Lisandro Sérgio Gomes Lopes

Datalogger Remoto – Monitorização e controlo de uma Bomba de Calor

Remote Datalogger – Heat Pump Control and Monitoring

"The data are no longer in the computers, we have come to see that the computers are in the data"

- Peter Lucas, Joe Ballay e Mickey McManus

Lisandro Sérgio Gomes Lopes

DataLogger Remoto – Monitorização e controlo de uma Bomba de Calor

Remote Datalogger – Heat Pump Control and Monitoring

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica de Telmo Cunha, Professor Associado/Investigador da Universidade de Aveiro.

Dissertation to be presented to the University of Aveiro to fulfil the necessary requirements to obtain the Master's Degree in Electronics and Telecommunications Engineering, under the scientific/technical guidance of Telmo Cunha, Assistant Professor / Researcher at Aveiro University.

O Júri

Presidente / President	**Alexandre Manuel Moutela Nunes da Mota** Professor Associado da Universidade de Aveiro
Vogal / Arguente Principal	**Paulo José Lopes Machado Portugal** Professor Auxiliar, Universidade do Porto - Faculdade de Engenharia
Vogal / Orientador	**Telmo Reis Cunha** Professor Auxiliar/ Investigador da Universidade de Aveiro

Agradecimentos

Começar por agradecer claro à minha família, em particular à minha mãe, pai e irmão, pela força, empenho e dedicação que mostraram e que incutiram em mim; por me terem dado todas as condições para que eu pudesse terminar o meu percurso académico rapidamente; por estarem sempre ao meu lado, nos bons e maus momentos. Tudo isto é graças a vocês, muito obrigado.

Aos amigos com quem partilhei casa no último ano da universidade e a todos os outros tantos que fiz durante estes 5 anos como estudante desta grande instituição de ensino, a Universidade de Aveiro, que, num momento ou noutro e por uma razão ou por outra, contribuíram certamente para que eu chegasse aonde estou hoje.

Em especial, um grande obrigado aos amigos que me acompanharam na aventura de trabalhar na Alemanha: vocês tornaram esta transição muito mais suave.

Um grande obrigado também ao líder da minha equipa na HP, Angelino, por me ter dado grande flexibilidade e autonomia relativamente às horas de trabalho, para que tivesse mais tempo útil para trabalhar neste documento.

Por fim, ao meu orientador, Telmo Cunha. Sempre que foi necessário pude contar com o seu conhecimento e experiência, seja ao nível do assunto aqui abordado, como também pela oportunidade de poder aprender com a grande experiencia profissional que possui e que faz transparecer com muita naturalidade.

Resumo

Um datalogger é um dispositivo capaz de recolher dados relativos a acontecimentos que surgem em seu redor, através da comunicação com sensores, estejam estes a operar de forma *standalone* ou sejam estes parte integrante de um sistema mais complexo. No caso em estudo, desenvolveu-se um datalogger dedicado à monitorização de uma bomba de calor da Bosch, equipamento responsável por transferir energia de uma fonte fria para uma fonte quente, contra o gradiente de temperatura. O objetivo final deste datalogger é integrar um largo espectro de dispositivos da Bosch, que utilizam um protocolo de comunicação proprietário. Todo o processo de monitorização é feito através da análise de informações recolhidas por uma ECU (*Electronic Control Unit*), elemento constituinte da bomba de calor.

Adicionalmente, a implementação deste datalogger teve que respeitar certas condições chave, apresentando como características imperativas a versatilidade, o baixo consumo energético, a capacidade de ligação à internet, capacidade de armazenamento de dados, a configuração remota e, a de se basear numa plataforma open-source.

Numa primeira fase, perante a ausência de equipamentos para teste, implementou-se um emulador em MATLAB capaz de simular os dados operacionais da bomba de calor.

Após a receção do material, o projeto do datalogger evoluiu e inúmeras funcionalidades foram adicionadas: A utilização da plataforma Xively ofereceu a possibilidade de controlo e monitorização remotos. Através desta, o controlo de alguns parâmetros do datalogger, bem como a consulta de informação relativa à bomba de calor, ficaram disponíveis remotamente.

palavras-chave DataLogger, Comunicação Machine to Machine (M2M), Micro-controlador, Arduino, Comunicação Série, emulador, Protocolo de Comunicação da Bosch, plataforma Xively.

keywords

Datalogger, Machine to Machine (M2M) *Communication, Microcontroller,* Arduino, *Serial Communication, Emulator,* Bosch *Communication Protocol,* Xively *platform*.

abstract

A datalogger is a device capable of gathering event related data from activities that take place in its surroundings, throughout communication with sensor units, operating as standalone devices or integrating part of a more complex system. This document is refered to the implementation of a datalogger that is capable of monitoring and control a heat pump system from Bosch, which is the equipment responsible for transferring energy from a cold source to a hot source, against the temperature gradient. This datalogger is able to integrate a wide range of different devices from Bosch which use a particular and proprietary communication protocol. The general monitoring process is achieved by analizing the operational data related to the heat pump, retrieved by an ECU (*Electronic Control Unit*), a constitutive element of the heat pump.

Additionally, this datalogger had to respect certain key conditions, having as main features the versatility, the ability to connect to the internet wirelessly, remote configuration, its low consumption, its storage capability and by having as base of development, an open-source platform.

In the early stages of this project, before the impossibility to access a real ECU, a MATLAB emulator was created in order to simulate the behavior of a heat pump.

After having received the real ECU, the project evolved and several functionalities were added: The Xively platform offered the possibility of remote monitoring and control of the heat pump. Through it, it was possible to remotely change specific datalogger parameters and to verifying information regarding the ECU.

Índice

Lista de Figuras

HDMI

High-Definition Multimedia Interface

I/O

Input/Output

ICSP

In Circuit Serial Programming

IDE

Integrated Development Environment

IoT

Internet Of Things

LAN

Local Area Network

LED

Light Emitting Diode

M2M

Machine to Machine Communication

MISO

Master In, Slave Out

MOSI

Master Out, Slave In

PSTN

Public switched telephone network

PWM

Pulse-width modulation

R&D

Research and Development

RTS

Ready to send Signal

RTU

Remote Terminal Unit

SD

Secure Digital Card

SDHC

Secure Digital High Capacity

SNR

Signal to Noise Ratio

SPI

Serial Peripheral Interface

SS

Shield Select

TCP/IP

Transmission Control Protocol/ Internet Protocol

TTL

Transistor–Transistor Logic

UPD

User Datagram Protocol

USART

Universal asynchronous receiver/ transmitter

USB

Universal Serial Bus

WiFi

Wireless Fidelity

Capítulo I

1. Introdução

Os dataloggers representam uma ferramenta fundamental para monitorizar todo o tipo de equipamentos, sejam eles industriais ou de uso doméstico. Com a constante renovação e aparecimento de novos equipamentos, muitas vezes operando em zonas de difícil acesso, foi imperativo que os Dataloggers acompanhassem esta evolução. *A priori*, a maior parte do registo de dados era feita de maneira manual ou através de Registadores[1]. Porém, o aumento do número de eventos disponíveis para monitorização/controlo em simultâneo (através do aparecimento de equipamentos com capacidade de analisar vários sensores ao mesmo tempo) e o surgimento do conceito de comunicação *Machine-to-Machine* contribuíram para o elevado desenvolvimento e para a imensa diversidade aplicacional destes sistemas de *data logging*.

Este trabalho incide sobre o desenvolvimento e implementação de um sistema de receção e análise de dados provenientes de sensores que constituem parte integrante de uma bomba de calor. Os dados processados são depois disponibilizados ao utilizador através de um *webserver* e/ou através de um programa de PC onde é possível visualizar a informação que flui pela porta série que estabelece a ligação entre o PC e este sistema. Para além da visualização, estes dois métodos permitem também o controlo do *datalogger*, através de pedidos de informação ou de mudança de parâmetros em tempo real.

A bomba de calor a monitorizar pelo datalogger possui sensores que disponibilizam, através de um protocolo de comunicação série, dados imperativos relativos à operação do equipamento, nomeadamente temperatura da água, pressão, etc. Estes dados provêm de uma interface denominada ECU, uma unidade eletrónica que controla uma série de sensores e que garante o correto funcionamento do equipamento ao qual está ligado. Em seguida, apresenta-se um breve enquadramento sobre o trabalho

[1] Em inglês, *Strip Chart Recorder* – Os primeiros dispositivos com esta nomenclatura eram dispositivos eletromecânicos que imprimiam *inputs* elétricos ou mecânicos numa folha de papel.

realizado, seguido dos objetivos pretendidos. A estrutura geral do documento finda este capítulo.

1.1. Enquadramento

Um *datalogger* é um equipamento eletrónico, computorizado que analisa e grava informações do ambiente onde está inserido. Com a evolução da tecnologia, os *dataloggers* foram se divergindo no que toca à sua aplicabilidade. Deste modo, as principais divergências entre estes são baseadas na forma em que os dados são recolhidos e onde são armazenados (*Cloud* ou localmente, por exemplo). Tipicamente são dispositivos pequenos, dependentes de baterias, estão equipados com um microprocessador, possuem capacidade de armazenamento e estão ligados a um ou mais sensores ou a dispositivos que contêm sensores.

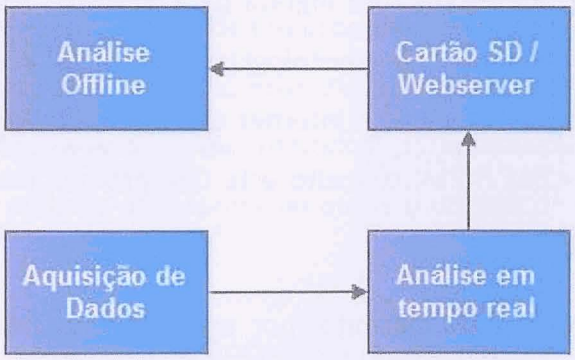
Este trabalho enquadra-se na "*Internet Of Things*" (IoT), ou usando termos mais intuitivos ou mais facilmente entendíveis e focando uma das áreas da IoT, refere-se a Comunicação M2M, nomenculatura inglesa para *Machine to Machine Communication*. Este termo surgiu quando três tecnologias que já se encontravam enraizadas na sociedade - os sensores *wireless*, a Internet e os computadores pessoais - se juntaram para criar comunicações M2M, conceito este que prometeu e cumpriu: revolucionou o mundo da telemetria.

A comunicação M2M permite, por exemplo, monitorizar de uma forma mais eficiente as condições de infraestruturas públicas, tais como estações de tratamento de água ou pontes, contribuindo para a redução da necessidade de intervenção humana. No mundo industrial, integra funções de realizações de inventário ou no auxílio de investigadores. Por outro lado, por se correlacionar facilmente com qualquer outra máquina (por exemplo: se utilizarem leitura de códigos de barras – tecnologia já existente há muito tempo), poderá também integrar funções domésticas, tais como intervir na criação de uma lista de compras num simples toque, na manutenção de jardins ou na limpeza de interiores: Em síntese, o conceito de M2M, apesar de presente na nossa

sociedade faz já alguns anos, só recentemente é que a indústria e a ciência começaram a apostar fortemente no seu desenvolvimento e na investigação nesta área, podendo-se dizer ainda que é um processo que continuará a expandir o papel da telemetria para além do seu uso na ciência, engenharia, aumentando a sua utilidade no nosso quotidiano.

Com a crescente expansão dos sistemas digitais no mundo tecnológico, são várias as aplicações onde é necessário armazenar dados provenientes de equipamentos eletrónicos. Os dataloggers podem ser configurados com o fim de obter pressão de gás, monitorizar quantidades presentes em tanques, na monitorização de transporte de equipamento, em estudos de qualidade, e na obtenção dos mais diversos dados ambientais, sejam eles características do solo, da água, da humidade, da radiação solar, etc.

O seu modo de operação é, conceptualmente, transparente. A figura 1 representa o diagrama de blocos de um datalogger básico.



1 - Diagrama de Blocos de um Datalogger

Durante a aquisição de dados, ocorre a conversão de sinais físicos em sinais digitais. Com a análise em tempo real, os valores digitais irão ser processados de modo a representarem as grandezas fixas lidas. Nesta fase, o microcontrolador desempenha um papel de extrema importância: sobre ele recai todo o processamento da informação

retirada dos sensores para algo regível pelo utilizador. A análise *offline* implica algum processamento que é feito depois de se efetuar o armazenamento dos dados.

Neste trabalho, iremos analisar uma das aplicações que utilizam comunicações M2M: os *Dataloggers*. Este trabalho surgiu da necessidade por parte da Bosch Termotecnologia em monitorizar, de uma forma eficiente, diversos equipamentos que utilizam protocolos de comunicação proprietários muito específicos, nomeadamente as suas bombas de calor. Para esse efeito, começou-se por efetuar o levantamento de todo o material necessário à sua montagem, tal como o microcontrolador e todos os módulos adicionais para dotar o datalogger de características fundamentais (como a conectividade sem fios).

Outra característica igualmente importante a referir é a modularidade: os *dataloggers* têm que ser facilmente expansíveis e devem ser facilmente configuráveis; o seu desenvolvimento deverá ter em consideração a fiabilidade, já que são construídos para operar de forma contínua e ininterrupta, mesmo em ambientes com variáveis externas que poderão interferir com o seu funcionamento. Por fim, a destacar a facilidade na sua utilização: se estes comunicam de forma lógica, a interpretação do seu modo de funcionamento torna-se mais fácil, e, por conseguinte, operar e expandir a sua aplicabilidade é, consequentemente, mais simples.

1.2. Objetivos

No sentido de monitorizar o funcionamento das bombas de calor da Bosch, especialmente quando estas se encontram em locais remotos ou de difícil acesso, o trabalho aqui apresentado considera a implementação de um sistema versátil e de fácil customização, composto por uma unidade de processamento baseada em plataforma *open-source* e que toma como necessidades essenciais ou primárias, a capacidade de armazenar dados em memória não volátil (Cartão SD) e a de se ligar a módulos de

comunicações, mais concretamente, a um módulo *wireless* com a tecnologia *Wi*-Fi implementada. Assim, possibilita o acesso a informações relativas ao funcionamento da bomba de calor, sem requerer uma avaliação presencial in loco, usufruindo das facilidades vulgarmente instaladas nos locais onde se situam as caldeiras (tipicamente, residências domésticas, muitas delas dispondo de ligação à internet). Deste modo, possibilita o estabelecimento de uma ligação de dados remota com um computador, tendo este a capacidade de controlar o dispositivo (datalogger, bomba de calor, ou mesmo outros equipamentos) e descarregar os dados armazenados remotamente. Como características secundárias, este dispositivo também apresenta um baixo consumo (ordem dos 5W) e possui, para além da capacidade de comunicação série com o equipamento a monitorizar, portas digitais que permitem analisar ou controlar sensores adicionais (1 *wire*). Esta dissertação resulta de uma colaboração entre o DETI-UA e a Bosch Termotecnologia, S.A.

1.3. Estrutura Geral da Tese

Este trabalho subdividiu-se em várias tarefas ao longo do ano letivo. Numa primeira fase, após se recolher a informação sobre as necessidades específicas da Bosch, efetuou-se o levantamento do estado da arte, de modo a avaliar todos os componentes que iriam integrar este projeto, assim como para analisar os dispositivos existentes no mercado relacionados com a aplicação anteriormente descrita. Após um período de pesquisa e de leitura exaustiva sobre o assunto, chegou a fase de aquisição de material e da elaboração do *sketch*[2]. Seguidamente efetuou-se a implementação e teste do datalogger projetado.

Este documento irá começar então por discutir e analisar o mercado atual dos dataloggers e componentes com alguma relevância para o trabalho em causa: começa

[2] Sketch – Nome dado a um script com extensão .ino, compatível com o ambiente de desenvolvimento Arduíno, baseado no software *Processing*

por tratar a evolução temporal da *Internet Of Things* e posteriormente, retrata as unidades comerciais disponíveis no mercado, inferindo sobre a necessidade de desenvolver este projeto; de seguida, critica as diversas opções ao nível da unidade de processamento que poderiam integrar este projeto.

Numa segunda fase será exposta a forma como todos os módulos que integram o datalogger estão interligados, que funcionalidades foram implementadas, tanto ao nível do datalogger, mas também ao nível da criação de um emulador para testes, e de que forma foi utilizado o serviço *Xively*. Para além disso, discutir-se-ão as técnicas utilizadas para ultrapassar certos problemas que foram surgindo pelo percurso, expondo, numa fase final, alguns testes do equipamento de data logging no campo.

Capítulo II

2. Soluções de Registo Remoto de Dados e Principais Componentes

Este capítulo irá cobrir todo o estudo realizado sobre as tecnologias envolvidas no conceito de *data logging*. Num nível mais abstrato, começa por analisar a evolução tecnológica desde os primórdios da Telemetria, passando depois pelas comunicações M2M, e findando no tema bem atual da IoT, a *Internet of Intelligent Objects*. Posteriormente, irão ser referidas e analisadas algumas soluções de *data logging* existentes no mercado. Por fim, perante o desafio que deu origem a este projeto, trata por efetuar uma comparação de duas das mais conceituadas unidades de processamento open-source existentes no mercado – Arduino e Raspberry Pi – que serviriam como constituinte base de um datalogger open-source.

2.1. Telemetria, M2M e a Internet Of Things

A Telemetria é a ciência de medição e comunicação de informações à distância, seja através de condutores, ondas rádio ou por satélite. Um dos primeiros circuitos de transmissão de dados foi desenvolvido em 1845, entre um dos palácios do Imperador Russo e a sede do exército. Num uso mais avançado, em 1874 foi instalado um sistema de sensores de temperatura e profundidade da neve em *Mont Blanc* que transmitia informação em tempo-real para Paris. Em 1912, linhas telefónicas transmitiam dados operacionais de uma central elétrica para um escritório (sistemas de supervisão)[1]. Estes exemplos aqui enunciados evidenciam a larga aplicabilidade dos sistemas telemétricos.

Os sensores presentes em redes telemétricas eram altamente especializados e requeriam fontes de transmissão de ondas rádio poderosas; para além disso, conforme a localização destes emissores (por exemplo, em locais com pouca receção de ondas rádio,

"*dead spots*"), a SNR (*Signal to Noise Ratio*) poderia ser gravemente afetada, e os dados poderiam ser transmitidos com erros. Atualmente existem técnicas avançadas de reconstrução de informação com elevada precisão (impulsionadas pelo aparecimento de processadores rápidos e ágeis).

Quase dois séculos depois, o conceito de telemetria foi amplamente explorado. Atualmente, os sistemas telemétricos utilizam redes *wireless* para transmissão de dados de estações aeroespaciais, são usados para o estudo da vida selvagem, na robótica, na domótica, etc.: até o GPS e a videoconferência poderão ser incluídos na área da telemetria.

Com os avanços na telemetria e na tecnologia em geral, introduziu-se, nos dias de hoje, um novo tipo de comunicação, a M2M, usada para referir a troca de informações automatizada entre máquinas. De notar que o termo *Machine* também pode referir-se a máquinas virtuais, como por exemplo aplicações de *software*. Definir em concreto M2M não é trivial: dependendo do contexto no qual é usado, possui várias definições. Por exemplo, no âmbito de redes como o GSM ou CDMA, poderá representar o envio de dados automatizados via SMS/GPRS/PSDN; No contexto de sistemas de rádio como o *Bluetooth* ou *Zigbee*, o M2M aparece como sinónimo de AMR (*Automated Meter Reading)*[2].

A comunicação autónoma entre dispositivos tornou-se, então, de elevada importância trazendo largos benefícios que foram aproveitados pela indústria. Um exemplo óbvio consiste na solução de comunicação com máquinas que são colocadas em locais de difícil acesso. A capacidade de enviar informação remotamente e de uma forma automatizada permite ao responsável pela manutenção da máquina descobrir erros, e até obter informações sobre a mesma e sobre o ambiente que a rodeia, sem a necessidade de deslocar-se até ao local. Isto permite às empresas pouparem esforços ao desenvolverem equipamentos de medição mais simples, concentrando toda a capacidade de processamento da informação no terminal onde irá ser enviado a informação recolhida no terreno. Assim, os recursos das empresas podem incluir um elevado número

de máquinas de medição simples e um número muito reduzido de unidades de processamento poderosas, conduzindo a soluções mais económicas.

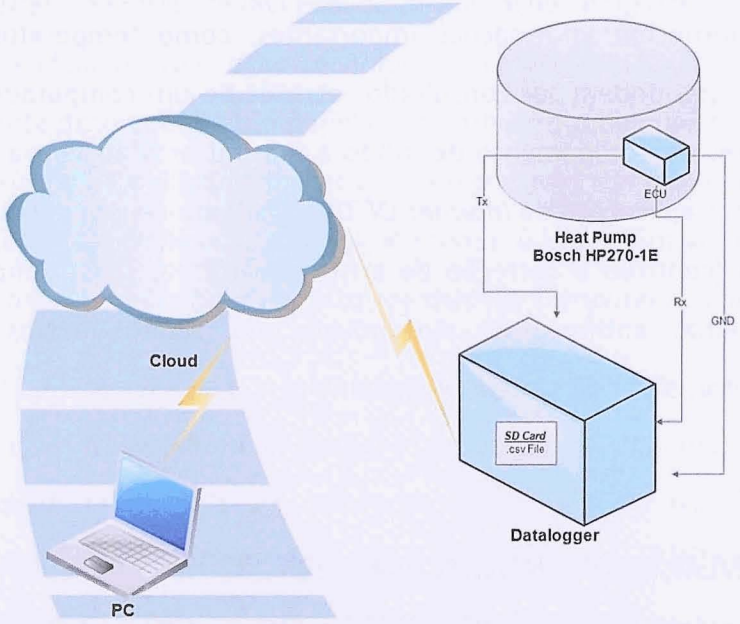
O progresso contínuo na área da Microeletrónica e nas Redes de Comunicações tornou possível a conceção de redes formadas entre objetos inteligentes, bem como a implementação de serviços sobre estes. Esta idealização de sistemas é denominada por *Internet of Things*: uma nova geração de *networked objects* com capacidade de comunicação, capacidade sensorial e de atuação para as mais diversas áreas de aplicação (ver ilustração na figura 2). É expetável que a comunicação entre objetos espalhados pelo mundo fora, que estejam ligados à rede e que possuam boa capacidade de processamento, dê origem a uma revolução ao nível da criação e disponibilização de serviços que mudará certamente a forma de interação com o mundo[2].

2 -Internet of Things - Fonte: [3].

Atualmente, um dos maiores passos na implementação de soluções comerciais M2M na Europa e nos E.U.A. foi realizado através de parcerias entre os principais operadores dos Estados Unidos, incluindo *AT&T, KPN, Rogers, Telcel*, etc., e outra entre a Vodafone, a *Verizon Wireless* e *nPhase*. Com isto, M2M e a sua aplicabilidade na denominada *Internet Of Things* está presente em diversos ramos, desde a segurança e saúde, até à indústria e sistemas de pagamento automático [4].

2.2. *Datalogger Remoto*

Características/Potencialidades fundamentais



4 – Conceptualização do Datalogger Remoto

O datalogger remoto aqui abordado e idealizado na figura 4 teve que seguir as seguintes especificações:

- Possuir capacidade de armazenamento;
- Ligação à internet sem fios;
- Plataforma *open-source;*
- Consumo baixo de energia – alta autonomia;
- Seja possível controlá-lo/configurá-lo remotamente;
- Dimensões do equipamento reduzidas.

As características enunciadas são de fato muito vagas. Estes requisitos, especificados pela Bosch Termotecnologia, permitiram um largo e vasto estudo ao nível das plataformas de base a utilizar.

A capacidade de processamento deste tipo de equipamentos, pequenos, com baixo consumo de energia, advém, normalmente, da utilização de dispositivos pequenos, de fácil customização, denominados microcontroladores ou microprocessadores.

Atualmente, no mercado, existem de facto inúmeras opções relativamente a dataloggers remotos. De seguida, apresentar-se-ão algumas propostas de produtos já existentes.

2.3. Unidades Comerciais de Data Logging

Nos dias que correm são imensas as possibilidades de *data logging* no mercado. Este está a tornar-se muito competitivo e os sistemas estão a evoluir, tornando-se cada vez mais complexos. Neste subcapítulo focou-se a análise em sistemas de *data logging* que possuíssem o maior número de características em comum com as pretendidas para a aplicação da Bosch. Descrevem-se, em seguida, as principais soluções encontradas no mercado.

2.3.1. IntelliLoggerTM

O *IntelliLogger*^{TM} (figura 5) é um sistema *standalone* de *data logging* poderoso que suporta FTP, correio eletrónico, mensagens (SMS), é capaz de servir um *webserver* de dados, e inclui um sistema de alerta na forma de alarme. Deste modo, possibilita o acesso ao mesmo através da *Web*, via LAN ou via Internet.[7]

5 - *IntelliLogger* IL-80 - Sistema Base. Fonte: [7]

Outras características fundamentais incluem uma capacidade de memória interna até 130000+ leituras, operação em modo *standalone* ou ligado à Internet, inclusão de um *socket* para uso de cartão de memória *flash* (CF), etc.

Relativamente à capacidade de comunicação, possui conetividade com periféricos por via USB, RS-232 e *Ethernet*. A estes portos podem ser ligados *modems* (PSTN – Dial Up, ou por Telefone), *gateways* de *WiFi*, módulos de rádio, etc. Possui um vasto leque aplicacional do qual se destaca os seguintes pontos:

- Monitorização remota de equipamento: Além de monitorizar o sistema, pode gravar parâmetros críticos de operação, gerar informação na forma de gráficos, páginas *web* e enviar alertas via correio eletrónico.
- Testes R&D de equipamento *in loco*;
- Monitorização de processos (longos períodos);

É um dispositivo *low-power* que oferece várias formas de alimentação: foto voltagem, bateria e/ou corrente doméstica. Como *software* de comunicação utiliza o HyperWare-II[3], um pacote de software multiuso com diversas funcionalidades.

Apesar da grande flexibilidade e robustez que este apresenta, as suas dimensões representam uma desvantagem marcante. Por acréscimo, este tipo de equipamentos não são programáveis pelo utilizador e não suportam todo o tipo de comunicação série: Suportam somente comunicação série com equipamentos que possuem *ModBus RTU*, um protocolo de comunicação de dados série específico. Então, para o tipo de aplicação que se pretende, possuidora de uma trama de dados particular, este *IntelliLogger* não é

---

[3] De acordo com a LogicBeach™, a HW-II pertence a uma nova geração de *software* icon-based de *data logging*, que inclui, em relação ao seu antecessor, capacidade de gerar gráficos, visualizar data em tempo real, construir páginas web customizáveis, etc.

adequado. Além disso, o preço deste equipamento, para o que se pretende, é consideravelmente elevado: 1170 euros.

Uma solução consideravelmente mais pequena e versátil é o *SerialGhost* da *KeeLog* que se analiza na seguinte subsecção.

2.3.2. *SerialGhost*

6 - SerialGhost Wi-Fi DB-9/DB-25 2GB. Fonte:[8]

O *SerialGhost* (figura 6), desenvolvido pela *Keelog*, é um *serial logger* de dados com uma capacidade de *logging* de 2GB, que pode ser acedido localmente como um *flash drive USB* ou remotamente através de Wireless LAN [8]. Dados bidirecionais presentes no barramento de dados são capturados e guardados na memória *flash* para consulta *a posteriori*. A utilização do *SerialGhost* implica o uso de um terceiro equipamento que envie os comandos à ECU a monitorizar. Quaisquer dados que passem pelo *SerialGhost* são guardados no cartão de memória nele existente, oferecendo ainda a possibilidade de enviar relatórios de dados para um correio eletrónico (via *WiFi*) definido nos ficheiros de configuração. Uma aplicação muito comum desde dispositivo é a de *reverse-engineering*[4].

Este datalogger representa, de facto, uma solução que não cumpre todas as especificações que se pretende, mas ainda assim, representa uma solução válida. Em

---

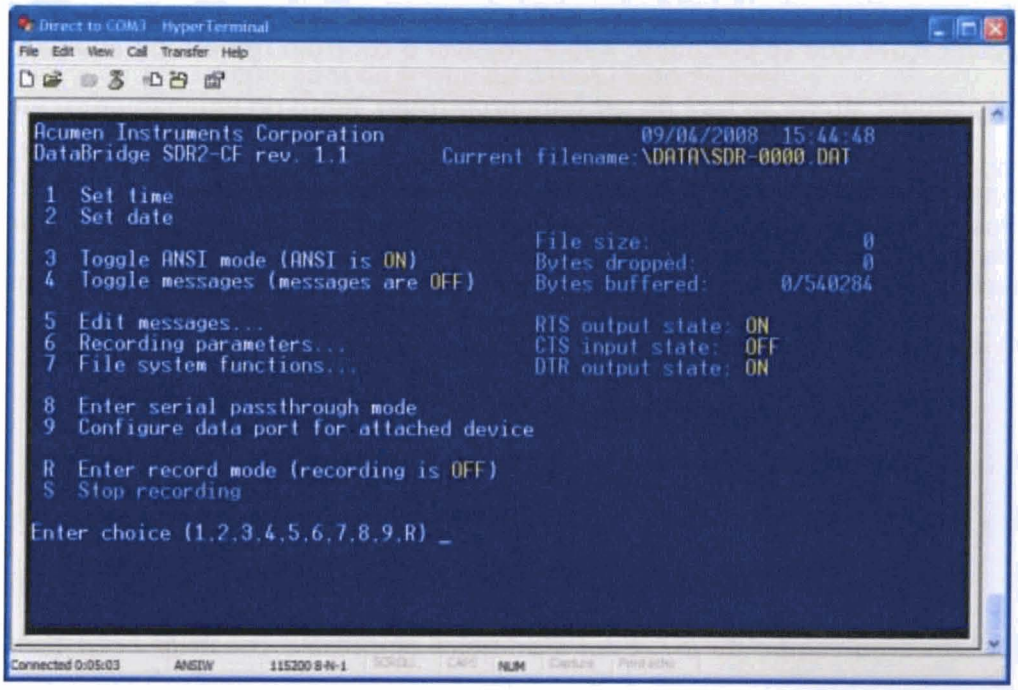
[4] - Reverse-engineering, quando referido ao nível do software, traduz a capacidade que este tem de analisar as estruturas de dados que por ele fluem e a guardá-los para uma análise posterior.

detrimento da utilização de uma plataforma aberta, totalmente customizável, o *SerialGhost* está limitado à gravação dos *bytes* sem qualquer interpretação da informação recolhida. Assim, seria necessária a utilização de um *script* que analisasse e processasse os dados lidos (guardados num ficheiro de texto, .txt) para valores interpretáveis facilmente pelo utilizador. Relativamente aos custos, este *SerialGhost* representaria um investimento de aproximadamente de 116 Euros.

2.3.3. DataBridge™ SDR2-CF

7 - DataBridge SDR2-CF - Serial Data Recorder. Fonte [9];

O *Serial Data Recorder* SDR2-CF[9] (figura 7) é um dispositivo *low-power*, *low-cost*, de pequenas dimensões, que permite a gravação de dados vindos de um periférico ligado por porta série. É uma solução de *hardware* facilmente configurável. Vem equipada por uma porta de configuração que, quando ligada a um computador, pode-se definir alguns parâmetros através do HyperTerminal (ver figura 8).



8 – Menu de Configuração do SDR2-CF (HyperTerminal). Fonte:[9]

Através deste menu é então possível configurar parâmetros de uma forma simples, nomeadamente:

- Configurar hora e dia atual;
- Especificar um diretório e mesmo um nome para o ficheiro onde irão ser guardados os dados a serem gravados;
- Garantir que o dispositivo de armazenamento que estamos a utilizar está corretamente configurado e que possui espaço suficiente para alocar dados.
- Verificar os *baudrates* de ambos os equipamentos envolvidos na comunicação série (o *baudrate* terá de ser garantidamente o mesmo).
- Estabelecer as mensagens de *output*, que neste caso, correspondem às tramas de dados utilizadas para pedir um determinado tipo de resposta à ECU. Estas mensagens são definidas pelo utilizador, e, no *startup* do equipamento, é possível também definir o intervalo de envio de cada mensagem ou seja, a frequência de leitura de dados do equipamento a monitorizar.

