UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

CURSO DE ENGENHARIA DA COMPUTAÇÃO

ANÁLISE DE PERFORMANCE DE APLICAÇÕES DESENVOLVIDAS UTILIZANDO O ESCORT

Luís Felipe Prado D'Andrada

Trabalho de Graduação

Recife
AGOSTO de 2013

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Luís Felipe Prado D'Andrada

ANÁLISE DE PERFORMANCE DE APLICAÇÕES DESENVOLVIDAS UTILIZANDO O ESCORT

Trabalho apresentado ao Programa de GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO do CENTRO DE INFORMÁTICA da UNIVERSIDADE FEDERAL DE PERNAMBUCO como requisito parcial para obtenção do grau de Bacharel em ENGENHARIA DA COMPUTAÇÃO.

Orientado por: Sergio Vanderlei Cavalcante

Recife
AGOSTO de 2013

Aos meus pais.

Agradecimentos

Agradeço primeiramente aos meus pais, que, cada um do seu jeito, me apoiaram, confiaram em mim e me fizeram crescer como pessoa.

Aos meus amigos de curso, especialmente Hector e Julio, os quais tive o prazer de trabalhar em conjunto inúmeras vezes.

Ao meu amigo e instrutor Gustavo Melo, que sempre teve paciência para me ajudar a crescer academicamente e profissionalmente.

Ao meu orientador Sérgio Cavalcante, que aceitou me orientar mesmo tendo uma agenda sempre cheia.

E a todos os professores do Centro de Informática, por todos os ensinamentos, sejam acadêmicos ou não.

Resumo

Com o aumento da capacidade de processamento e armazenamento dos componentes de hardware, além do barateamento, é possível atender a uma demanda por sistemas embarcados cada vez mais complexos. Apesar da maior complexidade exigida, o desejado time-to-market também está cada vez menor, assim se fazendo necessário o uso de uma ferramenta para o desenvolvimento de software embarcado que possibilite um desenvolvimento mais rápido, podendo se utilizar da constante melhora dos componentes de hardware.

O ESCoRT (Embedded Software Component: a Reuse Technology) é um arcabouço teórico, metodológico e operacional para o desenvolvimento de software embarcados utilizando componentes. O principal objetivo do ESCoRT é reduzir o tempo de desenvolvimento do software embarcado focando no reúso e portabilidade do software produzido.

Como o uso de memória e CPU continuam sendo limitações de um sistema embarcado, deve-se saber o quanto a utilização do ESCoRT no desenvolvimento de um software embarcado pode impactar sua performance. Para isto, foi comparado o desempenho de aplicações desenvolvidas tradicionalmente, utilizando a linguagem C e sem se preocupar com a reusabilidade do código produzido, e aplicações desenvolvidas utilizando os conceitos do ESCoRT.

Palavras-chave: Sistemas Embarcados, ESCoRT, Análise de performance, Orientação a Objetos, Modularidade, Reúso.

Lista de Figuras e Tabelas

Sumário

Capítulo 1

Introdução

Este capítulo apresenta o contexto o qual o ESCoRT está inserido e destaca a motivação deste trabalho. Em seguida é exposto como esta dissertação atacará o problema.

1.1 Motivação

A tecnologia está cada vez mais presente no nosso dia-a-dia. Seja através de poderosos smartphones, com aplicações cada vez mais impressionantes, computadores, ou mesmo através de automóveis e outros produtos que trazem consigo um sistema computacional. Muitos outros sistemas complexos também contam com algum sistema computacional embutido, também conhecido como Sistema Embarcado. Aparelhos que antigamente não tinham nenhum tipo de inteligência, como geladeiras, hoje em dia possuem sistemas computacionais embutidos, assim gerando produtos muito mais interessantes, além de aquecer ainda mais o mercado de Sistemas Embarcados. Uma das principais caracteristicas de um sistema embarcado é a sua limitação de recursos, como memória e consumo de energia.

A contínua melhora de desempenho dos sistemas computacionais não é vista apenas em computadores pessoais. No âmbito de sistemas embarcados, este avanço proporcionou uma grande melhora na capacidade de processamento e armazenamento dos sistemas presentes no mercado[12], como pode ser observado claramente nos celulares atuais, que contam com processadores cada vez mais poderosos e uma boa quantidade de memória disponível.

Com a crescente melhora de desempenho, também mencionada em [10], faz-se possível o uso de alguma ferramenta para o desenvolvimento de software embarcado que consiga atender as necessidades do mercado por sistemas cada vez mais complexos e com time-to-market cada vez menores. O ESCoRT (Embedded Software Component: a Reuse Techonology), um arcabouço teórico, metodológico e operacional para o

desenvolvimento de software embarcado introduzido por Gustavo Melo[1], propõe, entre outras coisas, reduzir o time-to-market de softwares embarcados utilizando conceitos de orientação a objetos para aumentar a capacidade de reúso e portabilidade do código produzido.

Uma das maiores preocupações no desenvolvimento de software embarcado é a capacidade de processamento e armazenamento do dispositivo a ser utilizado, como é mencionado em [11]. No mercado atual, existem inúmeras variedades de opções, porém, para obter um maior lucro em produtos a serem desenvolvidos, busca-se utilizar uma opção barata e que consiga executar as funcionalidades propostas com perfeição.

Sistemas operacionais, como Linux e Windows CE, também apresentam-se como uma boa opção para aumentar a velocidade do desenvolvimento de sistemas embarcados, pois provêm uma abstração de hardware e muitos códigos disponíveis através da comunidade. Porém, utilizar sistemas operacionais como estes requer um hardware muito mais potente, o que tornaria o sistema final muito mais caro, tornando-se uma opção pouco viável para muitas aplicações.

1.2 Justificativa

Antes do desenvolvimento do ESCoRT, a FRT Tecnologia Eletronica, empresa a qual esta metologia foi implantada, desenvolvia os softwares embarcados utilizando a linguagem C, sem utilizar os conceitos utilizados no ESCoRT, que foi idealizado para haver uma grande reusabilidade de código. O uso da linguagem C para o desenvolvimento de sistemas embarcados é bastante difundido e é bastante comum os desenvolvedores criarem códigos bastante específicos e otimizados, assim não se preocupando muito com a reusabilidade do mesmo.

Apesar do poder do ESCoRT em reduzir o tempo de desenvolvimento do software embarcado ter sido demonstrado em [1], não se tem uma boa noção da perda de performance gerada. Como a preocupação com a performance é bastante comum e necessária nessa área, para o ESCoRT ser amplamente aceito deve haver um estudo sobre o impacto na performance gerada, tanto no processamento (mais tempo para executar uma mesma tarefa) quanto em termos de memória.

Segundo Gustavo[1], o fato de se utilizar orientação a objetos e aumentar a modularização de um software pode afetar na performance e consumo de memória do sistema, questões muito importantes no desenvolvimento de software embarcado. Medir estes atributos é algo essencial para avaliar a aplicabilidade do ESCoRT.

1.3 Objetivos

Os objetivos deste trabalho são:

- Verificar se o uso do ESCoRT no desenvolvimento de uma aplicação causa prejuízo a performance e/ou aumenta o tamanho da memória necessária.
- Entender o porquê o uso de ESCoRT acarreta ou não num prejuízo da performance e uso da memória do sistema.
- Dar uma estimativa do quanto um projeto desenvolvido utilizando os conceitos do ESCoRT pode ser prejudicado na performance e no uso de memória.

1.4 Estrutura do documento

O capítulo 2 contém um background sobre o aumento da produtividade do software, sobre o desempenho da linguagem C++ em ambientes embarcados e, por fim, uma descrição do ESCoRT. No Capítulo 3 será mostrado como serão feitas as análises, além de explicações sobre o que pode causar prejuízo a performance. No Capítulo 4 serão introduzidos os exemplos que farão parte do experimento realizado. No Capítulo 5 serão expostos os resultados dos experimentos, além de explicações dos resultados obtidos. Por fim, o Capítulo 6 aprensenta as conclusões deste trabalho.

Capítulo 2

Background

2.1 Aumento de produtividade

2.1.1 A crise do software e a evolução das linguagens de programação

O termo “Crise do Software” é utilizado para expressar a dificuldade de desenvolvimento rápido de um software de qualidade, visto que a demanda por software e a complexidade destes são cada vez maiores.

Uma das primeiras menções a este termo foi feita em [17], por Dijkstra. Ele menciona o rápido crescimento da indústria de hardware e também a falta de métodos de desenvolvimento de software como uma das grandes dificuldades para os programadores. O rápido avanço da indústria de hardware, trazendo máquinas cada vez mais poderosas, gerava demanda para softwares cada vez mais complexos. Atender esta demanda e manter o software funcionando era cada vez mais dificil com as ferramentas existentes; linguagens como FORTRAN são criticadas por Dijsktra, que, neste texto, demonstra sua esperança que as futuras linguagens sejam bastante diferentes das atuais. Além disso, seriam necessárias também metodologias de desenvolvimento de software que pudessem gerar softwares de melhor qualidade em um intervalo de tempo menor. A produção de software na época era mais uma arte do que uma ciência.

