Universidade de São Paulo
Escola de Engenharia de São Carlos
Departamento de Engenharia Elétrica
Engenharia Elétrica - ênfase em Sistemas de Energia e Automação

Leonardo Brás Soares Passos

Plataforma para aplicações de Tempo-Real usando Linux embarcado em microcontroladores ARM

São Carlos
2012

Leonardo Brás Soares Passos

Plataforma para aplicações de Tempo-Real usando Linux embarcado em microcontroladores ARM

Trabalho de conclusão de curso apresentado ao Programa de Engenharia Elétrica da Escola de Engenharia de São Carlos como parte dos requisitos para a obtenção do título de Engenheiro Eletricista.

Área de concentração: Sistemas Embarcados

ORIENTADOR: Prof. Dr. Evandro Luís L. Rodrigues

São Carlos
2012

Passos, Leonardo Brás Soares
P289p Plataforma para aplicações de Tempo-Real usando Linux embarcado em microcontroladores ARM / Leonardo Brás Soares Passos; orientador Evandro Luís Linhari Rodrigues. São Carlos, 2012.

Monografia (Graduação em Engenharia Elétrica com ênfase em Sistemas de Energia e Automação) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2012.

1. Automação. 2. Microcontroladores. 3. Linux. 4. ARM. 5. Real Time. 6. Embarcado. 7. Software Livre. I. Título.

FOLHA DE APROVAÇÃO

Nome: Leonardo Bras Soares Passos

Título: "Plataforma para aplicações de Tempo-Real usando Linux embarcado em microcontroladores ARM"

Trabalho de Conclusão de Curso defendido e aprovado em 26/11/2012,

com NOTA 9,3 (nove, três), *pela Comissão Julgadora:*

Prof. Associado Evandro Luís Linhari Rodrigues (Orientador)
SEL/EESC/USP

Prof. Dr. José Roberto Boffino de Almeida Monteiro
SEL/EESC/USP

Eng. Renato Machado Monaro
SEL/EESC/USP

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Homero Schiabel

Imaginação é mais importante que o conhecimento. Conhecimento é limitado, enquanto a imaginação envolve todo o mundo, estimulando o progresso, dando vida à evolução. Ela é, de maneira rigorosa, um fator real na pesquisa científica.

Albert Eninstein, 1931

À minha família,
que sempre me apoiou nos momentos difíceis.

Agradecimentos

À minha mãe, Maria Aparecida, que me apoiou durante toda minha vida, e me ensinou a perseguir meus sonhos através de esforço e dedicação.

Ao meu pai, Juvenal Milton, que me ensinou a importância de aplicar meus conhecimentos de maneira prática.

À Lígia, minha irmã, que sempre me ensinou a não me contentar com meias vitórias e a continuamente buscar a melhoria.

À Giselle e ao Marcos, minha irmã e seu marido, que me ensinaram a importância da boa manutenção dos relacionamentos pessoais.

À Aline, minha irmã, que apesar dessa época de desentendimento, sempre me encorajou a manter um pensamento crítico.

Aos meus avós, pois cada um deles teve grande colaboração no meu crescimento pessoal.

Ao Prof. Evandro, que me orientou, aconselhou e animou nos momentos mais complicados da graduação.

Aos amigos que conheci durante a graduação, com quem aprendi a apreciar as diferenças entre as pessoas.

Aos professores do departamento, que me ensinaram preciosas lições, mesmo que, por vezes, da maneira difícil.

À Universidade de São Paulo, que me aceitou como aluno e disponibilizou recursos para minha graduação.

Aos cidadãos brasileiros, que pagam seus impostos e contribuem para a manutenção de universidades públicas, gratuitas e de qualidade.

Resumo

Devido às crescentes necessidades de automação e controle de sistemas de tempo-real, que têm rigorosos requisitos de previsibilidade do tempo de resposta, mostra-se viável a utilização de Sistemas Operacionais de tempo-real embarcados, cujo objetivo é, além de atender os requisitos impostos, simplificar e acelerar o desenvolvimento do software de controle. Nesse trabalho, usou-se o Linux, juntamente com o *patch RT*, também de código aberto, para construir um Sistema Operacional de Tempo-Real para a arquitetura ARM, o qual foi carregado no Kit de Desenvolvimento SAM9-L9260. Foram então realizados testes de desempenho, cuja função foi medir o tempo de resposta do conjunto. A partir de tais testes, usando-se de comparação com o Linux sem a aplicação do *patch*, constatou-se expressiva melhora de desempenho do sistema operacional em relação às tarefas de tempo-real, demonstrando elevada aplicabilidade e reduzido custo de implementação, tornando economicamente viáveis novas aplicações RT em automação.

Palavras-chave: Automação, Microcontroladores, Linux, ARM, Real Time, embarcado, Software Livre.

Abstract

Due to the growing needs of automation and control of real-time systems, which have strict requirements of response time predictability, it is shown viable the use of embedded real-time operating systems, whose goal is, in addition to meeting the imposed requirements, simplify and accelerate the development of the control software. In this final thesis, it was used Linux, along with the *RT patch*, also open source, to build a Real-Time operating system targeted to ARM architecture, which was loaded on a SAM9-L9260 Development Kit. So, there were conducted performance tests, whose function were measuring the response time of this ensemble. From these tests, using comparison with Linux without implementation of the patch, it was observed great improvement on Linux performance in regard to real-time tasks, demonstrating high applicability and reduced implementation cost, making economically viable new RT automation applications.

Keywords: Automation, Microcontrollers, MCU, Linux, ARM, Real-Time, Embedded, Open Source.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas

OS	Sistema Operacional – *Operating System*
RT	Tempo Real – *Real Time*
RTS	Sistema de Tempo Real – *Real Time System*
RTOS	Sistema Operacional de Tempo Real – *Real Time Operating System*
FOSS	Software Livre e de Código Aberto – *Free Open Source Software*
MCU	Microcontrolador – *Microcontroller Unit*
CPU	Microprocessador – *Central Processing Unit*
ARM	Máquinas RISC Avançadas – Advanced RISC Machines
RISC	Conjunto de Comandos Reduzido – *Reduced Intruction Set Computing*
CISC	Conjunto de Comandos Complexo – *Complex Intruction Set Computing*
RootFS	Partição Raiz do Sistema Operacional – *Root File System*
USB	Barramento Serial Universal – *Universal Serial Bus*
GPIO	Pino de Entrada e Saída Genérico – *General Pourpose Input Oputput*
APT	Ferramenta de Empacotamento Avançada – *Advanced Packaging Tool*
SSH	Terminal de Acesso Remoto Seguro – *Secure Shell*
PIC	Controlador de Interface Programável – *Programmable Interface Controller*

Sumário

Capítulo 1

Introdução

Nas duas últimas décadas, a utilização de Microcontroladores para Automação de Processos se tornou muito popular por sua elevada eficiência e custo reduzido. Dentre os microcontroladores mais simples, como o 8051 e alguns PICs (*Programmable Interface Controller*), é comum desenvolver as funcionalidades da aplicação usando linguagem C, ou, às vezes, simplesmente linguagem *Assembly*. Graças ao uso dessas linguagens de baixo nível é possível ter uma eficiente previsibilidade dos tempos de execução das aplicações desenvolvidas.

Ao longo dos anos, em busca de ganhos elevados de velocidade e eficiência, os microcontroladores vem sofrendo modificações que se inclinam para aumento de Conjunto de Comandos, Espaço de Endereçamento e Frequência de operação, além de modificações que incluem novos recursos mais avançados. Algumas dessas modificações, entretanto, tornam a programação do microcontrolador mais complexa, resultando na inviabilidade do uso de linguagens de baixo nível para sua programação.

Para auxiliar na solução do problema da complexidade, utiliza-se cada vez mais Sistemas Operacionais (SO) reduzidos, que ficam responsáveis por administrar os recursos da plataforma, e sobre os quais é realizado o desenvolvimento da aplicação desejada. No contexto geral, esse conceito trouxe grandes melhorias, inclusive no tempo de desenvolvimento [TANENBAUM (2003)]. No entanto, uma vez que os tempos e recursos do sistema não ficam mais sob controle do desenvolvedor, não há como garantir a previsibilidade dos tempos de execução das aplicações.

Para suprir a necessidade de algumas aplicações que precisam de previsibilidade de resposta (chamadas aplicações de Tempo Real) [LI; YAO (2003)], foram propostos os Sistemas Operacionais de Tempo Real (RTOS), que, para determinadas tarefas, tem a função de garantir que o sistema responda com um tempo previsível, e quando possível, reduzido.

Anteriormente, além do custo elevado de Microcontroladores de alto desempenho, os RTOS, que funcionavam sobre estes, constituíam-se principalmente de softwares proprie-

tários, fechados e com alto custo de aquisição, o que limitava o uso dessas plataformas às grandes corporações. Hoje, com a incrível redução de preços dos microcontroladores de alto desempenho, e o surgimento de algumas alternativas de RTOS em Software Livre, fez-se possível a exploração desses recursos para projetos de baixo custo.

A rápida evolução dos Microcontroladores nos últimos anos acontece também devido ao crescente mercado de *Smartphones* e *Tablets*. Esse mercado busca, cada vez mais, elevar o poder de processamento, reduzir a potência consumida e reduzir o custo de seus dispositivos. Assim, tal mercado optou por adotar, em sua maioria, o uso da arquitetura de microcontroladores ARM (*Advanced RISC Machines*) para seus produtos, e por isso o preço e a disponibilidade desses microcontroladores os tornaram interessantes para o desenvolvimento de projetos mais simples.

Além disso, o próprio mercado de *Tablets* e *Smartphones*, visando reduzir seu custo, tem investido bastante no desenvolvimento de software livre, sendo o Linux o maior beneficiado: mais de 64% dos Celulares vendidos no segundo trimestre de 2012 carregam o Android [Gartner Inc. (2012)], o SO do Google que utiliza kernel Linux, sendo que este, atualmente, é o mais bem sucedido projeto de SO Livre. Devido ao fato de seu código fonte ser aberto ao público e possuir uma boa documentação, há muita facilidade em adaptá-lo a necessidades particulares.

Nesse contexto, unindo a disponibilidade de microcontroladores com o crescente desenvolvimento do software livre, é viável a utilização do Linux em Sistemas Embarcados para realização de automação e controle de sistemas, incluindo os de tempo real.