Infelizmente, este equipamento também não é capaz de cumprir todas as especificações que se pretende: Não é possível o controlo remoto do equipamento, pelo que qualquer alteração terá de ser feita no terreno, junto do equipamento. Além disso, a incapacidade de este comunicar com a Internet impossibilita também a visualização de dados provenientes da ECU num *webserver*. Outro fator que torna esta opção menos viável é o seu preço e dimensões. Pela consulta do *site* do fabricante, este equipamento tem um custo aproximado de 480€.

2.3.4. H7708/H7710

A Hongdian, fundada em 1997 e localizada na China, foca a sua oferta de produtos em M2M e no campo de *IoT*, e foi a primeira empresa a construir *Data Terminal Units* (DTU) móveis. Atualmente, ocupa uma posição privilegiada no mercado relativamente ao fornecimento de *routers* 3G industriais. Possui várias alternativas com o mesmo objetivo: a monitorização de equipamentos remotamente.

Uma destas alternativas, o H7708 [10], um DTU, fornece o controlo remoto através de uma rede móvel GPRS. Pode ser utilizado nas mais diversas áreas para fornecer canais de comunicação M2M. Este tipo de dispositivos integra a secção da telemetria que se distancia da de *data logging,* mas com áreas de aplicação semelhantes: pertencem ao mundo das RTU (Remote Terminal Units), também denominadas por unidades de telemetria remota. Das características mais relevantes, pode-se destacar:

- Suportes para servidor/cliente TCP/IP, UPD/IP, DDP, SMS, AT.
- Vários modos de operação: *always-online* ou *data-triggered online* (nomenclatura auto descritiva);
- Ferramenta de configuração incluída (Ficheiro instalação PC);
- Configuração remota via PC, SMS e por Comandos AT;
- Suporta GSM/GPRS, RS232-TTL;
- Elevado nível de customização ao nível de informação:

- Definição do protocolo de transmissão (transparente ou não);
- Separador de trama de dados customizável;
- Intervalo de requisição de dados customizável;
- Possui um menu built-in de configuração dos diversos parâmetros;

- Suporta até 4 interfaces de comunicações com centros de recolha de dados em simultâneo (poderão até estar configurados de maneira diferente – interface altamente customizável).

A única limitação deste equipamento é que, na ausência de conectividade à internet, a recolha dos dados torna-se impossível, já que não possui capacidade de armazenamento interno (razão pela qual pertence ao grupo de RTUs e não de dataloggers). O ponto forte deste equipamento é mesmo a sua alta-fidelidade e customização que, por conseguinte, se traduz numa elevada diversidade aplicacional e desempenho.

2.3.5. Conclusão

Pode-se, então, inferir acerca da necessidade de se criar um dispositivo próprio para cumprir com os objetivos delineados inicialmente:

- Por motivos de proteção do próprio equipamento e minimização do overhead de *software*, a Bosch não implementa qualquer protocolo generalista, mas sim apenas o seu próprio protocolo de comunicação proprietário. Desta forma, garante-se total proteção do equipamento e otimização de todos os padrões de performance e segurança.
- A oferta de uma solução do mercado que possua todas as características pretendentes é insuficiente.

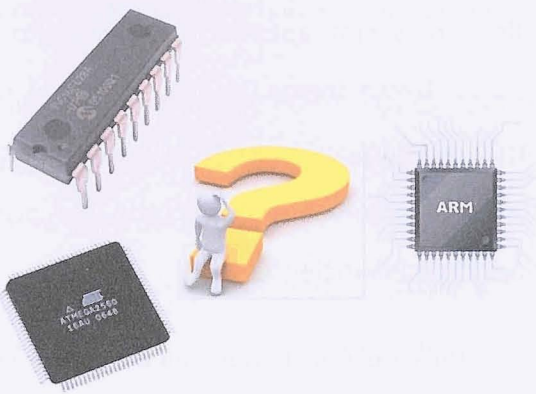
Assim, efetuou-se uma pesquisa sobre qual deveria ser a base do equipamento a ser desenvolvido. Um dos requisitos para o datalogger a implementar foi que se baseasse numa plataforma *open-source*. Estas possuem um largo especto de customização do

software, e permitem a fácil aprendizagem por parte de novos operadores que necessitem de customizar o datalogger. Por acréscimo, adicionar funções ou reprogramar o datalogger a qualquer altura nestas plataformas *open-source* é possível: realizar o mesmo utilizando um dispositivo comercial implica o desenvolvimento de uma nova solução ou uma revisão do mesmo, que acarretará custos acrescidos para o cliente. Assim torna-se evidente a escolha sobre o desenvolvimento de um datalogger de raiz.

Atualmente, a área de plataformas de prototipagem de sistemas é muito debatida. Duas plataformas muito utilizadas para a conceção deste tipo de sistemas são o Arduino e o Raspberry Pi. No entanto, cada um deles pertence a grupos distintos: O Arduino é um microcontrolador e o Raspberry Pi, por outro lado, integra o mundo dos microcomputadores. Segue-se então uma análise generalizada sobre estes dois mundos.

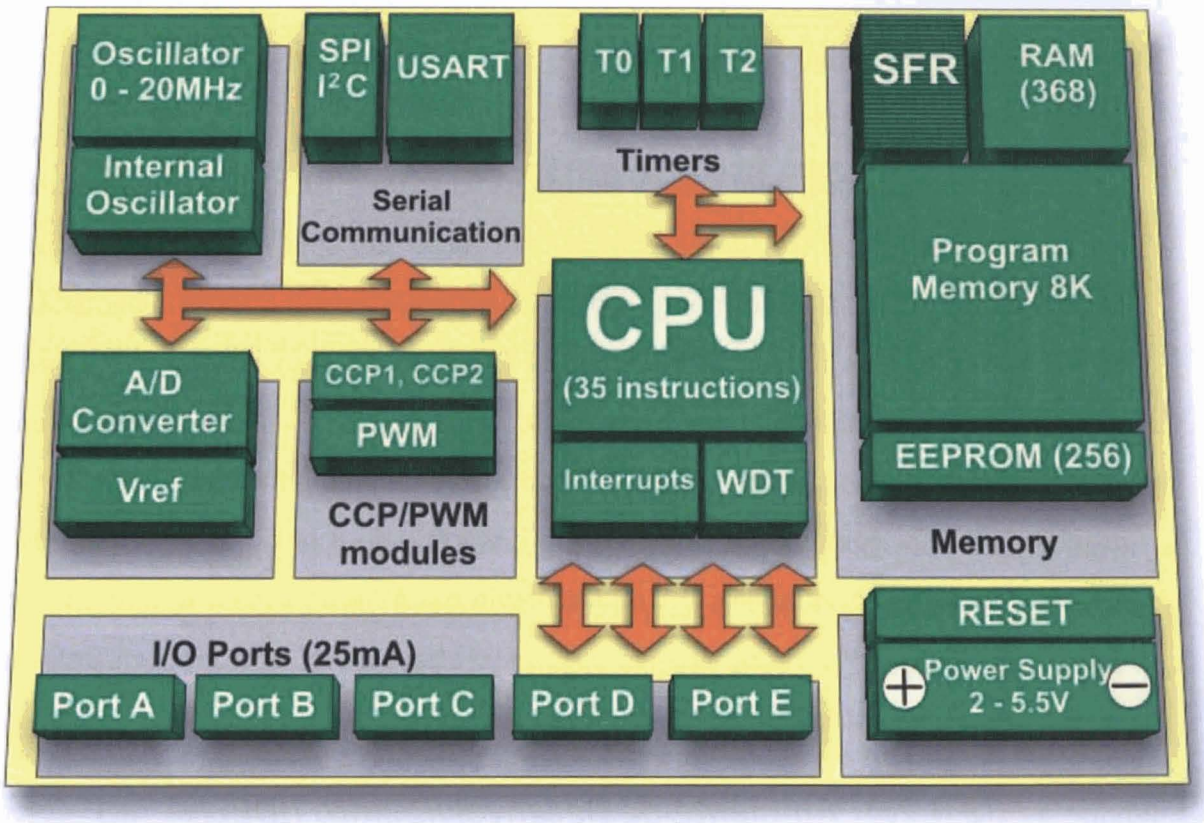
2.4. *Microcontrolador vs. Microcomputador*

A disposição das plataformas computacionais na figura 9 não foi feita ao acaso. De um lado temos processadores de 8 bits (Arduino) e do outro, processadores com arquitetura ARM (Raspberry Pi), de 32 bits. Efetuar uma análise comparativa direta entre eles é difícil, já que se enquadram em realidades completamente diferentes.



9 - ARM vs AVR & PIC

A figura 10 ilustra algumas das ferramentas que um microcontrolador apresenta nos dias de hoje. Um microcontrolador, MCU, ou µC, combina num pequeno dispositivo, as funcionalidades básicas de um computador: CPU *(Central Processing Unit),* memória volátil e não-volátil e a possibilidade de comunicar com o exterior através de I/O *(Input/Output).* Atualmente, para além destas funcionalidades básicas, as empresas que desenvolvem este tipo de sistemas têm vindo a tornar estes equipamentos o mais completo possível, com o objetivo de se distanciar da concorrência. A concorrência nesta área da eletrónica tem vindo a aumentar e, consequentemente, nos últimos anos, a evolução dos µC tem sido notória.



10 - Exemplo de um Microcontrolador ou µC. Fonte:[11]

Para além das características básicas que apresenta, nomeadamente portas I/O, o CPU e memória, este também pode apresentar algumas funcionalidades extras que poderão conter algumas diferenças que trazem grandes melhoramentos na *performance*

do mesmo, consentindo uma programação mais compacta e eficiente. Especificamente, destaque para as Interrupções, que permitem que um processador *single core* dê atenção a eventos externos para além do processamento do programa, em simultâneo; por outro lado, os ADC, os DAC e os módulos PWM permitem que uma maior gama de dispositivos sejam capazes de comunicar com o CPU, contribuindo, também, para a diversidade aplicacional do microcontrolador. Este constitui então, a unidade principal de um datalogger, pelo que a escolha deste será sempre feita tendo em conta as especificações e requisitos do datalogger, e deve ser feita com o maior rigor possível.

Começa-se, então, por analisar o cérebro de um microcontrolador através da análise dos mais reconhecidos microprocessadores de 8 bits do mercado: AVR da Atmel vs. PIC, da Microchip.

2.4.1. AVR vs PIC

Nos últimos anos, o mundo da eletrónica D.I.Y (*Do it Yourself*) tomou as rédeas do mercado dos microcontroladores com a introdução de unidades de processamento poderosas mas *low-cost*, facilmente programáveis (com introdução de bibliotecas alto-nível). A escolha dos microcontroladores de 8 bits neste caso baseou-se no facto de serem extremamente populares nos dias que correm e por se adequarem às características que se pretende para o dispositivo a implementar. São *chips* baratos e programáveis por computador. No mercado dos CPUs de 8 bits, destaque para os dois gigantes do mercado: a família PIC, da Microchip e a AVR, da Atmel. Apesar da decisão entre usar AVR ou PIC recair apenas na preferência pessoal de cada um, de seguida apresenta-se uma análise comparativa entre estes dois.

2.4.2. Ambiente de Desenvolvimento (IDE)

A AVR possui o AVRStudio, um ambiente estável, que utiliza os *frameworks* da Microsoft e é de fácil utilização, tanto em linguagem C, como em *Assembly*. Possui uma ferramenta de *debugging* razoavelmente robusta, com um funcionamento bom em geral. Por outro

lado, a PIC tem o MPLAB, estável e com características bem comparáveis com o AVR. No entanto, são ambos algo exigentes do ponto de vista computacional, pelo que um é necessário um computador com boa capacidade de processamento para suportar estes IDEs.

2.4.3. *Encapsulamento*

Em qualquer tipo de projetos de prototipagem é importante que estes chips sejam fáceis de trabalhar. Atualmente, ambas as plataformas em estudo possuem imensos processadores com encapsulamento em plástico *dual In-line*, referido como DIP (*Dual In-Line Package*).

2.4.4. *Linguagens de programação e assembler*

Um microcontrolador que ofereça simplicidade na programação irá abranger um mercado mais vasto de programadores. A linguagem que ambos obviamente suportam é o *Assembly*. Tanto com o PIC como o AVR possuem *assemblers* grátis. Porém, o *assembler* da AVR oferece maior simplicidade e, por conseguinte, eficiência. Dotado de um número elevado de instruções, possui ainda três ponteiros de 16-bit que simplificam o endereçamento e operações de palavras (2 *bytes*).

Por outro lado, o *assembly* da PIC é mais complexo, exigindo por exemplo o uso de *bank-switching*. Deste modo, a utilização de código *hand-written* torna-se muito difícil de implementar. Porém, estas operações são minimizadas por macros existentes no *assembler*.

Em conclusão, relativamente ao *assembler*, ambos são similares; porém, relativamente à linguagem em si, o AVR aparenta ter vantagens [12]. De qualquer forma, a grande maioria das aplicações são efetuadas em linguagem C, sendo esta automaticamente convertida para o código máquina respetivo através das ferramentas de software que ambas as empresas disponibilizam.

2.4.5. Compiladores

Ambos possuem compiladores grátis (Hitech ou CSS para a PIC e WINAVR para o AVR). Este último é baseado em ANSI C e no compilador GCC, facilitando a tarefa de importação de código e a utilização de bibliotecas *standard*. O tempo de compilação do código é rápido e o *sketch* ocupa relativamente pouco espaço (AVR foi desenvolvido desde o início para programação em C). [13]

Por outro lado, a PIC já não possui compiladores totalmente grátis. Aliás, existem versões *freeware*, com otimização limitada mas sem restrições de tamanho do código.

2.4.6. Outras características

Relativamente ao preço, ambos são razoavelmente baratos. Adicionalmente, ambos possuem velocidades de operação semelhantes, embora os *timers* e *clocks* da PIC ofereçam maior precisão.

2.4.7. Conclusão

Ambas as plataformas analisadas são consideradas adequadas para desenvolver projetos eficientes e baratos. O AVR é provavelmente uma plataforma mais acessível ao nível da facilidade de aprendizagem apesar de ambos oferecerem boas redes de suporte técnico (fórums, *templates*, tutoriais, etc.)

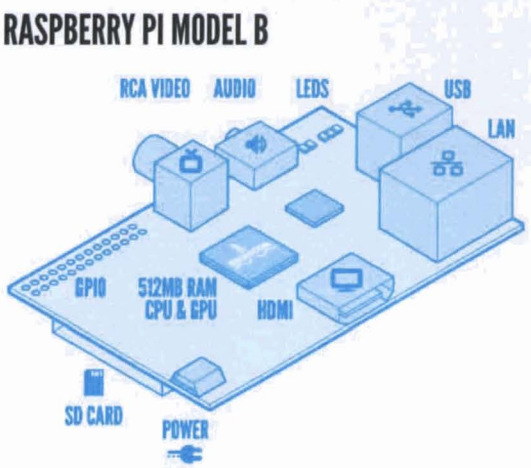
Mas no fim, a escolha entre estes dois microprocessadores recai, como já foi dito, na preferência pessoal de cada um.

Um fator que deverá pesar na escolha entre estas duas grandes famílias de processadores é ter em conta as plataformas de desenvolvimento de *hardware* que integram estes componentes.

Para este efeito, considerar-se-á o Arduino como a plataforma escolhida entre as possibilidades existentes no mercado que integram processadores de 8 bits. Do lado da

arquitetura ARM, é fácil restringirmo-nos ao Raspberry Pi: estas duas plataformas de prototipagem são atualmente líderes no sector onde se enquadram, não só pela qualidade dos seus produtos, mas também, pelo suporte técnico que oferecem. De seguida, comparar-se-ão exaustivamente as vantagens e desvantagens em utilizar cada uma destas plataformas.

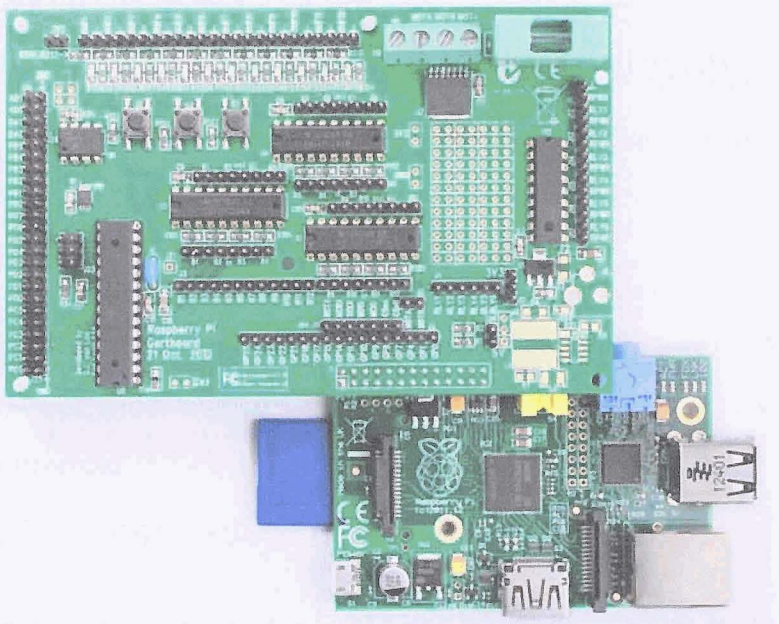
2.5. *RaspBerry Pi (ARM)*



11 - Microcomputador Raspberry Pi B. Fonte: [14]

Um microcomputador é um pequeno dispositivo e distingue-se de um microprocessador, já que possui menor capacidade de I/O de baixo nível, mas possui maior capacidade de processamento (muito mais rápido) bem como funcionalidades como HDMI e USB-Host. A figura 11 ilustra algumas características do RaspBerry Pi. Assim, é simples de inferir que um microcontrolador se insere num mercado completamente diferente do de um microcontrolador. O primeiro possui características de muito mais alto nível, como áudio, HDMI, GPU, mas, por outro lado, tem menor capacidade de controlo de *hardware* [14] . Claro que este controlo é possível: afinal de contas, é um computador. Basta para isso, ligar uma *board* de expansão como a Gertboard (figura 12), desenvolvida pelos fundadores da Raspberry, que dota o microcomputador com a capacidade de:

- Controlar motores;
- Detetar alterações no estado de *switches*;
- Alterar o estado de LEDs
- Controlar *Relays*;
- Detetar e fazer *output* de tensão;
- etc.



12 - GertBoard + RaspBerry Pi (Circuito de menor dimensão) Fonte: [14]

A Gertboard[15] é uma *board* de expansão GPIO (General Purpose I/O) para o Raspberry Pi. Tal como o nome indica, representa uma interface entre periféricos e um microprocessador. Analisando os componentes existentes na placa, destacam-se os seguintes: um microcontrolador AVR (Atmel), uma ADC e um DAC. O Raspberry Pi possibilita ainda a conexão de vários tipos de periféricos, tais como rato, teclado, monitor, cartão SD e, com a inclusão da Gertboard, qualquer periférico que um microcontrolador poderia controlar.

Deve-se, então, analisar se vale a pena usar um dispositivo tão poderoso como o RaspBerry Pi para a aplicação pretendida neste trabalho. A integração de um Raspberry Pi

no datalogger induzirá numa subutilização do microcomputador dada as funcionalidades de *high-performance* que este tem para oferecer, ou, por outras palavras, inutilizar-se-iam as várias funcionalidades que o distingue de dispositivos mais simples, vocacionados para a interação com o mundo "real", de onde se destaca o Arduino. Uma plataforma que simplesmente inclua um microprocessador e portos de saída que permitam o I/O de dados e que possua capacidade de armazenamento e ligação à internet teria todas as condições para satisfazer os requerimentos exigidos para o Datalogger.

A desvantagem mais preponderante deste em relação ao Arduino traduz-se pela maior complexidade do sistema (Ambiente UNIX) e pela menor existência de tutoriais que o tornem mais acessível a principiantes. Porém, está anunciado para Junho de 2013 [16] o lançamento de tal plataforma onde os utilizadores poderão trocar ideias, consultar os guias (tutoriais), referências e fazer o *download* de bibliotecas oficiais que irão facilitar a programação do Raspberry.

2.6. O Arduino

O Arduino é uma plataforma de prototipagem de *hardware open-source* ou livre baseado em *hardware* e *software* flexível e fácil de usar. Segundo a empresa, *"It's intended for artists, designers, hobbyist, and anyone interested in creating interactive objects or environments."* O Arduino recebe informações do exterior, através de sensores ou de outros equipamentos, e pode interagir com o que o rodeia, controlar luzes, motores, ou outros atuadores. Possui uma invejável plataforma onde os milhões de utilizadores podem contribuir com código, diagramas de circuitos, tutoriais, instruções DIY, dicas e truques, etc., e que existe faz já alguns anos; para além disso, existem inúmeros fóruns onde se pode esclarecer, com alguma celeridade, quaisquer dúvidas que possam surgir durante o desenvolvimento de um projeto.

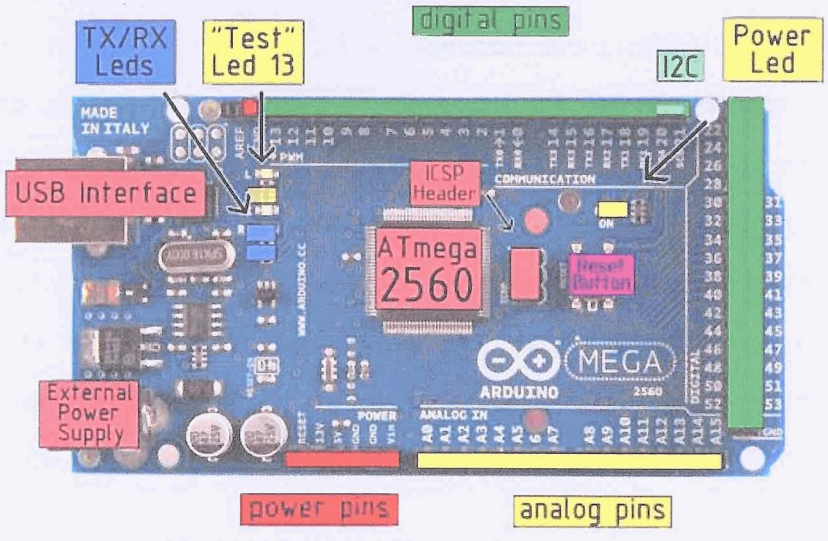
O microprocessador na placa, um AVR, da Atmel, é programado utilizando a linguagem de programação Arduino (baseada em *Wiring*) no ambiente de desenvolvimento Arduino (baseado no *Processing*). A linguagem de programação Arduino

é essencialmente baseada em C, mas com a particularidade de disponibilizar inúmeras bibliotecas de alto nível que facilita imenso a programação do microcontrolador.

No mercado existem inúmeras *boards* ou *shields* desenvolvidas, tanto pela própria Arduino como por outras empresas, que, dada a reputação do Arduino, desenvolveram *boards* capazes de adicionar funcionalidades ao Arduino, plug-and-play, indo ao encontro da versatilidade e simplicidade inerente ao Arduino. Exemplos de *shields* oficiais que poderão ser adicionados são:

- Ethernet, Wi-Fi ou GSM;
- SD;
- Motor Shield;
- XBee;
- GPS shield;
- etc.

Foi fácil notar que, para um projeto com esta complexidade, seria necessário um microcontrolador que possuísse memória interna considerável, já que a quantidade de código com que o processador iria lidar seria significativa. Assim, escolheu-se o Arduino Mega (R3), descrevendo-se este dispositivo em pormenor em seguida.[17]



13 - Arduino Mega 2560. Fonte: [17]

O Arduino Mega, representado na figura 13, é um microcontrolador baseado no ATmega2560. Possuiu 54 pins digitais que podem ser configuradores como *inputs* ou *outputs*, 16 portas analógicas, 4 USARTs e um oscilador de 16 MHz [18]. Opera a 5 V (baixo consumo) e possui uma memória *flash* de 256 Kbs, em que 8Kbs são utilizados pelo *bootloader*[5].

O Arduino Mega 2560 pode ser alimentado via USB ou via fonte de alimentação externa (2.1mm *jack*) e a escolha entre estas duas hipóteses é feita automaticamente. Os valores de alimentação propostos no *datasheet* estão entre a gama de 7-12 V, apesar do máximo suportável ser 6 a 20 V.

Esta *board* foi a primeira da geração Arduino a ser lançada sem um FTDI *USB-to-Serial driver chip*. Em vez disso, possui um Atmega16u2 programado para funcionar como um conversor *UST-to-Serial*. Esta abordagem será tratada em pormenor na secção deste capítulo referente à comunicação/interface série do Arduino Mega 2560.

Cada uma das portas digitais do Mega pode ser usada como *input* ou *output* conforme o uso das funções *pinMode*(), *digitalWrite*() e *digitalRead*(). Operando a 5V, cada um pode fornecer até um máximo de 40 mA e, por acréscimo, algumas portas ainda têm algumas funções associadas por *default*:

- Comunicação série: Os pins 0/1, 19/18, 17/16 e 15/14 funcionam como portas para comunicação série RX/TX, respetivamente. No entanto, é preciso levar em consideração que a Serial 0 (pins 0 e 1) estão também ligadas aos pins da ATMega16u2 *USB-to-TTL serial* chip.
- Interrupções externas: pins 2, 3, 18, 19, 20 e 21. Estes pins podem ser configurados para funcionarem como um *trigger* de uma interrupção perante um sinal *LOW*, *HIGH* ou perante um transição *LOW->HIGH* ou

[5] *Um bootloader é um programa, geralmente pequeno e permite a programação de microcontroladores sem a necessidade de utilizar um programador externo.*

HIGH->LOW, conforme estipulado pelo programador na função *attachInterrupt*()[19]

- Outputs PWM: Os pins 0 a 13, conforme descrito na função *analogWrite*(), podem funcionar como *outputs* PWM de 8 bits.
- LED: O pin 13 está também ligado a um LED *built-in.*
- I^2C: SDA presente no pin 20 e SCL no 21: Comunicação que utiliza a biblioteca *Wire*[20].
- *Inputs* Analógicos: existem 16 *inputs*, cada um deles com uma resolução de 10 bits (1024 valores possíveis). Por *default*, medem entre 0 e 5V, mas customizáveis através da alteração do valor do pin AREF e pelo uso da *analogReference*().

Para além destes pins I/O, esta placa de processamento suporta programação direta através do ICSP *header*, e tem capacidade de comunicar via SPI (pins 50 (MISO), 51 (MOSI), 52 (SCK) e 53 (SS)).

2.7. Comparação Arduino vs. Raspberry Pi

Resumindo, podemos incluir as principais virtudes de cada um das unidades de processamento na seguinte tabela[21][22][23]:

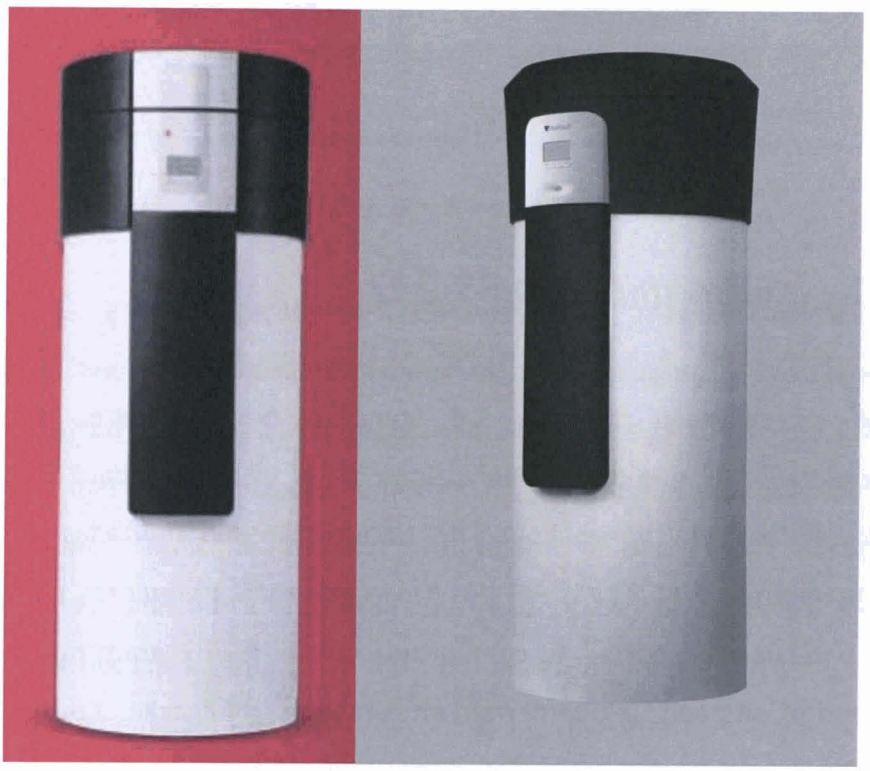
Tabela 1 - Comparação Arduino vs Raspberry Pi

Arduino	Raspberry Pi
Fácil adaptação e familiarização	Mais complexo processo de aprendizagem
Típico sistema embebido com *software* facilmente implementável	Todas as qualidades principais de um computador com Linux.
Bases de suporte técnico enormes (Tutoriais, fóruns, etc.)	Suporte limitado (Junho de 2013 anunciada centralização de informação[16])
Perfeito para controlar *hardware* (Robótica)	Possui um GPU extremamente potente e oferece suporte a conteúdo HD.
Atualmente, existe inúmeros *kits* e *shields* disponíveis, oficiais e não-oficiais	Tem muito poucas opções a nível de *shields*. Para inserir, por exemplo, capacidade de se ligar a um *Access Point* WiFi, teríamos que usar uma das duas portas USB existentes.
Consumo muito baixo de energia (<0.5W).	Consumo significativamente mais elevado do que Arduino (3.5 W).
Funcionalidades são acrescentadas através da adição de *shields*.	Oferece funcionalidades que não serão utilizadas na aplicação desejada (subutilização do equipamento)

Assim, tendo em conta todos estes fatores, a escolha para implementar o datalogger recaiu sobre o Arduino, pela sua notória flexibilidade e pelo facto de ser mais adequado a este projeto. O Raspberry Pi, apesar de possuir algumas características superiores em relação ao Arduino, perde nas ferramentas de controlo de *hardware*, pelo que, apesar de constituir um sistema mais completo, a sua integração num sistema de *data logging* resultaria numa subutilização da plataforma, já que muitas das funcionalidades não são úteis para o projeto.

Capítulo III

3. A bomba de Calor e a ECU



14 - HP270 – *Eletric Heat Pump Water Heater*

A HP270-1E, presente na figura 14, é uma bomba de calor para o aquecimento de água quente[24]. Genericamente, uma bomba de calor é um dispositivo que tem por objetivo a transferência de energia de uma fonte de calor para um recipiente, contra o gradiente de temperatura, ou seja, de uma fonte fria para uma fonte quente. Baseia-se na relação entre temperatura e pressão de líquidos de ponto de fusão muito baixo (líquidos de refrigeração). Controlando a pressão e o ciclo térmico do líquido, a bomba de calor consegue aquecer o interior de uma casa no inverno apenas utilizando a temperatura ambiente exterior como "fonte de calor".