O reúso de software era uma das soluções apontadas para acabar com a crise, segundo [19]. Porém, o reúso de software falhou em se tornar uma prática padrão de desenvolvimento de software, apesar de suas vantagens serem reconhecidas.

A evolução da linguagens de programação também deve-se a dificuldade de produção do software. Aumentando o nível de abstração é possível desenvolver programas mais complexos mais facilmente. Além disso, muitas linguagens já tinham bibliotecas com rotinas reusáveis para operações comumente utilizadas.

A fim de promover o reúso, maior modularidade e melhor nível de abstração do software produzido, surgiram as liguagens orientadas a objetos. Os conceitos de encapsulamento, herança, polimorfismo são essenciais para alcançar esta meta. Para

Deitel[13], fazer um projeto modular, orientado a objetos pode tornar grupos de desenvolvedores de software muito mais produtivos do que era possível com técnicas anteriores.

A dificuldade de se produzir um software não se limitou a época de Dijkstra. Segundo o "The Standish Group Report", datado de outubro de 2000, apenas 16,2% dos projetos de software são finalizados dentro do prazo e dentro do orçamento inicial [18].

Apenas o uso de linguagens com maior nível de abstração não resolve o problema da crise do software. Deve-se adotar o estilo de programação que consiga produzir melhores softwares em menos tempo. Para atingir este objetivo o reúso de software é uma poderosa opção, pois reusar componentes poupa o trabalho de reimplementá-los, além de que, como o componente é reusado em outras aplicações, já foi amplamente testado. Os conceitos de orientação a objetos permitem a criação de classes reusáveis, sendo assim, este pode ser o caminho para aumentar a produtividade e atender a demanda do mercado.

2.1.2 Reúso de software

Como vimos na sessão anterior, o reúso foi apontado como uma possível saída para a crise do software. O conceito de reúso é antigo, porém nunca conseguiu se tornar um padrão. Este conceito é basicamente utilizar artefatos de software existentes durante a construção de um novo sistema.

Serão mencionadas três técnicas de reúso, também vistas em [21]: Padrões de Software, Construção de Framework e Desenvolvimento Baseado em Componentes.

2.1.2.1 Padrões de Software

Descrevem soluções para problemas que ocorrem com frequência no desenvolvimento de software. Um padrão deve documentar um problema recorrente, uma solução e a situação em que deve ser aplicado.

2.1.2.2 Framework

Um framework é uma aplicação semi-completa reutilizável que, quando especializada, produz aplicações personalizadas. Esta técnica utiliza uma abordagem

top-down, na qual o desenvolvimento começa pelo entendimento do sistema contido no framework para, assim, detalhar as particularidades do sistema definido pelo usuário.

2.1.2.3 Componentes

Um componente é uma unidade de composição com intefaces bem definidas. Um software baseado em componentes é construído a partir de uma interligação de componentes, assim reduzindo o esforço necessário para produzir a aplicação, supondo que sejam utilizados componentes reutilizáveis já prontos.

2.2 Performance de C++ em ambientes embarcados

Um dos principais fatores que podem prejudicar na performance das aplicações desenvolvidas é o uso de orientação a objetos, neste caso específico a linguagem C++. A linguagem C já é bastante consolidada na área, porém o uso de C++ vem crescendo gradativamente [20], devido as suas inúmeras vantagens, além da popularidade do paradigma OO.

Nesta seção serão mostrados dois casos de estudo de performance de C++ em Sistemas Embarcados, ambas fazendo uma comparação de uma mesma aplicação desenvolvida em C. No primeiro exemplo serão utilizados benchmarks da OOPACK, que, segundo [3], são feitos de forma que o compilador possa transformar o código orientado a objetos em um código estilo C (procedural). Enquanto no segundo será comparado o desempenho de um kernel embarcado chamado DARK com sua versão orientada a objetos chamada de DARK++.

2.3.1 Benchmark OOPACK

Um dos propósitos desse trabalho é investigar o efeito de técnicas orientadas a objeto comparadas com o tradicional estilo de programação procedural em relação as suas performances. Com isso, o autor pretende mostrar que, caso não aplicada corretamente, o uso de orientação a objetos pode afetar significantemente a performance da aplicação.

A metodologia utilizada foi selecionar benchmarks de comparação de desempenho entre C e C++ e testá-los utilizando um compilador e um depurador para uma plataforma embarcada específica, neste caso ARM, assim obtendo os dados desejados.

As aplicações utilizadas foram:

2.3.1.1 “Max”

Mede o quão bem o compilador traduz um simples condicional.

Este exemplo usa uma função em ambos os estilos (C - procedural, C++ - orientado a objetos) para computar o máximo de um vetor. Enquanto o código estilo C faz uma comparação entre dois elementos explicitamente, a forma orientada a objetos faz a compação utilizando uma função inline.

2.3.1.2 Matrix

Mede quão bem o compilador propaga constantes e expressões invariantes.
Este exemplo multiplica duas matrizes contendo números reais para medir a eficiencia de duas clássicas otimizações: invariant hosting e strength-reduction. Invariant hosting é, basicamente, calcular fora do loop expressões que não mudam, assim não precisando recalcular o mesmo valor em todas as iterações. Enquanto o strength-reduction consiste em trocar uma operação custosa por outra equivalente, porém menos custosa.

2.3.1.3 Iterator

Mede o quão bem o compilador traduz pequenos objetos de vida curta.
Este exemplo computa um produto escalar usando um índice único comum na versão estilo C e usando um iterator em C++. Apesar de iterators serem considerados objetos "leves", eles podem causar um alto custo se compilados ineficientemente.

2.3.1.4 Complex

Mede quão bem o compilador elimina temporários.
Este exemplo mede a eficiência de C++ em lidar com aritmética complexa, multiplicando elementos de dois arrays contendo números complexos (definidos por uma classe). No estilo C, a operação é feita multiplicando explicitamente a parte imaginária e a parte real, enquanto em C++ a adição e multiplicação complexa são feitas utilizando operações sobrecarregadas (overload).

2.3.1.5 Resultados

Benchmark	code size (bytes)	Instructions	Cycles
OOPACK1_c	180	50536	77118
OOPACK1_oop	212	56032	91605
OOP Penalty	*17.78 %*	*10.88 %*	*18.79 %*
OOPACK2_c	308	5402229	8303851
OOPACK2_oop	424	5625529	9051974
OOP Penalty	*37.66 %*	*4.13 %*	*9.00 %*
OOPACK3_c	260	433042	635096
OOPACK3_oop	356	450049	677103
OOP Penalty	*36.92 %*	*3.93 %*	*6.61 %*
OOPACK4_c	620	1041241	1606642
OOPACK4_oop	804	1084256	1710665
OOP Penalty	*29.68 %*	*4.13 %*	*6.47 %*

Tabela 1 – Mostra o desempenho dos bechmarks OOPACK.

Pode se perceber através da tabela 1 que a utilização do estilo de programação orientado a objetos (utilizando C++) acaba trazendo um deterioramento na performance das aplicações. Em termos de ciclo de clock, a maior diferença ocorreu no benchmark 1, no qual foram necessários 18,79% ciclos de clock a mais para a execução da aplicação.

Através destes experimentos o autor conclui que deve-se tomar cuidado ao optar por utilizar um estilo de programação orientada a objetos, pois este pode gerar códigos bem maiores, como podemos ver na figura. Apesar disso, o autor destaca também as vantagens de OOP, principalmente sua capacidade em gerar códigos modularizados e reusáveis.

2.3.2 Kernel Embarcado DARK e DARK++

Neste artigo o autor[4] possui como um dos seus objetivos comparar a performance do DARK++, aplicação desenvolvida por ele, com o DARK, kernel já existente. Neste caso, é destacado que a aplicação DARK apresenta um código em C

com modularidade, reusabilidade e manutenção comparável ao desenvolvido em C++ (DARK++). Foi utilizado um simulador de uma plataforma embarcada para fazer os testes.

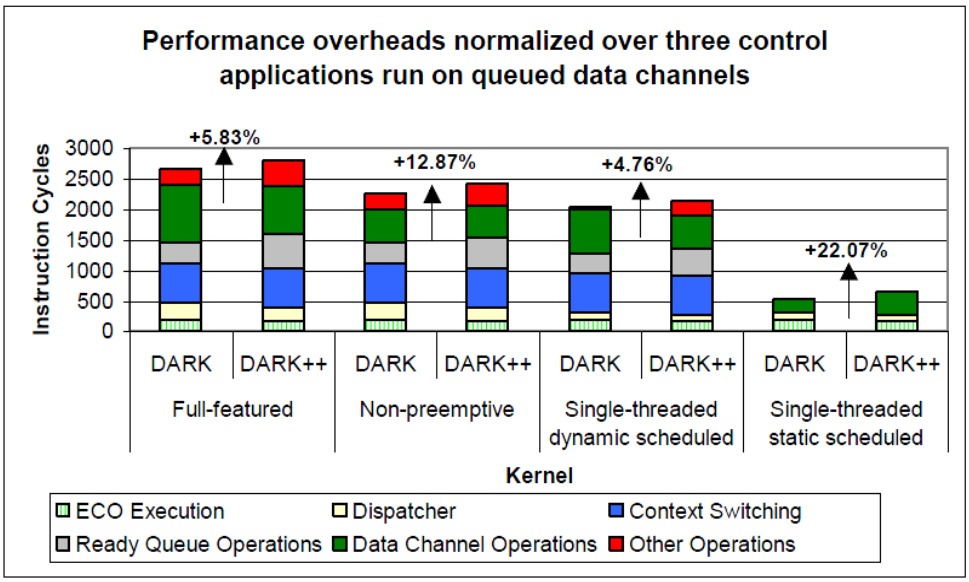

Figura 1 – Comparativo de performance DARK/DARK++ com filas.

