item*> diz respeito a uma imagem e usa o campo *drawable* para definir qual imagem a usar e o campo *state_pressed* para confirmar qual a imagem que corresponde ao estado pressionado. Segue-se o exemplo do código usado no ficheiro *ic_button_lock.xml*:

```
<selector
   <item android:drawable="@drawable/ic_button_lock_pressed"
         android:state_pressed="true" />
   <item android:drawable="@drawable/ic_button_lock_normal" />
</selector>
```

Serão apresentadas de seguida, as imagens referentes apenas à densidade mais elevada, ficando as outras subentendidas como "iguais".

Inicialmente tem-se as imagens referentes às abas (*tab1* e *tab2*).

Figura A.1 - Imagem usada nas abas (*ic_tab_home_unselected.png*, *ic_tab_home_selected.png*).

Figura A.2 - Imagem usada nas abas (*ic_tab_car_unselected.png*, *ic_tab_car_selected.png*).

Figura A.3 - Imagem usada nas abas (*ic_tab_track_unselected.png*, *ic_tab_track_selected.png*).

Figura A.4 - Imagem usada nas abas (*ic_tab_specs_unselected.png*, *ic_tab_specs_selected.png*).

Figura A.5 - Imagem usada nas abas (*ic_tab_alerts_unselected.png*, *ic_tab_alerts_selected.png*).

Figura A.6 - Imagem usada nas abas (*ic_tab_config_unselected.png*, *ic_tab_config_selected.png*).

Figura A.7 - Imagem usada nas abas (*ic_tab_security_unselected.png*, *ic_tab_security_selected.png*).

Figura A.8 - Imagem usada nas abas (*ic_tab_about_unselected.png*, *ic_tab_about_selected.png*).

Para cada grupo de imagens existe um ficheiro com os nomes *ic_tab_home.xml*, *ic_tab_car.xml*, *ic_tab_track.xml*, *ic_tab_specs.xml*, *ic_tab_alerts.xml*, *ic_tab_config.xml*, *ic_tab_security.xml* e *ic_tab_about.xml*, respetivamente.

De seguida, tem-se as imagens correspondentes aos itens dos *menus*.

Figura A.9 – Imagem *ic_menu_config.png*.

Figura A.10 - Imagem *ic_menu_car.png*.

Figura A.11 - Imagem *ic_menu_exit.png*.

Figura A.12 - Imagem *ic_menu_home.png*.

Figura A.13 - Imagem *ic_menu_newcar.png*.

Figura A.14 - Imagem *ic_menu_savepos.png*.

Figura A.15 - Imagem *ic_menu_deletepos.png*.

Figura A.16 - Imagem *ic_menu_layers.png*.

De notar que as imagens, à exceção das da Figura A.2, foram retiradas do *site* http://www.iconfinder.com, mencionado em 3.3.

Seguem-se as imagens dos botões, usados no *layout home.xml*.

Inicialmente foram criadas duas imagens semelhantes, que representam os estados normal e pressionado, como pode ser visto na Figura A.17.

Figura A.17 - Imagem original usada nos botões do *layout home.xml*. (normal à esquerda, pressionado à direita)

Depois foram usadas as imagens presentes na Figura A.18, para diferenciar os vários botões.

Figura A.18 - *Icons* para os diferentes botões do *layout home.xml*.

O resultado é o seguinte:

Figura A.19 – Imagem dos *botões ic_button_lock_normal.png* e *ic_button_lock_pressed.png*.

Figura A.20 - Imagem dos botões *ic_button_unlock_normal.png* e *ic_button_unlock_pressed.png*.

Figura A.21 - Imagem dos botões *ic_button_panic_normal.png* e *ic_button_panic_pressed.png*.

Figura A.22 - Imagem dos botões *ic_button_alarmon_normal.png* e *ic_button_alarmon_pressed.png*.

Figura A.23 - Imagem dos botões *ic_button_alarmoff_normal.png* e *ic_button_alarmoff_pressed.png*.

De notar, que os *icons* usados nos botões no estado pressionado, são ligeiramente mais pequenos que os *icons* usados no estado normal. Isto para dar um efeito de profundidade ao pressionar o botão.

Existem ainda os ficheiros que associam as diferentes imagens aos diferentes estados de cada botão. São eles *ic_button_lock.xml*, *ic_button_unlock.xml*, *ic_button_panic.xml*, *ic_button_alarmon.xml* e *ic_button_alarmoff.xml*.

Para terminar os botões, são apresentados de seguida os botões usados no *layout car.xml*.

Figura A.24 - *ic_button_engineoff_normal.png* e *ic_button_engineoff_pressed.png*.

Figura A.25 - *ic_button_engineon_normal.png* e *ic_button_engineon_pressed.png*.

Figura A.26 - *ic_button_camera_normal.png* e *ic_button_camera_pressed.png*.

Figura A.27 - *ic_button_windowup_normal.png* e *ic_button_windowup_pressed.png*.

Figura A.28 - *ic_button_windowdown_normal.png* e *ic_button_windowdown_pressed.png*.

Figura A.29 - *ic_button_trunk_normal.png* e *ic_button_trunk_pressed.png*.

Para estes botões foram também usados ficheiros para associar as diferentes imagens aos diferentes estados. São eles *ic_button_engineoff.xml*, *ic_button_engineon.xml*, *ic_button_camera.xml*, *ic_button_windowup.xml*, *ic_button_windowdown.xml* e *ic_button_trunk.xml*.