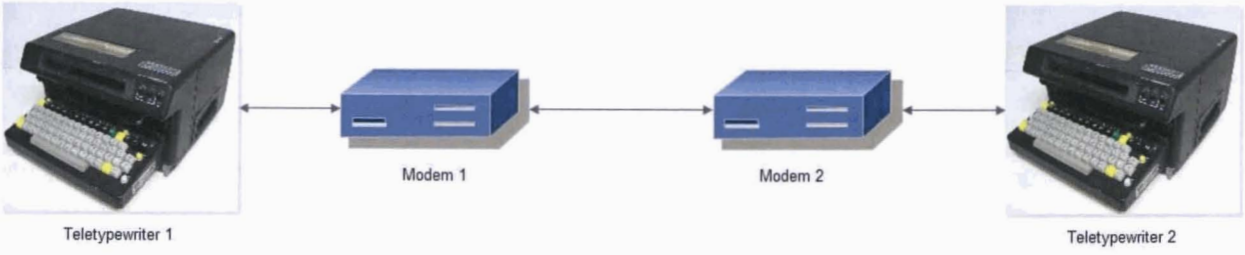
Para efeitos de teste, a Bosch forneceu uma ECU, equipamento pequeno e portátil, que utiliza a mesma versão de protocolo de dados do que a HP270, para que se pudesse testar o firmware desenvolvido sem a necessidade de uma deslocação a uma bomba de calor instalada. A comunicação é feita em série, tendo a ECU 6 pinos

disponíveis para serem utilizados: TX (linha de transmissão), RX (linha de receção), VO (*Voltage Output*), GND (*Ground)* e 2 pinos referentes a mecanismos de controlo de fluxo. Esta comunicação série é feita através de níveis TTL, pelo que não é necessária a transformação deste sinal para ser lido pelo Arduino, já que este utiliza estes níveis de tensão na comunicação série.

3.1. Comunicação RS-232

A comunicação série é, tal como o nome sugere, comunicação feita sequencialmente e que utiliza uma linha para transmitir data, ao passo que na comunicação paralela, várias linhas de transmissão são utilizadas para enviar dados simultaneamente. O RS-232 é um dos *standards* mais frequentes para a comunicação série e, antigamente, portas de comunicação série RS-232 eram muito frequentes nos computadores pessoais, já que permitiam estabelecer a ligação com um grande número de periféricos. As portas série dos dispositivos são responsáveis por providenciar a interface física que oferece o suporte para a comunicação série. Com o evoluir dos tempos, as portas série RS-232 foram preteridas pelas portas série USB. No entanto, o conceito de porta série não se alterou muito, sendo que muitos periféricos ainda utilizam o mesmo *standard* de comunicação série, mesmo que a conexão física (*hardware*) seja USB e não RS-232.

O *standard* RS-232 é *full duplex*, em que as linhas RxD e TxD podem ser utilizadas simultaneamente para receber e transmitir dados. A comunicação série não é, no entanto, tão simples: os dois dispositivos que estão a comunicar podem não ter a mesma capacidade de processamento da informação que lhes chega no RxD. Chega-se rapidamente a esta conclusão, especialmente se se pensar que, antigamente, este protocolo permitia a ligação entre dois PCs (*Data Terminal Equipment, DTE*) – Este exemplo está elucidado na figura 15.



15 - *Teletypewriter (First Generation DTE's) Communication*

Por vezes, a comunicação era feita através da utilização de *modems* como intermediários (*Data Communication Equipment, DCE*). Assim, é fácil inferir que há assimetria nesta comunicação bidirecional: esta leva à assunção de que o DTE recebe dados do DCE o mais rápido possível, e que o DCE poderá não ser capaz de lidar com a informação que lhe chega com a rapidez inerente do DTE. Deste modo, nestas situações, é usual usar-se mecanismos de controlo de fluxo típicos do *standard* RS-232. Estes mecanismos, que podem ser implementados por *hardware* ou por *software*, representam ferramentas fundamentais e até imperativas, em alguns sistemas, para a transmissão de dados com a menor taxa de erro possível. Na grande maioria dos mecanismos de controlo de fluxo por *hardware* implementados estão presentes os sinais RTS e CTS. Outra situação possível é utilizar-se controlo de fluxo por *software* (XON/XOFF), desde que o DTE e DCE tenham possibilidade de interpretação dos caracteres especiais de XON/XOFF a utilizar.

No caso presente haveria a possibilidade de implementar o controlo utilizando os sinais CTS/RTS, visto que a ECU disponibiliza estes sinais. Estes representam o denominado controlo de fluxo feito através de *handshaking* por *hardware*: basicamente atribui série aos dispositivos a capacidade de indicarem se estão prontos para receber/enviar dados. Estes sinais são particularmente úteis quando um dispositivo envolvido num processo de troca de dados é relativamente mais lento que o outro, ou, que precise de mais tempo para processar um bit recebido. Assim, de modo a prevenir a perda de informação (por exemplo quando o *buffer* de chegada de dados do Arduino fica cheio), este poderá sinalizar ao transmissor para parar o envio de dados por um determinado período de tempo.

O Arduino Mega, que contém um chip ATMEGA 2560, possui uma *built-in* USART de níveis TTL ligada ao Pin0 (RX) e Pin1 (TX) da placa de desenvolvimento. Todos os dados que são enviados por esta porta, denominada *Serial0*, poderão ser visualizados em tempo real numa das ferramentas que vêm embutidas no IDE do Arduino, o *Serial Monitor* (monitor da porta série), ferramenta esta que irá ser explicada em maior pormenor mais adiante. A comunicação entre o computador e o Arduino é feita através de um chip ATMEGA16u2, que reencaminha os pins 0 e 1 para a porta USB e faz com que o periférico apareça do *device manager* como uma porta série até que sejam instalados os drivers corretos. Neste ponto, o Arduino surge listado como "*USB Serial Port*". Assim, o Arduino utiliza protocolo de comunicação RS-232 em cima de uma ligação de *hardware* USB [25].

Uma vez que a USART no ATMEGA2560 usa apenas 2 pinos, não há controlo de fluxo por *hardware* baseado em RTS/CTS. De modo a minimizar a perda de informação, do ponto de vista do programador, há um *buffer* FIFO para a receção de dados de 128 *bytes* (ring buffer). Esta característica existe graças à biblioteca *HardwareSerial*, responsável por ler os dados que estão presentes no *buffer* de chegada da USART, que possui um tamanho de 20 *bytes* [26].

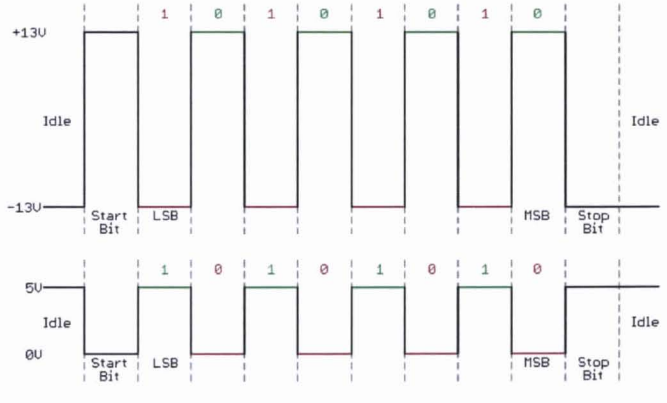
Por acréscimo, a trama do Arduino e da ECU utilizam uma estrutura de dados de 8 bits, 1 stop bit, sem paridade e sem controlo de fluxo. Relativamente ao *baudrate*, estes vai até aos 115200bps e deve ser escolhido para que não haja *overflow* dos *buffers* de receção de dados, tanto do buffer da HardwareSerial ou o buffer da USART [26]. Por conseguinte, e visto que um script em Arduino está em *loop*(), torna-se importante verificar as *flags Serial.available*() e evitar ciclos de processamento de chegada de *bytes* muito longos.

Quantos às restantes linhas disponíveis, falta referir a necessidade de se ligar o sinal GND de ambas as partes envolvidas na comunicação série de modo a assegurar que os dois dispositivos identifiquem os sinais lógicos da mesma maneira (referidos à mesma tensão de referência). Isto é absolutamente necessário especialmente quando os dispositivos envolvidos na troca de informação possuem fontes de referência diferentes (por exemplo, PC de um lado, e, alimentação da rede doméstica, do outro), em que os

níveis de referência poderão ser ligeiramente diferentes, pondo em risco a interpretação dos dados transmitidos.

Atualmente, a comunicação série é uma forma simples e muito poderosa de efetuar *debugging* de *software*. A forma como esta é feita é universal e, geralmente, os dispositivos que suportam comunicação série não encontram entraves muito significativos a nível de *software*. No entanto, poderá haver entraves a nível de *hardware*. Por exemplo, no caso das portas RS232, já em desuso no mercado atual, os '1's lógicos são representados por uma tensão negativa (-3 a 25 V) e os 0's são transmitidos como uma tensão positiva, entre os +3V e +25V (na grande maioria, os valores lógicos vinham em -13V e +13V, 1's e 0's, respetivamente). Por outro lado, temos a grande maioria dos microcontroladores, que atualmente trazem USARTs que podem ser usadas para enviar e receber dados via série. Neste formato de comunicação série, também chamado de TTL série, a comunicação ocorre utilizando uns níveis de tensão diferentes dos utilizados no *standard* RS232. Um '1' lógico é sempre representado por VCC, que tipicamente toma o valor de 5V ou 3V3, e um zero lógico como 0 V.

Como já foi dito anteriormente, no caso da ECU utilizada para testes e no caso do Arduino, existe uma interface de comunicação série que utiliza níveis TTL, pelo que a conversão entre os dois sinais é desnecessária (que teria que ser feita utilizando por exemplo MAX232 para inverter o sinal RS232). A figura 16 ilustra este mesmo caso, em que uma trama pode ser representada com níveis diferentes de tensão. Assim, o esquema de ligação entre a ECU e o Arduino é razoavelmente transparente, e será explicada mais adiante.

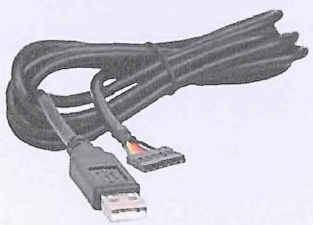


16 - Níveis RS-232 vs TTL

Numa primeira fase houve algumas dificuldades em obter uma ECU para testes. Por isso, implementou-se um emulador de bomba de calor (em que um computador simula a operação de uma bomba de calor) para obter os dados tipo e desenvolveu-se um script de processamento destes dados antes de se passar para uma ECU real. Para o efeito, utilizou-se o MATLAB para simular a ECU: recebe informação via porta USB, processa essa informação como um comando, elabora a trama de resposta correspondente e encaminha-a via série para ser processada pelo Arduino.

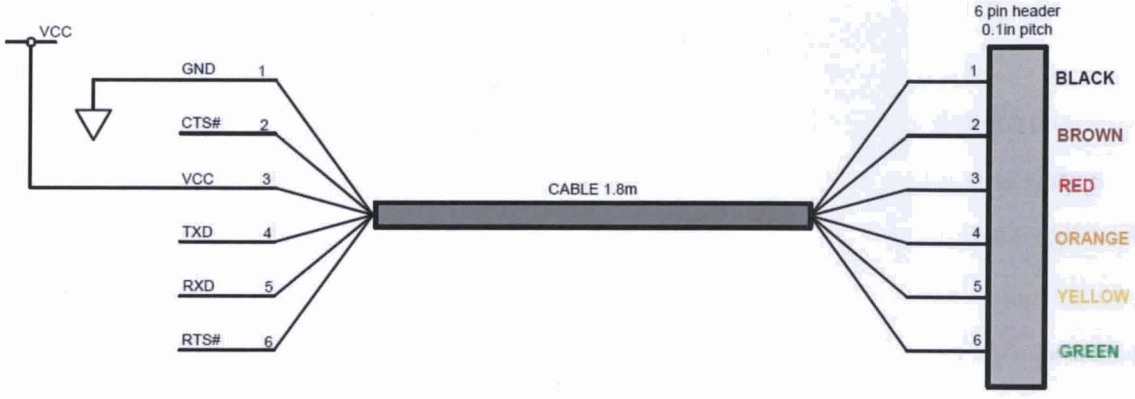
3.2. *Equipamento utilizado e setup do sistema de teste*

A ligação que já existe entre o Arduino e o PC (para efeitos de *upload* de código e para visualização de dados no *serial monitor* do Arduino IDE) não poderá ser usada para a comunicação série entre o MATLAB e o Arduino: cada porta série só poderá estar a ser utilizada por uma aplicação de cada vez. Assim, utilizou-se um cabo FTDI TTL 3V3, representado na figura 17 (e o respetivo pinout encontra-se na figura 18), que permitiu a utilização de outra porta série para a comunicação entre Arduino e o Matlab.



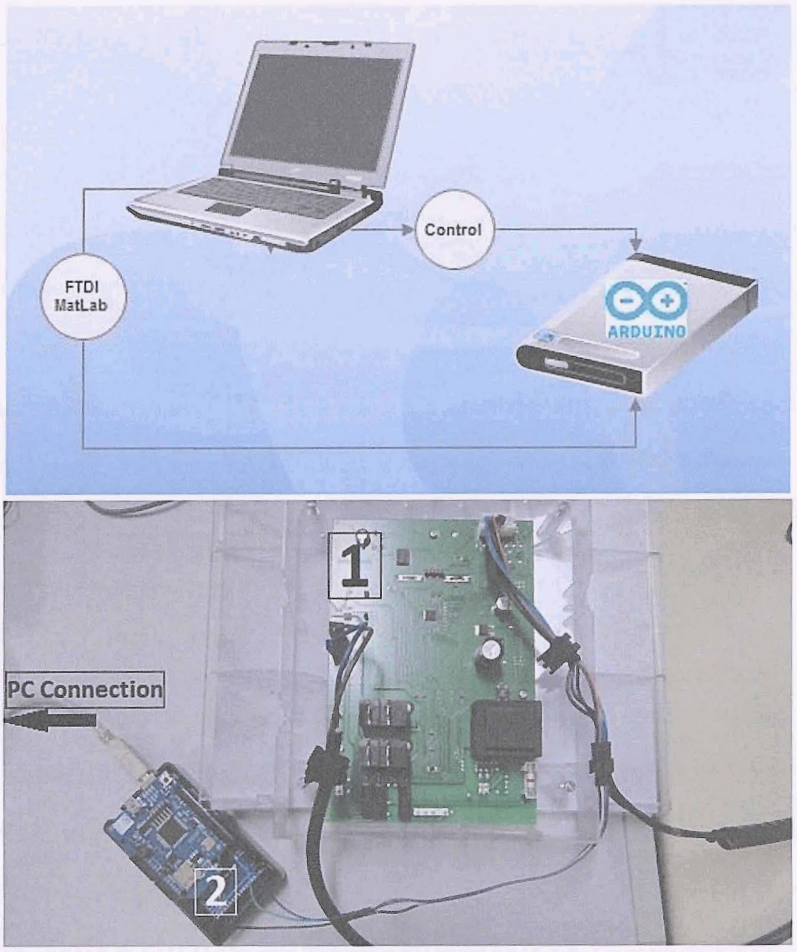
17 - Cabo FDTI TTL-232R TTL to USB Serial Converter.

A ligação entre ECU e Arduino é feita através da ligação entre o sinal Rx e o Tx, sem se recorrer aos sinais de controlo da comunicação série.



18 - Pin-out do FTDI cable.

O setup final do sistema implementado permite a troca de informação entre o Arduino e o emulador de bomba de calor (ver figura 19), além de permitir, em simultâneo, a monitorização e o controlo do Arduino: *upload* do *sketch*, navegação pelo menu, pedir um determinado tipo de informações, enfim, um vasto leque de funcionalidades que foram implementadas e que serão explicadas no capítulo 4.



19 – Em cima: Setup do ambiente de emulação da ECU.
Em baixo, Esquema real do sistema (1 – Board da ECU; 2- Arduino).

3.3. *Bosch Communication Protocol v2.0*

Anteriormente foi referido que a Bosch não utiliza nenhum protocolo genérico existente no mercado: utiliza a sua própria estrutura de trama de dados. A versão 2.0 dos comandos disponíveis para a ECU, disponibilizada para testes, permite efetuar vários pedidos diferentes. No entanto, no âmbito do trabalho, consideraram-se dois tipos de comandos: a trama correspondente a um *broadcast* de informação (figuras 24 e 25) e um pedido de informação específico *singlecast*) (figuras 20 e 21). A resposta a um pedido de

informação total (*broadcast*) retorna vários valores a serem representados ou armazenados.

Inicialmente, por se tratar de uma trama mais simples, começou-se por testar a comunicação Arduino-ECU através de um pedido de informação *singlecast* (figuras 20 e 21).

Os campos delimitados a azul na figura 20 são os chamados delimitadores da trama: o seu envio é obrigatório para a validação de uma trama. São estes que delimitam a trama e que assinalam a informação relevante que está entre o primeiro delimitador, o *startFrame*, representado pelo envio de dois caracteres chaveta para a direita e o segundo delimitador, o *endFrame*, representado por dupla chaveta para a esquerda. Cada um destes conjuntos de caracteres é representável e visível no *Serial Monitor* do Arduino IDE pela sua representação ASCII, correspondente ao número decimal 123 ou 0x7B em hexadecimal, e 125, ou 0x7D. O campo Data ID representa o tipo de dados a pedir à ECU, e, por conseguinte, à bomba de calor.

Start Frame	Command	ParamLocal	Data ID	CheckSum	End Frame
{{	D2	02	XX	XX	}}

20 - Bosch Communication Interface SingleData Frame Request

A resposta a esta trama é formulada utilizando o mesmo *byte* referente ao campo "*Command*", 0xD2. Segue-se o *DataID*, que deverá corresponder ao tipo de valor pedido, e os dois *bytes* referentes aos valores atuais associados a esse *DataID*. O primeiro, *IntDataValue* é o *byte* que vai originar a parte inteira do *float* requisitado e o *DecDataValue* corresponde à parte decimal. Resta depois o campo referente ao

CheckSum, que corresponde a uma soma *byte-wise* de todos os campos de data (a cinza na imagem).

Start Frame	Command	Data ID	Int Data Value	Dec Data Value	CheckSum	End Frame
{{	D2	XX	XX	XX	XX	}}

21 - Bosch DataFrame Response – Valid Data

Alguns dos diferentes valores possíveis para o campo *DataID*, para esta versão do protocolo, estão representados na Tabela que se segue:

ID	Data type	ID	Data type
0	Water Inlet Temperature	15	Fins inlet temperature
1	Water Outlet Temperature	16	Current Measurement
2	Water Flow	17	Water Pump Status
3	Tank Upper Temperature	18	Ventilator Status
4	Tank Lower Temperature	19	Compressor Status
5	Air inlet temperature/Box air temperature	20	Heating Element Status
6	Operation Mode	21	Solenoid Status
7	Only for use in EBHP - Heat Pump Compressor Rest Time (also reset 2 minutes fan working before start compressor)	22	Safety Valve 1 State
8	Burner Power	23	Safety Valve 2 State
9	Burner System Status	24	Safety Valve 3 State
10	Select Fan Speed	25	Ionization status
11	Reset 2 minutes fan time before compressor	26	Solar mode Status
12	reset Circulation pump time (1 min)	27	Fan Voltage
13	reset Heat Pump Compressor Rest Time	28	Fan Current
14	Open/Close Solenoid Valve 1- ON, 0-OFF	29	Fan Power
		30	Fan Speed

Tabela 2 - Data IDs

No decorrer da implementação do datalogger notou-se uma tendência da ECU em responder com uma determinada trama de dados, conforme o tipo de comando que esta

recebia e consoante o ambiente que rodeia a ECU, evidenciando situações que não estavam presentes na documentação fornecida. Então, inferiu-se acerca da estrutura da trama de dados enviada pela ECU, após a realização de várias experiências. Se por exemplo, a bomba de calor não estiver ligada fisicamente à ECU (que corresponde, de facto, ao caso envolvido nos testes) e independentemente do tipo de dados pedidos (referenciados pelo *DataID*), a resposta será sempre a mesma, como representado na figura 22.

Start Frame	Command	CheckSum	End Frame
{{	E0	**E0**	}}

22 - Bosch DataFrame Response – não há informação a disponibilizar por parte da ECU

Outra situação surge quando a ECU recebe uma trama que possui incoerências entre o *CheckSum* calculado internamente pela ECU e o *CheckSum* presente de facto na trama. Este comando é visto pela ECU como uma trama com conteúdo inválido e provoca a resposta ilustrada na figura 23.

Start Frame	Command	CheckSum	End Frame
{{	EC	**EC**	}}

23 - Bosch DataFrame: Resposta a um Comando recebido com *CheckSum* inválido

Posteriormente, adicionou-se a trama de dados que de facto contém grande parte de informação e que permite, com um só comando, requisitar diversos tipos de dados.

Start Frame	Command	CheckSum	End Frame
{{	A0	XX	}}

24 – Pedido de informação BroadCast

Um pedido de informação broadcast segue um formato simples e este está representado na figura 24. Para além dos delimitadores existe o *byte* relativo ao tipo de comando que informa a ECU acerca do tipo de informação que se pretende e o campo CheckSum, que neste caso, toma o valor de 'A0'. Por outro lado, a figura 25 apresenta a resposta ao comando enunciado acima e que é constituída por 28 *bytes*:

- Cada campo *Start/EndFrame* possui 2 *bytes*;
- 10 *Bytes* de dados correspondentes a 5 *floats* de informação;
- 12 *Bytes* relativos a outro tipo de dados
- 1 *Byte* correspondente ao *CheckSum* da trama e 1 *byte* relativo ao Comando.

Todos os campos de dados atribuem informação útil a ser armazenada. No entanto é importante destacar a situação em que a ECU não está ligada corretamente à bomba de calor pelo que assim, não poderá ler dados reais. Por conseguinte, a deteção da bomba de calor é feita através da análise do conteúdo do campo 12 (*Compressor State*) e o campo 13, *Heater State*. Através deste poder-se-á ver se o Compressor ou o *Electrical Heater* estão ativos (situação em que existe uma fonte de calor ativa – indicando que o dispositivo está ligado). No caso de a ECU não ter nada ligado a si mesma, o valor será sempre zero. Traduzindo esta situação para o ponto de vista do programador, a ECU está ligada se um *OR* lógico entre os dois *bytes* retornar um valor diferente de zero. Outra hipótese seria utilizar o campo da corrente, já que, se a corrente medida for zero, significa que a bomba de calor está desligada ou que não está ligada à ECU.

Start Frame	Command	Data0/1		Data2/3		Data4/5	
		NTC TOP (0.1°C)		HTC BOT (0.1°C)		HTC AIR (0.1°C)	
{{	A0	Int Part	Dec Part	Int Part	Dec Part	Int Part	Dec Part

Data6/7		Data8/9		Data10	Data11	Data12
HTC PINS		Current (0.1A)		Pump State	Fan State	Comp State
Int Part	Dec Part	Int Part	Dec Part			

Data13	Data14	Data15	Data16	Data17
Heater State	SolValve State	HP mode	HT[0]	HT[1]

Data18	Data19	Data20/21		CheckSum	End Frame
HT[2]	SwType	SwVersionN			
	0 - HPAL 2 - HPAF			XX	}}

25 – Protocolo de Comunicação Bosch –Resposta a um comando BroadCast

Ao longo do desenvolvimento de um produto a versão de firmware geralmente sofre algumas evoluções na sequência da colmatação de alguns erros ou da inclusão de novas funcionalidades. Notou-se que a ECU utilizada para testes continha um *firmware* que não era uma *release* final visto que rapidamente se verificou a existência de erros no protocolo de comunicação. Por exemplo, no caso da resposta a um comando *broadcast*, a ECU não respondia corretamente e o tamanho da trama de dados enviada não correspondia ao tamanho documentado na figura 25. Por acréscimo, durante o processo de desenvolvimento do *firmware* do datalogger apareceram alguns obstáculos que se interpuseram no processo evolutivo do projeto do datalogger. Primeiro, as versões relativas às estruturas das tramas utilizadas pelos equipamentos fornecidos para teste apresentavam diferenças de equipamento para equipamento. Tal implicava, aquando a atualização do protocolo, que o script desenvolvido para o Arduino tivesse que ser

espera. Assim, enviar este comando com tempo=0 ou enviar um pedido de informação *broadcast* é exatamente igual. A figura 27 apresenta a formatação correta do comando no caso específico de se configurar com tempo = 0 segundos.

Start Frame	Sender	Receiver	Message ID	Command	Data0	Data1	CheckSum	End Frame
{{	0x10	0x20	x	A1	0x00	Value (s)	ChkSum	}}

27 – Comando Configuração Broadcast

Com este último caso, 0 segundos no campo Data1 seria o valor escolhido para o pedido de *broadcast*, visto que quem controla a frequência dos *broadcasts* é o próprio script do Arduino e não a ECU. Assim, não há qualquer interesse em utilizar esta funcionalidade da nova versão protocolar e, portanto, sempre que se pretenda uma resposta *broadcast*, usar-se-á a configuração descrita na figura 27 (*broadcast*) em detrimento da representada na figura 26 (*configBroadcast*). O comando para realizar um pedido de dados *broadcast* é feito de forma equivalente à versão 2.0., mas com a adição dos campos de endereço, dependentes do tipo de equipamento, ou aplicação ligada à ECU; relativamente ao campo *Message ID,* este é usado como um *Acknowledged* e deverá ser mantido a 0x01 ou a 0x00: A figura 28 ilustra como este comando deverá ser estruturado.

Start Frame	Sender	Receiver	Message ID	Command	CheckSum	End Frame
{{	0x10	0x20	x	A0	ChkSum	}}

28 – Comando Broadcast

Relativamente aos campos já referidos dos endereços a utilizar, a tabela seguinte descreve a relação entre os valores hexadecimais escolhidos e o tipo de aplicação.

alterado de modo a colmatar estes *updates* e que a empresa tivesse que fornecer outro equipamento com a versão mais recente do *firmware*. Para manter a compatibilidade com os vários equipamentos, foram mantidas em estado funcional, no *firmware* do Arduino, as diferentes versões do protocolo.

Uma vez que a documentação fornecida não era coerente com o protocolo observado experimentalmente, houve algum investimento de tempo considerável na determinação experimental dos protocolos efetivamente implementados.

Desta forma, o datalogger foi tomando forma, e a sua rotina de tratamento de informação foi sofrendo alterações de modo a aceitar todos os tipos de resposta que pareciam apresentar alguma consistência (desconhecia-se na altura a existência destes erros). Por outras palavras, foi necessário adaptar a comunicação com a bomba de calor, tendo em vista a validação não só das tramas que a bomba enviava (com bugs) mas também tendo em consideração a versão do firmware que esta aparentava possuir.

Perante a coerência e consistência da resposta a um comando broadcast e pela escassa informação documental disponibilizada, validou-se a informação que a bomba de calor estava a transmitir. Tal foi também verificado no caso de um pedido de informação individual: a ECU devolvia sempre a mesma trama, e assumiu-se que esta estava no formato da trama usada pela ECU quando não havia bomba de calor ligada à ECU. Infelizmente, a *release* final do protocolo, a v2.2., foi disponibilizada numa fase muito tardia, pelo que todos os testes apresentados neste relatório não tiveram em conta a versão final do protocolo, mas sim a versão que estava de fato implementada na ECU que estava disponível. A *release* da versão final do protocolo, que aconteceu numa altura incompatível para a reformulação do datalogger, requeria que a ECU fosse substituída por outra já com este protocolo implementado, o que, devido a limitações temporais, não foi possível e não chegou a ser efetuado. Para enquadrar esta situação e tentar explicar melhor o que irá ser apresentado no capítulo dos Resultados, apresenta-se de seguida a estrutura de dados que de facto foi observável.

3.4. *Bosch Communication Protocol v2.0 – Experimental*

A comunicação através do *singlecast* processa-se identicamente ao enunciado nas figuras 19, 20 e 21.

Porém, em relação ao modo *broadcast*, notaram-se diferenças relativamente ao especificado: o tamanho da resposta ao comando efetuado pelo datalogger era diferente (+2 *bytes* de informação). Optou-se, no entanto, por manter o significado dos campos entre as duas versões (2.0 e 2.0 experimental) o mais próximos quanto possível. A figura 26 ilustra o significado adotado para os campos da versão aqui analisada, com base no que foi observado experimentalmente. Os dois *bytes* adicionais foram marcados com "??" e estão representados no campo 19 e 20 da trama (nos testes efetuados estes *bytes* eram sempre devolvidos a zero). A razão para a escolha destes campos como os campos incorretos ou não documentados foi simplesmente pelo facto da ECU conter alguma informação coerente relativamente aos campos que o sucedem, o *checksum*, *endframe* e possuir *SwType* e *SwVersion* diferentes de zero.

Start Frame	Command	Data 0/1		Data 2/3		Data 4/5	
		NTC TOP (0.1°C		HTC BOT (0.1°C		HTC AIR (0.1°C	
{{	A0	Int	Dec	Int	Dec	Int	Dec
Data 6/7	Data8/9	Data10		Data11		Data12	
HTC PINS	Current (0.1A)	Pump State		Fan State		Comp State	
Int Dec	Int Dec						
Data13	Data14	Data15		Data16		Data17	
SValve State	HP Mode	HT[0]		HT[1]		HT[1]	
Data18	Data19	Data20		Data21		Data22/23	
HT[2]	??	??		SwType		SwVersionN	
				0 - HPAL 2 - HPAF			
Check Sum	End Frame						
XX	}}						

26 - Bosch Communication Interface - BroadCast Frame Response

Posteriormente, a Bosch apresentou uma nova versão do protocolo em que alguns campos foram adicionados e alguns bugs corrigidos, constituindo então a versão 2.2., versão final a ser embutida na ECU final para o modelo HP270-E. Esta versão é descrita em seguida.

3.5. *Bosch Communication Frame v2.2*

Apesar de possuir uma estrutura diferente, o tipo de comandos existentes e que foram considerados para esta exposição incluem os considerados na versão antecedente: um comando para pedir um único tipo de dados, *GetParameter* e outro que efetua um pedido à ECU para transmitir diversas informações acerca do funcionamento da bomba de calor. Os campos adicionados à trama neste *update* do protocolo de comunicação fazem referência a uma forma encontrada para definir o tipo de aplicação que está a comunicar e a maneira que esta comunica. Para além disso, também foi incluído um comando diferente, representado pelo ID "A1", que representa uma alternativa para pedir respostas *broadcast* sem estar periodicamente a pedir, que se baseia no estabelecimento de um valor que representa o tempo de espera entre dois envios de informação *broadcast* por parte da ECU. A nomenclatura da versão final atribuída a esta versão protocolar veio colmatar todos os bugs verificados na versão anterior (2.0.). Na versão 2.0., relativamente à trama de *broadcast*, definiu-se a situação em que os dados não eram enviados da forma documentada. Assumiu-se que isto ocorria quando a aplicação (bomba de calor) não estava ligada a ECU. Então, foi importante definir a melhor forma de retirar informação acerca da ligação entre a aplicação e a ECU. A trama *broadcast* possui um campo onde se pode ver se existem aplicações ligadas à ECU (*Heater State* + *Compressor State*). Para isso, começar-se-á então a análise deste protocolo pela trama *broadcast*.

Por defeito, a ECU constrói a trama *broadcast* assim que o comando é recebido: a inclusão da nova funcionalidade enunciada acima deu origem a um comando novo, denominado por *configBroadcast,* que possui um campo correspondente ao tempo de

	Appliance Type	Default Adress	Range
Sender Adress	Gas Water Heater	0x00	0x00-0x0F
	Heat Pump Water Heater	0x10	0x10-0x19
	Electrical Water Heater	0x1A	0x1A-0x1F
Slave/Receiver Adress	Wired Remote Controls	0x20	0x20-0xBF
	CO detector	0xC0	0xC0
	Active Gateways	0xF1	0xF1-0xFE

Tabela 3 – Endereços Sender e Receiver

No caso em análise, os campos marcados a verde na tabela acima representam o tipo de aplicação a utilizar. Segue-se então a configuração propriamente dita da resposta a um comando broadcast. Esta é constituída por 32 *bytes* de informação, embutida numa trama de 41 *bytes*.