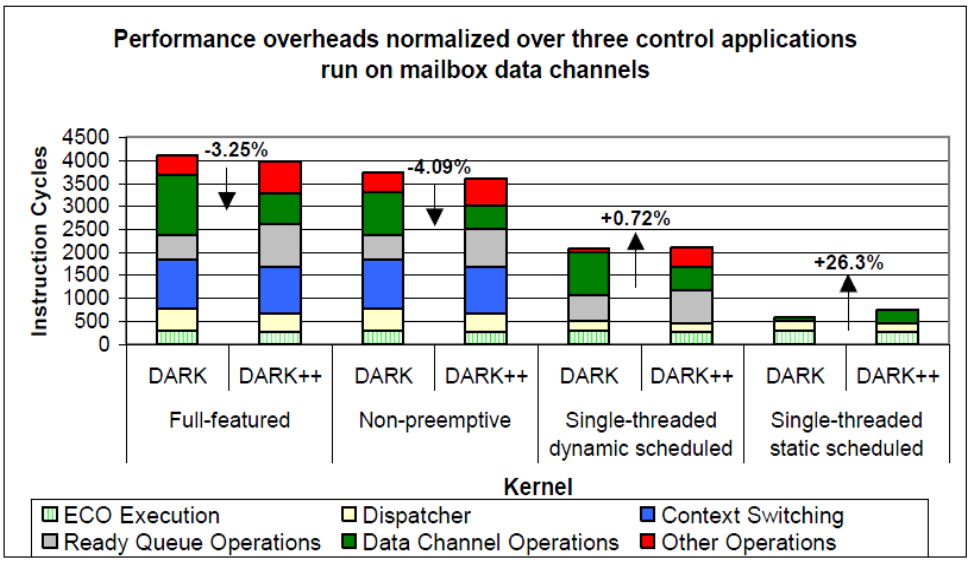

Figura 2 – Comparativo de performance DARK/DARK++ com mailboxes.

A figura 1 mostra o comparativo de desempenho do DARK com o DARK++ em 4 tipos de versões diferentes dos kerneis (Full-featured, Non-preemptive, Single-threaded dynamic scheduled e Single-threaded static scheduled) utilizando filas de mensagens, enquando a figura 2 faz o mesmo comparativo, porém utilizando mailboxes para a comunicação.

Pode-se observar uma performance similar do kernel DARK em comparação ao DARK++, porém, como mencionado pelo autor, a implementação orientada a objetos, neste caso, não traz muitos ganhos em relação a aplicação escrita em C no que diz respeito a reusabilidade e modularidade do codigo. O autor então conclui que códigos bem implementados em C++ podem ter perfomances similares aos implementados em C, usufruindo ainda das vantagens de C++.

2.3 ESCoRT

O ESCoRT (**E**mbedded **S**oftware **Co**mponent: a **R**euse **T**echnology) é um arcabouço teórico, metodológico e operacional para a construção de software embarcado. Seu principal objetivo é reduzir o tempo de desenvolvimento, através da maximização do reúso e da portabilidade.

Para conseguir uma maior portabilidade, a abordagem ESCoRT mantém o software modularizado ao máximo com interfaces padronizadas, de forma que, ao trocar de hardware, seja necessário apenas trocar componentes que se comunicam com o hardware. Além disso, ataca outra grande barreira para a portabilidade no campo de software embarcado, a existência de diversos compiladores com pequenas diferenças de sintaxes, utilizando uma forma de abstrair o compilador através de uma interface padronizada. Com isto, toda implementação especifica de um compilador estará isolada, de forma a garantir o desenvolvimento de componentes abstraindo-se o compilador.

Para alcançar a reusabilidade desejada, o ESCoRT utiliza-se de uma metodologia de desenvolvimento baseada em componentes. Se todos os componentes desejados para uma nova aplicação já foram desenvolvidos, o trabalho será apenas selecionar os componentes e "conectá-los". Uma das chaves para a reusabilidade no ESCoRT é, também, a modularização do código.

O ESCoRT aloca estaticamente os objetos que representam os componentes e executa suas inicializações. Isto descarta a necessidade de um heap para alocação dinâmica, que geralmente é necessário quando se utiliza C++, porém deve ser evitado em aplicações embarcadas.

2.3.1 Arquitetura

A arquitetura do ESCoRT é inspirada em outro trabalho [5] e utiliza o padrão de projetos Layered Pattern [6], de modo a organizar o software embarcado em camadas, assim facilitando o reúso e a manutenibilidade dos componentes. O principal objetivo da arquitetura é promover a portabilidade, organizando os componentes de acordo com sua função.

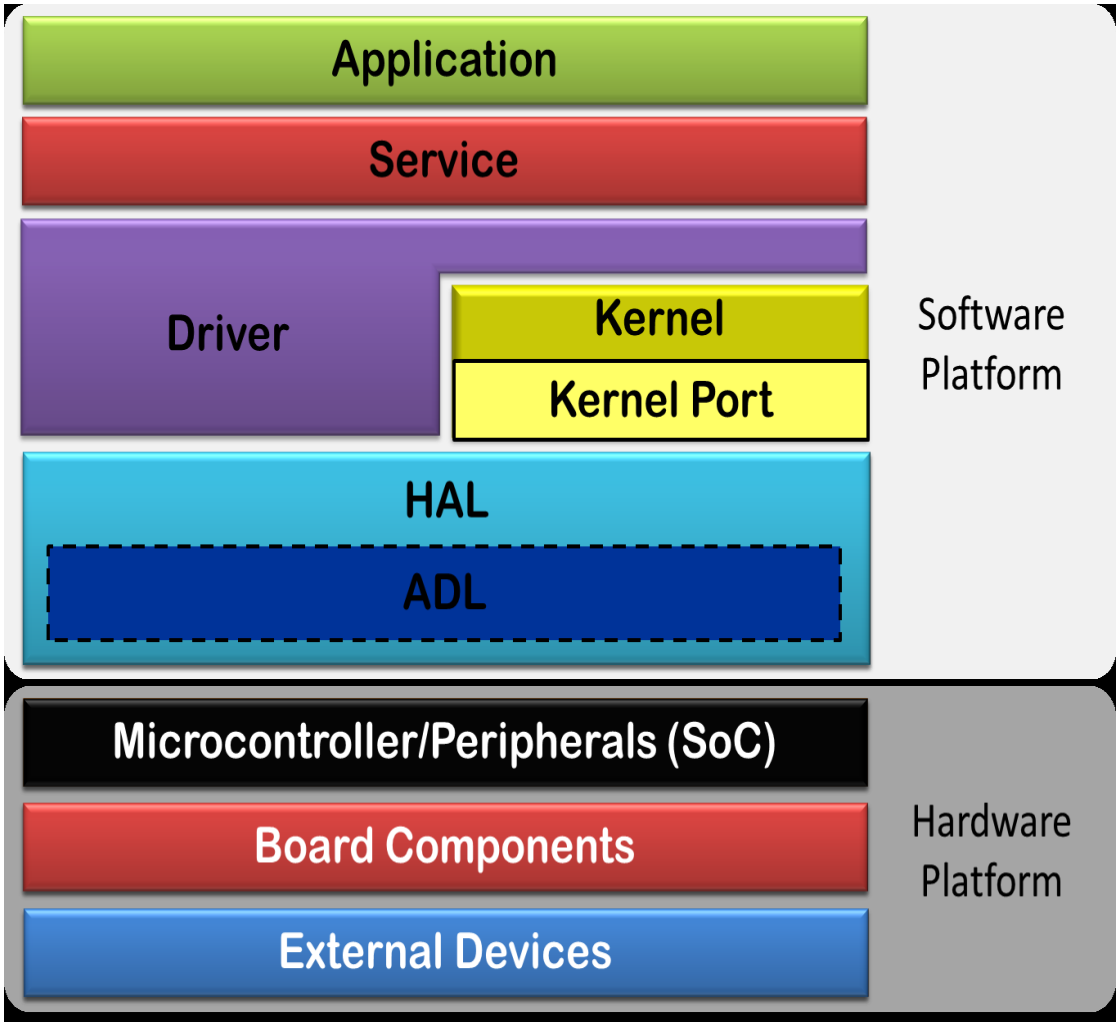

Figura 3 - Arquitetura ESCoRT.

A figura 3 mostra as camadas da arquitetura do ESCoRT que serão descritas nas seções subsequentes.

2.3.1.1 HAL

A camada HAL (Hardware Abstration Layer) é bastante difundida na área de Sistemas Embarcados. Como o próprio nome diz, esta camada visa a criação de uma abstração de hardware, ou seja, toda comunicação direta com o hardware é implementada nesta camada. Já que vários microcontroladores possuem periféricos em comum, é necessária a criação de uma API padronizada para estes periféricos. Assim, as camadas superiores podem ser indepedentes do hardware utilizado, pois irão se comunicar com este através da interface.