Por fim, as imagens que não fazem parte dos três grupos mencionados anteriormente.

Figura A.30 - Imagens usadas para marcar a posição do dispositivo (*ic_marker_blue.png*) e da viatura (*ic_marker_red.png*).

Na Figura A.30 estão representadas as imagens criadas, para serem usadas para marcar as posições do dispositivo e da viatura, no ecrã *Track* da aba principal da aplicação.

Figura A.31 - Imagem usada no *layout confabout.xml*. (*ic_ua.png*)

Figura A.32 – Imagem usada no *layout confabout.xml*. (*ic_android.png*)

Depois, na Figura A.31 e Figura A.32 estão representadas as imagens que são usadas no *layout confabout.xml*. A Figura A.31 diz respeito ao *icon* da Universidade de Aveiro, com o nome *ic_ua.png* e a Figura A.32 diz respeito ao *icon* do *Android*, com o nome *ic_android.png*.

Seguem-se as imagens criadas para serem usadas como fundos dos *layouts*. A Figura A.33 tem o nome *screen_car2.png* e diz respeito ao fundo do *layout car.xml*. A Figura A.34 é usada no fundo do *layout home.xml* e tem o nome *screen_leather.png*.

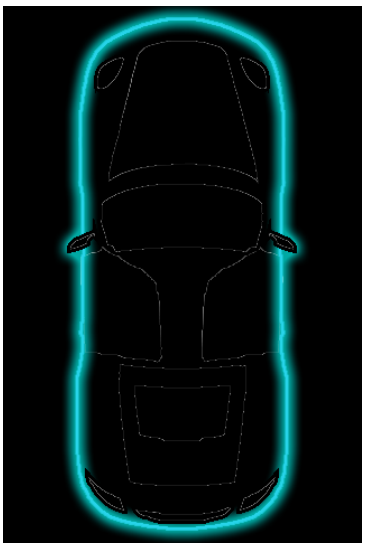

Figura A.33 - Imagem usada como fundo do *layout car.xml*. (*screen_car2.png*)

Figura A.34 - Imagem usada como fundo do *layout home.xml*. (*screen_leather.png*)

Segue-se a imagem criada para ser o símbolo da aplicação. Aqui foi tido em conta uma imagem que possa descrever a aplicação. Para isso usou-se uma roda de um automóvel, com

uma agulha de bússola e os pontos cardiais nos raios da jante. Isto para demonstrar que a aplicação, além de outras funcionalidades, oferece a opção de rastreamento de viaturas.

Foi também inserido o nome da aplicação, no pneu, para dar a ideia que o nome está gravado no próprio pneu, usado para a imagem.

Figura A.35 - Símbolo da aplicação. (ic_cdl.png)

Por fim, ainda nos recursos de *drawable*, foi criado um ficheiro que diz respeito ao aspeto gráfico dos *layouts* de *Dialog* usados pela aplicação. O ficheiro tem o nome *shape_alertdialog.xml* e usa o objeto <*shape*> que contém os objetos <*padding*> e <*gradient*>. O objeto *shape* possibilita criar a forma pretendida para um *layout*, com o auxílio dos objetos *padding* e *gradient*. Este ficheiro é usado como imagem de fundo, quando são criados os *layouts* que dizem respeito aos *Dialog*.

Relativamente aos **sons** usados pela aplicação, estes são guardados na pasta *raw*, que tem como função, guardar os ficheiros de áudio ou vídeo usados numa aplicação.

Os ficheiros usados são cinco e todos eles foram retirados do *site* http://soundjax.com, como referido em 3.3. Quatro dos sons, dizem respeito ao som possível de ser usado para ser notificado, aquando da chegada de um alerta. São eles *alarm1.mp3*, *alarm2.mp3*, *alarm3.mp3* e *alarm4.mp3*. Existe ainda outro som, com o nome *setalarmon.mp3*, que é usado para notificar o utilizador, quando for pressionado um dos botões do ecrã *Home*.

Por fim, existem as pastas *values* e *values-pt*, que contêm um ficheiro com o nome ***strings.xml***. As duas pastas dizem respeito ao idioma das palavras usadas na aplicação. Todas as palavras, vistas pelo utilizador, estão armazenadas neste ficheiro. Assim, torna-se simples usar vários idiomas por uma aplicação, uma vez que basta criar uma nova pasta e um novo ficheiro *strings.xml* e traduzir todas as palavras para o idioma pretendido.

Existe um ficheiro diferente em cada pasta, com o mesmo nome e serve para associar uma cadeia de carateres a uma palavra ou frase. Segue-se um exemplo:

```
<string name="Locked_Home">Locked</string>
<string name="Locked_Home">Trancado</string>
```

A primeira linha de código diz respeito ao ficheiro *strings.xml* usado na pasta *values* e a segunda diz respeito ao ficheiro *strings.xml* usado na pasta *values-pt*. Vê-se assim, que ao usar a cadeia de carateres *"Locked_Home"* num qualquer ficheiro de código da aplicação, será apresentado ao utilizador a palavra *Locked* ou Trancado, consoante o idioma que este pretenda. Cabe ao sistema usar o ficheiro da pasta correspondente ao idioma selecionado.