Start Frame	Sender	Receiver	Message ID	Command	Data0
{{	0x10	0x20	x	A0	SetPoint

Data1/2	Data3/4	Data5/6	Data7/8	Data9/10	Data11/12
NTC TOP (0.1°C)	NTC Bottom (0.1°C)	NTC Air (0.1°C)	NTC FINS(0.1°C)	NTC COIL (0.1°C)	NTC TANK (0.1°C)

Data13	Data14	Data15	Data16	Data17	Data18
Component Status	HP Mode	Working Mode	Working Flags	External Sys	Failure Mode

Data19/20	Data21/22/23/24	Data25	Data26	Data27
Current (0.1°C)	Power Consumption MSB LSB	HT[0]	HT[1]	HT[2]

Data28	Data29	Data30	Data31/32	CheckSum	End Frame
HT[3]	HT[4]	SwType	Sw Version Number	CheckSum	}}

29 - BroadCast Comand v2.2.

Tal como em versões anteriores, os *bytes* que correspondem ao *startFrame* e *endFrame* continuam os mesmos; as diferenças estão na adição de campos como os endereços (*Sender* e *Receiver*) cuja justificação foi já esclarecida acima, e na disponibilização de mais informações operacionais.

Relativamente à deteção da ligação entre a ECU e a bomba de calor dever-se-á fazer referência ao campo Data 13, mais concretamente para os bits referentes ao *Electrical Heater State* (bit4) e ao *Compressor State* (bit3): o caráter hexadecimal correspondente aos 4 bits menos significativos (que contém o Compressor Heater bit) estará compreendido entre 0 e 7 para o caso em que o bit3 seja 0 (LOW). No caso dos 4 bits mais significativos do *byte* (Data 13), o caráter hexadecimal será um número par visto que o bit4 é o bit menos significativo (quando este estiver a zero, ou seja, LOW). Assim,

obtém-se uma gama de valores hexadecimais pelos quais se pode identificar o estado da máquina. Para contextualizar o que foi descrito acima apresenta-se de seguida a estrutura do campo Data13 (figura 30):

Data13 - **Component Status**
bit0 - water pump state bit1 - Fan State bit2 - Fan Speed bit3 - Compressor Rate bit4 - Electrical Heater State bit5 - bypass Valve bit6 - External system coil status bit7 - Photovoltaic Status

30 - Campo Data13

Relativamente ao *GetParameter*, como já foi dito, foram efetuadas algumas alterações à sua estrutura em relação à v2.0. Agora, esta permite obter, numa só resposta, até 8 tipos de informação. Isto é efetuado pela inclusão de vários IDs de informação nos campos que se seguem ao *byte* "Command". Cada valor correspondente a um ID tem o tamanho de 2 *bytes* em que o primeiro (*byte* mais significativo da trama) contém o valor inteiro e o segundo *byte* (menos significativo na trama) contém os valores decimais.

Um exemplo de uma trama que efetua um pedido de informação relativo a 4 IDs está representado na figura 31 (*GetParameter*).

Start Frame	Sender	Receiver	Message ID	Command	Data0
{{	0x10	0x20	x	D2	0x00

Data1	Data2	Data3	Data4	CheckSum	End Frame
ID1	ID2	ID3	ID4	ChkSum	}}

31 - Trama com 4 IDS

A resposta ao comando evidenciado na figura anterior toma o formato representado na figura 32.

Start Frame	Sender	Receiver	Message ID	Command	Data0
{{	0x10	0x20	x	D2	ID1

Data1/2	Data3	Data4/5	Data6
Value	ID2	Value	ID3

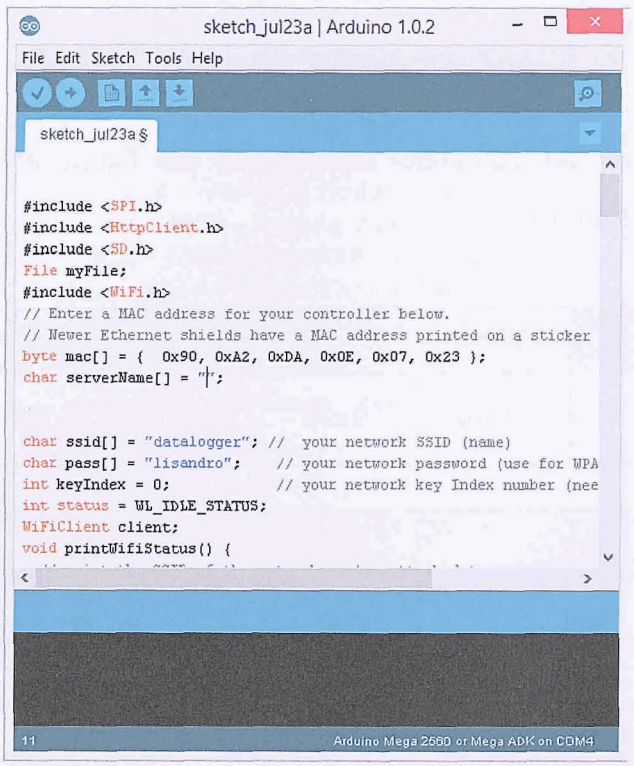
Data7	Data8/9	CheckSum	End Frame
ID4	Value	ChkSum	}}

32 - 4 IDS de resposta

A forma como estes dados são processados segue um processo diferente do da versão anterior, a 2.0: neste caso, considera-se *IntDataValue* um valor hexadecimal que, convertido em inteiro, corresponde ao valor 120. O valor real da temperatura (por exemplo) é 120-100=20°C; o mesmo acontece relativamente ao *DecDataValue*, com a particularidade de se ter que dividir por 100 para se obter o valor final desejável. Assim, exemplificando com o valor 156 (Decimal), tem-se 156-100 = 56/100 = 0.56. Neste

exemplo, o valor final requisitado pela trama de comando, e correspondente ao ID de informação referenciado pelo campo *DataID*, é 20.56°C.

3.6. Setup do Ambiente de desenvolvimento Arduino



33 - Ambiente de Desenvolvimento Arduino (IDE)

O IDE do Arduino, apresentado na figura 33, e tal como tudo relacionado com a plataforma de prototipagem Arduino, é um IDE que se rege pela flexibilidade e pela característica operacional de ser *plug-and-play*. Em poucos minutos é possível programar o microcontrolador. Este IDE é baseado no *Processing* IDE, um IDE idealmente concebido para se relacionar com um computador da mesma forma que o Arduino IDE se relaciona com os microcontroladores. De acordo com os *developers*, o *Processing* é uma plataforma para conceber ideias[27] e o Arduino rege-se pelos mesmos princípios. Relativamente aos ambientes de programação propriamente ditos, estes utilizam a mesma estrutura de

inicialização e de organização de código, mas cuja comparação sai do espectro deste trabalho, pelo que apenas a estrutura presente no Arduino IDE será discutida.

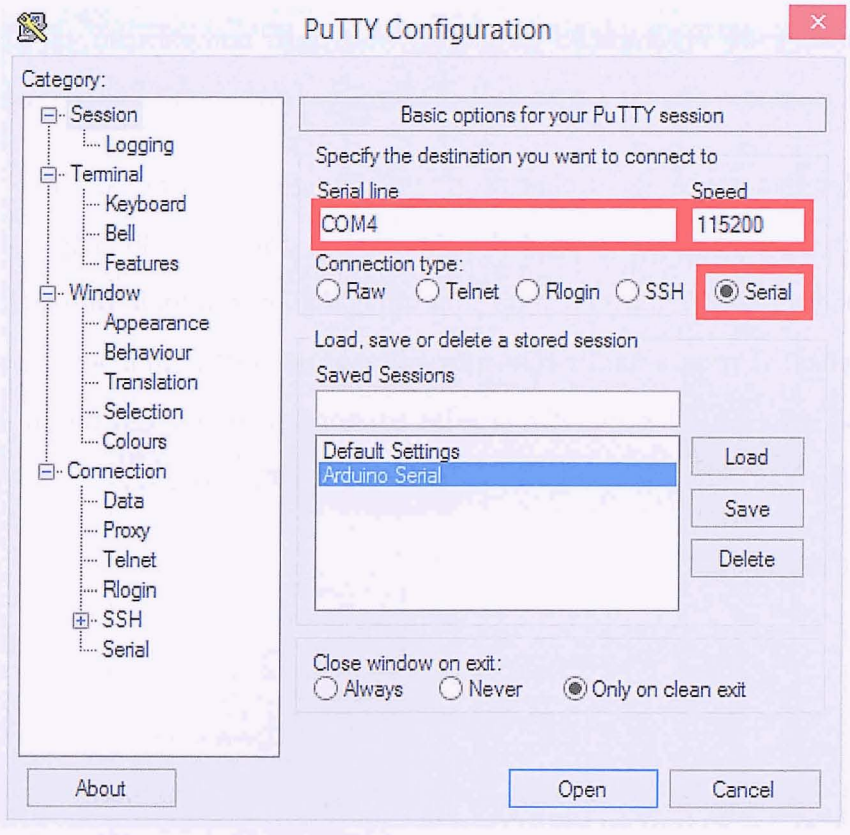
Relativamente à organização do código, o Arduino IDE possui algumas características extraordinárias. A função *main*() por exemplo, não está acessível pelo programador: esta vem preenchida com um código por *default*, que inclui a invocação de três funções. O programador têm que editar duas destas funções, a *setup*() e a *loop*(), invocadas pela *main*(), e que são criadas aquando a compilação de um *sketch*. A relação entre estas três entidades poderá ser evidenciada na figura 34.

```
int main(void)
{
init();

setup();

for (;;)
loop();

return 0;
}
```

34 – Estrutura interna do Código de um Arduíno Sketch (IDE)

O Arduino IDE vem integrado com um monitor da porta série, denominado Serial Monitor, que permite, tal como o nome indica, a monitorização e visualização dos dados a serem enviados bidireccionalmente entre o Arduino e o PC. No entanto, todos os resultados apresentados neste documento foram obtidos utilizando como *Serial Monitor* o PuTTy, uma ferramenta bem assente no mercado há muitos anos e a sua escolha baseou-se essencialmente na preferência pessoal. De seguida apresenta-se a configuração necessária para configurar o PuTTy de modo a analisar o que está a ser transmitido na porta USB.

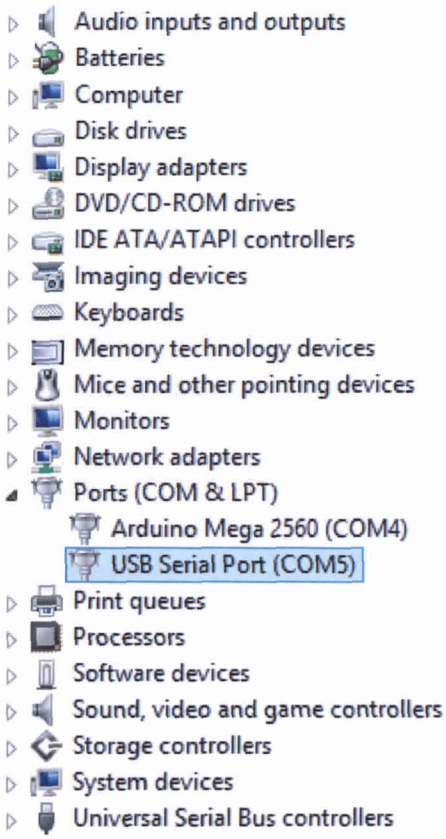
O PuTTY (figura 35) é uma ferramenta *open-source* capaz de realizar funções de cliente SSH e Telnet para as plataformas Windows e Unix. No caso presente foi utilizado como monitor de porta série, ou seja, para a visualização e monitorização dos bits que estão a ser transmitidos.



35 - Monitor de porta série PuTTY

A configuração do PuTTy passa por considerar 3 parâmetros muito importantes. Estes estão assinalados a vermelho na figura 35.

Primeiro, a linha *"Serial Line"* deverá ser preenchida pela porta Série que se encontra ligada a uma interface *miniUSB* (usada para programar). No caso de se desconhecer qual é esta porta, dever-se-á consultar o *Device Manager* do *host* da sessão de comunicação (figura 36) e escolher a porta correta.



36 - *Device Manager de um PC*

Pela análise da figura anterior, confirma-se a existência de duas portas utilizadas pelo *setup* utilizado para a comunicação entre o emulador de bomba de calor e o Arduino. A porta COM4 é utilizada para alimentar e programar o Arduino e a COM5 é utilizada pelo MATLAB para comunicar com o Arduino e emular então, a bomba de calor. Neste caso particular, dever-se-á preencher o campo "*Serial Line*" com COM4. Por fim, o terceiro parâmetro assinalado representa o *baudrate* ou o ritmo ao qual se irá fazer a troca de informação. Como a ECU comunica com 9600 de *baudrate*, este será o *baudrate* utilizado entre a comunicação MATLAB <-> Arduino. O *baudrate* de 115200 será o utilizado no PuTTy visto que tanto o PC como o Arduino suportam este *baudrate*.

3.7. Algoritmo de emulação

Pelo que foi dito em secções anteriores, a elaboração de um algoritmo foi algo extremamente importante de modo a emular-se a resposta de uma ECU, sendo capaz de a substituir completamente. Assim, seria possível elaborar um *sketch* que pudesse ser utilizado tanto para comunicar com uma ECU real como com a ECU virtual simulada em MATLAB.

Especificou-se no script desenvolvido em MATLAB o formato uint8 para o envio de *bytes*. Do lado do Arduino utiliza-se uma função da classe Serial, *Serial.read()*, que permite ler 1 *byte* (8 bits) da porta série. Desta forma garante-se que ambas as partes envolvidas na comunicação processam a informação na porta série como caracteres de 8 bits. A forma como o Arduino representa esta informação ao utilizador pode ser especificada durante a sessão de impressão na porta série correspondente ao monitor série do Arduino (porta ligada ao PC para controlo) através da definição do segundo parâmetro da função *Serial.print*(*byte*, <formato>). Este parâmetro poderá ser definido como HEX (caracteres hexadecimal), DEC (número decimal), etc.

O script desenvolvido no MATLAB começou por seguir a versão documentada do protocolo 2.0. No entanto, tendo em conta as diferenças que se notaram durante a sessão de testes efetuada com a ECU fornecida, foi importante modificar o código MATLAB para que este seja considerado um emulador capaz de simular a ECU real. Assim, permitiu-se ao utilizador selecionar a versão do protocolo do emulador: a resposta tendo em conta a versão v2.0 documentada e a resposta tendo em conta a versão experimental, verificada com a ECU fornecida.

Tendo em conta a situação descrita anteriormente, descreve-se, em seguida, a análise dos 4 casos observados na comunicação com o ECU real implementado com uma *release* de *software* não final da bomba de calor, versão protocolar 2.0..

Os 4 testes considerados em que se encontraram diferenças foram:

1. Envio de um comando *broadcast* na situação em que a ECU se encontra ligada a uma bomba de calor (assumiu-se que se tivéssemos uma bomba de calor ligada à ECU, esta iria responder como documentado na v2.0., justificando-se a sua inclusão no datalogger);
2. Envio de um comando *broadcast* num caso em que a ECU não se encontra corretamente ligada à bomba de calor (assumida como a versão experimental que foi de facto observável);
3. Envio de um *GetParameter* numa situação em que a ECU não se encontra ligada a uma bomba de calor;
4. Envio de um *GetParameter* para um sistema em que a ECU se encontra ligada a uma bomba de calor.

Estes quatro casos resultaram da assunção de que a ECU iria responder de acordo com o formato documentado e conforme o tipo de trama recebida do Arduino. O *layout* da GUI desenvolvida em MATLAB foi pensado de modo a facilitar a visualização dos dados enviados.

De seguida, apresenta-se o *layout* do script desenvolvido, explicando-se como se processa a comunicação, que tipos de processos estão envolvidos nos cálculos dos valores e a que tramas de dados correspondem cada um dos campos disponíveis ao utilizador através da GUI. Posteriormente, discute-se a forma como a GUI interage com o código MATLAB.

De forma a gerar valores típicos que a ECU obtém da bomba de calor, recorreu-se a algoritmos de geração aleatória de valores, sendo a trama final a ser devolvida pelo MATLAB resultante da concatenação dos *bytes* necessários à validação da trama (*startFrame*, *endFrame*, *checkSum*, etc.) com os *bytes* de dados gerados. A imagem seguinte evidencia como pode ser obtido um número hexadecimal de 1 *byte* no MATLAB.

```
S=2;
hex2dec(sprintf('%02x', uint32(rand(1,ceil(S/8)) * 99)+100))
outdec1 = ts1(1:S);
outdec1=upper(outdec1);
```

37 - Como Gerar dados aleatórios dentro de uma gama no MATLAB

Neste caso particular, considerou-se que Outdec1 poderá tomar valores entre 100 e 199 (formato decimal) ou 0x64 e 0xC7.

No fim, a trama é construída e enviada através do comando *fwrite* (s,datasend,'uint8'), em que 's' representa a porta série escolhida, '*datasend*' o *array* com os caracteres hexadecimais e *uint8* o tipo de dados a enviar.

A GUI criada para simular um ECU é invocada através da seguinte função:

- *function [] = myfuntester(porta,baud,operation);*

A *porta* terá que ser escolhida tendo em conta a nomenclatura 'COMX' em que X representa o número da porta série do *host* do MATLAB. A porta a utilizar é, fazendo referência à figura 35, a "USB Serial Port (COM5)". O *baudrate* irá ser configurado como o máximo que a ECU real suporta, que é de 9600bps. Relativamente ao modo de operação, esta função considera duas situações: A primeira, *operation=0*, representa o início da sessão de comunicação com a abertura da COM5. Com *operation=1*, esta sessão é fechada. Este parâmetro é necessário pois esta função baseia-se num ciclo infinito de comunicação e para que esta seja interrompida, o utilizador terá que manualmente interromper a sessão, com CTRL+C. Assim, para que o utilizador retome a sessão, terá antes que fechar a porta série. Se não o fizer, este será notificado com um erro (figura 38), indicando que a porta já se encontra a ser utilizada.

```
>> myfuntester('COM5', 9600, 0)
Error using serial/fopen (line 72)
Open failed: Port: COM5 is not available. Available ports:
COM4.
Use INSTRFIND to determine if other instrument objects are
connected to the requested device.

Error in myfuntester (line 12)
fopen(s);
```

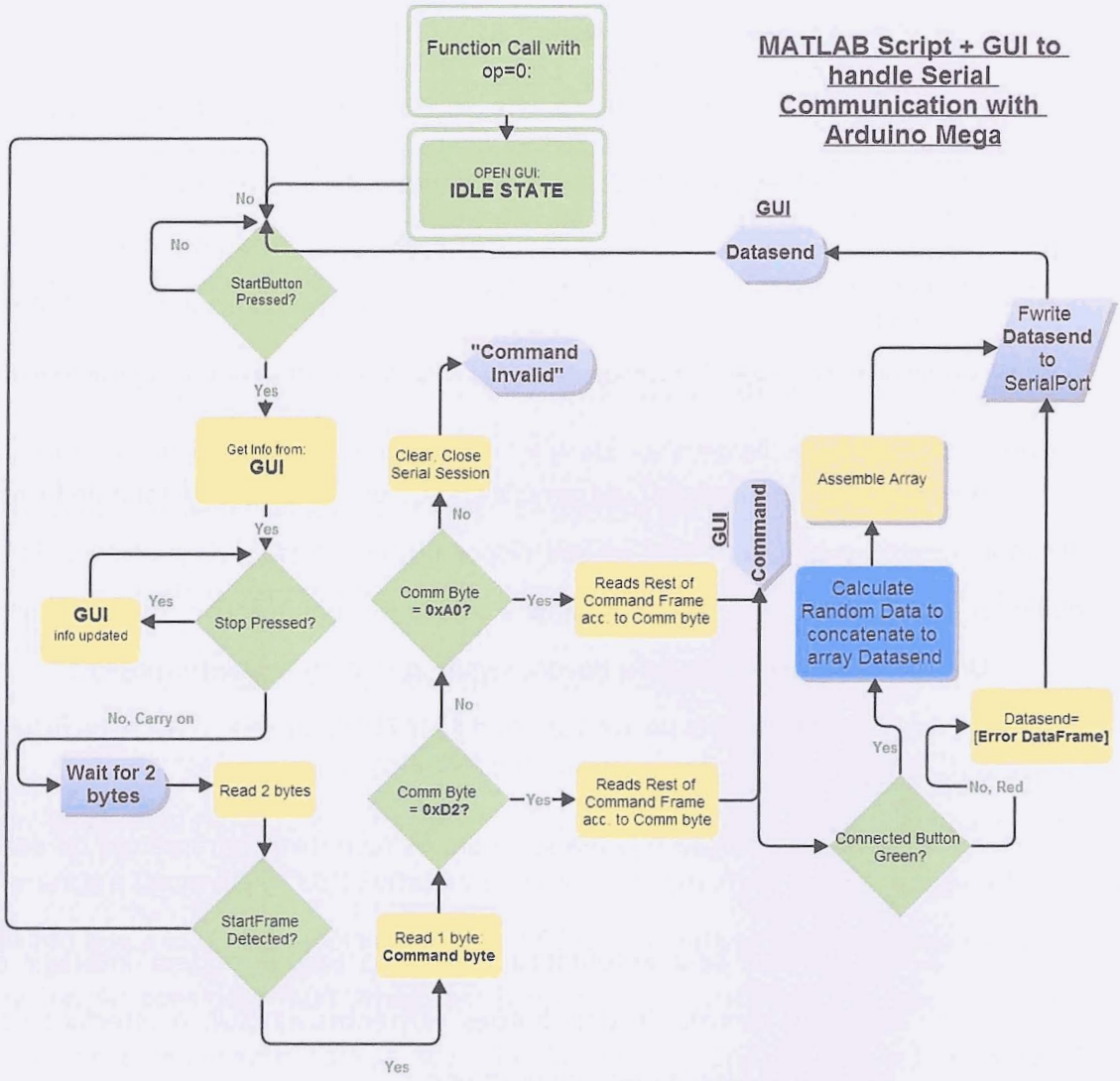
38 - Erro que ilustra a situação da interrupção da sessão Arduino <-> MATLAB via CTRL+C ou fecho da GUI

A versão final do emulador, desenvolvida posteriormente, funciona de forma mais autónoma, onde o utilizador tem ao seu dispor um botão *Start/Stop* através do qual, a qualquer altura, poderá terminar e retomar a sessão de comunicação com o Arduino.

Uma sessão de comunicação deverá seguir, então, os seguintes passos:

1. Incluir os scripts no *workspace* do MATLAB, ou seja, colocar os ficheiros no mesmo diretório;
2. Invocar a função *myfuntester* com os respetivos parâmetros de entrada e *operation=0*;
3. A GUI abrir-se-á automaticamente e o utilizar poderá interagir com o MATLAB através de dois botões existentes na GUI. A interface será em seguida analisada mais em pormenor;
4. A sessão poderá ser interrompida através destes mesmos botões. No entanto, se a janela da GUI for fechada, o utilizador terá que realizar o ponto 5, obrigatoriamente;
5. Invocar a função *myfuntester* com *operation=1* após a interrupção da sessão.

Relativamente ao funcionamento interno da função de emulação da ECU, apresenta-se na figura 39 um fluxograma ilustrativo da sua organização.



39 - Fluxograma do Script MATLAB + Interação com GUI

A sessão de comunicação é aberta quando se invoca a função da forma referida anteriormente. A GUI é inicializada e a função fica à espera, através de *polling* da porta série, que o utilizador inicialize a sessão de comunicação ao clicar no botão START. A cada botão existente está associado uma *flag* que é analisada no primeiro período de decisão do fluxograma representado acima. A função entra, então, no seu *loop* principal onde em cada ciclo completado com sucesso, uma resposta é enviada via porta série pela porta COM. Assim torna-se necessário fornecer uma forma de interromper este loop quando necessário, apresentado no fluxograma como o segundo momento de decisão, que irá

por dois botões, cujas funções já foram referidas anteriormente. Os dois últimos campos marcados representam os dados que chegam ao MATLAB (Ponto 3) e a trama construída pelo MATLAB como resposta ao comando (Ponto 4).

A existência destes dois botões traz quatro estados possíveis de resposta do sistema, tal como indicado na tabela apresentada de seguida:

	Start Button = OFF	**Start Button=ON**
Connected Button= OFF	Sistema em pausa	Envio de tramas indicativas da inexistência de uma aplicação ligada à ECU.
Connected Button=ON	Sistema em pausa	Envio de tramas indicativas da presença de uma aplicação ligada à ECU.

Tabela 4 – Combinações possíveis de estados do sistema

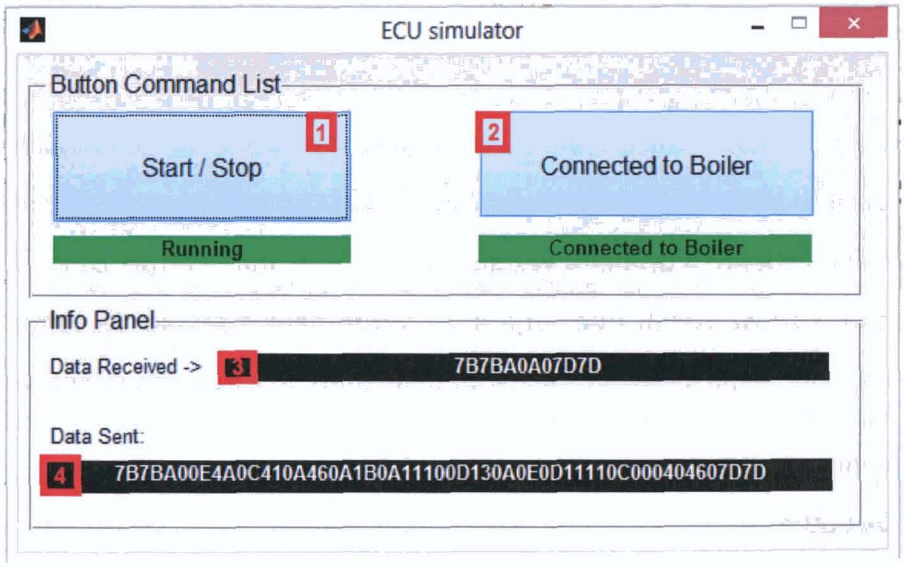
Com o *Start Button* em *OFF* o sistema não está a comunicar, pelo que, visualmente, o utilizador vê a última trama recebida e a última enviada para o Arduino (no caso antecedente à primeira vez que se clicar em *Start Button*, os campos 3 e 4 estão obviamente vazios). Assim que surja a luz verde para a sessão de comunicação (botão *Start*), o tipo de resposta irá alterar-se, dependendo do estado do *Connected Button*.

A figura 40 mostra uma das situações descritas no parágrafo anterior, em que ambos os botões estão ON, e que resulta num envio de uma resposta a um comando de dados *broadcast*. Para esta mesma situação, ao nível das flags ativas, uma trama *singleCast* é enviada da forma representada na figura 41:

reter o fluxo da função até que o utilizador volte a ativar a flag. O MATLAB entra, então, na fase em que aguarda a existência de *bytes* no *buffer* da porta série. Começa, assim, o processo de leitura e validação de dados. Primeiro terá que ser detetada a presença de um startFrame. Até que isso aconteça, a função lê e descarta todos os *bytes* existentes. Assim que a dupla chaveta é lida, inicia-se então a descodificação do comando recebido. *A posteriori*, decide, de acordo com o campo *Command*, o tamanho da trama que terá que construir, constrói a referida trama e escreve os dados na porta série.

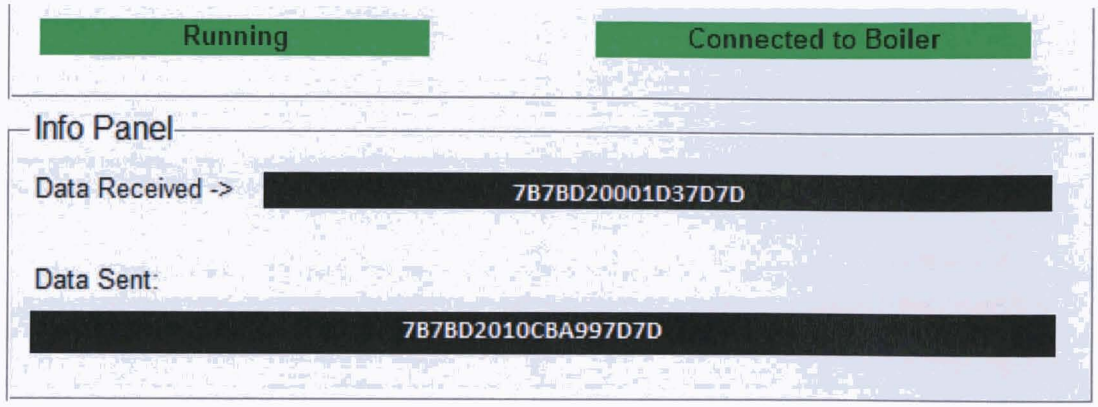
A forma com que este script *myfuntester* se relaciona com a GUI é através da existência de uma linha de comando que permite a importação do estado atual das variáveis.

A figura 40 apresenta a layout da GUI desenvolvida. Esta GUI é extremamente simples e tem como funções principais as de mostrar ao utilizador o fluxo de informação entre MATLAB e o Arduino, e por outro lado, selecionar entre as 2 opções relativamente à existência (ou não) de uma aplicação ligada à ECU (situação que surgiu devido a razões já esclarecidas anteriormente). O botão referenciado pelo ponto 2 da figura 40 permite escolher precisamente entre estas duas opções.



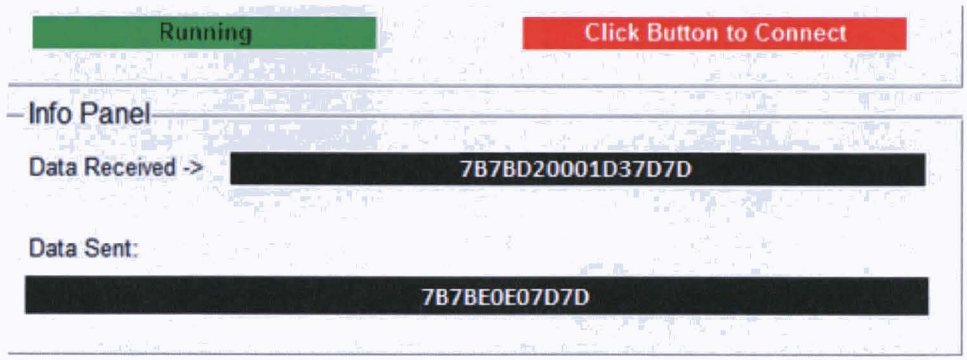
40 - ECU Emulator GUI

Como se pode ver pela figura 40, o emulador é constituído por 4 pontos essenciais de interação com o utilizador. Primeiramente surge um painel de Comandos, constituídos



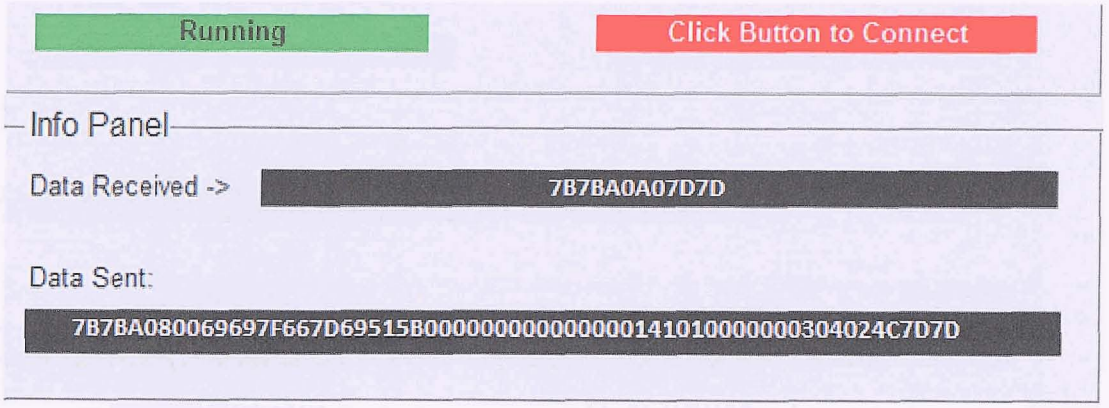
41 - Caso de um GetParameter na situação "ECU Connected to Boiler"

A situação descrita na figura 42 ilustra a situação onde o MATLAB recebeu um pedido de informação isolado (*singlecast*). Consultando o estado dos botões, infere-se que a sessão de comunicação está aberta mas que não existe aplicação ligada à ECU, pelo que não há valores reais para enviar. Assim, a trama enviada toma o formato já documentado anteriormente e apresentado na figura 21, dando origem à situação representada na figura 42.



