# 4. Implementação

Este capitulo trata de clarificar as soluções e opções adoptadas na implementação do projecto, bem como dos requisitos a nível de hardware e software necessários para correr a aplicação final.

## 4.1 Linguagem de programação

Um dos aspectos fulcrais para a realização deste projecto foi a boa escolha da linguagem programática, as opções eram à partida limitadas uma vez que como descrito em (INSERIR\_REF\_PA\_INTRO) era essencial que o código produzido no âmbito do projecto corresse em qualquer dispositivo com baixas ou altas características a nível de hardware, bem como sob qualquer sistema operativo no mercado. A escolha ficou reduzida às linguagens *C* e *C++*, ambas reconhecidas pela sua eficiência e propagação no mercado. A linguagem *C* é conhecida por ser uma das mais eficientes uma vez que é a linguagem imperativa "mais próxima" do hardware e pelos seus baixos custos a nível de memória. A linguagem *C++* é uma extensão do *C* introduzindo o paradigma orientado a objectos. Um dos maiores problemas da linguagem *C++* é o facto de não existir bons compiladores, especialmente quando o target são sistemas embebidos. Apesar disso a linguagem *C++* é muito menos propicia a erros do programador do que a linguagem *C*, por ser uma linguagem tipificada, o código produzido em *C++* é mais genérico (*template metaprograming* (ADD\_REF)) e mais legível levando assim a esta ser a linguagem adoptada na elaboração do projecto.

## 4.2. Algoritmo de Goertzel

No capitulo 3 (INSERIR\_REF) descreveu-se como funciona o algoritmo e que técnicas foram abordadas de forma a que o algoritmo seja utilizado no âmbito deste projecto. Nesta secção do capitulo tratara-se de explicar os detalhes de implementação de todos os mecanismos necessários para utilizar o algoritmo.

Uma das características mais interessantes do algoritmo de Goertzel é o facto deste ser paralelizável, essa característica é fulcral para sistemas em que o tempo de resposta deve ser o mais curto possível, como é o caso deste projecto, assim foi criada toda uma infra-estrutura (*GoertzelController* add ref 4.3) que tira-se partido e controlasse todo o processo de processamento de amostras.

### 4.3. GoertzelController

Neste subcapítulo, filtro de Goertzel refere-se à implementação de todas as funcionalidades descritas no CAP\_3\_REF (filtragem das amostras, implementação do algoritmo, etc) necessárias para o bom funcionamento do algoritmo de Goertzel.

A infra-estrutura *GoertzelController* é responsável por controlar:

* Os filtros Goertzel (INSERT\_REFE\_TO\_THAT).
* O buffer onde as amostras são guardadas.
* Quando é que as amostras devem ser entregues aos filtros.
* Quando e como os resultados dos filtros devem ser apresentados.
* Toda a sincronização necessária para aceder a variáveis partilhadas.

A representa o *pipeline* de processamento de sinal utilizando o algoritmo de *Goertzel*.

Existem três módulos intrínsecos na infra-estrutura, o primeiro passa por salvaguardar as amostras num buffer interno (*Samples Manager*) onde serão posteriormente enviadas para os filtro *Goertzel,* o segundo é a detecção das frequências presentes na amostra (*Goertzel Filters*) e o ultimo é o armazenamento e complemento dos resultados retornados pelos filtros com *metadata* gerada pelo controlador (*Results Controller*).

Figura - Diagrama de blocos do processamento de sinal.