1.1 Objetivos

Tendo em vista o cenário atual para desenvolvimento em plataformas embarcadas, o principal objetivo deste trabalho é desenvolver uma Plataforma de Automação com as seguintes características:

- Caracterize-se por um baixo custo de implementação;
- Tenha embarcado o sistema operacional Linux, com uma distribuição estável;
- Funcione a partir de um microcontrolador ARM de bom desempenho e baixo custo;
- Seja controlável via internet, através de uma interface Web;
- Ofereça uma segura previsibilidade de tempos de execução, ou seja, comporte-se como um RTOS;
- Seja altamente personalizável.

1.2 Justificativa

O desenvolvimento de uma plataforma de automação de baixo custo, que usufrua de RTOS deve possibilitar a oportunidade de desenvolvimento de aplicações para automação, de forma geral, hoje ainda pouco exploradas com esse conceito.

1.3 Organização do Trabalho

Este trabalho está estruturado por capítulos, sendo eles divididos da seguinte maneira:

- Fundamentação Teórica: Apresentação dos conceitos relevantes para o correto entendimento do presente trabalho;
- Metodologia: Apresentação dos materiais e métodos utilizados no trabalho;
- Resultados: Discussão e apresentação dos resultados obtidos a partir da Metodologia usada;
- Conclusão: Validação dos objetivos do trabalho, análise de aplicabilidade e conclusões finais.

Capítulo 2

Fundamentação Teórica

Este capítulo aborda os conceitos que serão necessários para entendimento do desenvolvimento do trabalho.

2.1 Sistema Operacional

Um sistema computacional moderno engloba processador(es), memória, dispositivos de armazenamento, dispositivos de interface com o usuário (como teclado, mouse, *touchpad*, monitor), dispositivos de rede, e outros milhares de periféricos mais específicos geralmente usados através da porta USB (*Universal Serial Bus*). Além disso, para cada uma dessas classes de dispositivo, existem vários padrões de operação que são escolhidos por preferência do fabricante.

Seria muito trabalhoso, se não inviável, modificar a aplicação para cada novo dispositivo de hardware que fosse inserido ou substituído no sistema computacional. Nesse contexto, surge a primeira função de um Sistema Operacional(OS): servir como uma camada de abstração de hardware para o programador [TANENBAUM (2003)]. Desse modo, o programador apenas se preocupa em enviar comandos a um dispositivo virtual e genérico, e o SO se encarrega de realizar a conversão desses comandos para o hardware utilizado. Uma demonstração gráfica desse recurso pode ser vista na Figura 2.1.

A segunda função do SO trata do gerenciamento de Recursos de hardware: Enquanto há apenas um processo rodando no sistema computacional, todos os recursos do mesmo estão disponíveis para esse processo. Mas a partir do surgimento da necessidade de se executar dois ou mais processos simultaneamente, há grande complexidade em se fazer gerenciamento inteligente dos recursos utilizados. Nesse ponto, o SO assume a tarefa, realizando o agendamento de uso de memória, tempo de processamento, e uso de periféricos, retirando da aplicação a responsabilidade de gerenciar os recursos disponíveis.

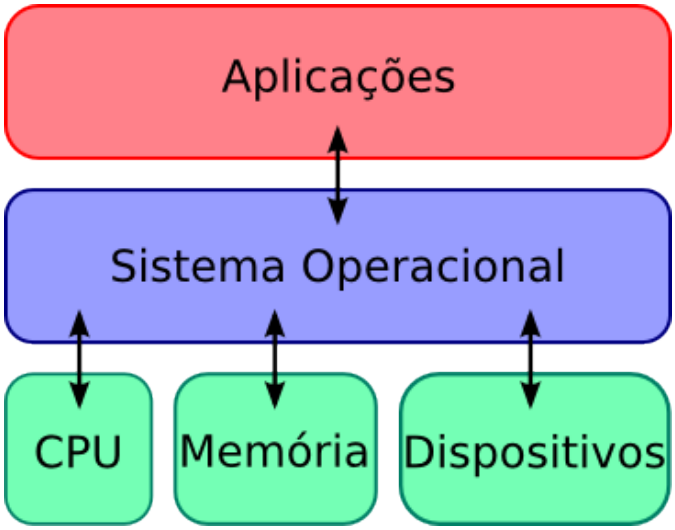

Figura 2.1: Diagrama Simplificado das camadas lógicas de um computador moderno.

2.2 Sistemas de Tempo Real

De maneira simplificada, um sistema de Tempo Real (RT) é aquele que exige, da entidade processadora, um limite restrito de tempo entre a aquisição dos dados e a emissão da resposta resultante do processamento dos dados aquisitados [LI; YAO (2003)]. Isso significa que o processamento deve, necessariamente, acontecer durante um determinado intervalo de tempo, cujo início é determinado pela saída do sistema.

O fato de o processamento não ter sido feito rápido o suficiente, é conhecido como perda de prazo (*deadline miss*). A reação do sistema à perda do prazo pode ser classificada em dois grupos:

1. *Soft Real-Time*: A perda do prazo, mesmo que em poucas situações, é indesejável, afetando negativamente o resultado do processo. Entretanto, a perda de poucos prazos não desestabiliza o sitema;

2. *Hard Real-Time*: Apenas uma perda de prazo tem consequências sérias, podendo inclusive causar instabilidade no sistema, fazendo com que o mesmo se comporte de maneira imprevista.

2.3 Sistemas Operacionais de Tempo Real

A função do Sistema Operacional de Tempo Real (RTOS) é, além de oferecer os recursos de um Sistema Operacional, suprir de maneira estável as necessidades de sistemas RT. Para isso, nele deve ser implementado um agendador de tarefas preemptivo, que possua a capacidade de interromper uma tarefa de menor prioridade para a execução de uma

tarefa com prioridade maior, sem que seja necessária colaboração da tarefa que estava em execução. Isso torna previsível o tempo entre a requisição da tarefa e o seu término.

Dessa maneira, se a tarefa de maior prioridade estiver associada ao sistema RT, todos os recursos de processamento do sistema estarão disponíveis para que a tarefa possa ser realizada no menor tempo possível, colocando assim todos os esforços da plataforma para realizar a tarefa crítica antes do prazo (*Deadline*).

É muito comum medir o desempenho de um RTOS pelo tempo entre a requisição originada pelo Sistema de Tempo Real e o início da tarefa de processamento relacionada. Esse tempo é conhecido como latência do sistema. A latência representa o intervalo de tempo que o RTOS gasta para detectar o requerimento de tarefa, interromper a tarefa atual e inciar a tarefa de alta prioridade.

2.4 FLOSS: *Free and Open Source Software*

Segundo a Free Software Foundation (2012), Software Livre é aquele em que todos os usuários, legalmente, tem todos os seguintes direitos:

- Executar o programa para qualquer propósito;
- Acesso público ao código fonte;
- Alterar o código da maneira desejada;
- Distribuir as alterações realizadas.

Esses direitos impactam de maneira positiva sobre projetos de pequeno porte, pois aproveitando as aplicações e ferramentas já implementadas pode-se, com pouco esforço, adaptar o Software Livre já existente para realizar as funções desejadas. Além disso, não há nada que impeça a venda de produtos com Software Livre embarcado, sendo isso um excelente incentivo para jovens empresas na indústria de tecnologia.

Atualmente existem diversas licenças de Software Livre, sendo que cada uma delas tem suas particularidades. Algumas delas são mais restritivas, como a GPL (GNU *Public Licence* [Open Source Initiative (2007)]), na qual consta a exigência de que todas as modificações realizadas no código sejam compartilhadas, e que os autores sejam mencionados. Outras licenças são mais permissivas, como a Licença BSD (Berkeley *Software Distribution* [Open Source Initiative (2008)]), que dá autonomia para que o usuário faça o que desejar com o código, inclusive transformar suas modificações no código em software proprietário.

Nesse trabalho, a grande maioria do software utilizado é licenciado pela GPL, que é normalmente adotada pelas distribuições Linux.

2.5 Linux

O Linux é um núcleo de Sistema Operacional compatível com a família de normas IEEE POSIX [LOCKE (2005)], e tem vasto suporte a diversas arquiteturas e dispositivos. Ele é desenvolvido sob o modelo de desenvolvimento de software livre, e tendo sido publicado sobre a licença GPL, é considerado atualmente o mais bem sucedido SO livre, podendo ser adquirido gratuitamente pela internet. [SILVA (2010)].

Devido à sua estabilidade, flexibilidade e padronização, o Linux é hoje usado em diversas aplicações, marcando forte presença em supercomputadores (estando em 462 dos 500 mais potentes do mundo [Top500 Supercomputer Sites (2012)]) e tendo crescente aceitação no mercado de *SmartPhones* [Gartner Inc. (2012)], onde é representado pelo Google Android.

Além das qualidades citadas previamente, o Linux conta com uma documentação bastante convidativa e uma boa comunidade de suporte, mostrando-se um SO excelente para o desenvolvimento de aplicações embarcadas.

2.6 Distribuições de Software

O objetivo de uma distribuição de software é facilitar a instalação de aplicativos e ferramentas em um sistema operacional. Isso acontece a partir do uso de um repositório de software, do qual o usuário tem a possibilidade de descarregar programas separados em determinadas coleções. Cada coleção tem diferentes objetivos: algumas delas prezam por softwares mais atualizados, enquanto outras priorizam a estabilidade do software utilizado.

Algumas distribuições de software mais conhecidas frequentemente oferecem outros recursos desejáveis:

- Gerenciador de Pacotes: É uma ferramenta que automatiza a aquisição de pacotes através do repositório e, geralmente, tem recursos de solução de dependência entre pacotes (adquire pacotes necessários para o funcionamento do pacote desejado);

- Pacotes Binários: A distribuição oferece os pacotes de software já pré-compilados e prontos para o uso.

Com o objetivo de evitar a concentração de tráfego de dados no servidor do repositório, as distribuições mais conhecidas recebem suporte de espelhos (*mirrors*), que são cópias exatas e verificadas do servidor original.

2.7 Interpretador de Comandos: *Shell*

A função do interpretador de comandos (*Shell*) é, como o nome deixa claro, interpretar os comandos enviados pelo usuário para o OS, servindo assim como interface que torna direta a comunicação entre estes [MATTHEW; STONES (2008)].

O *Shell* é, até a data de elaboração deste trabalho, a principal interface entre os sistemas Linux embarcados e o usuário, portanto seu uso é bastante requisitado nesse tipo de projeto. A partir do *Shell* é possível executar diversos aplicativos e utilizar basicamente todas as ferramentas disponíveis no sistema.