42 - Caso de um getParameter na situação "ECU Not Connected to Boiler"

A figura 43 representa o ambiente idêntico ao da figura 42 com a particularidade de aqui se estar diante de um pedido de informação *broadcast*.



43 - Caso de um get BroadCast na situação "ECU Not Connected to Boiler"

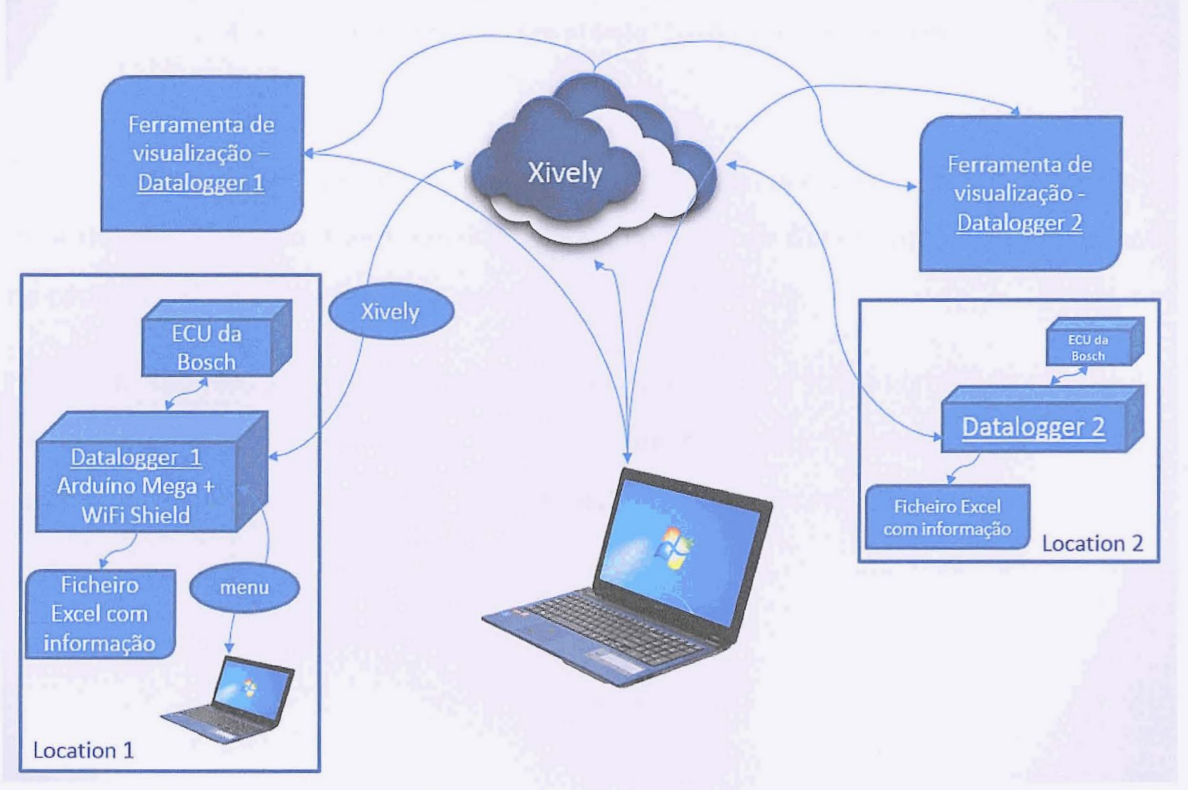
Para se obter estes dados do lado do Arduino foi necessário construir um *sketch* no Arduino que enviasse periodicamente um comando de *GetParameter* e um comando de pedido de dados *broadcast* via porta série para o MATLAB.

No capítulo seguinte irá ser abordado em detalhe o script elaborado no Arduino correspondente à versão final do datalogger desenvolvido, e apresenta-se no último capítulo os resultados das sessões de comunicação entre o Arduino e a ECU.

Capítulo IV

4. O dispositivo de *data logging*

O dispositivo de *data logging* envolve muitas funcionalidades. Esta secção descreve como foram cumpridas cada uma das funcionalidades requeridas para este datalogger. A figura 44 representa o panorama geral das funcionalidades idealizadas e implementadas para o sistema de *data logging* em causa.



44 – Panorama geral do sistema de data logging considerado

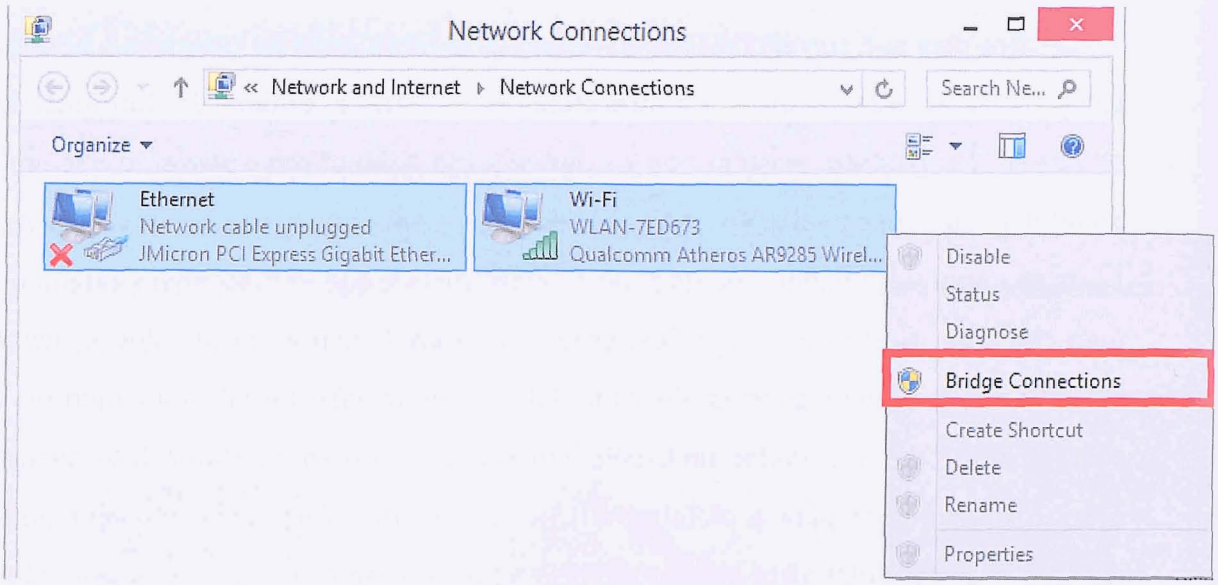
Pela análise da figura 44 evidenciam-se as seguintes características:

- Ficheiro Excel a efetuar *data logging* da informação;
- Menu *inloco* para efeitos de controlo local e de debug;
- Comunicação com a plataforma Xively;
- Centralização de informação de vários dataloggers em contextos temporais diferentes na mesma conta de utilizador do Xively;
- Ferramenta de visualização de dados individualizada (por cada datalogger).

Nos dias que correm, o Arduino possui uma gama vasta de *shields* que podem ser ligadas diretamente à placa de prototipagem e que a dotam de funcionalidades adicionais. No entanto, existem opções que não são compatíveis entre si, o que limita a quantidade de funcionalidades que se podem ter em simultâneo no Arduino. Assim, torna-se importante definir que tipos de características é que se pretendem para o que se quer construir. Neste caso específico, pretendia-se implementar um datalogger *wireless*. No entanto, por uma questão de comodidade, numa fase inicial, implementou-se a comunicação com o servidor utilizando um *shield* de Ethernet. Numa fase avançada, optou-se por migrar para a plataforma com *shield* WiFi. Verificou-se não ser possível juntar os dois módulos visto que ambos reservam os mesmos pinos de comunicação (via SPI - para comunicar com a *board* principal) e utilizam o mesmo PIN como *shield select* (tanto no caso da seleção do próprio *shield*, como também no de seleção do cartão SD), pelo que o Arduino não consegue alternar a comunicação entre os dois *shields.* Assim, este capítulo começa por referir os diferentes módulos de comunicação relevantes para a dissertação compatíveis com o Arduíno Mega.

4.1. Arduino Ethernet Shield

O *shield* Ethernet do Arduino efetua a ligação do Arduino à Internet de uma forma rápida. Depois de se efetuar a ligação do *shield* à *board* do Arduino, a ligação à internet é estabelecida através de um cabo RJ45. Por um lado tem-se o RJ45 ligado à porta Ethernet do *shield* e do outro, a ligação poderá ser feita por duas formas: ligar a um *access point* diretamente ou utilizar um PC como intermediário da ligação. Neste último caso, o computador terá que ter acesso à internet, seja via *WiFi* ou por cabo Ethernet (deverá ter uma porta Ethernet extra, neste caso). Após se efetuar as ligações físicas necessárias, foi necessário efetuar algumas alterações no menu de rede no computador. No caso do Windows 8 deve-se aceder ao menu *Network and Internet -> Network and Sharing Center*, e escolher-se a opção *Change adapter settings*. O menu apresentado na figura 45 ilustra o submenu que irá ser exibido.



45 - Como efetuar o bridge entre duas ligação de rede

Uma ponte ou *bridge* entre estas duas tecnologias atribui ao Arduino a capacidade de se ligar à internet, utilizando o PC como *access point*. Este *shield*, que opera a 5 V (fornecidos pelo Arduino), tem como constituinte principal o Controlador Ethernet W5100 [28]. Este Controlador, no mercado desde 2006, possui um *buffer* interno de 16KB, é largamente utilizado em aplicações *embedded* e atribui ao *shield* a capacidade para lidar com protocolos UDP e TCP.

O Arduino Ethernet, além de permitir velocidades de 10/100 Mbps, atribui outra característica importante ao sistema de *data logging* com a introdução de capacidade de armazenamento. Esta deve-se à presença de um SD *card slot* no *shield*. A inclusão deste *shield* num Arduino Mega implica a inutilização dos pinos 50, 51 e 52 do Arduino como input: estes deverão ser mantidos como output para que a interface SPI seja utilizável. O barramento SPI é partilhado entre o W5100 e o SD Card para efeitos de comunicação com a *board* principal: surge, então, a necessidade de se distinguir e separar o fluxo dos dados lidos pelo Arduino. Assim, utiliza-se o pino 10 para selecionar o W5100 (abrir uma sessão de comunicação com este torna-se possível) e o pino 4 para selecionar o cartão SD. O pino SS do Mega (pino 53), apesar de não ser utilizado para selecionar o W5100 (é a função do pino 10), deverá também ser mantido como output para que a comunicação via SPI seja possível. Outro ponto importante a referir é o modo que especifica as

camadas físicas e a subcamada MAC do modelo OSI: o Arduino Ethernet é compatível com o IEEE 802.3u e 802.3, que consistem nas especificações *standards* de comunicação Ethernet, mas cuja análise está fora do âmbito deste documento.

4.2. *Arduino WiFi Shield vs. Ethernet Shield vs. GSM Shield*

Após a implementação de um sistema funcional de *data logging* de informação utilizando o Arduino Mega e o Ethernet Shield, analisou-se as diversas hipóteses relativamente às plataformas disponíveis para atribuir ao sistema a capacidade de ligação à Internet, sem fios. No caso do Arduino existem algumas formas distintas de o fazer:

- Arduino Ethernet + Wireless Access Point
- Arduino GSM *Shield*
- Arduino WiFi *Shield*

A primeira é facilmente realizável sendo das três a mais barata. No entanto, não foi considerada como uma opção prática já que implicaria um *setup* mais complexo e com dimensões superiores, envolvendo, para além da programação de um microcontrolador, a configuração de um *access point*. Ora tendo em conta o objetivo deste datalogger, que é integrar um dispositivo comercial no mercado em grande escala, esta opção foi, assim, posta de parte.

A escolha entre o GSM *Shield* e o WiFi foi bastante evidente. Considerando os custos de utilização, o Arduino WiFi *Shield* destaca-se visto que:

- Preço do *shield* é mais baixo (sensivelmente 15 euros)
- O *GSM Shield* possui custos inerentes à ativação e manutenção de um tarifário para o envio dos dados via Módulo GSM (Cartão SIM necessário). Isto não acontece com o WiFi, visto que poderá simplesmente servir-se uma rede doméstica. Assim, é razoável inferir que não há custos provenientes do envio periódico de dados do datalogger: representam, nos

dias que correm, uma percentagem irrisória dos limites de tráfego (nos poucos casos em que ainda existem) presentes nos pacotes de Internet.

Por outro lado, o Arduino GSM *Shield* atribuiria uma característica muito interessante ao datalogger: a independência. Através da utilização da rede móvel para a transmissão de dados, elimina-se a necessidade da presença de um WiFi *access point* nas redondezas do datalogger que possibilite o envio dos pacotes de dados para o *webserver*.

Assim, ponderando mais os custos das comunicações, a escolha de um módulo que viabilizasse a ligação à Internet de forma *wireless* recaiu sobre o *WiFi Shield.*

Tal como o Arduino *Ethernet Shield,* o *WiFi Shield* utiliza o barramento SPI para comunicar com a *board* principal. Opera igualmente a 5V e como *standard* de comunicação utiliza as especificações 802.11 e suporta as versões b/n do mesmo. Quanto ao nível de encriptação, oferece suporte para as mais utilizadas em redes domésticas, a WEP e WPA2 pessoal. Como output, possui também uns conectores FTDI para serial *debugging* do *WiFi Shield* e uma porta Mini-USB para efetuar updates do firmware do Chip W5100. Como principais integrantes destaca-se o HDG104 [29], um SiP de muito baixo consumo energético. Tal como o Ethernet *Shield*, também oferece suporte para UDP e TCP através da utilização de um 32UC3. Por fim, possui embutido um micro SD *Slot* e é sobre este constituinte que o documento trata de seguida.

4.3. Formatação Necessária do SD Card

A incorporação de um cartão SD deve ter em conta algumas especificidades:

- Estar previamente formatado como FAT16 ou FAT32.
- O onboard slot suporta micro SDHC e micro SD

- Devido à sua formatação, os nomes que se dão aos ficheiros têm que se submeter à convenção 8.3. Além disso, destaque para o tamanho máximo de 8 caracteres para o nome dos ficheiros. [30]

Este cartão SD irá conter um ficheiro de extensão .csv que irá armazenar toda a informação de broadcast recolhida pelo datalogger.

4.4. *Uso da Plataforma REST Xively*

A *Internet of Things*, já discutida no início deste documento, promete revolucionar a relação entre as pessoas e o resto do mundo, induzindo uma interação muito simplificada entre dispositivos e seres humanos. O *Xively*, que foi o resultado de uma transformação e revolução da antiga plataforma *COSM*, apareceu num contexto em que para transformar uma grande ideia, com grande potencial inovador e económico para o mundo real, requer-se a utilização de plataformas, ferramentas, e serviços que geralmente estão associados à aplicação de muito tempo, dinheiro e *know-how*, que nem sempre são fáceis de obter. O aparecimento do Xively veio solucionar e facilitar esta situação: este providencia a plataforma e todos os serviços apresentados acima que simplificam e aceleram a criação de produtos competitivos que podem entrar no mercado muito mais facilmente. Uma pessoa criativa, por mais ideias inovadores que possua, se não tiver algo em concreto para apresentar a investidores, verá mais complexo e difícil a tarefa de obter financiamento para os seus projetos. Este é mais um aspeto em que o Xively intervém ativamente: permite aos criadores focarem o seu intelecto na inovação, na idealização de um projeto ou produto. Consiste, segundo os criadores, na primeira *IoT Cloud* pública do Mundo [31]: engloba um conjunto de serviços, focados diretamente na *Internet Of Things* que providenciam o armazenamento de dados em tempo real, a partilha e comunicação com inúmeros serviços na *cloud* e a comunicação bidirecional entre uma máquina e um utilizador, remotamente. Esta é, de facto, a palavra-chave: a possibilidade de consultar dados, de modificar parâmetros relativos ao funcionamento de uma aplicação de uma forma remota. No entanto, em alternativa, poder-se-ia projetar e implementar de um *webserver* criado especificamente para monitorizar e controlar

bombas de calor. A escolha do Xively em detrimento desta é facilmente justificável: cada bomba de calor, ou cada cliente, terá que ter uma página dedicada que permita que o utilizador consulte dados da sua bomba de calor. Isto implica um *webserver* por cliente ou que haja métodos de gestão de acesso às respetivas bombas de calor. A atribuição de custos à utilização de serviços de *webhosting* ou o aumento da complexidade inerente à utilização dos métodos de gestão de utilizadores, aumenta as vantagens de utilização do Xively.

De facto, o Xively possui, também, uma das características principais do Arduino, a flexibilidade. Juntos formam uma poderosa plataforma de prototipagem capaz de, em pouco tempo, levar uma ideia para a *cloud*, totalmente funcional. Existem inúmeros projetos que devem muito a esta plataforma: destaque por exemplo para a Elektron [24]. Ao utilizar esta plataforma, a Elektron tornou-se capaz de desenvolver rapidamente aplicações únicas no mundo da *IoT*, e concentram o seu trabalho aplicacional nas áreas de monitorização e controlo de serviços. O uso do Xively no mundo empresarial poderá levar à utilização do serviço *Premium* com inúmeras funcionalidades, em que não há limitações, por exemplo, no número de chamadas API por minuto. No contexto desta dissertação, refere-se a aplicabilidade do serviço base que a Xively oferece.

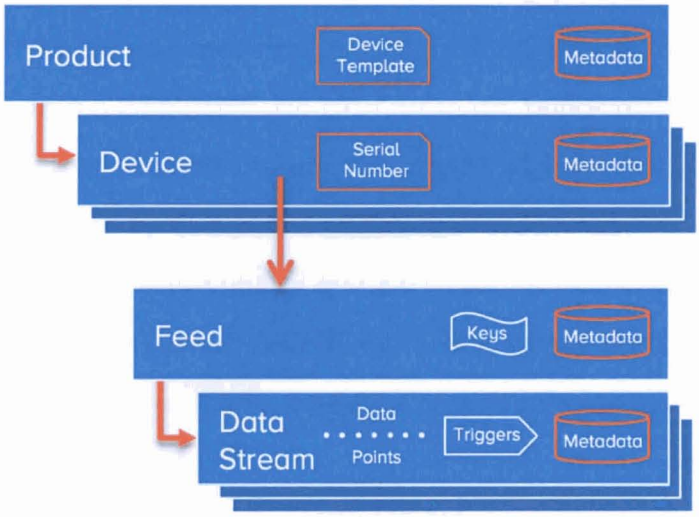
Tal como o Arduino, a Xively fornece bibliotecas de utilização livre e modificáveis que, juntamente com tutoriais e documentação, permitem uma ligação de um produto com o Xively, independentemente da plataforma utilizada (dada a diversidade de bibliotecas).

4.5. Xively API

O Xively é também denominado de uma *PaaS*, uma "*Platform as a Service*" para a *Internet Of Things*. Esta simplifica a interligação entre dispositivos, dados, pessoas e locais, acelerando a criação de soluções interessantes que transformam a forma como as pessoas interagem com o que as rodeia.

O *Xively Cloud Services*^{TM} oferece mensagens, armazenamento de dados, e comunicação bidirecional entre duas entidades através da *Xively API*. No caso do datalogger remoto, a utilização desta *API* teve o intuito de permitir o envio de comandos via HTTP e a obtenção e disponibilização das respostas em campos específicos da *Xively* API.

A relação entre os dados gerados por um dispositivo e a sua representação/armazenamento num dos campos da Xively API é representada pelo diagrama da figura 46.



46 – A Hierarquia de variáveis na Xively API. Fonte:[31]

A Xively API permite, após um registo autenticado de um utilizador, a criação de um *Produto*. Este *Produto*, que representa uma ideia, poderá ser constituído por vários *Dispositivos*. Esta nomenclatura, no caso do projeto corrente, poderá ser facilmente exemplificada através da utilização do datalogger remoto como *Produto* e de datalogger como *Dispositivo* (cada um identificado por um número série). Dentro de cada dispositivo ou datalogger, existem os dados, denominados por *datastreams*. Geralmente utiliza-se o termo *datastream* para referir um *array* que possui um ou mais elementos. Depois, a associação deste *datastream* a uma chave *(key)* faz com que seja possível referenciar o *array* através de um ID único, que irá ser utilizado pelos intervenientes da sessão de

comunicação (Arduino e Xively) para efetivar a comunicação entre eles. A cada valor lido de um *datastream* dá-se o nome de *datapoint*.

Esta API, aquando o desenvolvimento do datalogger, evoluiu de *COSM* para Xively, pelo que muitas das funcionalidades descritas acima só foram contempladas posteriormente e todos os *Feeds* (conjunto de datastreams relativos a um dispositivo) existentes até à data foram simplesmente denominados por *Legacy Feeds* e mantidos como tal. A figura 47 mostra um exemplo de uma API do Xively.

47 – Xively API – Legacy Feed

Terminologia associada à Xively API

A Xively API segue em conformidade com os princípios da *Representational State Transfer* (REST). O acesso REST utiliza comandos *HTTP* para determinar que ação tomar relativamente a um *datapoint*. Entre estes, existem 4 possibilidades:

- *GET*: obtém o valor atual do objeto de dados a que se refere o *datapoint*;
- *PUT*: altera o valor atual do objeto de dados;
- *POST*: cria um novo objeto de dados;
- *DELETE*: apaga um objeto de dados existente.

Estes comandos podem ser utilizados com os diversos recursos presentes na API do Xively. São eles os Produtos (*Products*), os Dispositivos (*Devices*), as Chaves (*Keys*), os *Feeds*, os *Triggers*, os *Datastreams* e os *Datapoints*. Segue-se então a análise em detalhe dos recursos mais importantes: os *Datastreams*, as *Keys* e os *Feeds*.

Um *datastream* é um canal bidirecional de comunicação que permite a troca de dados entre o Xively e algo associado a este, seja um dispositivo, uma aplicação ou um serviço. Poderá constituir um elemento ou um *array* de vários elementos. Cada elemento do *array datastream* representa um atributo específico, capaz de fornecer um tipo de informação. Estas variáveis podem ser apagadas, criadas automaticamente, ou até editadas através de alguns comandos, tanto do lado do Xively, como do lado da aplicação ou plataforma que está a comunicar. No caso presente, temos o Arduino que associa os dados de um dispositivo, uma ECU, a um *datastream* (1 key, 1 Dispositivo), sendo que este *datastream* é um array que possui toda a gama de valores que a ECU tem para oferecer. Adicionalmente é aconselhável que a criação dos datastreams seja feita através deste (e não através da API), resultando numa melhor gestão e eficiência do sistema de *data logging*. O *array* de *datastreams*, na API, é associado a um FEED_ID, que irá conter os *datapoints* num determinado instante. Adicionalmente, o sítio onde o FEED_ID está inserido, ou seja, a API que contém este FEED_ID, será identificado pela *Key*.

A forma como o Arduino comunica com o Xively é muito simplificada pela existência de uma biblioteca oficial do Arduino que traduz os comandos HTTP referidos acima para linguagem explícita e entendível a qualquer programador de Arduino, com ou sem conhecimentos em HTTP.

Inicialmente, a primeira secção de código que o programador terá que incluir no seu script terá que explicitar qual a chave *(key)* que contém o *datastream* na API que será afetada pelo script. Uma chave é definida por um conjunto de números e letras que são aglomerados num *array* de *chars*. Uma API *Key* permite a leitura e escrita de *datastreams* na API durante uma sessão de comunicação. O FEED_ID representa o "sítio" onde os dados irão ser guardados.

Os comandos HTTP são traduzidos para a linguagem de programação Arduino consoante o tipo de variável ou *datastream* que se pretende. Estes poderão ser inteiros, *Strings*, *arrays* de *chars* ou *floats*. Para o caso genérico de um número real, a forma de obter o *datapoint* mais recente traduz-se num comando *datastreams*[x].getFloat() e a forma de efetuar um *push* a um *datapoint* de um real traduz-se num comando *datastreams*[x].*setFloat*(Y), em que x é o índex que o elemento que se pretende alterar ocupa no array e Y o valor que se impõe ao elemento. Uma outra noção importante relativa a esta biblioteca é a capacidade de avaliar-se uma operação de leitura ou escrita de um *datastream* (*array* de *datapoints*). No caso de uma leitura ou escrita com sucesso (HTTP1.0 200, indicativo de um OK) há a devolução de um 0. Um valor negativo é indicativo de um erro (HTTP1.0 40x) que poderá ser traduzido da seguinte forma [32]:

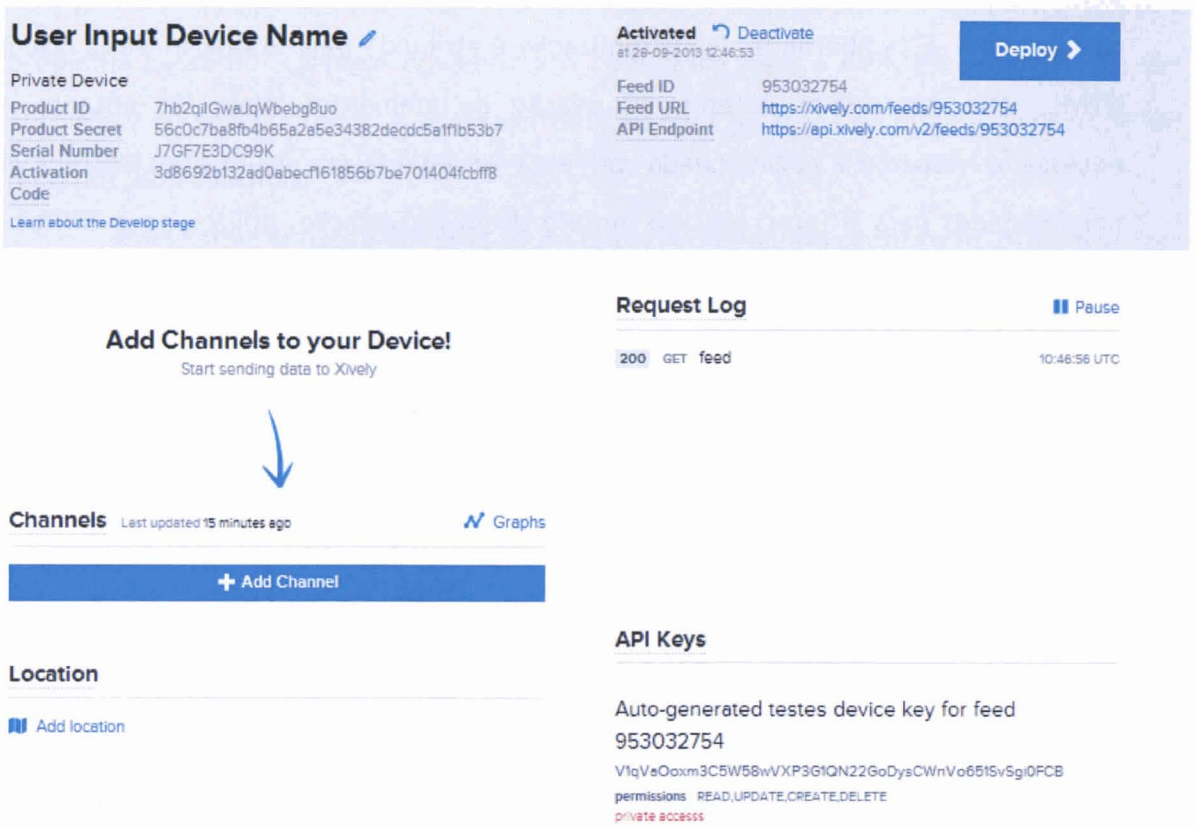
- -1: Falha na ligação à API;
- -2: Um método HTTP foi invocado erradamente;
- -3: Conexão à API ultrapassou o tempo máximo *(timed-out).*
- -4 : Resposta do Servidor inválida ou inesperada

Para explicar melhor o estabelecimento de um produto no *Xively API* enumeram-se em seguida os passos da criação de um dispositivo até se obter o sistema representado na figura 48.

1. Numa primeira fase, estabelece-se um *"development device"*. Nesta altura, ter-se-á que definir um nome, uma descrição, e definir restrições de privacidade a implementar no dispositivo (visível a todos ou privado – só visível pelo administrador da conta).
2. A definição do ponto 1 gerará automaticamente um ID referente ao Produto ao qual este *device* pertence e um *serial number* referente ao dispositivo acabado de criar. Para além disso, um FEED_ID é gerado, sendo possível aceder ao respetivo *feed* onde irão ser guardados os *datastreams* gerados por este dispositivo através de https://xively.com/feeds/<FEED_ID>. Adicionalmente, uma *APIKEY* é

gerada automaticamente, atribuindo direitos ao utilizador para escrita, leitura, criação e remoção de *datastreams* do *Feed*.

3. Neste ponto é possível adicionar um canal ou *datastreams* via API. No entanto, é aconselhável a definição do tipo de dados que se pretende através do *sketch* do Arduino. Os datapoints definidos nesse ambiente serão automaticamente atualizados para a Xively API
4. A obtenção da API KEY, do FEED_ID e a sua inclusão no *sketch* Arduino, nos campos respetivos, permitirá associar este dispositivo criado na API ao Arduino.



48 – Exemplo de um dispositivo criado no Xively

Existem vários serviços associados a esta aplicação. Um deles advém da capacidade de cada *datapoint* ser associado a um *trigger* ou uma ação concreta: por exemplo, se um dado valor inteiro ultrapassar determinado *threshold* será possível despoletar o envio de um *e-mail*. Este *add-on* extremamente útil da *Xively* é denominado

por Zapier, um serviço que automatiza tarefas entre aplicações Web. Através da ligação de um *trigger* do *Xively* a este serviço é possível, por exemplo, enviar um *tweet*, acionar uma chamada de voz ou enviar um SMS através do *Twilio* e interagir com várias outras aplicações web [24]. A forma como este serviço deteta o *trigger* é a mesma com que o Arduino comunica com o *Xively*: através de *HTTP posts*, visto que um *trigger* do *Xively* toma a forma de um *HTTP Post* para o Zapier. O tutorial presente na página da *Xively* acerca deste serviço é extremamente simples e permite a utilização do Zapier em alguns minutos.

Outro serviço que funciona como um *add-on* com algum potencial é a capacidade de se visualizar os dados numa plataforma externa de uma forma totalmente customizável. Esta liberdade de customização é atribuída pela utilização de JavaScript e HTML, uma ferramenta potente na criação de ambientes Web. No entanto, será necessário hospedar a página criada com esta ferramenta em algum sítio. Existem várias possibilidades para o fazer, mas no âmbito desta dissertação, utilizou-se o *GitHub*. No *Github* existe uma página associada ao *Xively* que, através da transposição do código aí presente para uma conta pessoal (ato denominado de *fork*) e da inserção da chave relativa ao *Datastream* que se pretende visualizar, faz com que a transposição de plataforma seja simples e, por conseguinte, obtém-se uma forma de visualização dos dados muito facilitada e customizável.