### 4.3.1 Configuração e Parametrização

Os módulos do *GoertzelController* funcionam com valores gerados/calculados previamente à sua execução, (coeficientes dos filtros, gama de frequências, etc) para tal foi necessário criar um estrutura que defina esses valores de forma a que fosse possível implementar código genérico ao nível de funcionamento. As estruturas são as seguintes:

* ***GoertzelFrequency*** - representa uma frequência, contem os valores da frequência e do coeficiente de *Goertzel*.
* ***GoertzelFrequenciesBlock*** - representa uma gama de frequências, nesta está presente a frequência de amostragem da gama, o seu valor de *N*, o salto necessário para realizar a divisão da frequência de amostragem por software(CAP3\_RESOLVER\_RESO\_DO\_GOERTZEL), um array com os valores do filtro a utilizar sobre esta gama e finalmente o array de frequências que pertencem a esta gama.
* ***GoertzelResult*** - representa o resultado produzido pelo filtro de Goertzel, contem uma referencia para a frequência encontrada (*GoertzelFrequency*) e uma percentagem com a diferença entre a energia total do sinal e a energia relativa calculada pelo algoritmo de *Goertzel*.
* ***GoertzelResultCollection*** - representa todos os resultados que a infra-estrutura produziu num dado momento, contem um *array* de resultados (*GoertzelResult*), quantos resultados estão presentes no *array* e quantos **blocos** foram utilizados para produzir os resultados.

Um **bloco** representa uma sequencia de amostras com o tamanho do máximo *N* presente ao longo de todas as gamas.

O *GoertzelController* precisa apenas de um *array* com as gamas de frequências que deve processar (*GoertzelFrequenciesBlock*) sendo a única parametrização necessária para o funcionamento da infra-estrutura.

.

### 4.3.2 Samples Manager

Este modulo da infra-estrutura tem como principal funcionalidade de salvaguardar as amostras que vão sendo fornecidas ao *GoertzelController*, para além de ir calculando a energia das amostras à medida que vão sendo fornecidas, evitando assim que seja necessário percorrer todas as amostras para calcular a sua energia.

Este módulo tem ainda como responsabilidade saber quando dar amostras aos filtros de *Goertzel*, para tal é necessário ter em *buffer* pelo menos *N* amostras, sendo o valor de *N* o máximo *N* de todas as gamas de frequências previamente calculados (CAP 3 TAB das gamas). Por fim o módulo ainda permite descartar blocos quando o controlador achar necessário faze-lo, avisando igualmente os filtros que foram blocos descartados.

### 4.3.3 Goertzel Filters

A infra-estrutura tira partido da paralelização do algoritmo através da criação de diferentes fios de execução (tarefas) para executar cada um dos filtros, como ilustra a Figura 2.



Figura - Diagrama de blocos do módulo Goertzel Filters.

O segundo módulo do controladoré responsável por filtrar o sinal para a sua gama de frequências e de produzir os resultados. Este filtro usa a mesma estratégia utilizada no módulo *Samples Manager* de ir fazendo trabalho à medida que seja possível*.* Uma particularidade destes filtros é que estes só invocam o *Goertzel* caso exista evidencias que alguma das frequências da gama que está a processar se encontra nas amostras, para tal este vai guardando a energia do sinal filtrado à medida que filtra as amostras, assim quando a condição para chamar o *Goertzel* (ter as *N* amostras) esteja cumprida é comparado o valor da energia do sinal (calculado pelo *Samples Manager*) com a energia do sinal filtrado (calculado no filtro), assim o *Goertzel* é apenas invocado quando a energia do sinal filtrado esteja sobre um limiar configurável. Estes filtros estão ainda preparados para descartar todo o trabalho (filtragem, calculo da energia) caso recebem essa ordem do módulo *Samples Manager.*

### 4.3.4 Results Controller

O *Results Controller* é o módulo responsável por gerir os resultados, esta gestão é necessária uma vez que uma das particularidades desta infra-estrutura é que os filtros podem produzir resultados em alturas diferentes, por exemplo enquanto que a frequência de 4186.01Hz necessita apenas de 179 amostras para ser possível saber se está presente no sinal ou não, a frequência 55Hz necessita de 2704 amostras (CAP3 TRATAMENTO DA RESOLUCAO\_DO\_ALGO) , assim foi utilizada uma técnica de *double buffering(INSERT\_REFERENCE)* de forma a que os filtros tenham sempre algum local onde preservar os seus resultados. Este módulo gera ainda *metadata* para ser utilizada pela aplicação que utilize esta infra-estrutura, nomeadamente quantas frequências foram encontradas e quantos blocos de amostras foram necessários para encontrar todas as frequências encontradas, com esta informação é possível por exemplo que uma aplicação saiba quanto tempo é que uma dada frequência esteve activa nas amostras passadas ao controlador.