No intuito de realizar tarefas repetitivas de maneira automática, pode-se fazer um *Shell Script*. Seu funcionamento é simples: basta abrir qualquer editor de texto e escrever os comandos que devem ser realizados, na ordem desejada e separando-os por linha e salvá-los em um arquivo. Feito isto, basta conceder permissão para que o sistema operacional use arquivo de *Shell Script* como executável, e as tarefas nele descritas serão realizadas automaticamente. Comandos de condição (*if, else*) e laço (*for, while, until*) também estão disponíveis para a construção do *script*.

2.8 Microcontroladores ARM

ARM (*Advanced RISC Machines*) é uma arquitetura de microprocessadores em crescente desenvolvimento, que conta com as seguintes características [SLOSS; SYMES; WRIGHT (2004)]:

- Construída sobre o Design RISC (*Reduced Instruction Set Computer*) : visa proporcionar um conjunto com comandos reduzido, que sejam mais rápidos e possam ser organizados de forma a gerar instruções complexas mais flexíveis;
- Alguns Conceitos não RISC: Possui chaveamento para realizar algumas instruções complexas e instruções estendidas para processamento numérico de alto desempenho;
- Eficiência Elétrica: Projetada para ter baixo consumo e medidas mecânicas reduzidas, é ideal para equipamentos móveis;
- Alta Densidade de Código: O arquivo executável (binário) ocupa menos espaço em memória, o que é desejável em sistemas mais limitados.

Entretanto, para manter o baixo consumo elétrico, a maioria dos processadores ARM não dispõem de grande poder de processamento. A geração de binários, se realizada nativamente no ARM, pode ser bastante demorada e até inviável. Portanto é prática bastante comum a compilação cruzada (*Cross-Compiling*), que visa usar máquinas de

maior desempenho a fim de gerar binários que podem ser executados na arquitetura ARM.

2.9 Plataforma Embarcada com Linux

No geral, as plataformas com Linux embarcado contém os seguintes componentes de software:

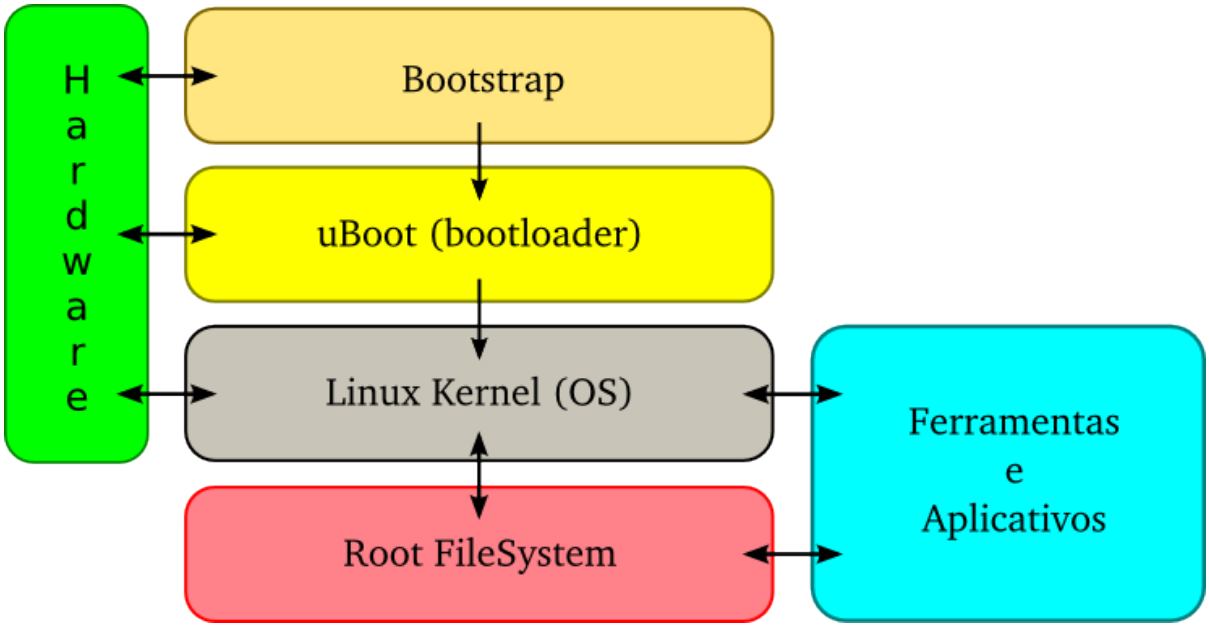

Figura 2.2: Diagrama Simplificado da comunicação entre os componentes de uma plataforma com Linux embarcado.

- *Root FileSystem* (RootFS): É a partição onde ficam os arquivos básicos do sistema, e geralmente todas as ferramentas e aplicações disponíveis pelo sistema. Essa partição pode opcionalmente guardar aquivos modificados, e ser usada para descarregar novos softwares. É essencial para o funcionamento do sistema operacional;
- *Kernel* Linux: É o núcleo do SO Linux, onde estão disponíveis as principais ferramentas de gerenciamento de hardware;
- Gerenciador de Boot (*Bootloader*): Software responsável por indicar a localização do *Kernel* e do RootFS, de modo que o OS possa ser iniciado. Alguns *Bootloaders* tem recursos avançados como suporte a rede e passagem de variáveis ao *kernel*;
- *Bootstrap*: Presente em alguns casos, esse software realiza o carregamento do *Bootloader*.

Na Figura 2.2 é mostrada a comunicação entre os componentes da Plataforma Embarcada com Linux.

Capítulo 3

Materiais e Metodologia

Este capítulo divide a metodologia utilizada em duas sessões: a primeira apresenta os materiais utilizados, englobando hardware e software. A segunda aborda os métodos utilizados para a obtenção dos resultados.

3.1 Materiais Usados

Nesta sessão são apresentados os materiais utilizados na construção do trabalho. O principal hardware utilizado foi o Kit de Desenvolvimento SAM9-L9260 (descrito na Seção 3.1.1), enquanto os principais softwares utilizados foram o código fonte do Linux (Seção 3.1.2), o *patch* RT (Seção 3.1.3) e a ferramenta de testes CyclicTest (Seção 3.1.4).

3.1.1 Kit de Desenvolvimento SAM9-L9260

Com o intuito de evitar o trabalho relacionado à montagem de hardware, foi adquirido o kit de desenvolvimento da Olimex® SAM9-L9260, que contém um microcontrolador ARM embarcado, bem como memórias volátil e NAND Flash. As configurações mais relevantes ao trabalho estão descritas a seguir [Olimex Ltd (2009)]:

- MCU AT91SAM9260 de 16/32 bits arquitetura ARM9^{TM} rodando a 200MHz;
- Frequência principal do sistema igual a 50MHz;
- 64 MB SDRAM;
- 512MB NAND *Flash* para armazenamento (RootFS);
- Conector *Ethernet* 100Mbit;
- Conectores USB *host* (tipo B) e USB *device* (tipo A);
- Interface Serial (RS232), usada como terminal serial;

- Conector para cartões de memória SD/MMC;
- *Bootloader* uBoot, e *Bootstrap* já configurados;
- *Kernel* Linux 2.6.31-rc3 customizado;
- Distribuição de Software Debian versão 4.0 codinome "Etch".

A Figura 3.1 mostra visões de perspectiva isométrica (esquerda) e visão de fundo (direita) da placa utilizada.

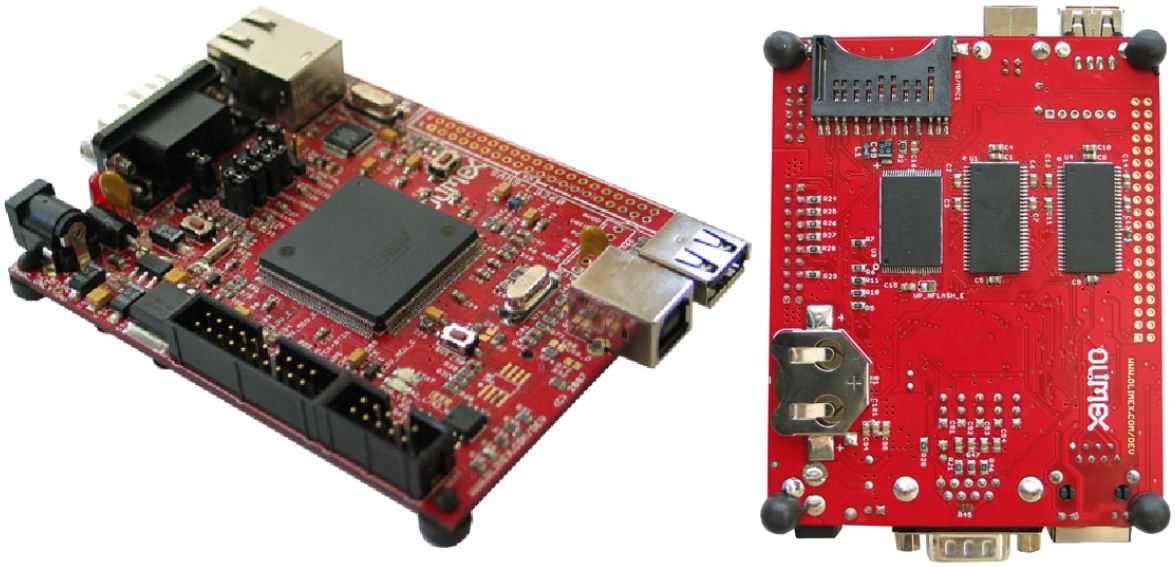

Figura 3.1: Fotos do Kit SAM9L9260.[Olimex Ltd (2009)]

O *uBoot* permite *download* de kernel via protocolo TFTP (*Trivial File Tranfer Protocol*), e oferece a possibilidade de usar um Root FileSystem (RootFS) a partir de cartão de memória SD/MMC ou Pendrive (USB *device*) ao invés da NAND *Flash* interna.

3.1.2 Código Fonte do *Kernel* Linux

O código fonte do *Kernel* Linux está disponível na internet [Kernel.ORG (2012a)]. O download pode ser realizado também em um dos espelhos disponibilizados nesse mesmo endereço. A partir do código foram feitas as modificações necessárias para tornar o Linux um RTOS.