4.6. Cuidados a ter na Programação do Arduino

Este subcapítulo e os seguintes apresentam a parte relativa ao contributo do Arduino para o sistema de *data logging*. Para isso começa-se por explicar a resolução de um bug extremamente importante, presente há alguns anos na plataforma do Arduino Mega e, apesar de ter havido novas revisões da plataforma e muita discussão nos fóruns [33], nada foi alterado de modo a colmatar este problema.

Em qualquer ambiente de programação, uma boa forma de efetuar *debugging* de software é inserir algumas linhas que tenham algum impacto visual direto para o programador. Um bom exemplo é a biblioteca Serial. A inserção de um *Serial.println()* foi

e é muito usada no debug de *sketches* do Arduino, mas o uso de impressões para um monitor é algo facilmente abrangente a métodos de debugging de outras plataformas. No entanto, existem alguns caracteres que se devem evitar. A utilização de um *Serial.println*("!!!") induz o envio de 3 caracteres consecutivos para a porta série. Ora, o Arduino Mega considera este comando (quando enviado para a porta série 0) como um pedido para entrar no *monitor mode*. Isto faz com que o Arduino deixe de ser capaz de interpretar quando um novo código está a ser carregado para a memória. Este estado de funcionamento é indicado pelo piscar do LED 13 (built-in na board) ou mesmo pelo estado *ON* deste mesmo LED. Assim, o microcontrolador fica incapaz de responder a qualquer ação por parte do utilizador/programador. A solução encontrada para resolver o problema passa por carregar novamente o *bootloader* para o AVR, efetuando assim um *hard reset* no sistema todo ao se impor todas as configurações de origem. Este pode ser feito por várias formas.

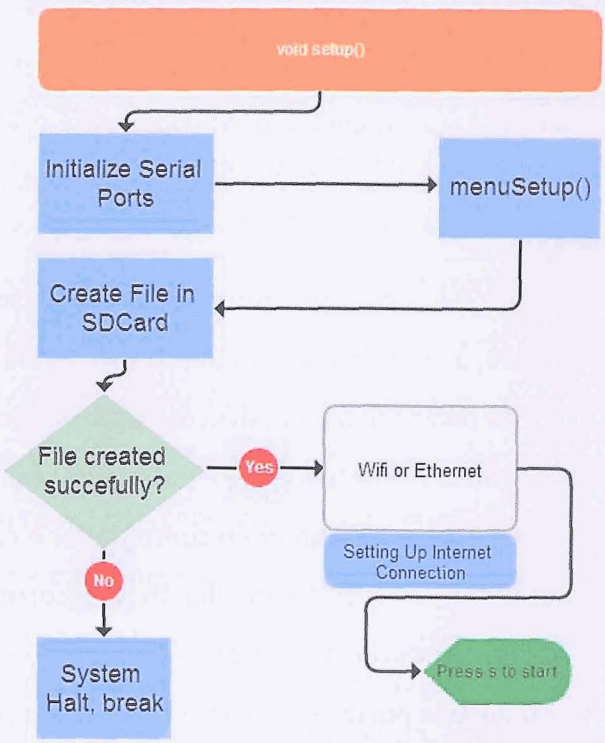
Uma forma consiste em utilizar um outro Arduino para servir como programador externo do *AVR*. No entanto, a solução mais simples passa por se utilizar um programador externo de AVR. Com este dispositivo é possível carregar um ficheiro .bin diretamente para o Arduino. É um processo transparente, e passa simplesmente por ligar os dois dispositivos via ICSP, ligar o *AVR programmer* ao computador e carregar o ficheiro .bin do *bootloader* para o computador através da escolha do item correto do menu do Arduino IDE. Depois do ambiente de reprogramação estar todo montado, resta utilizar o *software* apropriado para enviar o .bin via porta série para o *AVR Programmer*, que por sua vez irá programar o Arduino Mega de volta ao seu estado original. A figura 49 ilustra as ligações que são necessárias para permitir a reprogramação do processador de um Arduino.

49 - AVR Programmer + Arduino

4.7. Funções integrantes

Como já foi dito anteriormente, a estrutura do código num *sketch* Arduino contempla 2 funções principais, o *setup*() e o *loop*(). De seguida irá ser apresentado o que é que cada uma destas funções faz no caso particular do projeto em causa.

4.7.1. Setup()



50 – setup();

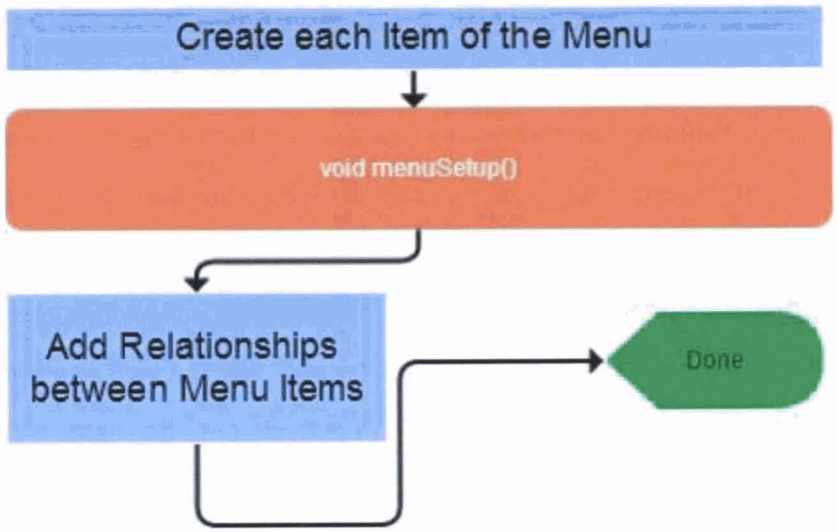
A análise do texto que se segue deverá ser feita em paralelo com a figura 50. A função *setup*() começa por inicializar as portas série, tanto a porta série responsável pela troca de informação entre computador e Arduino mas também a porta que interliga o Arduino com a ECU. Assim, inicializam-se as portas série, especificando o *baudrate* a que cada uma delas irá comunicar. Por uma questão organizacional, o *setup* do menu foi

incluído numa função à parte: o seu funcionamento irá ser descrito no contexto da descrição do menu, que irá ser feita em seguida.

Após este passo, a função trata de criar um ficheiro no cartão *SD* com um determinado nome e com determinadas permissões de escrita, leitura e remoção. Se o ficheiro for criado com sucesso a função continua. Se não, o sistema é interrompido e será necessário efetuar um reset para que este funcione corretamente. No caso positivo, segue-se a configuração da ligação à Internet. Todos as verificações e inicializações aqui efetuadas são visíveis pelo utilizador, já que existem alguns *prompts* de *strings* conforme é executado cada um dos passos, nomeadamente *"Connecting to Internet", "Initializing SD card"*, etc.

4.7.2. *menuSetup()*

O *menuSetup*(), tal como o nome indica, é responsável pela criação e configuração do menu. Este menu utiliza uma biblioteca específica e muito completa criada por Bill Greiman [34]. O seu funcionamento irá também ser descrito de uma forma mais abrangente ainda neste capítulo: por agora, irá ser descrito o funcionamento de *menuSetup*(). A figura 51 mostra as suas funcionalidades:



51 – menuSetup() – Estabelecimento de relações entre itens;

Cada Item do menu pode ser interpretado como um ponto que tem ligado a si vários nós segundo uma direção: cima, baixo, esquerda, direita. Estas direções permitem navegar pelo menu: navegar entre os itens que estão no mesmo nível (cima, baixo), navegar para itens que estão em níveis superiores (esquerda) ou para subníveis (direita). A relação entre estes níveis é feita no *menuSetup*(), ilustrado na figura **51**, onde se estabelece as relações entre todos os itens do menu. Antes desta função terão que ser definidos todos os objetos correspondentes aos itens do menu (caixas a azul na figura 52).

4.7.3. Menu()

A implementação deste tipo de menu foi importante, especialmente para efeitos de debug, já que permite obter informações acerca das atividades que o datalogger está a realizar em tempo real.

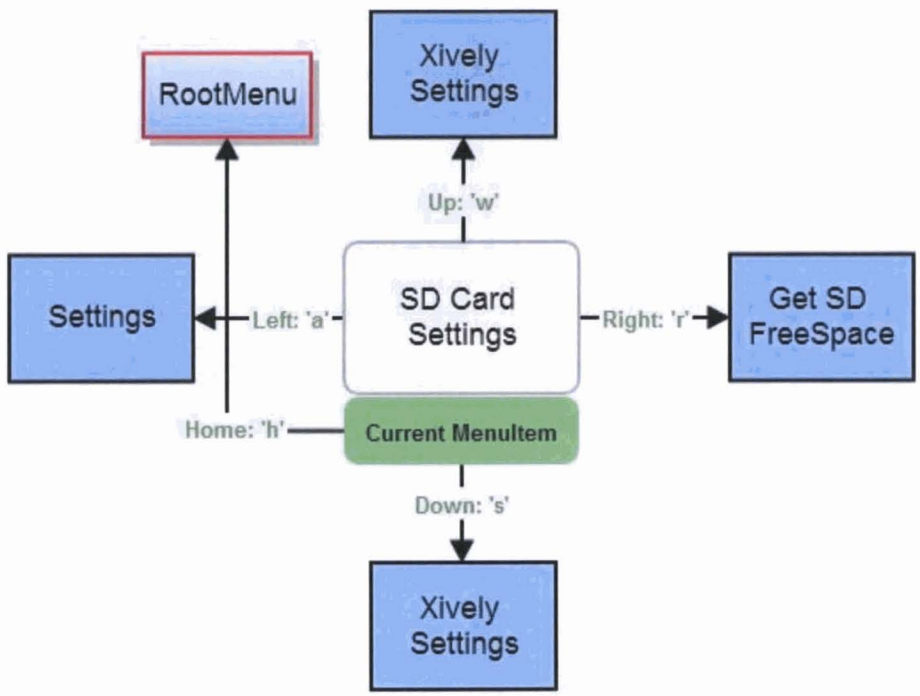
Tendo os itens inicializados e definidos, existem outras funções que ajudam na eficiência do menu. O Menu, denominado *MenuBackend* foi desenvolvido e integrado numa biblioteca com o mesmo nome por um entusiasta norueguês e foi modificado para efeitos de utilização com o sistema de data logging. Esta biblioteca, para além da criação de um tipo de dados *MenuBackend*, inclui a presença de duas outras funções: *menuUseEvent*() e *menuChangeEvent*(). A primeira, *menuUseEvent*() é a função que é invocada sempre que se opta por "utilizar (*use*)" um dos itens de menu, ou seja, que ações é que se tomam aquando a utilização de um item do menu. A segunda, *menuChangeEvent*() é onde se incluem todas as ações que ocorrem quando o utilizador abandona ou passa por um item (durante a navegação para subníveis): no caso concreto, é onde se efetua o display de todos os itens que existem no nível corrente para que se consiga saber que opções é que se tem ao nível da navegação. Para além destas funcionalidades foram também acrescentadas a este sistema do menu a opção de circular para o menu principal (representado pela saída dos subníveis até ao nível primário), e a função de imprimir o Menu todo.

O menu terá, então, que usar uma porta série para receber *inputs* do utilizador. Existem várias hipóteses de *input*. Antes de apresentar as formas de *input*, consideremos o menu:

- RootMenu

1. Options:
   1.1. Check DataTransfer
   1.2. Ask BroadData
2. Settings
   2.1. SD Card Settings;
      2.1.1. Get SD FreeSpace;
      2.1.2. Set SD Rec Rate;
   2.2. Xively Settings;
      2.2.1. Set Xively UpRate

Considerando que, em determinado momento, estamos no ponto *SD Card Settings* (2.1.1), as diversas possibilidades de navegação estão representadas na figura 52:



52 – Navegação entre menus/submenus

Como a imagem sugere, os *inputs* disponíveis ao utilizador do menu são os seguintes:

- W -> Circular no mesmo nível, deslocando-se para cima; funciona de maneira circular pelo que se o utilizador se encontrar no primeiro item (x.1), a escolha desta opção levá-lo-á para o último item no mesmo nível (x.2 neste caso particular)

- S -> Circular no mesmo nível, deslocando-se para baixo; funciona de maneira circular;
- D -> Circular para a direita, ou seja, entrar num subnível; não funciona de maneira circular;
- A -> Circular para a esquerda, ou seja, subir um nível; não funciona de maneira circular;
- H -> ("Home"). Circula para a esquerda as vezes que forem necessárias até entrar no menu *RootMenu*, um nível acima do referido como primário (1., 2.).

A zona do script que está atenta aos *inputs* do utilizador será o *loop*() que, ao repetir-se infinitamente, está também atento ao fluxo de dados das portas série. Deste modo, assegura-se que qualquer *input* do utilizador relativo a um comando é atendido assim que possível - até lá, o *buffer* de atendimento implementado por software, intrínseco ao Arduino, armazena os *bytes* recebidos.

A movimentação entre menus é feita pela associação da leitura de um *byte* da porta série (inserido pelo utilizador) e as respetivas ações presentes na estrutura 'menu' *(menu.moveUp(), menu.moveDown(), menu.moveLeft(), menu.moveRight()* e *menu.use()).*

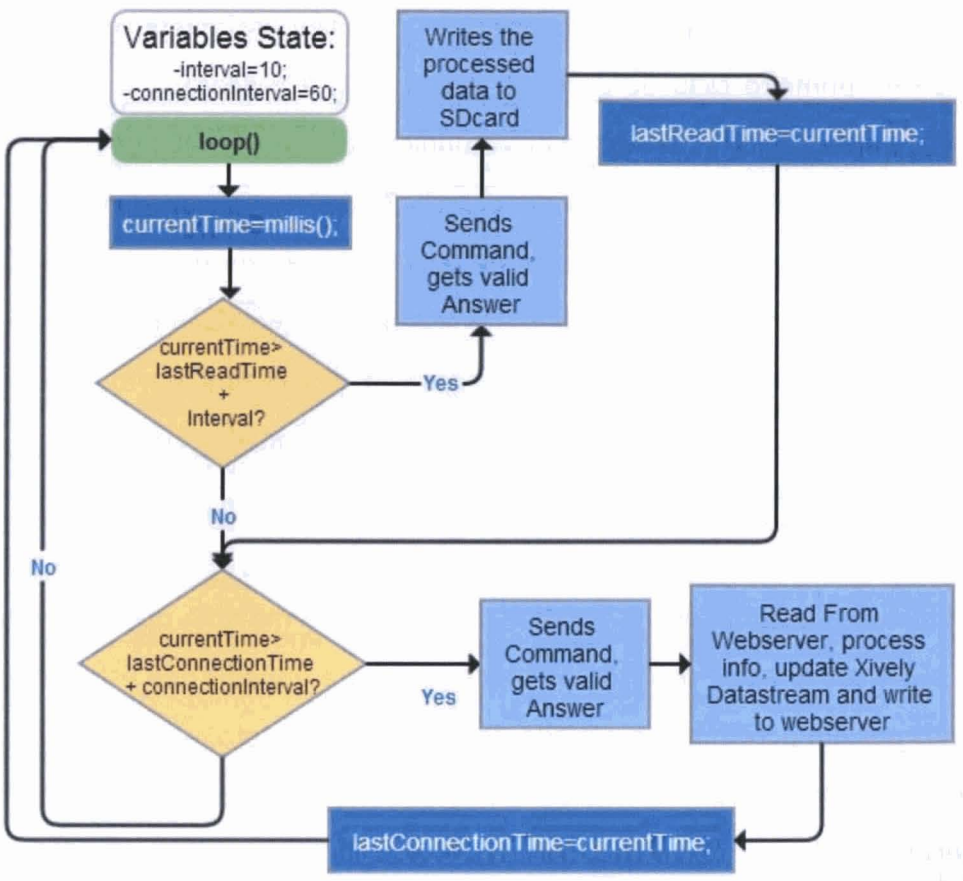
Seguidamente será analisada a secção principal do *sketch* desenvolvido: a função *loop().*

4.7.4. Loop()

O *loop()* é uma função que corre infinitamente e repete-se tanto mais rápido quanto mais rápida for a execução de um ciclo. No entanto, pretende-se neste caso efetuar ações com um determinado ritmo: por exemplo, escrever no cartão de memória de 10 em 10 segundos, mas só aceder e gravar dados no *webserver* de minuto a minuto. Ora, no ambiente de programação Arduino existem formas eficientes para lidar com esta situação, de modo a evitar a deterioração da robustez e eficiência do código com a introdução de atrasos impostos pelo programador. Assim, no Arduino, utiliza-se a função interna *millis*(): esta função conta os segundos que passaram desde o início de execução

do programa e retorna este valor, quando invocada, para uma variável *unsigned long*, variável de 4 *bytes* (32 bits). O diagrama representado na figura 53 demonstra como é feita esta gestão temporal no *sketch*.

O ciclo de *loop()* é formado por duas secções de código que correm com diferentes frequências. Essa frequência é determinada pela validação dos ciclos *if* que possuem estas duas secções de código. A forma de validação é feita através de uma operação aritmética entre três variáveis. Primeiro temos a definição de duas constantes, o interval e o *connectionInterval*, que são customizáveis através do *webserver* mas que tomam valores *default* de 10 s e 60 s, respetivamente.



53 – Gestão temporal das secções de código dentro da função loop()

A figura 53 evidencia esta relação através dos dois circuitos de decisão nesta presentes. Numa primeira fase do *loop()* ocorre a atualização da variável do tipo *unsigned*

long int (inteiro, precisão 32 *bytes*, sem sinal) denominada por *currentTime*. Esta variável guarda o valor retornado pela função já incluída nas bibliotecas pré-definidas do Arduino, a *millis*(). No caso de *overflow*, o valor da contagem é posto a zero e a contagem reinicia-se. Resta então referir como é definida a variável *lastReadTime*. Esta variável é atualizada também com o valor retornado pela função *millis*(), mas não no mesmo caso do *currentTime*() (que ocorre sempre que o ciclo *loop* é reiniciado). Esta é atualizada cada vez que a resposta ao circuito de decisão é *"Yes"*, pelo que corresponde a um momento em que de facto houve troca de informação entre o *Arduino* e a ECU e onde foram guardados dados relativos ao funcionamento da bomba de calor.

O mesmo acontece no segundo ciclo de decisão, onde se usou o mesmo critério para a escolha das variáveis (a nomenclatura utilizada é facilmente correlacionada com a utilizada no primeiro ciclo de decisão). Assim, com a validação do circuito de decisão segue-se a abertura de uma sessão de comunicação com a bomba de calor onde, através da utilização das funções *serialEvent* e *serialBroadEvent*, é possível obter a resposta obtida (se segue a estrutura presente na documentação da trama). Estas duas funções irão ser discutidas mais em detalhe noutra secção deste documento. Para já é importante evidenciar o resultado produzido pela invocação destas funções: o Arduino envia um comando à ECU e, após um *delay* imposto pelo programador (100-200ms), invoca uma das rotinas referidas acima, que será responsável por analisar e validar a informação existente no *buffer* da porta série que está ligada à ECU e, consequentemente, à bomba de calor. A utilização de *delays* geralmente não representa uma opção correta, visto que estão a ser consumidos recursos ao processador e a impedir que este reaja a impulsos externos por um determinado período de tempo. No entanto, neste caso, a comunicação segue um modelo pergunta-resposta em que o momento de chegada de informação é conhecido, sendo o *delay* utilizado para dar tempo suficiente à ECU para formular e enviar a trama de resposta para o Arduino.

No caso de haver uma resposta válida por parte da bomba de calor, esta é lida e são tomadas as respetivas ações conforme a zona da secção do *loop* onde foi feito o pedido de informação (se numa secção de escrita no cartão de memória - circuito de

decisão 1 – ou numa secção de comunicação com o *Webserver* - circuito de decisão 2). Em qualquer um destes circuitos de decisão existem *prompts* realizados para a porta ligada ao PC (e ao *Serial Monitor* do Arduino) para que haja noção do tipo de informação que está a ser gerada por ambas as partes intervenientes na comunicação.

É importante explicar como a informação é disponibilizada ao utilizador. No caso de um comando de *broadcast*, os *bytes* recebidos são convertidos em caracteres hexadecimais e concatenados a uma *string*, que ocupa um espaço reservado na memória do Arduino. No caso *singlecast*, optou-se por utilizar um array de *bytes* com um tamanho pré-definido. Este critério é definido pelo tamanho máximo que uma resposta por parte da ECU pode tomar. Porém, foi preciso tomar algumas precauções relativamente à escrita no monitor série deste array de *bytes*: devido às diferenças no tamanho da resposta, utiliza-se uma função que imprime os índices do *array* em conformidade com o valor indicado no 3º índice do *array*, coincidente ao campo do tipo de comando. Estas situações serão detalhadas em seguida.

4.7.5. *serialEvent() e serialBroadEvent();*

Segue-se, então, a análise das funções mais importantes do datalogger: *serialEvent()* e *serialBroadEvent()* - que operam identicamente a uma RSI, associando a ocorrência de um evento a uma função. Neste caso, a utilização da classe *Serial* permite obter informações relativamente importantes sobre o estado da porta série, e assim, consegue-se efetuar uma boa gestão de recursos do microprocessador através da verificação de *flags* (usando condições *if*), e sem a necessidade de se efetuar *polling* de dados. Para além disso, como foi visto na exposição das características do Arduino no Capítulo 2, as bibliotecas do Arduino dotam o microprocessador com *buffers* de chegada de informação das portas série. Assim, não se achou relevante utilizar interrupções externas visto que, para o caso da comunicação série no Arduino, já existe um *serial handler* implementado.

Assim, cada uma destas funções inicializa-se inquirindo se existem ou não dados à espera de serem processados na porta série. Se houverem, continua; se não, volta ao

estado inicial e abandona a rotina, atualizando o *Program Counter (PC)* para o *loop*() e para a instrução que segue a invocação desta rotina. Importa referir que cada uma destas funções opera utilizando duas *flags*, a *startFrame* e *frameCompleted*, que permitem evidenciar situações diferentes conforme o seu estado:

- por *default*, aquando a chamada das rotinas de tratamento de informação, estas duas *flags* são falsas (valor lógico '0')
- a *startFrame* toma o valor lógico 1 assim que sejam lidos da porta série os caracteres que representam o início de uma *frame* (' {{ '). Esta *flag* só volta a ser '0' se eventualmente a *frame* recebida possuir caracteres inesperados e representativos de uma *frame* inválida, incorreta, ou quando se validar uma trama de dados, abrindo caminho para a validação de outra trama, assim que disponível no *buffer* de dados.
- A *frameCompleted* toma o valor lógico 1 no caso único em que foi lida uma *frame* válida e que os caracteres presentes na variável correspondente à resposta recebida correspondam de facto a caracteres válidos.

Estas funções foram alteradas com alguma frequência de modo a colmatar a situação já referida acima: a consecutiva atualização do protocolo de comunicação que estas bombas de calor da *Bosch* utilizam. No entanto, estas alterações não afetaram a estrutura da mesma em grande escala, que distingue os casos referidos acima. Claro que a validação da *flag frameCompleted* irá ser tanto mais longa quanto mais longa for a *frame* a receber e a analisar.

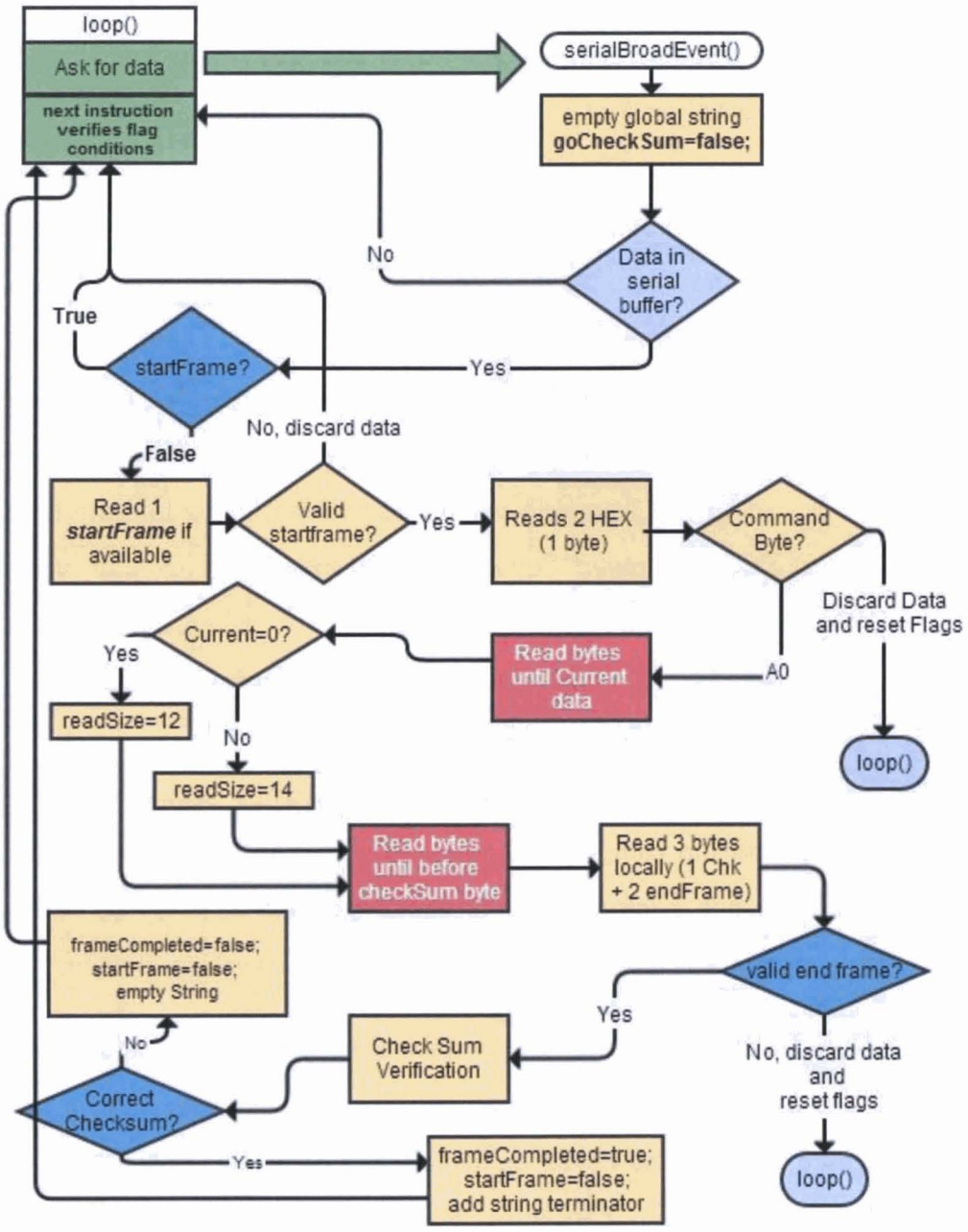
Customizar esta função para que aceite outro tipo de respostas é relativamente simples: para isso, basta fazer referência à documentação existente sobre o protocolo de comunicação e fazer as alterações necessárias à rotina de tratamento de informação. Deste modo, sempre que houver um pedido de informação específica, a rotina de tratamento de trama irá interpretar este pedido de acordo com o campo "*Command*" e

especificar o tamanho total da trama a ler, bem como o tipo de dados que o *Arduino* irá receber.

A figura 54 apresenta o fluxograma associado à função de tratamento de dados presentes no *buffer* da porta série do Arduino. Após o pedido de informação por parte do Arduino e após um período de espera curto (200ms *delay*), a função *serialBroadEvent*() é invocada. Inicialmente o estado das *flags*, *startFrame*, *frameCompleted* e *goCheckSum* estão a zero. A cada ciclo de leitura de informação é verificado se existem *bytes* no buffer de dados através da utilização da função *Serial.available*(), verificação esta omitida do fluxograma abaixo de modo a facilitar a análise do mesmo. Outra omissão do fluxograma representado foi a verificação dos *bytes* lidos para que sejam descartados na eventualidade de coincidirem com o código ASCII 125 ou 127 (caracteres que compõem um *startframe* e *endFrame*, respetivamente). A *flag startFrame* só é verdadeira após a leitura e validação dos *bytes* correspondentes ao *startFrame*. Esta situação é descrita pelo circuito de decisão "*Valid startFrame*?". Enquanto esta *flag* tiver ativa, não há leitura de *bytes* representativos de inícios de trama, pelo que quaisquer dados são descartados e a sessão de análise do conteúdo da trama não começa. De seguida, segue-se a leitura de 1 *byte* que representa o tipo de trama que está a ser transmitida. Após a validação deste *byte* como sendo resposta a um comando de *broadcast* (A0), segue-se a leitura da informação que a trama possui. Nesta etapa, os *bytes* mais importantes representam a corrente. Quando este valor é zero corresponde à situação em que não há ligação física entre a ECU e a bomba de calor e, por conseguinte, infere-se acerca do tamanho da trama a ler: este assumiu-se diferente no caso de teste em relação ao protocolo documentado, pelo que foram tomadas diligências para que se conseguisse validar algum tipo de informação proveniente da ECU testada.

Deste modo, distinguem-se 2 valores para a quantidade de dados que faltam ler após o campo corrente (ver figuras 25 e 26), apesar de só ter sido possível testar para o caso de teste (trama com o formato representado na figura 26). Assim, sabemos que quantidade de informação é que existe entre o *byte* corrente e o *byte CheckSum*.

Finalmente, os *bytes CheckSum* e os *bytes* relativos ao *endFrame* são lidos para variáveis locais. Deste modo é possível avaliar e validar o *endFrame* antes de se passar à etapa de verificação do *CheckSum*, etapa esta que consome mais recursos relativamente à anterior. Se o valor lido do *CheckSum* coincidir com o valor calculado (soma de todos os *bytes*, com início no *CommandByte* (inclusive) e com fim no *byte* anterior ao *CheckSum*), a trama é validada, e a *flag* global *frameCompleted* toma o valor de *true* e retoma-se a função *loop*(), que através da verificação desta mesma *flag*, analisa se o conteúdo da *string* é válido, e toma as diligências necessárias ao processamento da informação. No fim, apaga o conteúdo da *string* e impõe os valores por defeito às *flags*, tornando então possível a análise e validação da frame seguinte.