### 4.3.5 Funcionamento e Características

A Figura 3 representa o *flowchart* do controlador:

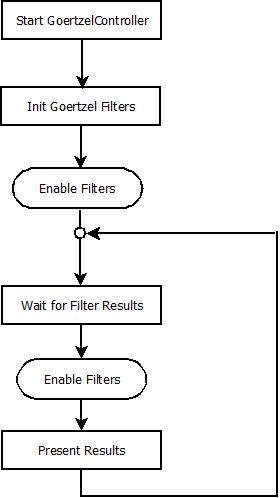


Figura - Flowchart do funcionamento do GoertzelController.

Como já foi referido esta infra-estrutura não funciona apenas com um fio de execução (INSERIR\_REF\_PA\_GOERTZEL\_FILTERS) mas sim com vários por isso é natural que esta apresente os resultados de forma assíncrona, assim para que uma aplicação obtenha resultados desta infra-estrutura é necessário registar um *callback* com um protótipo definido, representado na Listagem 1.

typedef void (\*GoertzelControllerCallback)(GoertzelResultCollection \* results);

Listagem - Protótipo do callback do GoertzelController.

Na Figura 3 essa característica também é visível uma vez que o controlador depois de esperar pelos resultados do *ResultsController* reactiva os filtros e antes de chamar o *callback* definido (*Present Results*).

A interface publica do controlador permite ainda acesso ao módulo *Samples Manager* de forma a que a infra-estrutura não esteja comprometida com a aquisição do sinal. Este acesso reduz-se a dois modos:

* *Single mode*, a aplicação irá fornecer ao controlador amostras singulares ao longo do tempo para este processar, neste modo o controlador toma conta de gerir toda a informação sobre a amostra e o bloco onde se encontra.
* *Burst mode*, a aplicação irá utilizar um ponteiro fornecido pelo controlador para guardar amostras, a aplicação neste caso é responsável por calcular a energia do sinal bem como toda a lógica necessária para evitar *buffer overflow*. (INSERT\_REFERENCE\_TO\_BUFFER\_OVERFLOW), após as amostras desse bloco estarem preenchidas o controlador deve ser avisado de forma a libertar acesso ao bloco para os filtros.

Independentemente do modo de utilização, a aplicação terá sempre de contactar o controlador à medida ou quando acabar de preencher o bloco, assim existe um ponto após a aquisição de sinal onde é possível manter sempre controlo no que diz respeito ao processamento de sinal.

Esta infra-estrutura foi desenhada para ser portável entre arquitecturas de alta ou baixa gama de processamento, assim não existe nenhum compromisso com memória dinâmica nem funções especificas com o sistema, apesar disso existe algum compromisso entre arquitecturas uma vez que para esta infra-estrutura funcionar plenamente é necessário ser possível de criar múltiplos fios de execução (Tarefas).

Outra característica da infra-estrutura é o facto de no inicio da aplicação ser calculado o valor da energia do ambiente, ou seja, por um tempo configurável nenhuma amostra é avaliada pelos filtros mas sim pelo próprio controlador de forma a detectar o ruído presente no ambiente. Assim é possível avaliar, sempre que um bloco de sinal é adquirido, se existe alguma nota presente no sinal, através da comparação entre a energia do sinal recolhida no arranque da infra-estrutura com a energia do bloco adquirido. A Figura 4 representa essa característica da infra-estrutura, apesar de estar desenhada para o modo *Single* o comportamento no modo *Burst* é idêntico.