A versão escolhida para a execução do trabalho foi a 3.4.9, sendo esta a versão mais nova que mantinha compatibilidade com o *patch Real-Time*, considerando o período em que ocorreu a execução deste trabalho.

3.1.3 *Patch Real Time* para Linux

Patch é um arquivo que traz algumas modificações que podem ser aplicadas ao código fonte de um software para fazê-lo realizar tarefas diferentes, corrigir algum defeito, ou adicionar suporte a novas plataformas. Quando o *patch* atinge um certo nível de maturidade e relevância para o projeto original geralmente ele é absorvido e se torna uma opção de configuração a mais na hora da compilação.

A função desse *patch* é aumentar a preempção do Linux, permitindo melhor atendimento das tarefas de alta prioridade. Isso é feito na tentativa de adaptar o Linux para que ele funcione como um Sistema Operacional de Tempo-Real. O *patch* RT é um projeto oficial do Kernel.org e o *Wiki* do projeto está disponível na internet [Kernel.ORG (2012b)].

3.1.4 Ferramenta de Testes *Cyclictest*

Cyclictest é uma ferramenta usada para levantar estatísticas sobre os tempos de latência da plataforma em que é executado. Essa ferramenta é recomendada pela comunidade Embedded Linux [eLinux.ORG (2012)] como uma boa prática de teste para sistemas RT.

O Cyclictest funciona requerendo, em uma frequência definida e bastante precisa, que o OS realize uma tarefa de alta prioridade. O tempo de latência é então calculado e armazenado. Ao final do teste tem-se o valor máximo, mínimo, médio e o número de vezes que a tarefa foi requerida.

3.2 Métodos Usados

Nesta sessão são explicados os métodos utilizados para a obtenção dos resultados, onde as Seções de 3.2.1 até 3.2.3 explicam os como foi feita a geração e a configuração da plataforma, enquanto a Seção 3.2.4 explica os procedimentos de teste.

3.2.1 Compilação Cruzada

Antes de iniciar a explicação sobre a compilação cruzada, é necessário salientar que em necessidade de executar algum processo como super-usuário ou administrador (*root*), os seguintes comandos foram usados antes do início do bloco:

```
1 sudo su  #(sistemas com sudo instalado) ou
2 su       #(sistemas sem sudo instalado).
```

Nada impede de que todos os processos sejam executados como root, mas por boa prática serão citados aqueles que exigem esse nível de privilégio.

A primeira etapa foi obter o código fonte do *Kernel* Linux e prepará-lo para aplicação do patch. Para isso, seguiu-se os seguintes passos:

```
1  # Realiza o Download do Linux versão 3.4.9.
2  wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.9.tar.xz
3  # Extrai o pacote em uma pasta de mesmo nome:
4  tar -xJf linux-3.4.9.tar.xz
5  # Para entrar na pasta referente:
6  cd linux-3.4.9
```

A próxima etapa foi obter o arquivo de *patch* e aplicá-lo ao código do Linux:

```
1 # Realiza o Download do Patch RT para o Linux 3.4.9
2   wget http://www.kernel.org/pub/linux/kernel/projects/rt/3.4/older/patch
        -3.4.9-rt17.patch.xz
3 # Extrai o Patch
4   xz -d patch-3.4.9-rt17.patch.xz
5 # Aplica o patch ao código.
6   patch -p1 < patch-3.4.9-rt17.patch
```

Com esses comandos, o código do Linux foi alterado com as modificações propostas pelo *patch*. Antes de continuar, foi necessário instalar alguns pacotes para possibilitar a compilação cruzada entre o computador usado e o kit de desenvolvimento. Os comandos a seguir foram usados em distribuições Debian ou descendentes (como o Ubuntu) que usam gerenciador de pacotes APT (*Advanced Packaging Tool*). Caso seja necessário realizar esse processo em outra distribuição é indicado usar o gerenciador de pacotes da mesma, ou o processo por ela indicado para realizar a instalação.

```
1 # Os comandos a seguir devem ser executados como Super-Usuário.
2 # Instalando o Compilador Cruzado
3  apt-get install gcc-arm-linux-gnueabi
4 # Instalando bibliotecas básicas para compilar o kernel.
5  apt-get install gcc make libncurses-dev
6 # Instalando bibliotecas necessárias para usar o configurador gráfico do
     kernel. (Opcional)
7  apt-get install libqt4-dev
```

Nesse ponto iniciou-se a configuração do Linux para o kit de desenvolvimento utilizado. A linha 8 gera a configuração padrão do kit, e foi obtida em seu manual [Olimex Ltd (2009)].

```
# Limpa qualquer tentativa de compilação anterior.
 make clean
# Exporta as variáveis para a Compilação Cruzada.
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-

# Aplica a configuração de compilação padrão do Kit.
  make sam9_l9260_defconfig
# Para configurar o Linux com a ferramenta gráfica:
  make xconfig
# Para configurar o Linux com a ferramenta de texto:
  make menuconfig
```

Para a configuração deste Kit foi necessário adicionar configurações específicas, sendo algumas para funcionamento em geral e outras para o funcionamento em Real-Time. Para mais detalhes veja o Apêndice A.

Uma vez acabados os ajustes, foram executados os seguintes comandos para realizar a compilação do Linux e a sua cópia para a pasta atual:

```
# Realiza a compilação, usando os 4 núcleos do processador para reduzir o
    tempo,
  make uImage -j4
# Move o Linux compilado para o diretório atual,
  mv arch/arm/boot/uImage .
```

Para finalizar, foi necessário mover o arquivo uImage para um servidor de TFTP(*Trivial File Tranfer Protocol*), de onde ele pudesse ser obtido pelo *uBoot*. O endereço IP do servidor era 10.235.0.130. Para mais detalhes, veja o Apêndice B.

3.2.2 Criação do *Root FileSystem*

Para a criação do *Root FileSystem* (RootFS) foi usada uma ferramenta da Distribuição Debian (que também está presente nas distribuições descendentes) chamada Debootstrap. Esta ferramenta usa os repositórios da distribuição Debian (ou descendentes) para montar um RootFS básico. Além desta, foi necessário instalar algumas ferramentas de emulação para concluir a configuração do Debian, de modo a transferir o RootFS pronto para o Kit. Os procedimentos de configuração são mostrado nos trechos de código a seguir:

```
# Os comandos a seguir devem ser executados como Super-Usuário.
#Instala o Debootstrap e os emuladores (qemu).
 apt-get install binfmt-support qemu qemu-user-static debootstrap
```

Após o término da instalação, via APT, do Debootstrap e das ferramentas de emulação, criou-se uma pasta onde foi feito o processo do Debootstrap. O processo foi realizado como mostrado a seguir:

```
# Os comandos a seguir devem ser executados como Super-Usuário.
# Criando e entrando na pasta usada no Debootstrap:
 mkdir Debootstrap
 cd Debootstrap
# Usando o Debootstrap para criar, na pasta debian, um RootFS da Versão Wheezy (7.0) usando a arquitetura armel (ARM),a partir do espelho da USP do repositório Debian.
 debootstrap --foreign --arch armel wheezy debian/ http://sft.if.usp.br/debian/
```

O próximo passo foi usar a pasta debian como RootFS e realizar a emulação da plataforma ARM para terminar a configuração. Para isso, foi necessário copiar o emulador para dentro da pasta debian, pois caso contrário a emulação falharia:

```
# Os comandos a seguir devem ser executados como Super-Usuário.
# Copia o emulador para a pasta debian.
  cp /usr/bin/qemu-arm-static debian/usr/bin/
# O comando a seguir exporta algumas variaveis e usa o comando chroot para iniciar a emulação de um Microcontrolador ARM.
  DEBIAN_FRONTEND=noninteractive DEBCONF_NONINTERACTIVE_SEEN=true LC_ALL=C LANGUAGE=C LANG=C chroot debian/

# Dentro da emulação, instala-se todos os pacotes DEB baixados com o comando debootstrap
dpkg --force-all -i /var/cache/apt/archives/*.deb

# Caso ocorra algum erro na configuração do base-files. comente esta linha (coloque um # antes de rmdir) para corrigir o bug.
# O comando a seguir o levará diretamente na linha 30, que precisa ser comentada.
  nano +30 /var/lib/dpkg/info/base-files.postinst
# ctrl+o para salvar e ctrl+x para sair.
```

Para possibilitar o uso do APT, adicionou-se os endereços dos repositórios de onde poderia ser feita a aquisição de pacotes. O repositório escolhido foi o da USP por se mostrar com maior taxa de transferência dentro do campus. O procedimento é mostrado a seguir:

```
# Adiciona o repositório SFT da USP na lista de repositórios.
  echo "deb http://sft.if.usp.br/debian wheezy main contrib non-free" > /etc/apt/sources.list
# Baixa a listagem do repositório e realiza as atualizações, além de corrigir os possíveis problemas com o base-files.
  apt-get update && apt-get dist-upgrade -y
#Para sair da emulação, use o comando exit
exit
```

Após terminada a emulação, a pasta debian continha todos os recursos básicos para, juntamente com um kernel, iniciar o sistema operacional. Por fim, foram transferidos os conteúdos da pasta debian para um *pendrive*, que foi usado no processo de *boot*.

```
# Os comandos a seguir devem ser executados como Super-Usuário.
# Formata (em EXT3) o pendrive (/dev/sdb)
  mkfs.ext3 -L "pendrive"  /dev/sdb
# Monta o pendrive na pasta /media/pendrive
  mount -t ext3 /dev/sdb /mnt/
# Copia conteúdo do RootFS para o pendrive.
  cp -r debian/* /mnt/
  umount /dev/sdb
```

3.2.3 Iniciando o Sistema

Foi necessário conectar, através de um cabo serial, o kit e um computador de mesa a fim de usar um emulador de terminal serial (Minicom, Teraterm) para alterar as configurações do *uBoot*. O emulador foi ajustado nas seguintes configurações:

- *Baud Rate*: 115200;
- Bits de Dados: 8;
- *Stop* Bits: 1;
- Sem controle de fluxo.