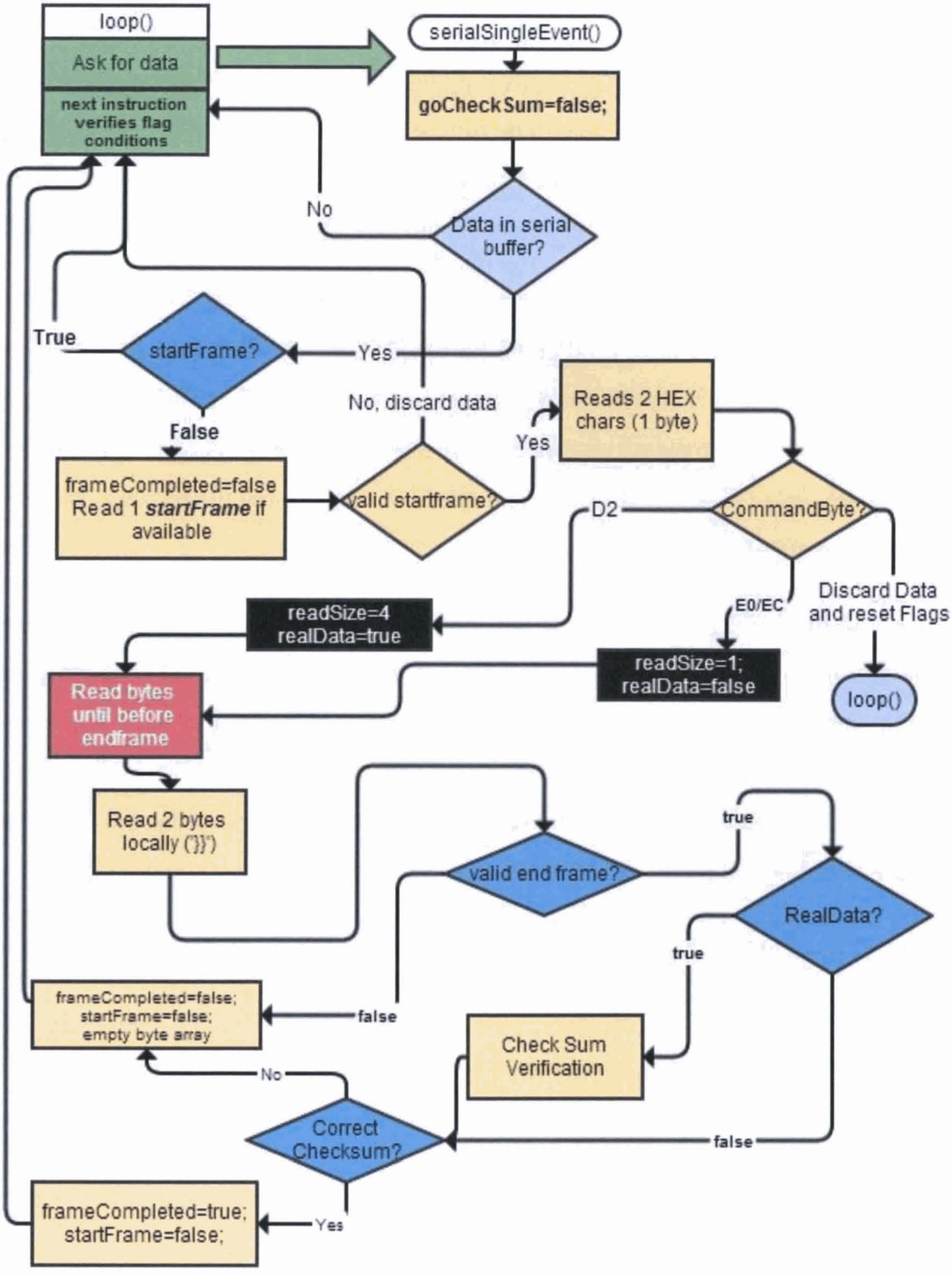
54 – Fluxograma elucidativo do funcionamento da SerialBroadEvent()

A operação de validação de uma trama de resposta a um *GetParameter* é feita de forma análoga. A diferença fundamental está no tipo de variável que é usada para guardar os dados recebidos. No caso anterior era uma *String*, enquanto que agora optou-se por utilizar um *array* de *bytes*, de tamanho pré-definido. A forma utilizada para diferenciar, tanto no processo de escrita na porta série monitorizada pelo Arduino IDE, tanto no

processo de leitura de informação, teve que considerar os diversos tamanhos possíveis de resposta consoante o campo *Command* da trama. Assim, através da análise do fluxograma da figura 55, é possível evidenciar 3 situações distintas que darão origem a 4 tipos de *prompts* de mensagens no monitor:

- Campo "*Command*" 'D2': segue-se a leitura de 4 *bytes* de informação. Este comando diz respeito a uma trama real com informação real (*realData=true*).
- Campo "*Command*" 'E0/EC': segue-se a leitura do *byte* relativo ao *CheckSum*. Estamos numa situação de erro (*realData=false*), onde:
    - E0: A trama de pedido de informação contém um byte no campo "*Command*" que não corresponde a nenhum valor da gama existente de tipos de comando;
    - EC: O comando enviado pelo Arduino possui um *CheckSum* incorreto.
- Campo "*Command*" diferente dos descritos acima: A resposta da ECU possui um campo de "*Command*" que não é reconhecível pelo Arduino, pelo que a informação até agora agregada ao *array* de *bytes* deverá ser descartada e as flags reestabelecidas com os valores default.

Através da análise do fluxograma abaixo, observa-se que, após a verificação da existência dos *bytes* correspondentes ao "*endFrame*", segue-se então a verificação do *CheckSum*, para que se possa validar ou não a trama recebida. Consoante a veracidade da informação (*flag realData*), conclui-se acerca da quantidade de informação a ser contabilizada na operação de verificação da soma *byte* a *byte* (*CheckSum*). No caso em que o comando é E0/EC, só se tem o *byte* relativo ao próprio comando. Assim, uma simples comparação entre este e o *byte* correspondente ao *CheckSum* permite inferir acerca da validade da trama recebida. Por outro lado, no caso em que há informação real no *array* de *bytes*, torna-se necessário verificar a soma *byte-wise* do conteúdo do *array* e comparar com o campo da frame recebida correspondente ao *CheckSum*.



55 – Fluxograma elucidativo do funcionamento da SerialSingleEvent()

4.7.6. Customização do SD Rate e Xively Rate

Existem várias bibliotecas relativas à utilização do cartão de memória. No entanto, algumas oferecem um largo espectro de funções enquanto outras regem-se pela leveza e simplicidade dos serviços que disponibilizam. A *SDFat* é uma biblioteca extremamente completa e que oferece inúmeras possibilidades que vão para além da criação, exclusão, leitura e gravação de ficheiros *.csv*.

Numa primeira fase, e tal como na função *setup(),* verifica-se se o ficheiro está criado e se possui permissões de escrita-leitura. Se assim for, é possível a edição e a escrita num ficheiro presente no cartão *SD.* A extensão deste ficheiro é *.csv* e foi escolhida para que fosse possível abrir diretamente no Excel, onde poderemos visualizar a informação mais facilmente.

O ritmo de gravação no cartão poderá ser customizado através do *webserver* (plataforma *Xively*) e poderá ser escolhido entre a gama de 10 e 30 segundos, valores limite impostos pelo programador. Por outro lado, a frequência com que ocorre o *upload/download* para/do *webserver* também pode ser customizado através da edição de um campo presente no *webserver*. Estes dois campos serão analisados em mais detalhe no capítulo dos resultados.

Estas são apenas umas das formas de comunicação bidirecional que o datalogger oferece entre o Arduino e a bomba de calor.

4.7.7. Ocupação do Cartão SD

Outra informação que se optou por indicar foi a ocupação atual do cartão de memória. Foi esta funcionalidade que levou à utilização da biblioteca *SDFat* já que, em detrimento da *SD*, é maior e torna o *sketch* mais pesado. No entanto, só com esta biblioteca é que é possível contar os *clusters* de memória livre e os ocupados e então assim obter o espaço total ocupado/livre existente. Esta funcionalidade é importante já que permite saber se é necessário efetuar a substituição do cartão SD.

4.7.8. Conexão WiFi

A conexão *WiFi* é inicializada através da função *setupWifi()*. Aqui é definido o nome da rede à qual se pretende ligar e qual a *password* associada a esta. Esta função, presente em muitos dos exemplos associados à biblioteca oficial do *WiFi* do *Arduino*, oferece a possibilidade de obtermos dados relativos à força do sinal *WiFi* relativo à rede que se pretende ligar, e impõe um atraso ao programa de 10 segundos, tempo necessário para garantir que a ligação à rede escolhida foi feita com sucesso. A existência dos referidos exemplos de ligação a uma rede *wireless* permite, desde logo, um mecanismo *plug-and-play* nesta plataforma. A transposição para outra forma de conexão à internet é feita de maneira muito simples, e neste caso verificou-se a hipótese de se ligar via Ethernet.

4.7.9. Conexão Ethernet

Tal como a plataforma *WiFi*, a biblioteca associada ao *shield* Ethernet vem, por defeito, dotada de exemplos de utilização que atribui a esta ferramenta a característica de *plug-and-play.*

Utilizar ambas as formas de ligação (que assegurasse que se pelo menos uma destas falhasse, a outra tomava o controlo) é algo complicado de se conseguir: primeiro, pelos custos que acarretam (cada uma destes *shields* custam entre 80 e 100 euros); depois, ambas utilizam a mesma porta de comunicação, pelo que seria necessário acrescentar um *shield* ao sistema, que iria funcionar como um adaptador, e assim consentir a existência das duas formas de ligação à internet. A existência de tais equipamentos, só iria deteriorar a processo de logística do datalogger numa caixa pequena.

4.7.10. *Xively*

A linguagem e as diferentes nomenclaturas associadas a esta plataforma foram já descritas anteriormente. No entanto, considerou-se adequado incluir nesta subsecção informação concreta relativa à forma como é efetuada, do ponto de vista da programação, ligação à plataforma *Xively*.

Numa primeira fase é necessário definir que tipos de dados é que irão ser associados a um *feed* ID. Para isso, inicializa-se um array de *datastreams* e para cada um dos elementos deste *array* é imperativo definir o tipo de variáveis e outras características que dependem do tipo de variável que se escolha. Seguidamente, procede-se à associação entre o *datastream* criado, uma *key* e um FEED_ID, para que a sessão de comunicação entre um *webserver* e os *datastreams* criados no *sketch* do Arduino possa efetivamente ser aberta.

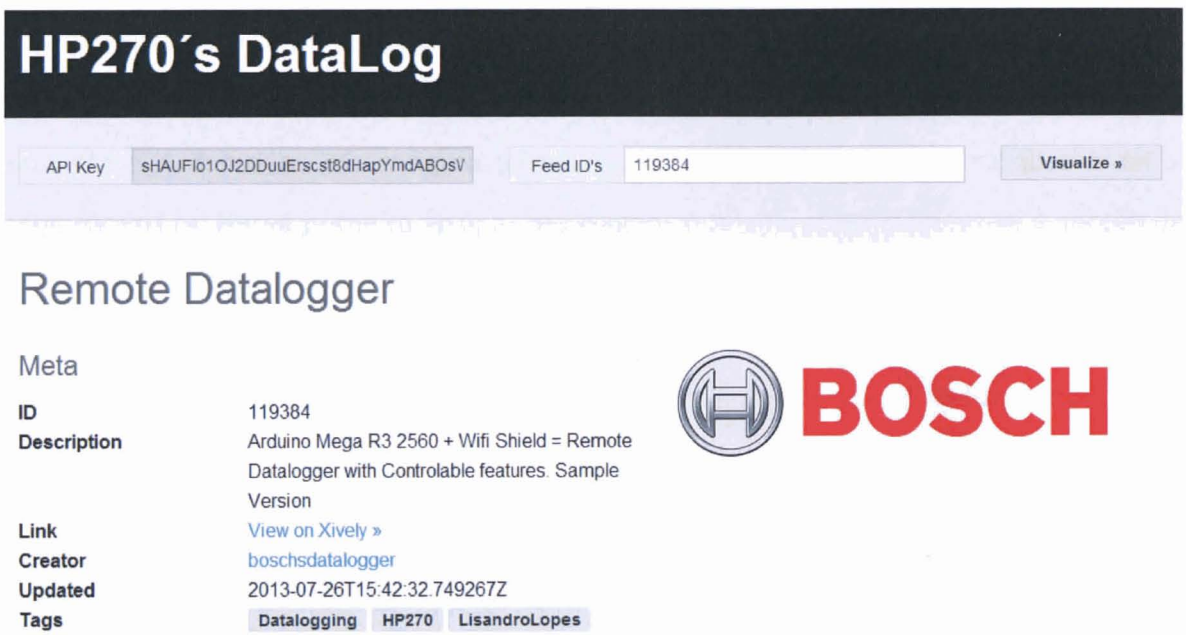
Pela análise da figura 55, é possível verificar-se que, em cada ciclo de execução da parte de código correspondente a uma resposta afirmativa ao circuito de decisão, o Arduino acede ao *Webserver*, retira o estado atual do *array* de *datastreams* existentes, analisa-os e, conforme os valores de cada índice do array, altera localmente o valor destas variáveis. De seguida, independentemente do estado atual das variáveis do *webserver*, o Arduino efetua um pedido de informação e só depois de obter a resposta e processar toda a informação, atualiza todas as variáveis locais referentes ao *array* de *datastreams* e, posteriormente efetua o upload do *feed* de informações para o *webserver* através de uma única operação de escrita (*upload* do *array* de *datastreams*).

4.7.11. *JavaScript Visualization Tool – Github Hosted*

Anteriormente referiu-se alguns dos serviços que o *Xively* oferece. Um destes serviços é uma ferramenta de visualização de *feeds* em *Javascript*, *online*, hospedado no *GitHub*. Esta ferramenta foi customizada para incluir na mesma página, o logotipo da *Bosch*, e mostrar um determinado tipo de gráficos, em que cada *datapoint* (definido

anteriormente como um valor de um *datastream* num determinado momento) é representado por um ponto único.

A figura 56 apresenta o resultado das pequenas modificações que se efetuaram a um dos tutoriais existentes no site da *Xively* [35] com efeitos a nível do layout da ferramenta de visualização em JavaScript.



56 – Layout da customização efetuada ao código base presente no *GitHub*

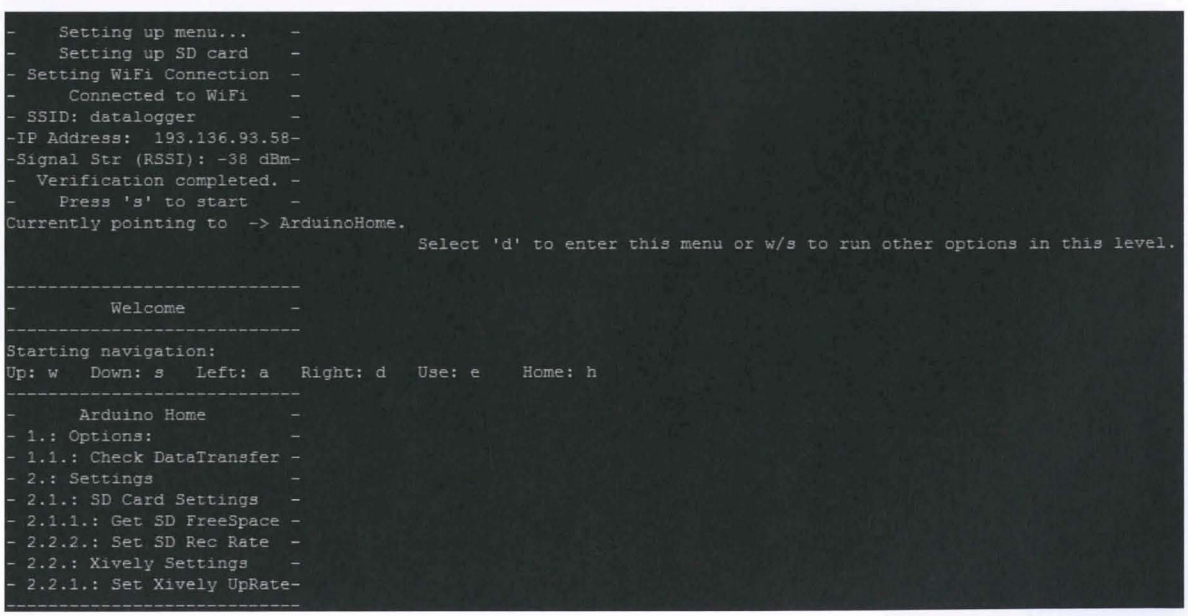
A configuração desta ferramenta de modo a apresentar os *datastreams* do datalogger é muito simples. Basta indicar qual a KEY e qual o FEED_ID associada à coleção de datastreams que se pretende visualizar. A partir daí, pode-se customizar a página web através da edição das secções HTML e JavaScript da mesma.

Capítulo V

5. Resultados

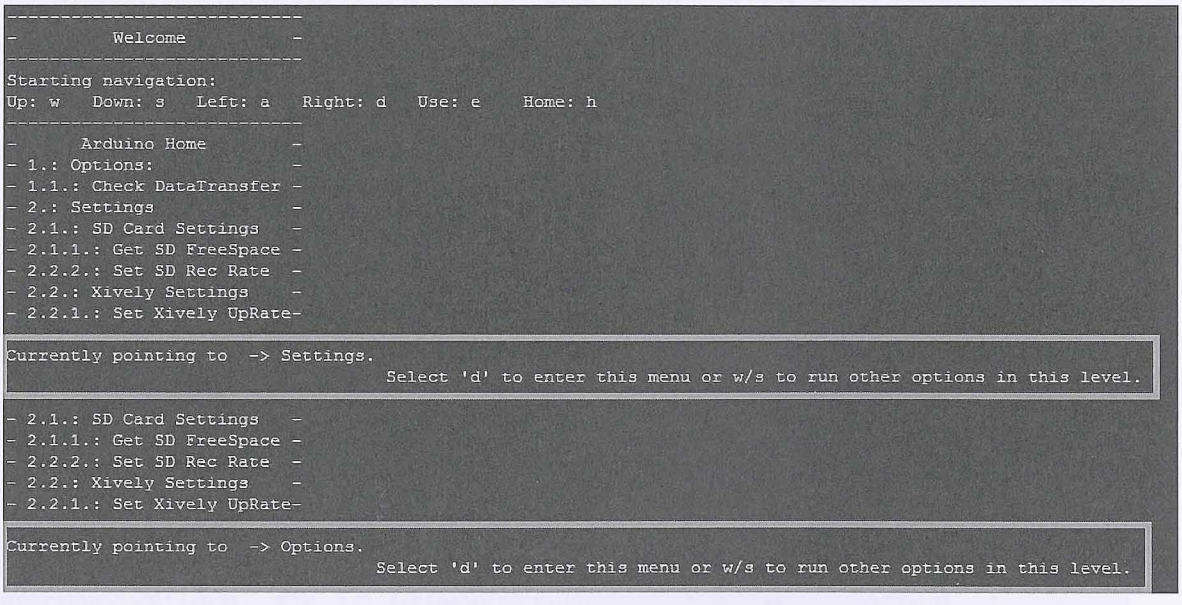
5.1. O menu;

Após o *upload* do ficheiro .bin para a memória do Arduino através do *Arduino IDE* é possível analisar que dados estão a ser transmitidos e recebidos na porta que liga o *Arduino* à ECU, e controlar o menu através de *inputs*. Estes serão escritos na porta série que liga o Arduino ao PC. A figura 57 apresenta o momento em que se abre o *serial monitor PuTTy*. Numa primeira fase, as mensagens que aparecem referem-se à secção de código do *setup()*. Como foi referido, é nesta função que as portas série são inicializadas, que se define a rede sem fios que irá ser utilizada para ligar o *datalogger* à Internet e onde se inicializa os ficheiros do cartão SD onde irão ser guardados os dados recolhidos.



57 – Layout do Menu

Após esta inicialização e verificação de que o cartão SD está presente e corretamente formatado, apresenta-se então o layout do menu principal se e só se o utilizador inserir no campo de input o caráter 's', seguido de *enter*. Por conseguinte, ficam disponíveis outras *keys* que o utilizador poderá utilizar para navegar no menu, sendo elas, 's', 'w', 'a', 'd', 'e' e 'h'. Seguindo o método de navegação já descrito anteriormente, se o utilizador inserir uma determinada combinação, (poderá ser intercalada ou não com "*enter*"), o menu irá apontar para um determinado subitem do menu. A localização atual do ponteiro traduz-se na impressão de uma *string*, representada na figura 58. Posteriormente, se se enviar o caráter que representa a utilização desse item, o 'e', o programa irá agir em conformidade. Nas secções seguintes deste capítulo irão ser descritos os diversos tipos de ações que este menu oferece, explicitando a forma de navegação pelo menu.



58 – Navegação pelo Menu

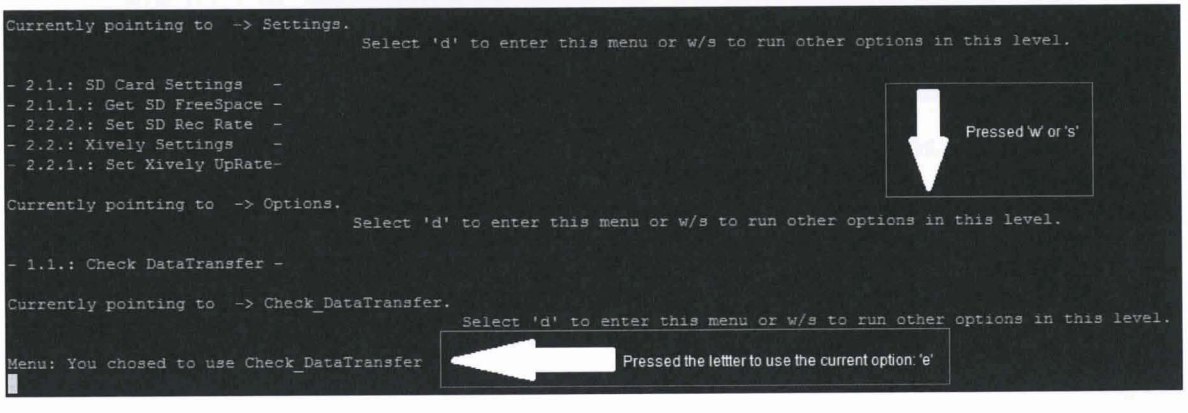
5.2. *Pedido de um GetParameter;*

Ao escolher-se a opção *Check_DataTransfer*, uma *flag* é ativa, levando a que o programa imprima no ecrã o comando enviado e a consecutiva resposta da ECU, bem como os dados já processados da trama recebida, seguindo o protocolo de comunicação

descrito em capítulos anteriores. Para além disso, a ativação da *flag* permite a impressão no monitor da porta série *(Serial Monitor)* muitas outras mensagens que indicam qual o estado atual do programa e o que está a ser executado num determinado momento.

Um ciclo completo de execução consiste essencialmente no envio de um comando *Broadcast*, de onde se retira toda a informação que a ECU pode fornecer e numa verificação dos valores que estão no *Webserver* através de uma leitura de dados por parte do *datalogger*. Se o parâmetro ID do *Webserver* contiver um valor diferente do valor lido no ciclo anterior significa que houve um *input* por parte do utilizador do *webserver* correspondente a um pedido de informação. Deste modo, uma trama de *getParameter* é construída utilizando o último valor lido do campo ID para ser enviada ao *datalogger*. Da mesma forma são analisados os campos do ritmo de gravação no cartão de memória e o ritmo de *upload* para o *webserver* e determinadas ações são tomadas de acordo com os valores lidos. São estes aspetos que irão ser abordados de seguida.

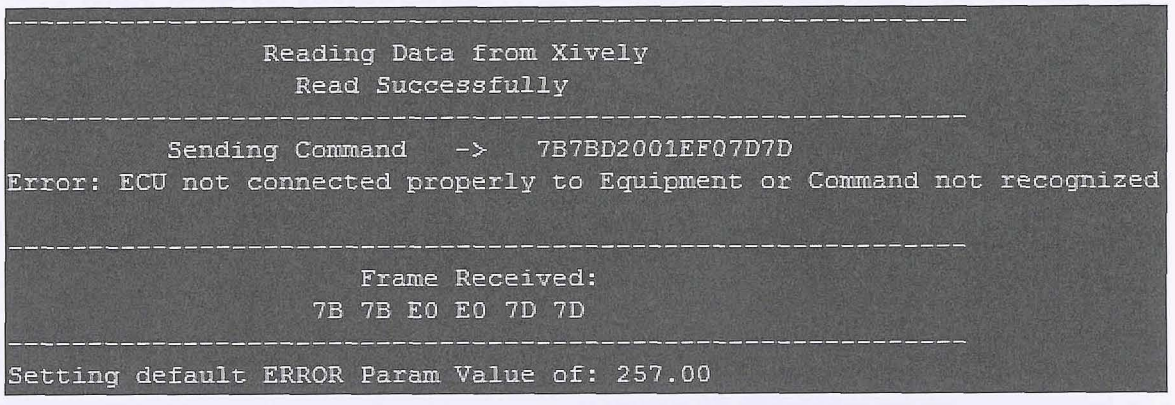
A figura 59 apresenta a troca de dados ocorrente durante um *GetParameter* quando a opção *Check_DataTransfer* se encontra selecionada (e a *flag* associada a esta opção está ativa) e quando existe um pedido de informação singular do lado do *webserver*.



59 – Navegação pelo menu (2) – Escolha da Opção Check_DataTransfer

Após a leitura dos dados do *webserver*, se não se verificar a existência de um pedido de informação, o *sketch* imprime a *string* representada na figura 60.

60 – Inexistência de pedido de informação por parte do Webserver



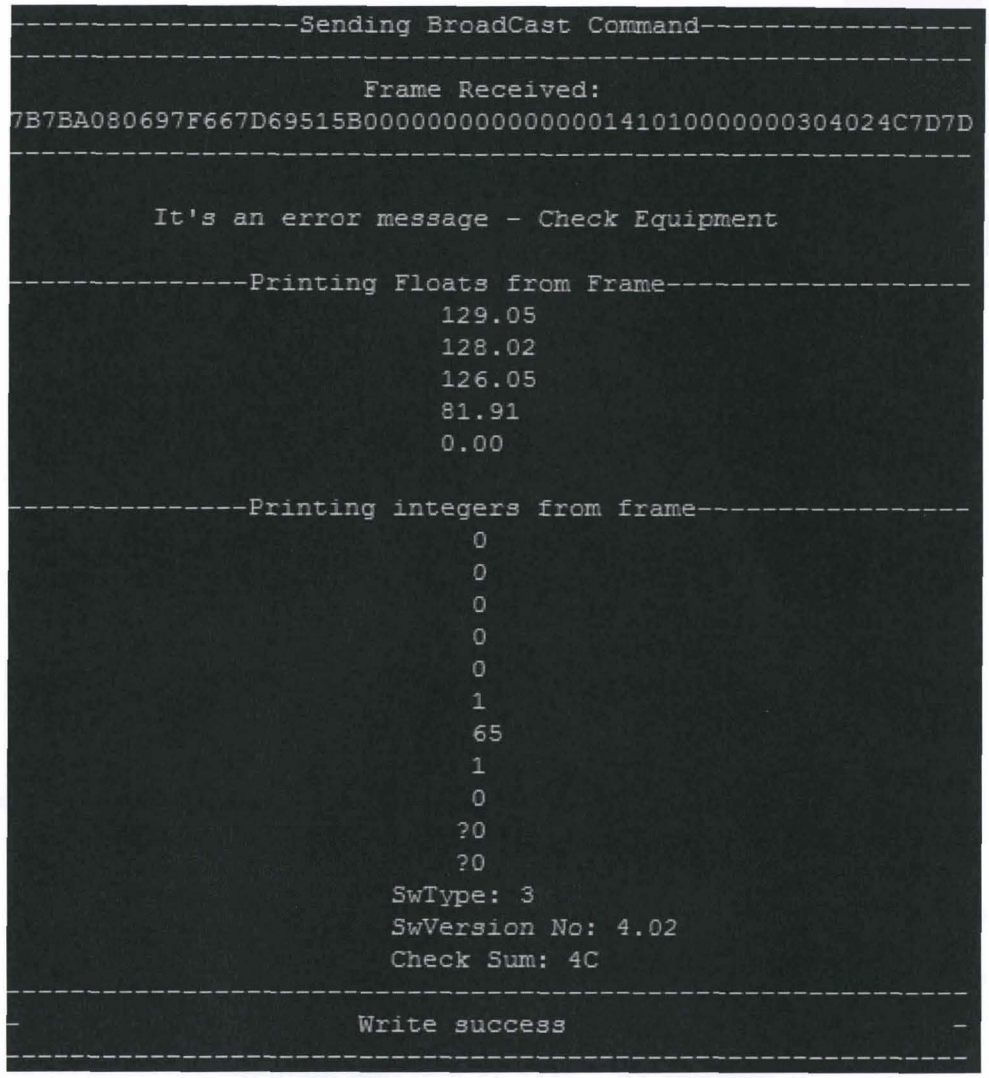
61 – Existência de um pedido de informação por parte do Webserver

Tal como foi referido, a ECU responde a um *GetParameter* com uma estrutura de trama indicativa na inexistência de uma aplicação a si ligada e envia para o *webserver* um valor indicativo desse mesmo erro. Para o efeito, o valor escolhido foi o de 257.0, por se encontrar fora da gama possível de resposta documentada (0-255). Esta situação está indicada na figura 61, onde, após a verificação de existência de um pedido de informação via *webserver*, o *datalogger* envia um comando à ECU, recebe a resposta e envia o valor de 257.0 para a *cloud* no próximo ciclo de *upload* para o *webserver* assim que processar a trama e inferir acerca do tipo de dados que esta transmitiu.

5.3. *Pedido de um BroadCast;*

Relativamente ao comando de *Broadcast*, este é feito de forma periódica, consoante o valor atual do *SDRate*, visto que sempre que há troca de dados *Broadcast*, este é guardado no cartão de memória. Assim, periodicamente poderá avaliar-se e atualizar-se a informação, tanto do lado do *Webserver* como do lado do cartão de

memória. A figura 62 reporta o fluir da informação e como a informação é disponibilizada para o monitor série.



62 – BroadCast de informação no monitor PuTTy

Verificou-se que a resposta da ECU a uma trama *broadcast* variava com um padrão desconhecido. Não foi possível inferir, com a utilização da ECU disponibilizada, acerca dos dados que se recebe num determinado momento. Para além disso, o tamanho da trama recebida não corresponde ao documentado na versão 2.0. No entanto, foi possível verificar o funcionamento correto da RSI relativamente à existência de um *endframe* ou *startframe* ('{' ou '}') no meio da trama: o *datalogger* deverá descartar qualquer caráter deste género. Outro processo de decisão importante é se o *sketch* é capaz de inferir

acerca da existência ou não de uma aplicação ligada à ECU: na versão protocolar 2.0 da *Bosch*, escolheu-se o *byte* referente à corrente: se esta for nula, os valores lidos não são corretos e uma mensagem de alerta é impressa na porta série monitorizada pelo *PuTTy* indicada por *"It's an error message – Check Equipment".*

Como já foi referido no capítulo 2.2, relativamente à diferenciação feita entre a versão experimental do protocolo 2.0 e o que está de facto documentado, optou-se por definir o seguinte momento de decisão:

- Se a corrente lida é zero (situação que ocorria sempre neste sistema), a trama expectável é aquela que foi obtida experimentalmente (o tamanho era sempre fixo).

- De modo a contemplar a situação documentada, que corresponderia a um sistema real (aplicação ligada à ECU), definiu-se que, quando a corrente fosse diferente de zero, o sistema iria responder da forma que está de facto documentada na versão de comunicação 2.0 da Bosch.

A primeira situação foi a única possível de documentar neste capítulo de resultados, correspondendo ao caso apresentado na figura 62 (tamanho da trama diferente da documentada –notar a correspondência entre os campos '?' (Data19 e 20) da figura 26 com os '?' da figura 62).

Relativamente à impressão dos dados da figura 62, assumiu-se um processamento idêntico ao da versão documentada: a existência de 5 números de vírgula flutuante, a existência de inteiros (mesmo num número diferente ao documentado - +2 *bytes* indicados pelos dois '?' junto aos caracteres representados na figura 62), a leitura do campo de tipo de software (*SWType*), *SwVersion* (versão de software), e por fim o valor do *CheckSum* e a impressão da indicação do sucesso da escrita no cartão de memória, dada pela impressão da *string "Write Success".*

5.4. *Gravação de dados no cartão SD*

O datalogger implementado permite também a exportação dos dados recolhidos da ECU para um ficheiro com extensão .csv existente no cartão SD embutido no Wi-Fi *Shield*. Cada vez que o Arduino sofre um *reset*, ou quando é efetuado o carregamento de um script pela primeira vez, a função *setup*() verifica a existência do cartão SD e cria os títulos das colunas que identificam o tipo de dados que essa coluna possuirá. O caráter de navegação entre colunas é o *';'*, podendo ser usado num *Serial.println()* para imprimir, numa única linha, todos os dados separados por colunas.