Ainda existe uma subcamada da HAL, a ADL (Architecture Dependent Layer). Nesta camada estão todas as definições de registradores necessários para o controle do hardware, normalmente feitas em arquivos header (.h) que são fornecidos pelo compilador.

2.3.1.2 Kernel

A função desta camada é encapsular o kernel utilizado, assim facilitando o reúso. Para promover uma maior modularidade, foi criada uma subdivisão chamada de Kernel Port, que implementa componentes dependentes do processador utilizado. Assim, é possível trocar de processador sem precisar criar outro componente Kernel.

2.3.1.3 Driver

Esta camada, assim como a camada HAL, provê uma abstração de hardware. Porém, os componentes HAL apenas fazem alteração e leitura de dados dos periféricos, enquanto os drivers implementam protocolos de acesso e tratamento de interrupções. Um componente Driver não é dependente do hardware, pois ele é construído com base na API da camada HAL.

2.3.1.4 Service

Algumas funcionalidades numa plataforma de software podem estar além do controle de dispositivos de I/O. Tais funcionalidades normalmente são serviços compartilhados pelo sistema operacional, tais como File System e TCP/IP. Por poderem ser reusados em várias aplicações, foi criada a camada Service.

2.3.1.5 Application

Nesta camada ficam componentes relativos a aplicação em si, que podem utilizar toda a plataforma construida nas camadas subjacentes.

2.3.2 Metodologia de Desenvolvimento

A metodologia baseada em componentes do ESCoRT propõe que primeiramente seja construida uma plataforma de software para prover os serviços necessários a aplicação. Depois, a aplicação é construida em cima desta plataforma.

Desta forma é possível que profissionais sem muita experiência no desenvolvimento de software embarcado possa desenvolver uma aplicação embarcada, utilizando uma plataforma criada por um profissional especialista na área.

Então, para se desenvolver um software usando a metolodia do ESCoRT, deve-se seguir o Fluxo de atividades proposto por Gustavo[1]:

2.3.2.1 Fluxo das atividades

1. O fluxo do trabalho começa com o Desenvolvedor do Compilador encapsulando a sintaxe específica do compilador, resultando no artefato Abstração do Compilador.
2. O Fabricante do Microcontrolador cria abstrações de hardware para os periféricos (como descrito na Seção 2.3.1.1), resultando em artefatos Componente HAL.
3. Em seguida o Desenvolvedor da Plataforma se responsabiliza por criar a configuração da plataforma com o compilador desejado.
4. Então o Desenvolvedor da Plataforma adiciona alguns dos componentes criados pelo Fabricante do Microcontrolador necessários para construir a plataforma. Assim o artefato Configuração Plataforma começará a ser refinado.
5. O Desenvolvedor da Plataforma cria o componente Kernel e depois o adiciona na plataforma. Dessa forma o artefato Configuração Plataforma é incrementado.
6. Depois o Desenvolvedor da Plataforma deve criar os componentes Driver necessários e em seguida adicioná-los à plataforma para refinar o artefato Configuração Plataforma.
7. Nesta etapa o Desenvolvedor da Plataforma irá criar os componentes Service e depois adicioná-los, para assim finalizar a implementação da plataforma de software embarcado.
8. Então entra a figura do Desenvolvedor da Aplicação, que começa por criar a configuração da aplicação a partir da configuração da plataforma.
9. Depois o Desenvolvedor da Aplicação cria os componentes que implementam a aplicação embarcada e em seguida os adiciona ao sistema, que neste ponto do desenvolvimento está completamente instanciado.

2.3.3 Arquivos Gerados Automaticamente pelo ESCoRT

Ao criar um componente, o ESCoRT deve gerar um arquivo .xml que conterá informações sobre o componente como, por exemplo, que este é uma especialização de outro determinado componente e que deve utilizar um determinado driver. Além do arquivo .xml o ESCoRT irá gerar um arquivo .cpp e um .h, o qual conterão apenas o "esqueleto" da classe relativa ao componente recém criado. Nesta classe deve ser implementada todos os métodos relativos ao componente.

Após a criação dos componentes deve-se adicioná-los ao sistema (no arquivo de configuração configuration.xml), assim o ESCoRT irá gerar os arquivos config.cpp e config.h que irão conter todo o instaciamento dos componentes e definicões feitas durante a configuração do sistema, além de rotinas para inicializar cada uma das camadas da arquitetura do ESCoRT. Assim, deve ser criado um arquivo Main.cpp para chamar os métodos de inicialização de cada camada e o método de execução do componente de aplicação desejado.

Deve ficar claro que os arquivos .xml são utilizados pelo ESCoRT apenas para gerar o código referente as configurações realizadas em cada componente. Este código é gerado nos arquivos config.h e config.cpp. Os arquivos .cpp e .h gerados para cada componente criado são apenas "esqueletos" que podem ser modificados completamente pelo usuário.

2.3.4 Considerações Finais

A arquitetura do ESCoRT é um grande trunfo. Ela permite, entre outras coisas, a troca de hardware de um projeto apenas modificando os componentes dependentes do hardware, ou seja, reusando todo código das camadas Driver, Kernel, Service e Application. E, ao desenvolver uma nova aplicação para um mesmo hardware, será possível reusar todos os componentes da camada HAL desejados na nova aplicação. Além disso a arquitetura permite que um profissional não especialista consiga desenvolver uma aplicação embarcada, quando de posse de uma plataforma pronta.

Além da fácil mudança de plataforma alvo e reúso de componentes ao se utilizar uma mesma plataforma alvo, a arquitetura do ESCoRT proporciona uma maior

manutenibilidade do software desenvolvido, devido ao aumento de modularidade proposto.

Capítulo 3

Abordagem da análise

Para Gustavo [1], o uso de orientação a objetos e o aumento da modularização do software são os dois fatores que podem causar algum prejuizo a performance da aplicação.

Em relação ao uso de C++, os principais motivos de preocupação com a performance, segundo [4] são: Memória de Heap, Vinculação Dinâmica e Chamada de Métodos. O ESCoRT aloca estaticamente todos os componentes de uma configuração, declarando todos os componentes e suas definições nos arquivos config.cpp e config.h, assim evitando ao máximo o uso de memória de heap. Como em orientação a objetos são criados inúmeros métodos que realizam pequenas operações o número de chamadas de métodos deve aumentar. Deve-se notar, porém, que este comportamento é necessário para o objetivo de aumentar a reusabilidade do software. Além disso, há o uso de vinculação dinâmica, que ocorre normalmente devido ao uso de métodos virtuais. Como no ESCoRT são utilizadas classes abstratas que servem como interface, métodos virtuais são utilizados. Um objeto de uma classe que contenha algum método virtual deve conter uma referência para a vTable desta classe, uma tabela que contem referências para os métodos virtuais a serem utilizados. Deve-se notar, porém, que estes possíveis overheads são necessários para atingir o nível de modularização desejado pelo ESCoRT, assim também podem ser considerados overheads gerados pela maior modularização do software.

O aumento de modularização mencionado refere-se ao fato de se utilizar componentes com interfaces bem definidas, que proporcionam a fácil reusabilidade dos componentes. Alguns autores, como [13] citam que aumentar a reusabilidade de um software pode causar um prejuizo na performance, porém não há nenhum dado mensurável. Para tentar mensurar esse prejuizo foram criadas algumas aplicações simples que irão mostrar porque o reúso pode impactar na performance do software. Foram desenvolvidas duas versões de cada aplicação com mesmas funcionalidades, sendo uma em C, sem se preocupar com reúso ou manutenibilidade do software, e outra utilizando os conceitos do ESCoRT, em C++.

Para obter nossas métricas será utilizado um software para compilar e depurar os programas, assim como foi feito em[3], porém utilizando um outro software: o Keil uVision.

Nos programas desenvolvidos serão selecionadas algumas funcionalidades as quais serão analisadas quanto ao tempo de execução. Por exemplo, quanto tempo leva-se para inicializar uma serial ou quanto tempo leva-se para setar um pino da GPIO (General Purpose Input/Output). Para mensurar este tempo de execução, será feito como explicado em [14]. Como nos casos de análise utilizados não há uso de SO, os tempos de execução sempre são iguais, indepedente de quantas vezes formos executar as funcionalidades analisadas; e, para confirmar isto, os testes realizados foram executados mais de uma vez, sempre dando o mesmo resultado.

Em relação ao uso de memória, serão obtidos dados que o próprio software utilizado nos mostra após uma compilação bem sucedida. È importante mencionar que não será feita uma análise da pilha utilizada, sendo utilizada uma pilha de tamanho igual para cada caso de teste.

Capítulo 4

Análise de eficiência do ESCoRT

4.1 Metodologia

Ao compilar um código o Keil uVision fornece os requisitos de memória que sua aplicação tem. Ele separa a memória utilizada em 4 tipos: “Code”, “RO-Data”, “RW-Data” e “ZI-Data”. Estes significam:

- Code é o tamanho do código gerado;
- RO-Data significa Read-Only Data (dados contantes);
- RW-Data significa Read/Write Data (variáveis globais);
- ZI-Data significa Zero-Initialized Data (Dados que foram iniciados com valor 0).