Ao ligar a placa, apareceu a opção de cancelar o *boot* automático. Quando cancelado, foi mostrado o terminal do *uBoot*, onde foram feitas as configurações para realizar o *boot* a partir do Kernel presente no servidor TFTP e o RootFS do *pendrive*. Uma vez que o arquivo *uImage* já estava no servidor TFTP e o *pendrive* já havia sido construído, bastou inserir-lo na entrada USB da placa (USB *device*), conectar o cabo de rede e usar os seguintes comandos de configuração (adaptados de [Olimex Ltd (2009)]):

```
# Aponta o local do servidor
  setenv serverip 10.235.0.130
# Atribui o endereço local
  setenv ipaddr 10.235.0.136
# Descarrega o uImage via TFTP e o carrega no endereço 0x22200000
  tftpboot 22200000 10.235.0.130:uImage
# Ajusta a localização do RootFS para o pendrive (/dev/sda1)
  setenv bootargs mem=64M console=ttyS0,115200 root=/dev/sda rootdelay=10
# Realiza o Boot
  bootm
```

Foi então realizado o *boot* da maneira desejada e o Sistema Operacional iniciou como esperado. Para listar as características do mesmo foi usado o comando *uname*:

```
# Comando para listar características do OS,
  uname -a
# Retorno
"Linux SAM9-L9260 3.4.9-rt17 #11 PREEMPT RT Mon Sep 10 17:25:28 BRT 2012
    armv5tejl GNU/Linux"
```

A partir destas listagens pode-se verificar que o processo de construção do Linux RT ocorreu com sucesso.

3.2.4 Testes de Desempenho

Nesta Seção são detalhados os procedimentos de teste utilizados para demonstrar as características obtidas pelo Linux após a configuração e a aplicação do *patch* RT.

Teste de *Clock*

O teste de *clock* foi um teste que teve por objetivo de verificar a diferença ocasionada pelo uso do comando *chrt* no comportamento do ambiente em diferentes situações.

A função do comando *chrt* é manipular os atributos RT de um processo, aumentando ou diminuindo a sua prioridade de execução. Através dele é possível selecionar os processos que serão vinculados ao sistema de tempo real, e portanto serão prioritários [SIEVER et al. (2009)].

O teste gerava sinal de *clock* a partir de um pino de GPIO(*General Purpose Input Output* - Pino de uso geral) o mais rápido possível. Com o auxílio de um osciloscópio, mediu-se o período de meia onda, que representa o tempo de chaveamento do GPIO.

Para isso, dois meios foram propostos para gerar o sinal: *Bash Script* e um programa escrito em linguagem C. Cada um desses meios foi testado de três maneiras:

1. Usando o Linux com o *patch* RT, sem alterar a prioridade da tarefa, que tem o mesmo resultado do Linux sem o *patch* RT;

2. Usando o Linux sem o *patch* RT, alterando a prioridade da tarefa para o máximo (*chrt*);

3. Usando o Linux com o *patch* RT, alterando a prioridade da tarefa para o máximo (*chrt*).

A partir desses valores foi possível perceber a influência do *patch* nas temporizações do Linux, e também os efeitos provocados pelo comando *chrt* no comportamento do Linux. Os tempos observados nesses testes foram referenciados como "períodos de meia onda", e equivalem ao tempo que leva ao ambiente realizar uma troca de valores em um GPIO. Os detalhes sobre a codificação do teste podem ser encontrados no Apêndice C.

Teste de Latência por GPIO

Esse teste já foi um pouco mais elaborado, e teve por objetivo encontrar, no kit testado, o tempo entre a recepção de um sinal por um pino de entrada e a resposta a esse sinal por um pino de saída. Esse tempo é denominado tempo de resposta do ambiente.

O sinal de entrada era proveniente do pino de GPIO de um Kit Auxiliar, e tinha a forma de onda quadrada com período de 2ms. A resposta aparecia a cada borda de transição do sinal de entrada, e era dada na forma de um pulso. Ambas entrada e saída foram colocadas nos dois canais de um osciloscópio, que foi colocado em modo de persistência infinita de imagem, e teve o *trigger* fixado nas bordas de transição da entrada. Isso ocasionou que todos os pulsos de saída fossem marcados na imagem, possibilitando a medida dos tempos de resposta. O diagrama de montagem do teste é representado na Figura 3.2.

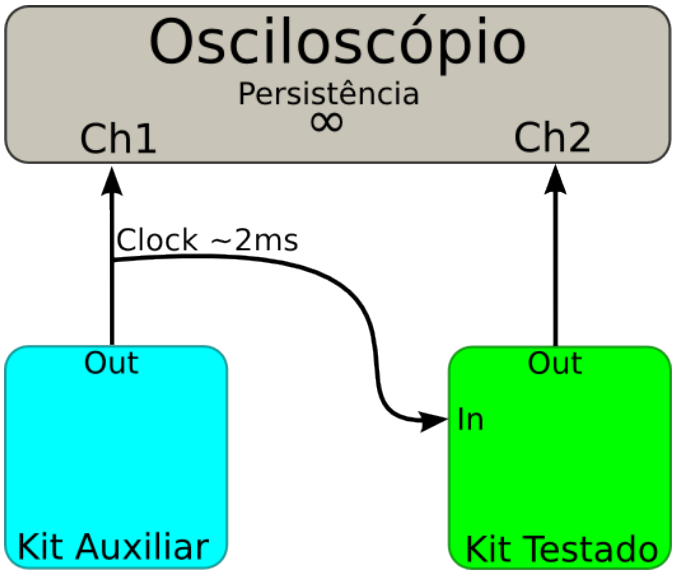

Figura 3.2: Diagrama de Montagem do teste de GPIO.

O teste foi aplicado em duas versões do Linux: uma com o *patch* RT, e a outra sem o *patch* RT, sendo que em cada versão o teste foi feito com o ambiente sobrecarregado de tarefas e repetido com operação em baixa carga.

Esse teste foi muito importante, por representar uma situação realística de aplicação da plataforma para automação.

Através de experiências empíricas, percebeu-se que infelizmente o teste não poderia ter duração muito longa, já que este utilizava a função de persistência de imagem no osciloscópio. Assim, a imagem poderia ser borrada por perda de sincronia, ocasionada por algumas oscilações na rede, ou quaisquer fatores externos.

Os tempos observados nesses testes serão referenciados como "tempos de resposta", e são referentes à soma dos tempos de latência e de alternância de nível no pino do GPIO.

Mais detalhes sobre a construção do teste encontram-se no Apêndice D.

CyclicTest

Antes de usar o *CyclicTest*, melhor explicado na sessão 3.1.4 , foi necessário obter o código fonte dos repositórios *Kernel.org* e compilá-lo no Kit já em funcionamento. O processo de compilação é simples e pode ser visto no Apêndice E.

Uma vez compilado e instalado, o CyclicTest foi chamado da seguinte maneira:

```
1 time cyclictest -m -a -t -n -p99
```

Onde:

- *time*: Comando que executa o comando a seguir e retorna o tempo levado para sua execução;
- -m : Trava as alocações de memória da maneira atual. Necessário para evitar erros de Segmentação;
- -a : Indica para todos os testes serem feitos com processador 1;
- -t : Usa uma *thread* por processador, pois o sistema possui apenas 1 *thread*;
- -n : Faz a temporização com a função *nanosleep()*, que é mais precisa;
- -p99 : Aumenta a prioridade do teste ao máximo, ou seja, o mesmo não pode ser interrompido, e sempre interrompe qualquer outra tarefa.

Por padrão, o *CyclicTest* executa um teste a cada 1ms e usa cada tempo medido para realizar as estatísticas do teste.

Capítulo 4

Resultados

Este capítulo apresenta os resultados dos testes apresentados no Capítulo 3. A Seção 4.1 apresenta os resultados dos testes de *clock*. Na Seção 4.2 são apresentados os resultados dos testes de GPIO. Já a Seção 4.3 apresenta os resultados do *CyclicTest*.

Note que nos resultados dos testes mostrados da Figura 4.1 até a Figura 4.10 deve-se verificar as medidas de tempo utilizando os valores dos cursores a direita de cada figura, onde são representados pela variável ΔX, visto que a escala de cada imagem foi ajustada para preservar a riqueza de detalhes.

4.1 Teste de *Clock*

O teste de *clock* foi realizado de duas maneiras: uma por *Bash-Script*, mostrado na primeira Seção, e outra por um executável escrito em linguagem C, mostrado na segunda Seção.
A constução de ambos os testes é mostrada no Apêndice C.

4.1.1 *Clock* em *Bash-Script*

A Figura 4.1 é resultado do programa *bash-clock* rodado no Linux com *patch* RT, a partir do seguinte comando:

```
1 ./bash-clock
```

A Figura 4.2 é resultado do programa *bash-clock* rodado no Linux sem o *patch* RT, a partir do comando:

```
1 chrt -f 99./bash-clock
```

A Figura 4.3, é fruto do programa *bash-clock* rodado no Linux com *patch* RT a partir do seguinte comando:

```
1   chrt -f 99 ./bash-clock
```

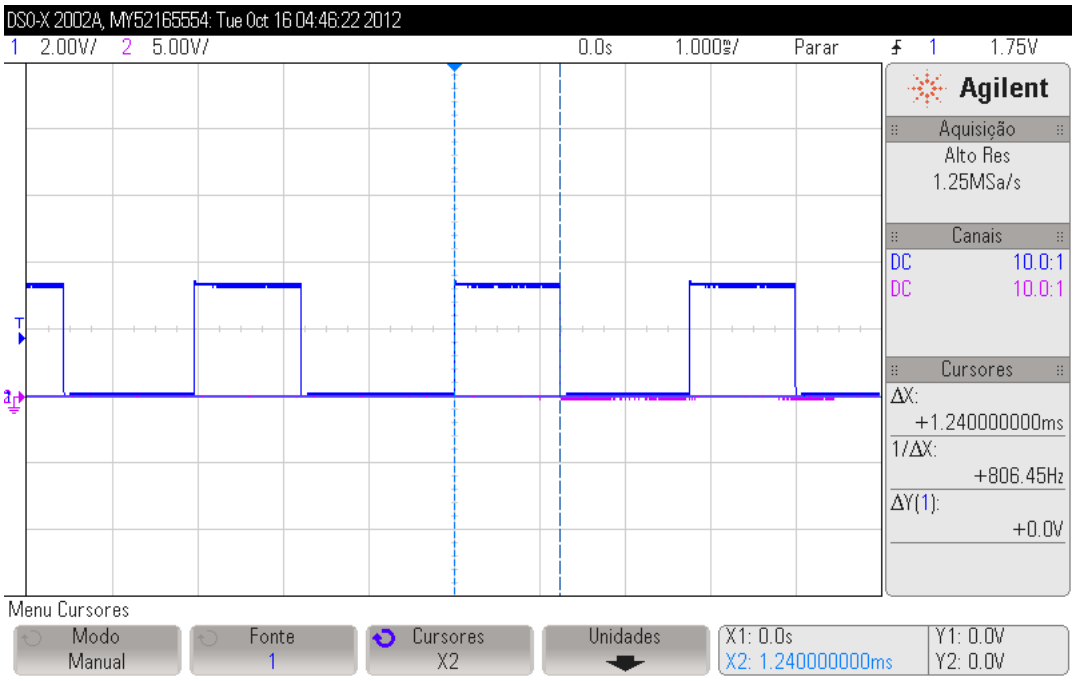

Figura 4.1: *Clock* via *Bash-Script* no Linux com o *patch* RT.