## C:\Users\DVD\Desktop\SamplesModeFlowchart.jpeg

Figura - Flowchart to tratamento de ruído da infra-estrutura.

## 4.4 Arquitectura PC (Windows)

Toda a implementação da infra-estrutura *GoertzelController* foi realizada sobre a arquitectura de PC de modo a testar a funcionalidade do algoritmo *Goertzel* bem como para calcular o tempo de processamento (LINK\_TO\_TESTS).

Para fazer a aquisição de sinal foi utilizado a biblioteca de multimédia do Windows, onde é possível configurar a placa de som do sistema para colocar amostras num array. Posteriormente à recepção das amostras da placa de som, estas são passadas ao controlador através do modo *Single*.

Uma vez que a implementação é executada sob um sistema operativo, a implementação estática da infra-estrutura *GortezelController* é algo desvantajosa uma vez que não utiliza todos os recursos que poderia utilizar (e.g memoria dinâmica) , mas por outro lado como actualmente praticamente todos os PCs têm múltiplos cores a escalabilidade da infra-estrutura é uma grande vantagem no que diz respeito a tempos de processamento(TESTES) neste tipo de arquitecturas.

## 4.5 Arquitectura ARM7

Na INTRODUÇÃO/DESCRIÇÃO foram referidas as especificações do *yaab2294*, kit de desenvolvimento que foi usado para a elaboração deste projecto, as especificações deste hardware estão muito próximas das usadas no mercado (para este tipo de aplicações) tendo sido essa uma das razões para a escolha do hardware.

### 4.5.1 Aquisição de sinal

A aquisição de sinal foi realizada com o ADC a 8800Hz, este gera uma interrupção sempre que tem uma amostra pronta, o tratamento desta trata de guardar o valor da amostra adquirida num *array* fornecido pela infra-estrutura *GoertzelController* e de calcular a contribuição da amostra para a energia total do bloco.

### 4.5.2 Sistema Operativo

Uma das grandes diferenças entre executar a infra-estrutura implementada num PC e num ARM é o simples facto do PC ter um sistema operativo em execução a virtualizar o hardware e como a infra-estrutura foi desenhada para correr sob alguma abstracção (nomeadamente de tarefas) foi necessário encontrar um sistema operativo para portar a infra-estrutura para o *ARM*. Existem muitos sistemas operativos *open-source* para a arquitectura *ARM*, para a realização deste projecto foram abordados três:

* ***eCos*** (ADD\_REF) - um dos maiores sistemas operativos no mercado, conhecido por estar disponível para várias arquitecturas e *targets* diferentes.
* ***FreeRTOS*(**ADD**\_**REF**)** - um sistema bastante mais pequeno que o *eCos*, mas igualmente com alguma presença no mercado.
* **TNKernel**(ADD\_REF) - o sistema mais pequeno de todos, praticamente só trás consigo mecanismos de sincronização e de escalonamento de tarefas.

O único que seria necessário fazer o *port* seria o *eCos*, uma vez que para os outros dois existem já versões para o processador que se usou no projecto. Na escolha do sistema operativo foi tido em conta a latência do tratamento de interrupções, uma vez que era importante que fosse o mais rápido possível para não perder amostras, o tempo de comutação de uma tarefa, já que a infra-estrutura funciona com múltiplas tarefas em simultâneo é importante que este tempo seja baixo, por fim a implementação dos sincronizadores, uma vez que é importante saber se os sincronizadores desligam as interrupções de forma a obter exclusão para secções criticas, pela mesma razão referida em cima.  
 Com os critérios definidos, foram avaliados os três sistemas escolhidos, o *eCos* foi excluído quase de imediato devido aos seus elevados tempos de comutação (LINK\_PARA\_TESTES), tanto o *TNKernel* e *FreeRTOS* desligam as interrupções em chamadas a secções criticas inviabilizando-os. A solução ideal seria um sistema operativo que apenas desliga-se as interrupções quando fosse mesmo necessário (e.g. *context switch*). Optou-se então por fazer um sistema operativo que cumprisse todos os requisitos necessários.