A partir destes dados temos que o tamanho total da ROM (Read Only Memory) utilizada é o tamanho de “Code” mais o tamanho de RO-Data. Enquanto o tamanho da RAM (Random Acess Memory) utilizada é igual a RW-Data mais ZI-Data.

O tempo de execução de algumas funcionalidades será medido utilizando o debug do Keil, definindo um breakpoint antes e ao fim da execução da funcionalidade, assim, subtraindo o tempo final pelo inicial, teremos o tempo de execução. Apenas no caso do Bootloader não foi utilizada a simulação no debug, pois o teste envolvia uma atualização de firmware utilizando um pen drive e arquivos específicos, sendo o programa depurado utilizando-se o ULINK[15].

O exemplo Blinky (em C) foi obtido de [7]. Já o exemplo Bootloader Lightdot faz parte de um produto da FRT Tecnologia Eletrônica e sua implementação em C teve que ser desenvolvida para ser feita a comparação. Foram implementadas todas as outras versões de exemplos.

Foi utilizado o mesmo arquivo de startup para cada versão do mesmo caso de análise, assim utilizando mesmo tamanho de pilha e heap.

4.2 Casos de Análise

4.2.1 Vinculação Dinâmica

Para mensurar o overhead causado pela vinculação dinâmica foram implementados 2 exemplos para serem comparados entre si. No primeiro exemplo, temos uma forma de implementação bastante utilizada no ESCoRT, enquando a segunda seria uma forma alternativa de se implementar, porém sem as vantagens de modularidade e reúso que a primeira proporciona. Este exemplo foi testado no Keil uVision utilizando como dispositivo alvo a STM32F103RB[9], utilizando um clock de 72Mhz.

4.2.1.1 Exemplo 1

```
1  class InterfaceTeste
2  {
3    public:
4    virtual void metodoVirtual()=0;
5
6  };
7
8  class Teste:InterfaceTeste
9  {
10   public:
11   virtual void metodoVirtual()
12   {
13     for (int i =0; i<50 ; i++);
14   }
15 };
16
17 class Execucao
18 {
19   public:
20   Execucao(InterfaceTeste* x):
21   x(x)
22   {}
23   void metodoTeste()
24   {
25     x->metodoVirtual();
26     x->metodoVirtual();
27   }
28   private :
29     InterfaceTeste* x;
30 };
```

```
32 Teste teste;
33 InterfaceTeste *x = (InterfaceTeste*) &teste;
34 Execucao execucao(x);
35 int main()
36 {
37   execucao.metodoTeste();
38   execucao.metodoTeste();
39   return 0;
40 }
41
```

Figura 4 – Código de exemplo 1

Esta configuração é bastante comum ao se utilizar o ESCoRT. A classe Execução recebe um objeto do tipo InterfaceTeste, uma classe abstrata, em seu construtor e o utiliza em métodos da classe. Um exemplo desta configuração na prática

é a seguinte: Um componente UARTDriver recebe, em seu construtor, um componente da camada HAL chamado IUART, que é uma interface que tem métodos virtuais puros como *setBaudrate*, *send* e *receive*. No arquivo de configuração de sistema do ESCoRT, será definido qual objeto será passado para o componente UARTDriver no lugar do IUART. Este objeto pode pertencer a qualquer componente que herde do IUART, ou seja, um componente UART a ser utilizada no projeto. Assim, o componente UARTDriver tem a capacidade de chamar o método *receive* de qualquer componente UART, seja este implementado para qualquer hardware.

4.2.1.2 Exemplo 2

```
 1 class Teste
 2 {
 3   public:
 4    void metodoVirtual()
 5   {
 6     for (int i =0; i<50 ; i++);
 7   }
 8 };
 9
10 class Execucao
11 {
12   public:
13   Execucao(Teste* x):
14   x(x)
15   {}
16   void metodoTeste()
17   {
18     x->metodoVirtual();
19     x->metodoVirtual();
20   }
21   private :
22     Teste* x;
23 };
```

```
24 Teste teste;
25 Teste *x =  &teste;
26 Execucao execucao(x);
27 int main()
28 {
29   execucao.metodoTeste();
30   execucao.metodoTeste();
31   return 0;
32 }
```

Figura 5 – Código de exemplo 2

Este exemplo retrata uma outra forma de implementar a mesma funcionalidade do Exemplo 1, porém não utiliza interface. Não utilizando uma interface faz com que a classe Execução só possa receber objetos do tipo Teste em seu construtor. Na prática, caso se utilizasse esse tipo de implementação, haveria uma perda da capacidade de reúso no ESCoRT, pois, por exemplo, um componente UARTDriver seria dependente do componente UART da camada HAL, ou seja, o driver seria dependente da plataforma alvo.

4.2.2 Aumento de modularização

Para ter uma noção do custo adicional de tornar o código mais modularizado e mais fácil de ser reutilizado, iremos comparar implementações de 3 projetos, sendo 2 exemplos de teste bastante conhecidos e o outro um projeto um pouco maior, que foi implementado na FRT Tecnologia Eletronica.

4.2.2.1 Echo

A aplicação Echo é um exemplo bastante conhecido. Esta aplicação recebe caracteres pela serial e os envia de volta, também pela serial. Serão comparadas duas implementações, uma em C, não se preocupando com reúso e modularidade do código, e outra em C++, utilizando os conceitos do ESCoRT. Para ambas implementações foram usados os mesmos tamanhos de buffer para a serial. Este exemplo foi implementado para o NXP LPC2368[8], utilizando um clock de 60Mhz. Será chamada de ECHO-CPP a aplicação Echo em C++ e ECHO-C a aplicação desenvolvida em C.

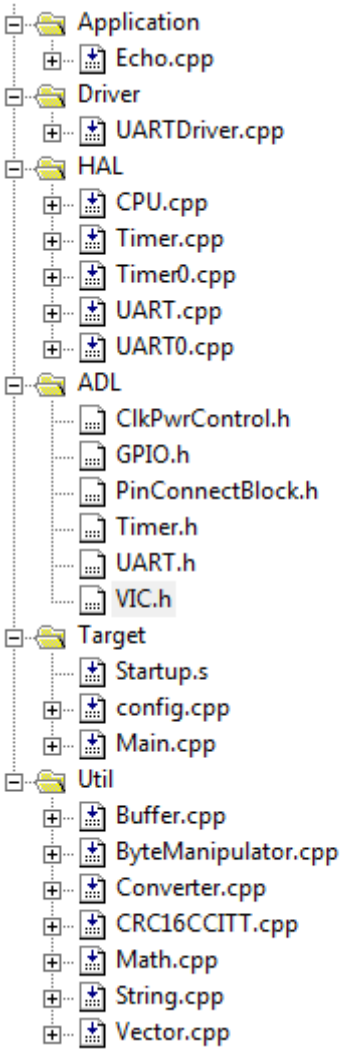

Figura 6 – Arquivos do projeto ECHO-CPP

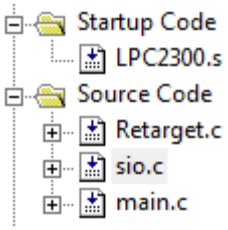

Figura 7 – Arquivos do projeto ECHO-C

Percebe-se que a aplicação ECHO-CPP contém muito mais arquivos. Essa quantidade extra de funcionalidades implementadas facilita o reúso dos componentes, pois, por exemplo, o componente UARTDriver implementa um timeout, que é necessário para diversas aplicações. Porém, para a aplicação Echo, o timeout não seria necessário, assim apenas trazendo um overhead para a aplicação.

4.2.2.2 Blinky

A aplicação Blinky também é um exemplo bastante conhecido. Sua funcionalidade é testar pinos da GPIO, setando um pino no nível lógico 1 e, depois de um tempo, no nível lógico 0, e repetindo este processo para os próximos pinos. Nesta análise teremos a comparação entre a versão Blinky em C, que será chamada Blinky-C, e a versão desenvolvida utilizando os conceitos do ESCoRT, que será chamada Blinky-CPP. Este exemplo foi implementado para o NXP LPC2129[15], utilizando um clock de 60Mhz

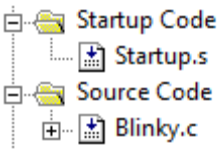

Figura 8 – Arquivos Blinky-C

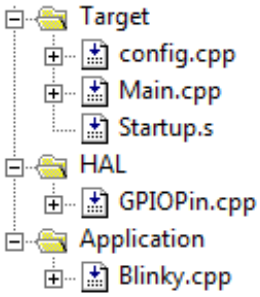

Figura 9 – Arquivos Blinky-CPP

Como a aplicação Blinky deve lidar diretamente com os pinos da GPIO, o componente da camada Application *Blinky* deve se comunicar diretamente com o componente da camada HAL *GPIOPin*. O componente Blinky manipula o objeto GPIOPin gerado através de uma interface IGPIOPin, componente que possui apenas uma arquivo do tipo header (.h). Enquanto isso, na versão em C, os pinos são manipulados diretamente. Uma grande diferença entre as duas aplicações está no uso dos pinos da GPIO, pois, enquanto Blinky-C utiliza acesso direto aos registradores, podendo modificar o estado de vários pinos de uma vez, o Blinky-CPP trata cada pino como um componente GPIO, assim modificando o estado de um pino de cada vez.