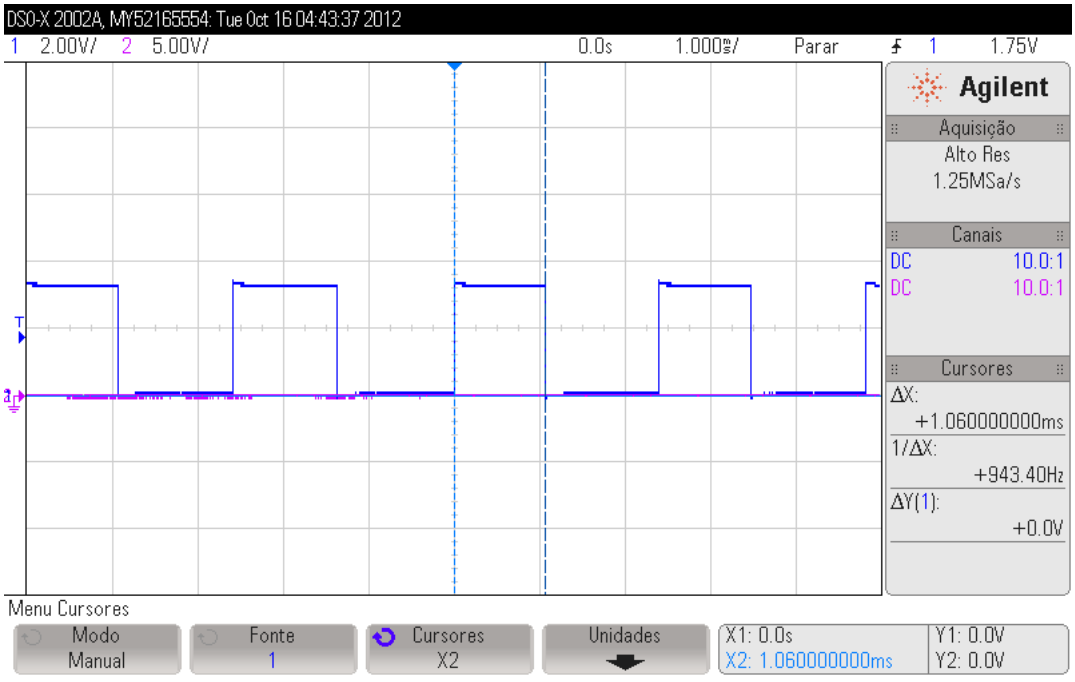

Figura 4.2: *Clock* via *Bash-Script* no Linux sem o *patch* RT, com *chrt*.

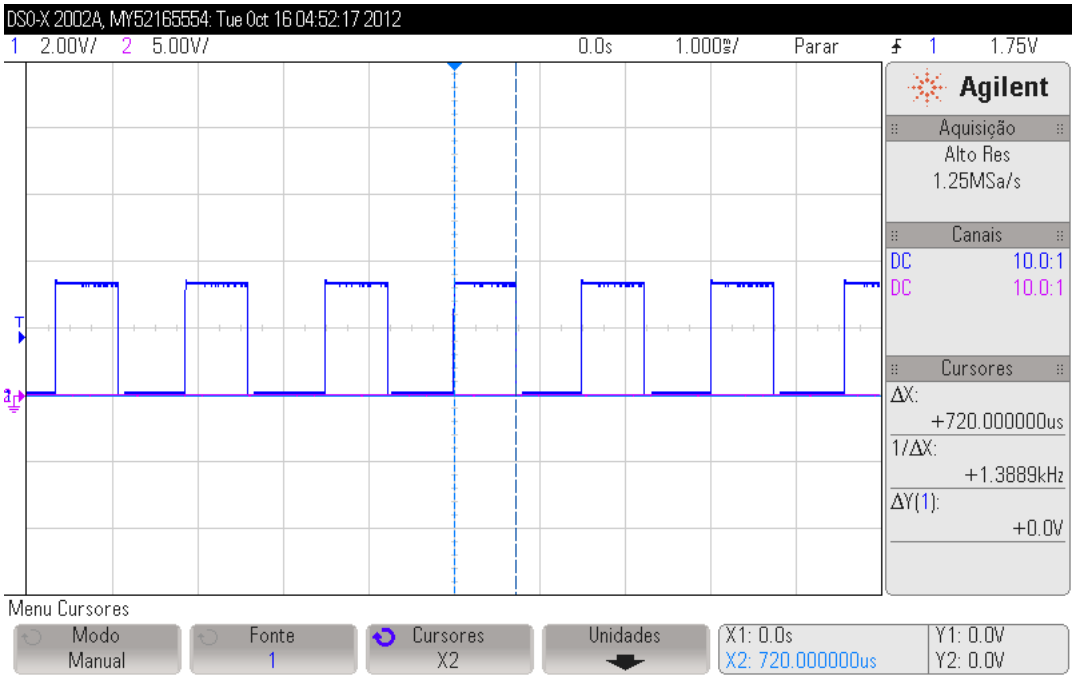

Figura 4.3: *Clock* via *Bash-Script* no Linux com o *patch* RT, usando *chrt*.

Todos os resultados do teste de *clock* escrito em Bash-Script estão concentrados na tabela 4.1.

Tabela 4.1: Resultados do Teste de Clock - *Bash-Script*

	Ambiente	*Bash-Script*
A	Linux + patch RT (sem chrt)	1240 us
B	Linux - patch RT (com chrt)	1060 us
C	Linux + patch RT (com chrt)	720 us
	Comparativo	
	B/A	85,4%
	C/A	58,0%
	C/B	67,9%

Pode-se verificar, a partir das figuras apresentadas, que o uso do *chrt* reduz os períodos de semi-ciclo, e que o efeito é mais expressivo no caso do teste usando chrt, mostrado na Figura 4.3, onde o valor equivale a menos de 60% do tempo da execução sem o uso do comando *chrt*, conforme mostra a Figura 4.1. Sabendo que o processo executado era exatamente o mesmo em todos os casos, pode-se confirmar que o *chrt* produz melhores resultados evitando que a tarefa de *clock* seja interrompida por outras tarefas.

4.1.2 *Clock* escrito em Linguagem C

A Figura 4.4 é resultado do programa *c-clock* rodado no Linux com *patch* RT, a partir do seguinte comando:

```
./c-clock
```

A Figura 4.5 é resultado do Teste de *Clock* para o Linux sem o *patch* RT, a partir do comando:

```
chrt -f 99./c-clock
```

A Figura 4.6, é fruto do programa *c-clock* rodado no Linux com *patch* RT a partir do seguinte comando:

```
  chrt -f 99 ./c-clock
```

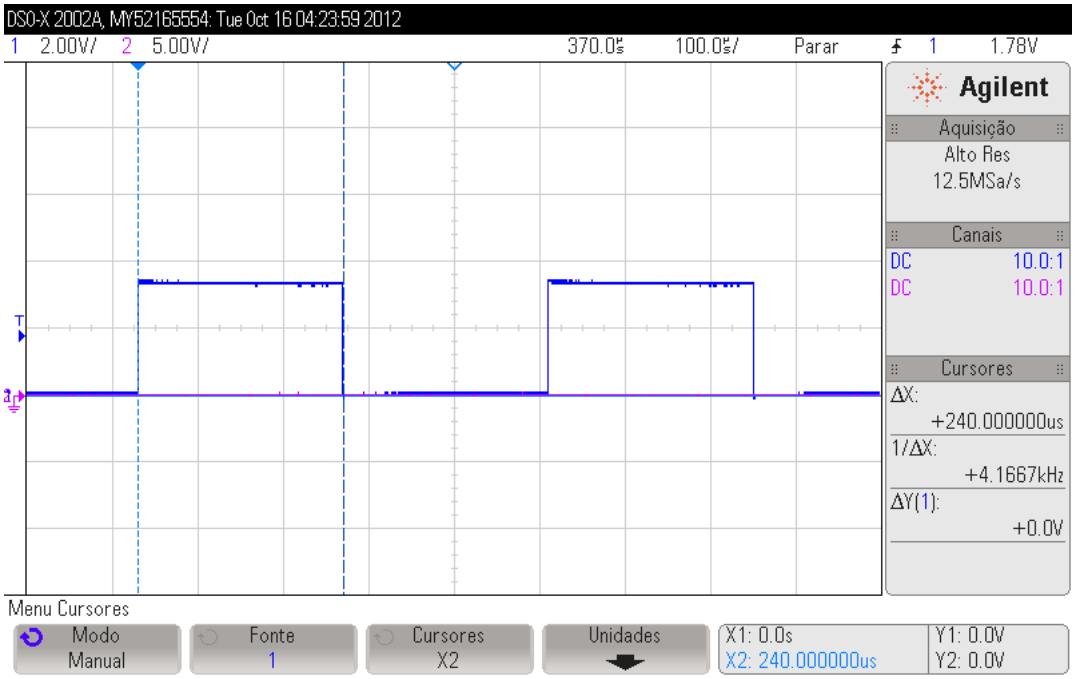

Figura 4.4: *Clock* escrito em Linguagem C, no Linux com o *patch* RT.