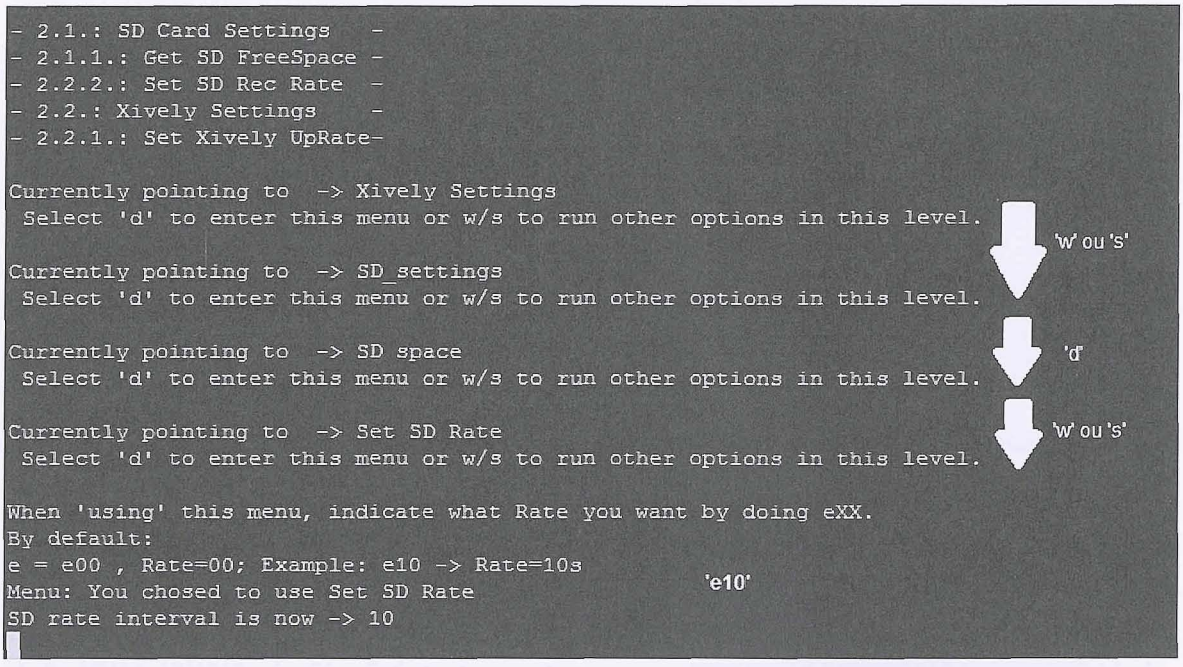
A figura 63 apresenta um exemplo de como os dados ficam organizados no cartão de memória.

NTC TOP	HTC BOT	HTC AIR	HTC FINS	CURRENT	Pump St	Fan State	Comp St	HeaterSt	SolValve	HPmode	HT[0]	HT[1]	HT[2]	?	?	SwType	SwVersion
126.02	126.00	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.00	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.01	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.00	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02
126.02	126.00	125.08	81.91	0.00	0	0	0	0	0	1	65	1	0	0	0	3	4.02

63 – Layout da informação no ficheiro .csv

5.5. *Alteração do ritmo de gravação no cartão SD e consulta de dados*

A alteração do ritmo de gravação no cartão de memória pode ser feita de duas formas: ou através do *webserver*, ou através da escolha do item apropriado do menu. Nesta secção irá ser descrita a segunda situação. A utilização do item *"Set SD Rate"* (efetuado pelo envio do caráter 'e' juntamente com um valor inteiro quando o ponteiro do menu está a apontar para este) conduz à impressão, no ecrã, de como é que o utilizador poderá alterar o ritmo de gravação no cartão de memória. A figura 64 representa precisamente esta situação.



64 – BroadCast de informação no monitor PuTTy

Este valor é atualizado localmente (numa variável do *sketch*) e será enviado para o *webserver* no fim de um ciclo da função *loop()* se e só se respeitar o valor mínimo de 10 segundos, imposto pelo *sketch*. Outra forma de atualizar este valor é através do *webserver* e esta situação irá ser descrita na secção referente à "Comunicação bidirecional com o *Xively*".

Para além desta funcionalidade, é possível também requerer a impressão, na porta a ser monitorizada, do tipo de dados que se encontram no cartão de memória, bem como do espaço total livre no cartão de memória. Esta operação requer um tempo de execução significativo pela necessidade de contagem dos *clusters* de memória do cartão de memória, para se medir a quantidade de clusters que se encontram livres e ocupados. A utilização de cartões de memória de menor tamanho poderá ter os seus benefícios ao nível da eficiência desta secção do *sketch*; no entanto, terá que haver um compromisso entre este tempo e a capacidade de armazenamento do *datalogger*. A figura 65 reporta o momento em que se pede a consulta da informação referida.

```
Menu: You chosed to use SD space
Wiring is correct and a card is present.

Volume type is FAT16

Volume size (Mbytes): 1926816
Free space MB: 1898048
Card Occupation: 1%
BROAD.CSV       2000-01-01 01:00:00 209456
TEST.TXT        2000-01-01 01:00:00 0
APP.TXT         2000-01-01 01:00:00 216
MUSIC.TXT       2000-01-01 01:00:00 0
DATALOG.TXT     2000-01-01 01:00:00 226
BROADE.CSV      2000-01-01 01:00:00 59436
AVF_INFO/       2013-07-24 21:29:52
  AVIN0001.INT    2013-07-24 21:29:54 2457600
  AVIN0001.BNP    2013-07-24 22:54:10 6815744
  AVIN0001.INP    2013-07-24 22:54:08 6815744
  STAT0000.BIN    2013-07-24 22:54:10 20
  PRV00001.BIN    2013-07-24 22:54:08 736512
DCIM/           2013-07-24 22:39:14
  101MSDCF/       2013-07-24 22:39:14
    DSC02847.JPG    2013-07-24 22:39:10 823890
    DSC02848.JPG    2013-07-24 22:39:40 846081
    DSC02849.JPG    2013-07-24 22:40:02 865908
    DSC02850.JPG    2013-07-24 22:40:34 876365
    DSC02851.JPG    2013-07-24 22:40:44 820132
    DSC02852.JPG    2013-07-24 22:40:54 858677
    DSC02853.JPG    2013-07-24 22:41:02 883571
    DSC02854.JPG    2013-07-24 22:41:14 841491
    DSC02855.JPG    2013-07-24 22:41:46 793658
    DSC02856.JPG    2013-07-24 22:41:56 737742
    DSC02857.JPG    2013-07-24 22:43:26 851326
    DSC02858.JPG    2013-07-24 22:43:54 910685
    DSC02859.JPG    2013-07-24 22:44:00 923352
    DSC02860.JPG    2013-07-24 22:54:04 817322
MP_ROOT/        2013-07-24 22:39:14
  101MNV01/       2013-07-24 22:39:14
```

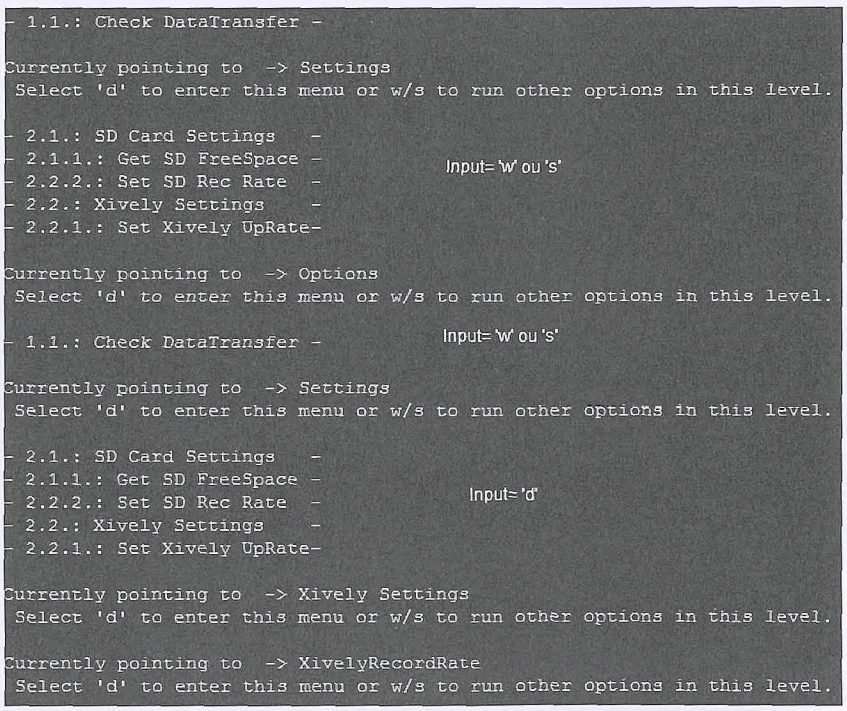
65 – Consulta do conteúdo do Cartão SD (monitor PuTTy)

Primeiro é indicado o tamanho total do cartão em *Mbytes*, depois o espaço livre existente. Relativamente à informação contida no cartão de memória, cada linha impressa segue o seguinte formato:

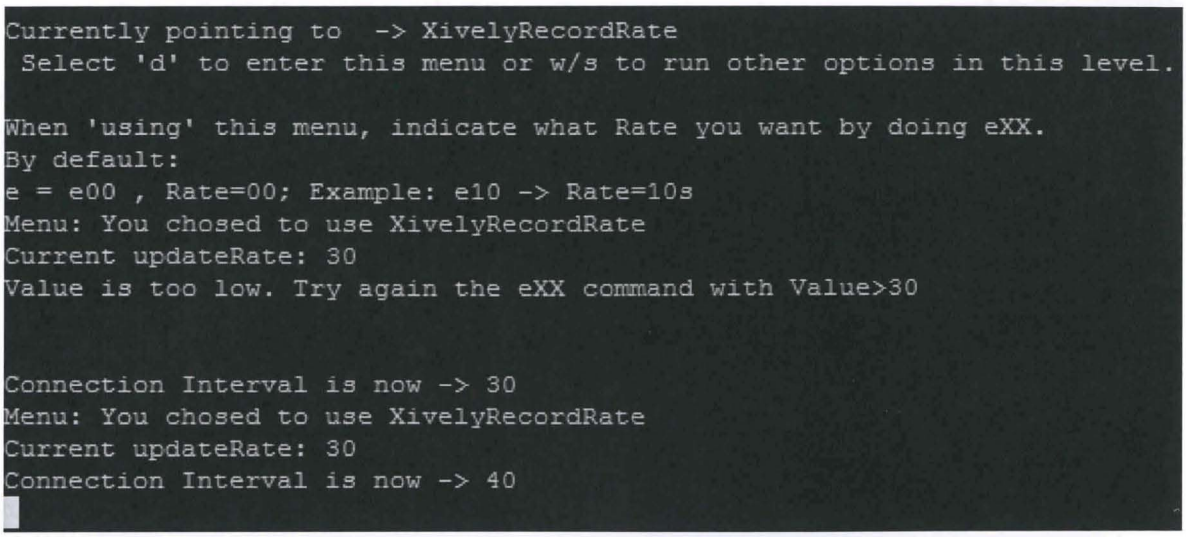
- Nome do ficheiro em causa;
- Data e hora de criação
- Tamanho em *bytes* do ficheiro em análise.

5.6. *Alteração do ritmo de gravação no Xively*

A alteração do ritmo de *upload* de informação para o *webserver*, tal como acontece com o cartão de memória, pode ser feita de duas formas: ou através do *webserver*, ou através da escolha do item apropriado do menu. Este último caso será de seguida analisado. Para isso, considera-se a continuidade do momento de execução do *sketch* do subcapítulo anterior em que o estado atual do menu é o *CheckDataTransfer*. A navegação deste ponto para o ponto pretendido é referida na figura 66 e faz referência ao que já foi indicado relativamente ao menu.



66 – Navegação para alterar o ritmo de upload via menu (Putty)



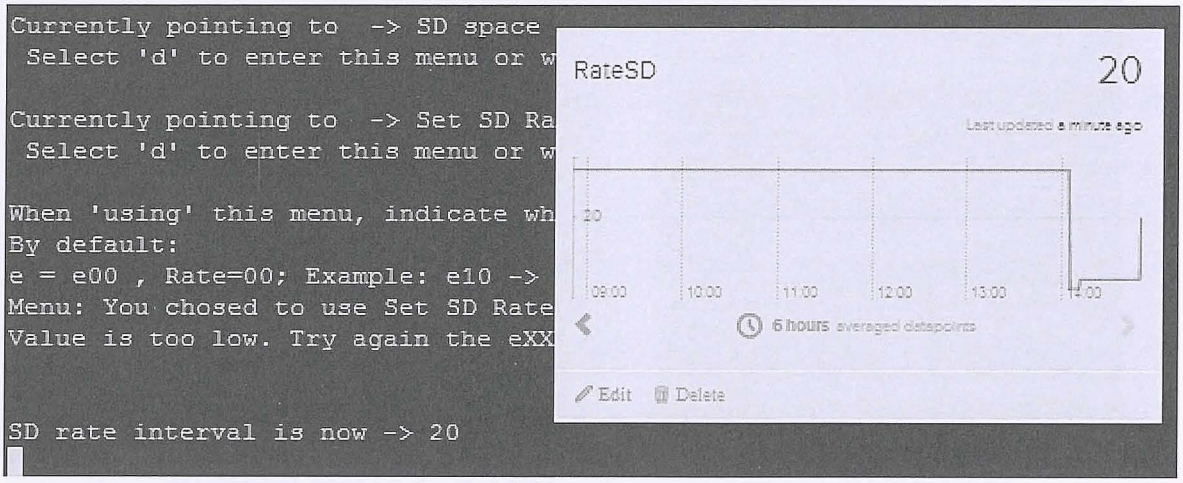
67 – Alteração do ritmo de upload via menu (Putty)

Assim ao escolher-se o item do menu *SetXivelyRate* e seguindo o formato indicado na figura 67, é possível alterar o ritmo com que se faz *upload* de dados, que toma efeito a partir do próximo ciclo de execução do *loop*. O limite designado como ritmo mínimo de *upload* de dados para o servidor é de 30 segundos, já que, tal como a consulta de informação de dados e de espaço livre no cartão de memória, esta secção afeta a eficiência temporal do *sketch*, visto que implica a abertura de uma sessão de comunicação com o servidor, que ocorre tanto mais frequente quanto menor for o valor escolhido para o *XivelyRate*.

5.7. Comunicação bidirecional com o Xively

De seguida segue-se a análise do controlo efetuado pelo *Webserver* através da comunicação bidirecional entre este e o *datalogger*. Esta comunicação permite alterar o ritmo de gravação de dados no cartão de memória e a cadência de *upload* para o servidor, para além do que foi analisado acima referente ao pedido de um parâmetro à ECU. Para complementar a descrição, as figuras que compõem este subcapítulo apresentam a evolução do processo no *webserver* e no *sketch*. Relativamente ao pedido de alteração do ritmo de gravação no cartão SD, assim que o valor lido do *webserver* não corresponder ao valor ao ritmo atual de gravação, este irá ser atualizado.

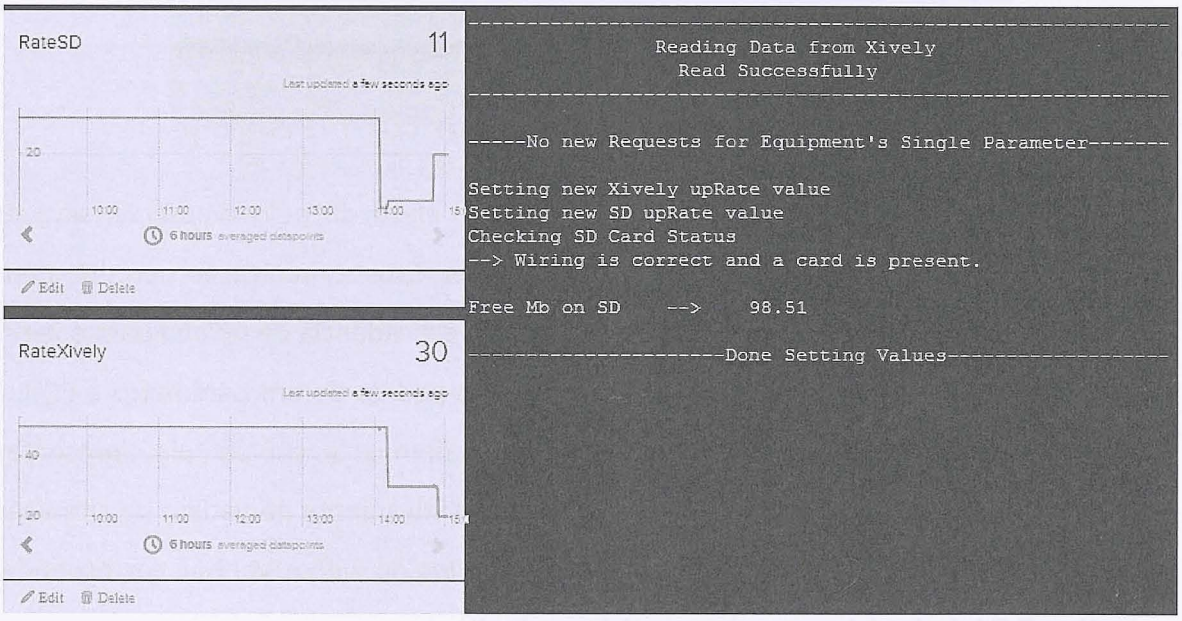
A figura 68 representa o efeito da alteração efetuada via *PuTTy* no *webserver*.



68 – Navegação para alterar o ritmo de gravação no cartão SD via menu (Putty)

O processo de alteração do *upload* de dados para o *Xively* é muito idêntico ao método de alteração de ritmo de gravação de dados no cartão, pelo que não irá ser descrito.

Por fim refere-se o que acontece quando se utiliza o *Webserver* para impor um ritmo de gravação ou de *upload* para o *Xively* diferente do atual. O momento em que ocorre o *update* dos valores para o *webserver* está descrito na figura 69.

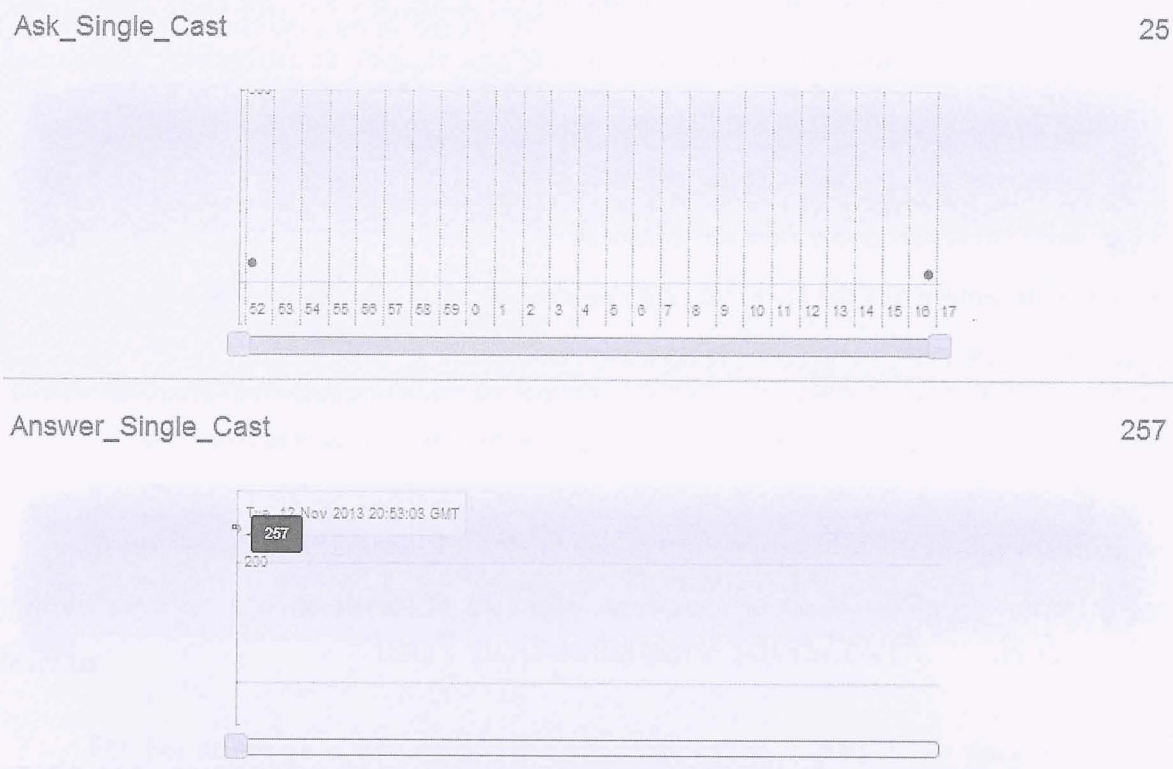


69 – Alteração dos "Rates" via Webserver (Putty)

Através da análise da figura 69 é possível verificar-se que, após a leitura do array de *datastreams* por parte do Arduino, este consegue detetar que houve alterações efetuadas aos valores através da comparação entre o valor atual das duas variáveis indicadas e toma diligências para efetivar esse pedido, através da alteração destes valores, que irão ser utilizados a partir do próximo ciclo de execução do *loop*. De modo a confirmar estas alterações, o *upload* do *datastream* atualizado é realizado e aí o utilizador saberá se o valor se alterou ou não (a alteração deste valor indica que o valor foi alterado corretamente e que faz parte do *array* de *datastreams* que existe localmente no Arduino).

5.8. *JavaScript Visualization Tool*

A ferramenta de visualização aqui analisada foi pensada como uma alternativa para a visualização dos dados. No entanto, esta ferramenta não permite a edição dos campos visualizados, mas oferece a disponibilização dos dados recolhidos da bomba de calor com maior detalhe, já que os gráficos possuem maior resolução temporal, que vai até aos 90 dias de rastreamento de informação. No caso de se escolher um período de 6 horas é possível obter-se o *layout* representado na figura 70 (note-se que o gráfico correspondente ao *Ask_Single_Cast* foi definido inicialmente para 6 horas mas a figura mostra algo diferente: esta ferramenta oferece também a funcionalidade de *zoom in/out).*



70 – JavaScript Visualization Tool

Um pedido efetuado através da plataforma *Xively* (e não através desta ferramenta de visualização) pela alteração do campo *Ask_Single_Cast* conduz a uma resposta representada no gráfico *Answer_Single_Cast.*

Capítulo VI

6. Conclusões

Aquando o seu aparecimento, os *dataloggers* tiveram grande impacto no mundo industrial. Através da associação às comunicações M2M, estes tiveram o seu espectro aplicacional enormemente alargado.

A necessidade de se desenvolver um *datalogger* especifico, através de uma plataforma *open-source*, adveio do facto da Bosch ter já desenvolvidos módulos de software reaproveitáveis. Além disso, é uma tecnologia barata, uma vez que não é necessário adquirir *software* específico que dependa de um só fabricante (que, de um momento para o outro, possa deixar de dar continuidade a um produto).

Muitos obstáculos foram surgindo durante o desenvolvimento do *datalogger*. Por se tratar de uma aplicação que iria integrar uma nova bomba de calor desenvolvida especificamente pela Bosch Termotecnologia, muitas condicionantes surgiram: a consecutiva renovação do protocolo de comunicação e a não atualização da ECU disponibilizada para testes com estes mesmos *updates* veio impossibilitar a criação de um *datalogger* que contemplasse a versão final da bomba de calor. De notar que esta nova bomba de calor só tomou uma versão final em finais de julho, altura em que o trabalho prático referente à implementação do *datalogger*, representado no capítulo 4, estava essencialmente concluída. Esta, aliada ao surgimento de alguns obstáculos pessoais, que trouxeram imensos constrangimentos temporais, veio impossibilitar a contemplação destas alterações finais do protocolo no *sketch* desenvolvido. As repercussões discutidas neste parágrafo resultaram na utilização, ao longo deste trabalho, de uma ECU que possuía diversos bugs ao nível do *firmware*.

O trabalho aqui exposto incidiu na criação de um *datalogger* seguindo uma filosofia de *plug-and-play*, através da utilização de um plataforma de prototipagem

largamente utilizada nos dias que correm. Este datalogger permite a comunicação bidirecional com um *webserver* e permite também o controlo da aplicação a ele ligado pela utilização de um menu. Desta forma são disponibilizadas duas formas de controlo: controlo remoto e controlo *inloco*.

O datalogger desenvolvido é um dispositivo de pequenas dimensões e integra, no seu interior, uma *board* com capacidade de processamento e uma outra *board* acoplada a esta, que lhe atribui a capacidade de ligação à internet através de um protocolo *wireless* (*WiFi Shield*) e capacidade de armazenamento através da integração de um cartão SD.

Entende-se, assim, que apesar das limitações temporais encontradas, tanto da minha parte, como da falta de especificações corretas relativamente ao protocolo de comunicações com a ECU, o *datalogger* implementado cumpre as especificações impostas inicialmente e permite, devido à larga versatilidade considerada na sua implementação, a fácil adaptação do respetivo *sketch* a qualquer protocolo de comunicação Bosch que possa surgir, apenas requerendo ligeiras alterações em partes específicas do sistema.

Relativamente a trabalho futuro, deixo como proposta a atualização do sistema aqui desenvolvido para a última versão protocolar do sistema de comunicação da Bosch. Adicionalmente, uma adição importante para este projeto seria a possibilidade de programar o Arduino remotamente. Uma das formas de se concretizar este ponto seria alterar a forma como o *bootloader* está organizado, algo que foi analisado durante o decorrer do trabalho mas que, por falta de tempo útil para o concretizar, não foi implementado.

Bibliografia

[1] W. J. Mayo-Wells, *The Origins of Space Telemetry*, 4th ed. Society for the History of Technology, 1963.

[2] J. Brow, "Machine 2 Machine - Internet of Things - Real World Internet," 2011. [Online]. Available: http://www.slideshare.net/jackebrown/machine-2-machine-internet-of-things-real-world-internet-8996257. [Accessed: 10-Jul-2013].

[3] H. Research, "Where are the opportunities of Internet of Things?," 2013. [Online]. Available: http://harborresearch.com/tag/business-models/.

[4] "Machine to Machine Communication - Not a Fiction anymore." [Online]. Available: http://www.diplointernetgovernance.org/profiles/blogs/machine-to-machine-communication-not-a-fiction-anymore. [Accessed: 09-Jul-2013].

[5] P. Lucas, J. Ballay, and M. McManus, *Trillions: Thriving in the Emergin Information Ecology*, 1st ed. Wiley, 2012.

[6] T. Cantegrel, B. Cabé, and G. Morice, "M2M Workshop - EclipseCon Europe," 2011.

[7] "IntelliLogger Data Logging and Alarming System with Network Connectivity - Logic Beach." [Online]. Available: http://www.logicbeach.com/intellilogger/intellilogger.html. [Accessed: 10-Jul-2013].

[8] KeeLog, "SerialGhostWi-fi User's Guide," 2013. [Online]. Available: https://www.keelog.com/files/SerialGhostUsersGuide.pdf. [Accessed: 10-Jul-2013].

[9] "DataBridge SDR2-CF Serial Data Recorder RS-232 Data Logger Datasheet." [Online]. Available: http://www.acumeninstruments.com/products/SDR2-CF/SDR2-CF.shtml. [Accessed: 10-Jul-2013].

[10] "H7708 GSM/ GPRS unit, Hongdian Product Details from Shenzhen Hongdian Technologies Corporation." [Online]. Available: http://hongdian.en.alibaba.com/product/202188811-200074774/H7708_GSM_GPRS_unit.html. [Accessed: 10-Jul-2013].

[11] Robotee, "What is a microcontroller?," 2013. [Online]. Available: http://www.robotee.com/index.php/what-is-a-microcontroller-51059/. [Accessed: 01-Mar-2013].

[12] "PIC vs. AVR smackdown." [Online]. Available: http://www.ladyada.net/library/picvsavr.html. [Accessed: 11-Jul-2013].

[13] "PIC versus AVR." [Online]. Available: http://www.kanda.com/pic-vs-avr.php. [Accessed: 11-Jul-2013].

[14] A. Sowards and A. Goltz, "Introducing Raspberry Pi (RPi) – A Fully Programmable PC." [Online]. Available: http://infinigeek.com/introducing-raspberry-pi-rpi-a-fully-programmable-pc/. [Accessed: 10-Mar-2013].

[15] G. Van Loo and M. Vaninwegen, "Gertboard User Manual," 2012. [Online]. Available: http://www.farnell.com/datasheets/1683444.pdf.

[16] Liz, "RaspBerry Pi Official News," *Archived*, 2013. [Online]. Available: http://www.raspberrypi.org/archives/4100.

[17] Arduino, "Arduino Mega Datasheet." [Online]. Available: http://www.mantech.co.za/datasheets/products/A000047.pdf.

[18] I. Sram and W. E. Cycles, "– 8-bit Atmel Microcontroller ATmega640 / ATmega1280 / ATmega1281 / ATmega2560 / ATmega2561 / " 2012.

[19] "Arduino - AttachInterrupt." [Online]. Available: http://arduino.cc/en/Reference/attachInterrupt. [Accessed: 01-Sep-2013].

[20] "Wire \ Libraries \ Wiring." [Online]. Available: http://wiring.org.co/reference/libraries/Wire/index.html. [Accessed: 01-Sep-2013].

[21] Quora, "Arduino: What are the differences between Arduino and Raspberry Pi? -." [Online]. Available: http://www.quora.com/Arduino/What-are-the-differences-between-Arduino-and-Raspberry-Pi#. [Accessed: 11-Jul-2013].

[22] M. Engineering, "What are the differences between Arduino and Raspberry pi?" [Online]. Available: http://minuteeng.blogspot.com.br/2012/08/what-are-differences-between-arduino.html. [Accessed: 11-Jul-2013].

[23] A. Industries, "'Whats the difference between Arduino, Raspberry Pi, BeagleBoard, etc?'," 2012. [Online]. Available: http://www.adafruit.com/blog/2012/06/18/ask-an-educator-whats-the-difference-between-arduino-raspberry-pi-beagleboard-etc/. [Accessed: 11-Jul-2013].

[24] Bosch, *Electric Heat Pump Water Heater HP270 Manual*. pp. 1–36.

[25] Binglong's Space, "Arduino Serial Port Communication | Binglong's space," *October 26, 2011*. [Online]. Available: http://binglongx.wordpress.com/2011/10/26/arduino-serial-port-communication/. [Accessed: 01-Sep-2013].

[26] Atmel, "UART interrupt example for the ATmega2560: mega_uart_interrupt_example.c File Reference," 2011. [Online]. Available: http://asf.atmel.com/docs/2.11.1/mega.applications.mega_uart_interrupt_example.mega2560_stk600/html/mega__uart__interrupt__example_8c.html. [Accessed: 01-Sep-2013].

[27] Processing, "Overview. A short introduction to the Processing software and projects from the community." [Online]. Available: http://processing.org/overview/. [Accessed: 10-Apr-2013].

[28] W. Co and A. R. Reserved, "W5100 Datasheet," 2008. [Online]. Available: https://www.sparkfun.com/datasheets/DevTools/Arduino/W5100_Datasheet_v1_1_6.pdf.

[29] H. Wireless, "HDG104-details." [Online]. Available: http://www.hd-wireless.se/index.php?option=com_content&view=article&catid=3:content&id=49:hdg104-details. [Accessed: 24-Sep-2013].

[30] Microsoft, "How Windows Generates 8.3 File Names from Long File Names." [Online]. Available: http://support.microsoft.com/kb/142982/en-us.

[31] Xively, "What is Xively?," 2012. [Online]. Available: https://xively.com/whats_xively/.

[32] P. Bellamy, "Xively - Arduino Library," 2012. [Online]. Available: https://github.com/xively/xively_arduino.

[33] "Bug with !!! Using Mega 2560 - Arduino Forum." [Online]. Available: http://forum.arduino.cc/index.php/topic,46966.0.html. [Accessed: 28-Sep-2013].

[34] B. Greiman, "sdfatlib - A FAT16/FAT32 Arduino library for SD/SDHC cards - Google Project Hosting." [Online]. Available: https://code.google.com/p/sdfatlib/. [Accessed: 29-Sep-2013].

[35] "Tutorials - Xively." [Online]. Available: https://xively.com/dev/tutorials/.