4.2.2.3 Bootloader Lightdot

O “Bootloader-Lightdot” é um projeto comercial que serve como Bootloader para o produto Lightdot da FRT Tecnologia Eletrônica. Entre os os principais componentes implementados neste projeto estão: Sistema de arquivos, USB, driver de um display LCD, além da aplicação, que tem como funcionalidade a análise de arquivos de um pendrive para uma possível atualização de firmware do produto LightDot e, caso necessário, a atualização de firmware em si. Foi desenvolvida uma versão em C (Bootloader-C) a partir da versão já desenvolvida em C++ (Bootloader-CPP, que seguia os conceitos do ESCoRT. A principal diferença entre as duas é o uso de interfaces e falta de preocupação com o reúso do código, apesar de que também há uma boa modularidade do software na versão Bootloader-C. Este projeto foi implementado para a STM32F107VC[9], utilizando clock de 72Mhz.

4.3 Considerações

No exemplo de vinculação dinâmica espera-se verificar o overhead causado pela mesma, sendo este o único prejuízo que o uso do ESCoRT não pode evitar, pois é justamente o uso de métodos virtuais que possibilitam um maior reuso de componentes e a troca de plataforma alvo sem grandes problemas.

Nos outros exemplos espera-se verificar o prejuízo causado pelo uso do ESCoRT em geral, considerando o aumento da modularidade, uso dos componentes e o uso de métodos virtuais.

Capítulo 5

Resultados

5.1 Vinculação Dinâmica

Assim como mencionado na sessão 4.2.1, o Exemplo 1 representará o código utilizando interface e método virtual, enquanto o Exemplo 2 será o código que não utiliza interface e nem método virtual. O tamanho da pilha utilizada neste exemplo foi de 1024 bytes e o heap de 512 bytes. A tabela 2 mostra os requisitos de memória de ambos os exemplos, enquanto a tabela 3 mostra o tempo de execução do "for" utilizado no método de ambos os exemplos e o tempo de execução do método *execucao.metodoTeste*.

Aplicação \ Dados (Bytes)	Code	RO	RW	ZI	Total ROM	Total RAM
Exemplo 1	1092	396	32	1632	1488	1664
Exemplo 2	928	372	32	1632	1300	1664
Penalidade	17,67 %	6,45%	0%	0%	14,46%	0%

Tabela 2 – Memória utilizada nos Exemplos 1 e 2

Aplicação \ Tempo (ms)	for	*execucao.metodoTeste*
Exemplo 1	0,0356	0,0767
Exemplo 2	0,0356	0,0756
Penalidade ESCoRT	0%	1,45%

Tabela 3 – Tempo de execução de funcionalidades dos Exemplos 1 e 2

Como era esperado, o Exemplo 1, que contém interface e método virtual, demanda mais memória para o código e mais memória do tipo Read-Only. A diferença de memória read-only está nas vTables criadas. Cada classe contendo pelo menos um método virtual deve conter uma vTable para selecionar o método a ser executado; e no Exemplo 1 existem 2 classes deste tipo. A diferença no tamanho do código pode ser explicada pelo uso da classe de interface, além de instruções a mais na chamada de

métodos virtuais. Essas instruções também acabam a aumentar o tempo de execução do método *execucao.metodoTeste* que contém duas chamadas a métodos virtuais. A penalidade de 1,45% pode ser interpretada como a penalidade que um método virtual com um for de 50 iterações sofre para ser executado completamente. Também foi medido o tempo de execução do laço, apenas para confirmar que utilizar C++ não implica em uma perda de desempenho por si só.

5.2 Echo

Chamaremos de ECHO-C a versão C (estilo procedural) e ECHO-CPP a versão em C++, desenvolvida utilizando o ESCoRT. Neste caso de análise foi utilizada uma pilha de 1024 bytes para o usuário, 256 bytes para o Fast Interrupt Mode e 8 bytes para o SuperVisor Mode; enquanto não foi reservado nada para o heap. Ambas as aplicações utilizaram buffers de 32 bytes, tanto para entrada de dados pela serial, quanto para a saída. Neste experimento foi medido também o tempo gasto pelas inicializações realizadas na aplicação, ou seja, o tempo para que a aplicação comece a rodar. Em ECHO-C foi necessária a inicialização da serial, enquanto, em ECHO-CPP, a inicialização de todos os componentes.

Aplicação \ Dados (Bytes)	Code	RO	RW	ZI	Total ROM	Total RAM
ECHO-C	1908	32	12	1372	1946	1384
ECHO-CPP	8016	420	28	1452	8436	1480
Penalidade ESCoRT	320,12%	1212,5%	133,33%	5,83%	333,50%	6,93%

Tabela 4 - Memória utilizada no projeto ECHO.

Aplicação \ Tempo (ms)	Inicialização
ECHO-C	0.00947
ECHO-CPP	0,13392
Penalidade ESCoRT	1314,15%

Tabela 5 - Tempo de execução de funcionalidades no projeto ECHO.

Podemos perceber nessa aplicação uma grande diferença no tamanho da memória ROM necessária para cada aplicação. Por ser muito mais modular, o ECHO-CPP acaba gerando mais código que não tem utilidade para esta aplicação específica, como por exemplo timeout de leitura da serial. A grande diferença no tamanho do código se dá principalmente pelo fato de ECHO-C ser bastante específico para esta funcionalidade, enquanto os módulos utilizados em ECHO-CPP são mais gerais, podendo ser estendido para outras funcionalidades. Além disso, por utilizar um timeout no componente driver da serial, o ECHO-CPP acaba por ter que implementar um timer, colaborando ainda mais para a grande diferença entre as duas aplicações. Uma das razões para o aumento da RO Memory é o uso de métodos virtuais, necessários para a implementação de interfaces, pois se faz necessário o uso de vTables. O aumento da RAM pode ser explicado pelo uso de funcionalidades não necessárias para esta aplicação específica, porém, devido ao código ser reusável por outras aplicações, a existência dessas funcionalidades se faz necessária.

Em relação ao grande aumento no tempo de execução na inicialização do sistema, já era esperado, pois a aplicação ECHO-CPP necessita inicializar todos os componentes do sistema, enquanto o ECHO-C só necessita inicializar a serial. Fica bastante claro que as aplicações desenvolvidas com o ESCoRT irão necessitar de muito mais tempo para serem inicializadas.

5.3 Blinky

Chamaremos de Blinky-C a versão C (estilo procedural) e Blinky-CPP a versão em C++, desenvolvida utilizando o ESCoRT. Neste caso de análise foi utilizada uma pilha de 1024 bytes para o usuário, 128 bytes para o Fast Interrupt Mode e 8 bytes para o SuperVisor Mode; enquanto que não foi reservado nada para o heap. Neste experimento foi medido o tempo gasto para: Definir todos os pinos desejados como pinos de saída e Definir o nível lógico de um pino.

Aplicação \ Dados (Bytes)	Code	RO	RW	ZI	Total ROM	Total RAM
Blinky-C	528	16	0	1160	544	1160
Blinky-CPP	1888	104	100	1260	1992	1360
Penalidade ESCoRT	257,57%	550%	-	8,62%	266,17%	17,24%

Tabela 6 - Memória utilizado no projeto Blinky.

Aplicação \ Tempo (ms)	Definir como Saída	Definir nível lógico
Blinky-C	0,00010	0,00008
Blinky-CPP	0,01177	0,00072
Penalidade ESCoRT	11670%	800%

Tabela 7 - Tempo de execução de funcionalidades no projeto Blinky.

Novamente pode-se perceber uma diferença no tamanho do código gerado, desta vez não tão grande quanto no exemplo anterior, pois, neste caso, não houveram funcionalidades extras que não seriam úteis a esta aplicação específica. O aumento da RO Data se deve ao uso de métodos virtuais, ou seja, uso de vTables, enquanto o aumento na ZI Data deve-se ao instaciamento dos objetos. É perceptível que o uso do ESCoRT acarreta num aumento do tamanho do código gerado, além do inevitável aumento da RO Data, devido ao uso de interfaces.

Tratar cada pino individualmente também custou mais tempo na operação “Definir como Saída” para a aplicação Blinky-CPP, visto que a configuração de Input/Output de todos os pinos utilizados na aplicação está em apenas um endereço de memória, ou seja, foi necessário percorrer todos os GPIOPins e setá-los como pinos de saída um por um, enquanto tudo isto poderia ser feito em apenas uma instrução. As abstrações para definir o nível lógico de um pino também causaram uma piora na performance na operação “Definir nível lógico”, porém bem menor que a operação anterior.

5.4 Bootloader-Lightdot

Chamaremos de Bootloader-C a versão C (estilo procedural) e Bootloader-CPP a versão em C++, desenvolvida utilizando o ESCoRT. Foi utilizada uma pilha de 1024 bytes e um heap de 512 bytes. Neste experimento foi medido o tempo gasto para as inicializações necessárias, mostrar a mensagem desejada no display LCD, e para fazer uma atualização de firmware, sendo dados como entrada para o programa os mesmos dados em todos os testes deste tipo.