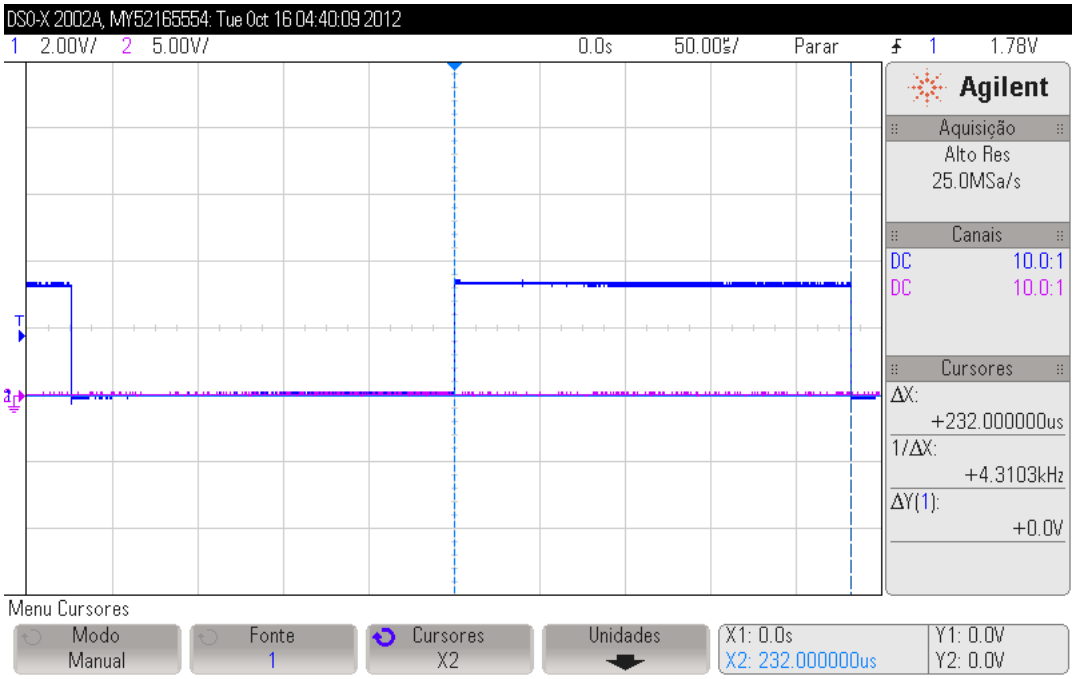

Figura 4.5: *Clock* escrito em Linguagem C, no Linux sem o *patch* RT, usando *chrt*.

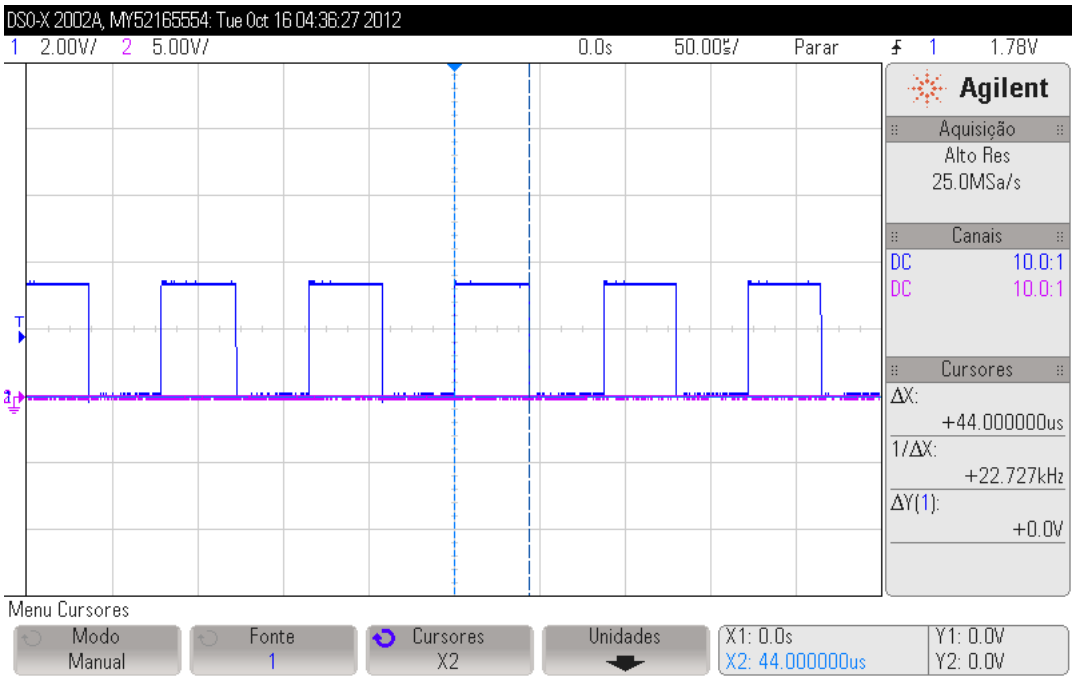

Figura 4.6: *Clock* escrito em Linguagem C, no Linux com o *patch* RT, usando *chrt*.

Todos os resultados do teste de *clock* escrito em Linguagem C estão concentrados na tabela 4.2.

Tabela 4.2: Resultados do Teste de Clock - Linguagem C

	Ambiente	Linguagem C
A	Linux + patch RT (sem chrt)	240 us
B	Linux - patch RT (com chrt)	232 us
C	Linux + patch RT (com chrt)	44 us
	Comparativo	
	B/A	96,7%
	C/A	18,3%
	C/B	19,0%

Através destes testes de *Clock*, pode-se notar a aplicação escrita em C tem uma eficiência maior, pelo fato de que o teste em *Bash-Script* realiza a abertura e fechamento do *file-descriptor* do GPIO a cada mudança de nível, enquanto o teste escrito em C realiza a abertura do *file descriptor* apenas uma vez no início do programa. Isso significa uma boa redução de chamadas do sistema e explica a redução drástica nos valores de tempo de meia-onda.

Pode-se verificar também que, novamente, apesar de também aparecer no Linux sem *patch* RT (Figura 4.5), a redução no tempo de semi-ciclo foi mais drástica no caso do Linux com *patch* RT (Figura 4.6), atingindo o valor menor que 20% do total percebido no mesmo ambiente sem o uso de *chrt*.

É importante ressaltar que, após a chamada do comando com *chrt*, no caso do Linux com *patch* RT, o Sistema Operacional parou de responder, indicando que toda a prioridade de execução havia sido dada ao programa de teste, a ponto de não haver condições de o Sistema Operacional atender os requerimentos do usuário.

4.2 Teste de Latência por GPIO

O teste de latência foi aplicado a dois ambientes distintos: O primeiro ambiente, descrito na Seção 4.2.1, opera com baixa carga de processamento, e portanto há pouca concorrência entre tarefas. O segundo ambiente, descrito na Seção 4.2.2, opera com sobrecarga de processamento, e portanto há mais concorrência entre tarefas.

4.2.1 Ambiente operando com Baixa Carga de processamento

A Figura 4.7 é resultado do Teste de GPIO no ambiente Linux com *patch* RT rodado pelo seguinte comando:

```
1 chrt -f 99 ./gpio-test
```

A Figura 4.8 é resultado do Teste de GPIO no ambiente Linux sem *patch* RT rodado pelo seguinte comando:

```
1  chrt -f 99 ./gpio-test
```

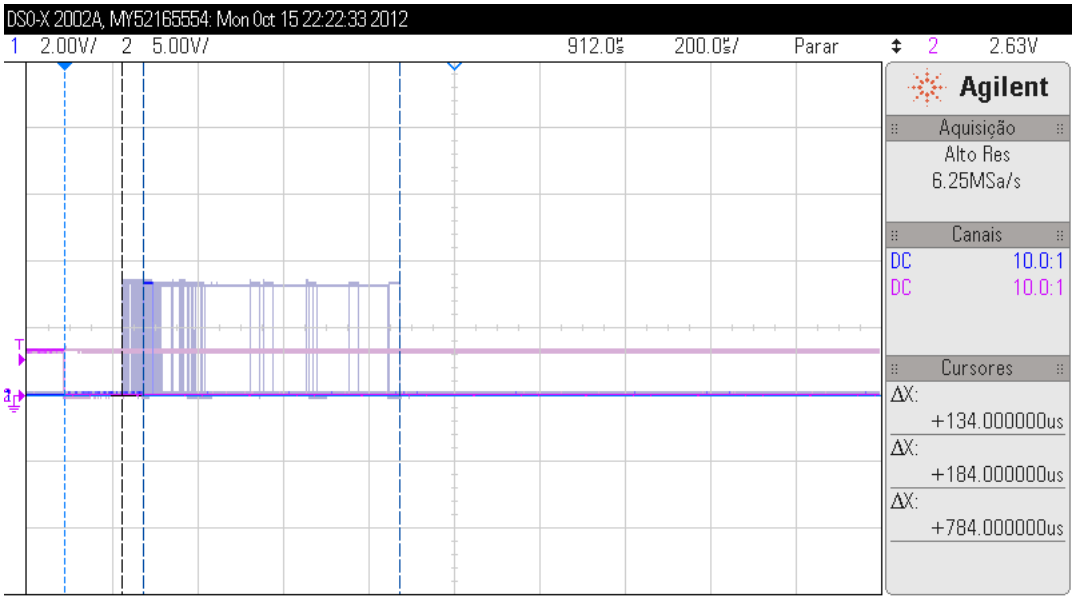

Figura 4.7: Teste por GPIO, com baixa carga, no Linux sem o *patch* RT.

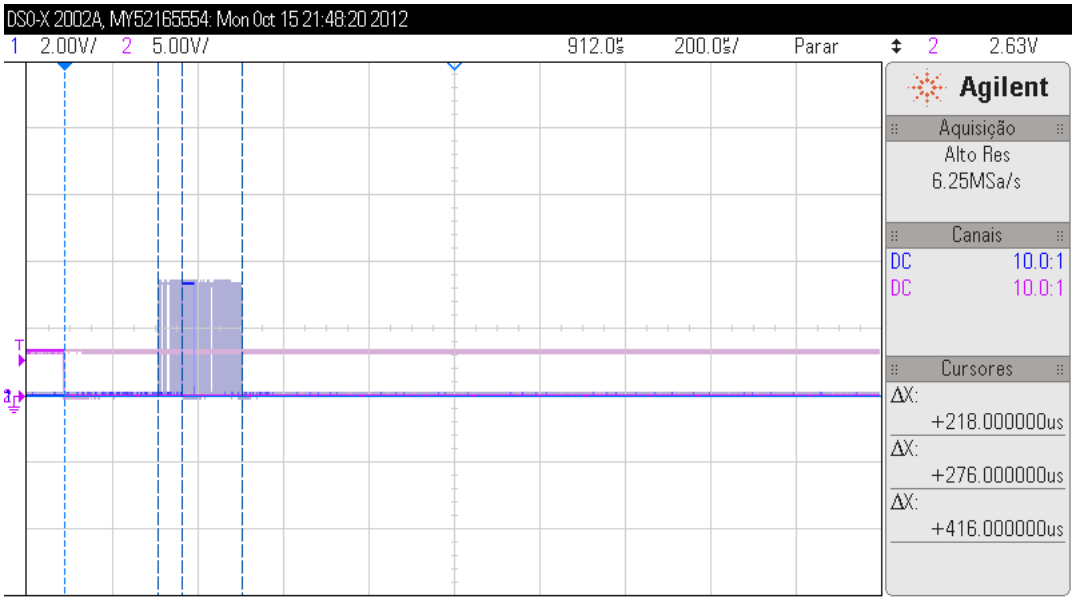

Figura 4.8: Teste por GPIO, com baixa carga, no Linux com o *patch* RT.