#### 4.5.2.1. Micro Operating System (mos)

O *mos* foi criado no âmbito deste projecto de forma a cumprir todos os requisitos necessários para correr a infra-estrutura, este é baseado numa versão em C++ do núcleo *uthreads* leccionado nas aulas de Programação Concorrente. Em comparação com os sistemas operativos referidos neste capitulo (*FreeRTOS* e *TNKernel*) o *mos* oferece alguma virtualização de periféricos básicos no mundo dos embebidos (*e.g. SPI*, *UART*, *TIMER*, etc). Apesar de ter sido criado para cumprir os requisitos da infra-estrutura de processamento de sinal, o *mos* é completamente independente e adjacente a esse facto.

O mos está dividido em três partes:

* *Kernel* - o núcleo do sistema operativo, trata de todo o sistema multi-tarefa bem como o tratamento de interrupções.
* *Port* - todo o código comprometido com o hardware, este módulo está dividido em arquitectura, onde contém todo o código comum à arquitectura e *target* onde se encontra as *drivers e startups* especificas de um processador/*board*.
* System - a virtualização do hardware, aqui são definidas contractos que cada *Port* deverá cumprir e/ou implementar para o funcionamento do sistema.

Todos os requisitos da infra-estrutura referidos anteriormente foram cumpridos no kernel do sistema:

* *Tempo de comutação de tarefas reduzido*, este requisito foi implementado sob o *context switch* do sistema, este apenas salvaguarda a informação necessária para que a tarefa possa ser resposta futuramente.
* *Tempo de latência das interrupções baixa*, para diminuir o tempo de latência foi implementado um esquema semelhante ao do *eCos* cuja principal característica é delegar o processamento (que não seja crucial) da interrupção para um tarefa definida pelo sistema, que correrá já com as interrupções ligadas. Outra solução teria sido fazer o tratamento das interrupções dentro do contexto da tarefa que estava a executar previamente ao pedido de interrupção, mas esta solução trás é mais propicia a erros nomeadamente a *buffer overflow.*
* *Sincronização sem desligar as interrupções,* para resolver este requisito foi criado um *lock* de sistema, este *lock* é um simples contador que quando está diferente de zero significa que está adquirido por alguma tarefa. O contador não é mais que uma "*flag*" reentrante que informa o sistema de tratamento de interrupções que não deve alterar nenhuma variável do sistema (*e.g* não deve trocar de tarefas), levando a que o sistema esteja sempre sensível a interrupções.

### 4.5.3 *Port* da infra-estrutura para *ARM*

Após a implementação do sistema operativo *mos* e da aquisição de sinal o *port* da infra-estrutura reduziu-se a alterar as criações das tarefas para usar a API do *mos* à do *windows*.

Uma vez que a infra-estrutura utiliza aritmética com *floating-point* foi utilizada a biblioteca da *tool-chain* que sabe realizar tais operações por software. O problema da utilização desta biblioteca é que acrescentou um peso considerável ao processamento de sinal (INSERT\_TESTS\_OF\_OPERATIONS), uma vez que as operações realizadas nos filtros e no *Goertzel* são multiplicações e adições de *doubles.*   
 Após a implementação foram realizadas testes à infra-estrutura (INSERT\_TESTS\_AGAIN) e reparou-se que a infra-estrutura demorava cerca de 17 segundos para processar um bloco de 1 segundo (8800 amostras). Este facto é contra um dos objectivos do trabalho, latência mínima entre a acção de tocar o instrumento e a reflexão dessa na pauta.

A solução encontrada foi abandonar a precisão *floating-point* e usar inteiros a 64bits, para não perder toda os valores decimais utilizados foram multiplicados por uma constante. No final foi possível reduzir o tempo até XX (INSERT\_REF).