Aplicação \ Dados (Bytes)	Code	RO	RW	ZI	Total ROM	Total RAM
Bootloader-C	24004	1296	180	6340	25300	6520
Bootloader-CPP	26028	2564	244	6452	28592	6696
Penalidade ESCoRT	8,43%	97,84%	35,55%	1,76%	13,01%	2,70%

Tabela 8 - Memória utilizado no projeto Bootloader-Lightdot.

Aplicação \ Tempo (ms)	Inicializações	LCD	Atualização de Firmware
Bootloader-C	1.942,61980	5,31349	4.769,33965
Bootloader-CPP	1.889,06734	6,20721	4.749,72841
Penalidade ESCoRT	-2,75%	16,82%	-0,41%

Tabela 9 - Tempo de execução de funcionalidades no teste Bootloader-Lightdot.

Como o código do Bootloader-C foi baseado no código do Bootloader-CPP, a modularidade também é uma característica dele. Entre as principais diferenças estão: o uso de interfaces, de herança, ou seja, algumas características que mostram a preocupação com a reusabilidade do código e que são características de linguagens orientadas a objetos. A maior modularidade se faz necessária em aplicações maiores, assim aumentando a manutenibilidade do sofware.

Em aplicações mais complexas, percebe-se que a diferença do tamanho do código não é mais tão grande quanto nas aplicações anteriores, que eram bem menos complexas. Neste caso isso deve-se ao número de métodos ou funções serem similares em ambos os projetos. O aumento da RO Data neste caso também deve-se ao uso de interfaces e métodos virtuais. O aumento da RAM necessária também deve-se ao

aumento da reusabilidade do código, ou seja, o uso de componentes mais gerais, não específicos para a aplicação.

O tempo das operações do LCD foi menor na aplicação em C, pois trata os pinos da GPIO diretamente, sem utilizar uma abstração, ou seja, sem um componente GPIOPin. No geral, o desempenho da aplicação Bootloader-CPP foi próximo, e até um pouco melhor, dependendo também da eficiência do código produzido, o que demonstra que mesmo em C++ a performance do software pode continuar similiar a performance em C, resultado também obtido por [4].

5.5 Considerações

Foi verificado em todos os casos de teste um aumento da memória necessária, principalmente a memória ROM, que contém o código e dados do tipo Read-Only. Este aumento ocorre devido principalmente ao aumento da modularidade do código e do uso de métodos virtuais. Já o overhead de memória RAM foi bem menor, sendo este gerado por alguns dados a mais para controle individual de alguns componentes, por exemplo, um componente GPIOPin contém a informação de qual pino ele representa.

Em relação ao tempo de execução, na comparação do projeto Bootloader LightDot, em que o nível de modularidade entre os códigos comparados era semelhante, não houve diferenças tão notáveis. Porém, nas outras aplicações, em que o código em C era mais otimizado, enquanto o código criado com a ferramenta ESCoRT tinha mais funcionalidades não utilizadas, a diferença foi grande, principalmente nas rotinas de inicialização.

Capítulo 6

Conclusão

A utilização do ESCoRT traz inegáveis ganhos na portabilidade e reusabilidade do software embarcado, como foi demonstrado por Gustavo[1]. Apesar disso, foi constatado que a utilização dessa metodologia pode causar um deterioramento da performance da aplicação, seja numa necessidade de mais memória, como em termos de tempo de execução.

Como visto no exemplo do DARK e DARK++ [4], o uso de C++ por si só não implica em uma penalidade considerável na performance da aplicação. A perda de performance está ligada ao aumento da modularização do sistema, que, além trazer mais camadas de abstração para a aplicação, pode incluir funcionalidades não necessárias a aplicação a ser desenvolvida, pois os componentes utilizados devem servir para outras aplicações também. Além disso, por criar uma interface única para os componentes da camada HAL, pode acabar perdendo oportunidades de melhorar o desempenho num hardware específico.

O uso de interfaces, métodos virtuais, mais chamadas de métodos é característica do estilo de programação orientado a objetos. Esse estilo proporciona uma alta modularidade, reusabilidade, portabilidade e manutenibilidade ao código, porém trazendo consigo uma necessidade maior de memória, seja de código, Read-Only, ou RAM. Este trabalho mostrou o overhead proporcionado por uma vTable de uma classe com apenas um método virtual num ambiente embarcado, além de mostrar o impacto geral causado pelo ESCoRT em aplicações.

Em sistemas mais complexos existe uma necessidade de escrita de um código mais legível e modularizado, assim facilitando a manutenção do sistema. Além disso, utilizar um código portável para outras plataformas alvo pode ser de grande interesse para diversos projetos. Com hardwares cada vez mais poderosos, é possível que em muitas situações valha a pena pagar o trade-off de se utilizar os conceitos do ESCoRT, uma vez que os benefícios da utilização da metodologia também são muito significantes.

Em sistemas que necessitam um acesso rápido ao hardware é possível que o overhead causado pela abstração utilizada, a fim de reutilizar o código, possa impossibilitar o uso do ESCoRT. Supondo que a equipe que irá desenvolver um projeto

deste tipo já tenha um repositório de componentes e conhecimento do ESCoRT, é também possível utilizar uma abordagem mista na qual as partes mais baixo nível, como componentes da camada HAL, fossem implementadas visando uma melhor performance. Essa abordagem mista poderia melhorar a performance a o uso de memória nas partes do projeto desenvolvidas fora do ESCoRT, mas continuaria com os overheads das camadas mais altas, que, dependendo da aplicação, poderiam ser aceitáveis.

Foi demonstrado que a utilização do ESCoRT pode trazer um grande prejuízo a performance de uma aplicação, bem como a necessidade maior de memória. Cabe ao engenheiro de software embarcado decidir pela sua utilização, considerando as limitações do seu projeto e as vantagens do ESCoRT. Espera-se que em projetos que não haja grande limitação dos recursos a utilização do ESCoRT seja escolhida, devido as suas vantagens de reusabilidade, portabilidade, manutenibilidade e modularidade.

6.1 Trabalhos Futuros

Neste trabalho não foi abordado sobre um possível aumento no consumo de energia ao se utilizar o ESCoRT. Como esta é uma das grandes preocupações na área de sistemas embarcados seria interessante fazer um estudo sobre este possível impacto.

Além disso, ainda não foi feito um estudo aprofundado sobre os ganhos de produtividade ao se utilizar o ESCoRT. Mesmo tendo um poder óbvio para melhorar a produtividade, seria interessante uma análise mais a fundo.

Uma outra análisa interessante a se fazer seria mensurar o impacto do ESCoRT na utilização da pilha, pois um uso maior da pilha poderia significar uma maior necessidade de memória reservada para ela.

Referências Bibliográficas

[1] MELO, G. ESCORT: UM FRAMEWORK DE COMPONENTES PARA A CONSTRUÇÃO DE SOFTWARE EMBARCADO. 2011. 101 f. Tese (Mestrado em Ciência da Computação) - Centro de Informática, Universidade Federal de Pernambuco, Recife, 2011.

[2] Xu, Dong, et al., "Towards a Software Framework for Building Highly Flexible Component-Based Embedded Operating Systems." Berlin : Springer-Verlag, 2007.

[3] Chatzigeorgiou, Alexander, et al., " Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors." Berlin : Springer-Verlag, 2002.

[4] Bhakthavatsalam, S. Measuring the Perceived Overhead Imposed by Object-Oriented Programming in a Real-time Embedded System. 2003. 151 f. Tese (Master of Science in Computer Science and Applications) - Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2003.

[5] Melo, Gustavo A. F. B. e Cavalcante, Sérgio V., "Modelo de Arquitetura para Construção de Plataformas de Software Embarcado." 2010. 12th Brazilian Workshop on Real-Time and Embedded Systems.

[6] Douglass, Bruce Powel., "Layered Pattern." [A. do livro] Douglass Powel Bruce. Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. s.l. : Addison Wesley, 2003, 4.1, pp. 95-99.

[7] Keil, Download Page. Disponível em <http://www.keil.com/download/>. Acesso em 15 fev. 2013.

[8] NXP., "LPC2364/66/68/78 User manual.". Disponível em <www.keil.com/dd/docs/datashts/philips/lpc23xx_um.pdf>. Acesso em 15 fev. 2013.

[9] STMicroelectronics., "RM0008 Reference Manual - STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs." Disponível em: <www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf> . Acesso em 15 fev. 2013.

[10] Lima, Hilton. “Desenvolvimento de uma rede neural artificial para detecção de descargas parciais em cadeias de isoladores de linhas de alta tensão”. Disponivel em: <tcc.ecomp.poli.br/20092/TCC_final_Hilton.pdf>. Acesso em 25 jul. 2013.

[11] Heath, Steve. “Embedded System Design”. 2ª Ed. Newnes, 2002. 430p.

[12] Sangiovanni-Vincetelli, Alberto. Martin, Grant. “Platform-Based Design and

Software Design Methodology for Embedded Systems". Disponível em : < http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=970421&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F54%2F20927%2F00970421>. Acesso em 25 jul. 2013.
[13] Deitel, Paul. Deitel, Harvey. "C++ How to Program". 8ªEd. Prentice Hall, 2011. 1104p.
[14] Keil, "µVISION DEBUGGER: MEASURING EXECUTION TIME". Disponivel em: <http://www.keil.com/support/docs/971.htm>. Acesso em: 16 fev. 2013.
[15] Keil, "ULINK Family of Debug and Trace Adapters". Disponível em: <http://www.keil.com/ulink/> . Acesso em: 16 fev. 2013.
[16] NXP, "LPC21xx and LPC22xx User manual". Disponível em : <http://www.nxp.com/documents/user_manual/UM10114.pdf>. Acesso em: 15 fev. 2013.
[17] Dijkstra, Edsger. "The Humble Programmer". Disponível em : <http://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF> . Acesso em 25 jul. 2013
[18] "The Standish Group Report". Disponível em: <https://cs.nmt.edu/~cs328/reading/Standish.pdf> . Acesso em: 26 jul. 2013.
[19] Krueger, Charles. "Software Reuse". Disponível em: <http://dl.acm.org/citation.cfm?id=130856> . Acesso em 25 jul. 2013.
[20] "An A+ for C++". Disponível em : <http://blog.vdcresearch.com/embedded_sw/2012/09/an-a-for-c.html> . Acesso em 26 jul. 2013.
[21] Oliveira, Karina. et al. "Abordagens de Reúso de Software no Desenvolvimento de Aplicac¸oes Orientadas a Objetos". Disponivel em: <http://www.munif.com.br/munif/arquivos/reúso-1.pdf?id=54> . Acesso em : 27 jul. 2013.

Apêndice A

Blinky_ESCoRT

Será mostrado como se organiza o projeto Blinky desenvolvido utilizando o ESCoRT, além de códigos dos componente implementados.

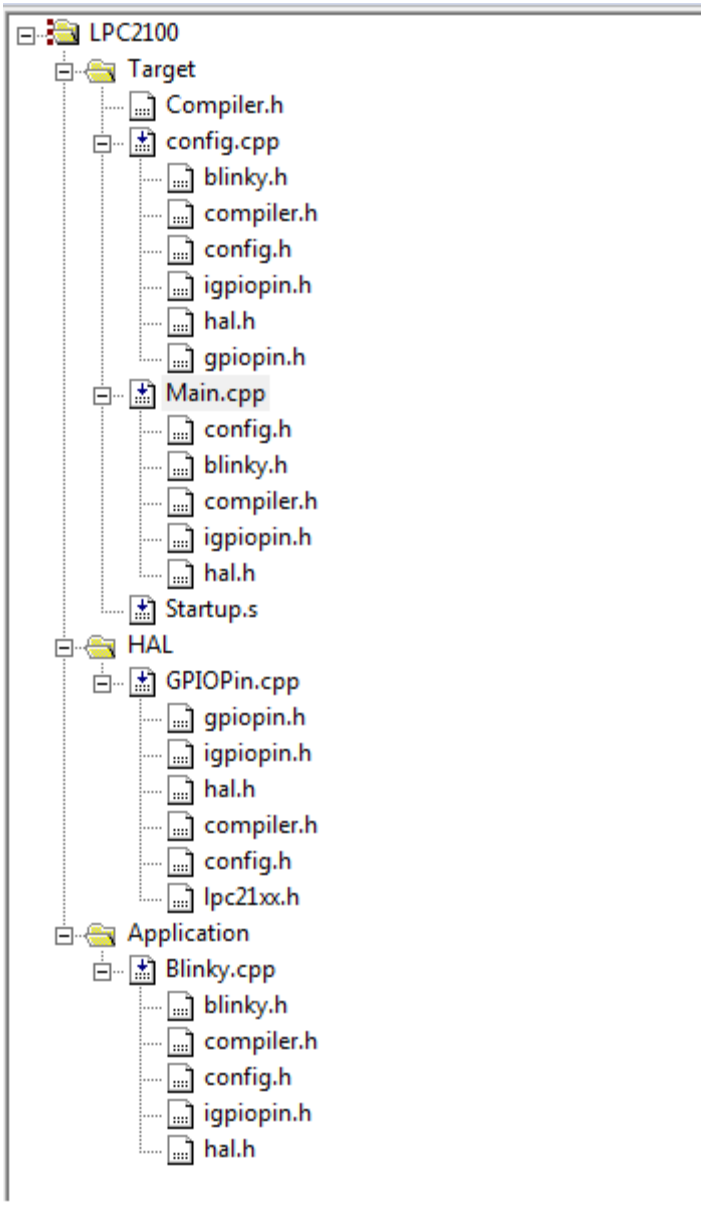

Figura 10 – Projeto Blinky

A.1 Arquivo Main.cpp

```
#include "config.h"
#include <Blinky.h>
//*****************************************************************
***************
// External variables
//*****************************************************************
***************
extern tg::application::Blinky app_blinky;
//*****************************************************************
***************
// Prototypes
//*****************************************************************
***************
void initHALComponents();
void initKernelComponents();
void initDriverComponents();
void initServiceComponents();
void initApplicationComponents();

//*****************************************************************
***************
// Functions
//*****************************************************************
***************

void initSystem() {
	initHALComponents();
	initKernelComponents();
	initDriverComponents();
	initServiceComponents();
	initApplicationComponents();

}
```

```
//************************************************
// Main
//************************************************
int main() {
	initSystem();
	app_blinky.execute();

	return 0;
}
```

A.2 Arquivo config.cpp

```
#include <Blinky.h>
#include <GPIOPin.h>


lpc::hal::GPIOPin hal_led0(0);
lpc::hal::GPIOPin hal_led1(1);
lpc::hal::GPIOPin hal_led2(2);
lpc::hal::GPIOPin hal_led3(3);
lpc::hal::GPIOPin hal_led4(4);
lpc::hal::GPIOPin hal_led5(5);
lpc::hal::GPIOPin hal_led6(6);
lpc::hal::GPIOPin hal_led7(7);

escort::hal::IGPIOPin* pinos[] =
{&hal_led0, &hal_led1, &hal_led2, &hal_led3, &hal_led4, &hal_led5,
&hal_led6, &hal_led7};


tg::application::Blinky app_blinky(pinos);

void initHALComponents()
```

```
{

        hal_led0.init();
        hal_led1.init();
        hal_led2.init();
        hal_led3.init();
        hal_led4.init();
        hal_led5.init();
        hal_led6.init();
        hal_led7.init();

}

void initKernelComponents()
{

}

void initDriverComponents()
{

}

void initServiceComponents()
{

}

void initApplicationComponents()
{
        app_blinky.init();

}
```

A.3 Arquivo GPIOPin.cpp

```
#include "GPIOPin.h"
#include "../../Target/config.h"
#include "../../Target/Compiler.h"

#ifdef __lpc_hal_GPIOPin__


extern "C" {
	#include "../lpc.adl/LPC21xx.H"
}



namespace lpc {
namespace hal {

GPIOPin::GPIOPin(U8 pino)
{
	this->pinNumber = pino;
}

void GPIOPin::init()
{

}

void GPIOPin::setDirection(escort::hal::IGPIOPin::GPIOPinDirection direction)
{
	U32 mask=1<<16;
	for(int i = 0; i < this->pinNumber; i++, mask = mask<<1);
	if(direction == OUTPUT)
	{
```

```
                IODIR1 = IODIR1 | mask;
        }
        else
        {
                IODIR1 = IODIR1 & (~mask);
        }

}

escort::hal::IGPIOPin::GPIOPinDirection GPIOPin::getDirection()
{
        U32 mask=1<<16;
        for(int i = 0; i < this->pinNumber; i++, mask = mask<<1);
        if(IODIR1 & mask)
        {
                return OUTPUT;
        }
        return INPUT;
}

void GPIOPin::setStatus(bool status)
{
        U32 mask=1<<16;
        for(int i = 0; i < this->pinNumber; i++, mask = mask<<1);

        if(status)IOSET1 = mask;
        else IOCLR1 = mask;
}

bool GPIOPin::getStatus()
{
        U32 mask=1<<16;
        for(int i = 0; i < this->pinNumber; i++, mask = mask<<1);
```

```
        if(IOPIN1&mask) return true;
        return false;
}

}

}

#endif
```

A.4 Arquivo Blinky.cpp

```
#include <Blinky.h>


#ifdef __tg_application_Blinky__

namespace tg {
namespace application {

Blinky::Blinky( IGPIOPin **hal_leds)
{
        leds = hal_leds;
}


void Blinky::init()
{

}


void Blinky::execute()
```

```
{

	for (U8 i = 0; i <= 7; ++i) {
		(leds[i])->setDirection(escort::hal::IGPIOPin::OUTPUT);
	}
	while(true)
	{
		for (U8 i = 0; i < 7; i++) {
			(leds[i])->setStatus(true);
			this->wait();
			(leds[i])->setStatus(false);
		}
		for (U8 i = 7; i > 0; i--) {
			(leds[i])->setStatus(true);
			this->wait();
			(leds[i])->setStatus(false);
		}

	}
}

void Blinky::wait()
{
	  int  d;
	  for (d = 0; d < 1000000; d++);
}



}
}

#endif
```