Os resultados do teste em baixa carga estão concentrados na tabela 4.3.

Tabela 4.3: Resultados do Teste de Latência por GPIO - Baixa Carga

	Ambiente	Latência Mínima	Latência Maxima
A	Linux padrão	134 us	784 us
B	Linux com o patch RT	218 us	416 us
	Comparativo		
	B/A	162,7%	53,0 %

Como o binário rodado era o mesmo, pode-se perceber que a previsibilidade do tempo de resposta da Figura 4.7 é bastante inferior à da 4.8, que apresentou um tempo máximo de resposta muito mais elevado. Isso ocorre porque, mesmo com o *chrt*, o ambiente sem o *patch* RT não tem as condições de evitar que a tarefa seja interrompida por outras. Já o ambiente com *patch* RT já mostra resultados mais condizentes com as necessidades de um Sistema de Tempo-Real.

4.2.2 Ambiente operando com Sobrecarga de processamento

O algoritmo que gera a carga no ambiente pode ser encontrado no Apêndice F. Ele é executado antes do comando de teste, como será mostrado a seguir.

A Figura 4.9 é resultado do Teste de GPIO no ambiente Linux com *patch* RT, com algoritmo de sobrecarga, rodado pelo seguinte comando:

```
1  ./doload.sh 65 &; chrt -f 99 ./gpio-test
2 # O primeiro comando (antes do ';') é o algoritmo de carga, e ele é rodado
    em segundo plano, evidenciado pelo uso do &.
```

A Figura 4.10 é resultado do Teste de GPIO no ambiente Linux sem *patch* RT, com algoritmo de sobrecarga, rodado pelo seguinte comando:

```
1  ./doload.sh 65 &; chrt -f 99 ./gpio-test
```

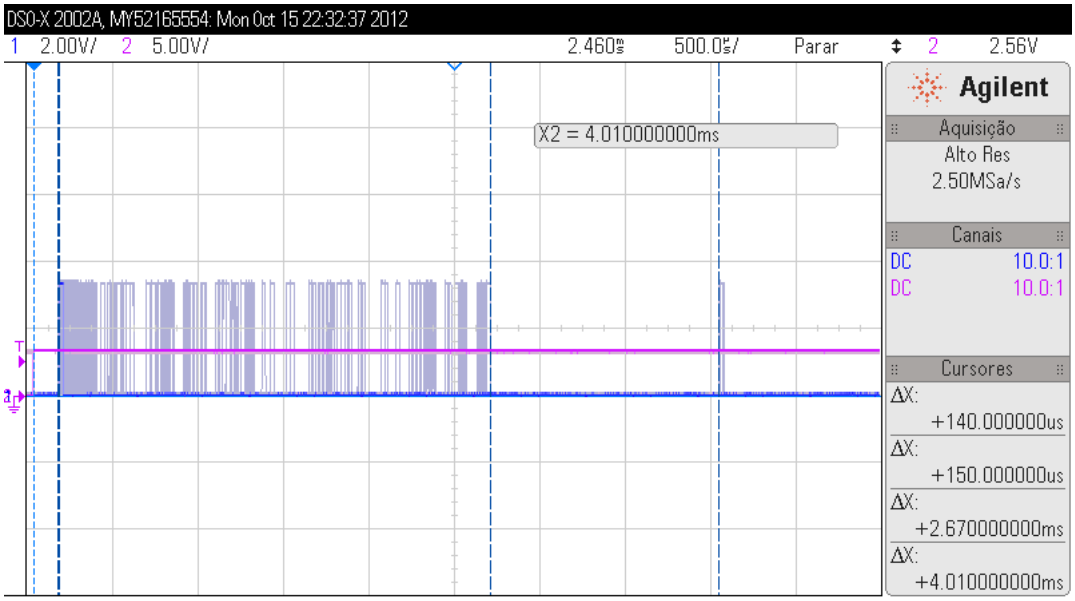

Figura 4.9: Teste por GPIO, com sobrecarga, no Linux sem o *patch* RT.

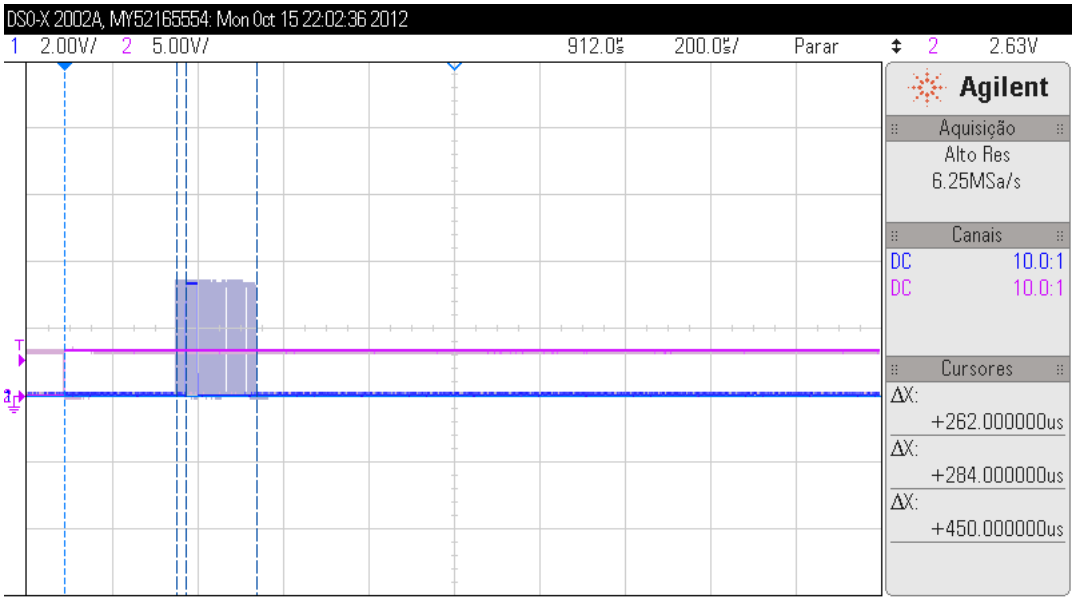

Figura 4.10: Teste por GPIO, com sobrecarga, no Linux com o *patch* RT.

Os resultados do teste em sobrecarga estão concentrados na tabela 4.4.

Tabela 4.4: Resultados do Teste de Latência por GPIO - Sobrecarga

	Ambiente	Latência Mínima	Latência Maxima
A	Linux padrão	140 us	4000 us
B	Linux com o patch RT	262 us	450 us
	Comparativo		
	B/A	187,1,7%	11,3%

É bem simples visualizar que a previsibilidade de tempos de resposta do ambiente com o *patch* RT é muito mais elevada do que a do ambiente sem o *patch*. Isso fica mais evidente no ambiente com sobrecarga, pois a quantidade de tarefas concorrendo com o teste realizado cresce, ocasionando que o teste no ambiente sem o *patch* RT seja interrompido mais vezes do que quando não há carga. Já no ambiente com o *patch* não há necessidade de preocupações com a carga, já que a tarefa de teste tem a maior prioridade possível.

Em um caso prático, a tarefa de teste é substituída pela tarefa crítica do Sistema de Tempo Real, que precisa ser atendida dentro de um prazo definido. O ambiente com o *patch* RT sempre seria capaz de atender a tarefa se o prazo fosse menor que 500us, mesmo quando houvesse sobrecarga no sistema. Já o ambiente sem o *patch* teria perdido vários prazos e poderia ter levado à instabilidade do sistema de Tempo Real.

4.3 *CyclicTest*

Para o experimento, dois Kits foram carregados com o mesmo RootFS, e dois Linux diferentes, sendo um deles com o *patch* RT e o outro sem. Ambos os sistemas foram acessados via acesso remoto (SSH - *Secure Shell*) por terminais diferentes, como mostrado abaixo:

```
1 # No Terminal #1, kit sem patch RT.
2   ssh root@10.235.0.135
3 # No Terminal #2, kit com patch RT.
4   ssh root@10.235.0.136
```

Em cada um dos terminais, foram solicitadas duas tarefas de cada placa, sendo a primeira com o objetivo de listar as características do ambiente e outra o próprio *CyclicTest*. Os comandos para tal são mostrados a seguir:

```
1 # Lista características.
2   uname -a
3 # Executa o Cyclictest
4   time cyclictest -m -a -t -n -p99
```

Após isso, o computador host, que acessou ambos os Kits via SSH, foi bloqueado e só voltou a ser desbloqueado ao final de um pouco mais de 90h. Os resultados foram obtidos após o desbloqueio do computador, quando os testes foram parados. No primeiro Kit, em que não foi aplicado o *patch* RT no Linux, foi obtido o seguinte resultado:

```
# No Terminal #1, kit sem patch RT.
# Resultado do uname -a
"Linux SAM9-L9260 3.4.9 #3 PREEMPT Mon Oct 15 12:44:15 BRT 2012 armv5tejl
    GNU/Linux"
# Resultado do CyclicTest
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.42 0.41 0.41 1/39 2284

T: 0 ( 1153) P:99 I:1000 C:326973304 Min: 61 Act: 132 Avg: 123 Max: 19178
^C
real  5449m33.611s
user  371m19.100s
sys 269m35.670s

"O tempo 'real' convertido é: == 326973,304s == 90h49m33s"
```

O resultado obtido pelo segundo Kit, no qual o Linux recebeu aplicação do *patch* RT, é mostrado a seguir:

```
# No Terminal #2, kit com patch RT.
# Resultado do uname -a
"Linux SAM9-L9260 3.4.9-rt17 #11 PREEMPT RT Mon Sep 10 17:25:28 BRT 2012
    armv5tejl GNU/Linux"
# Resultado do CyclicTest
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.49 0.59 0.59 1/48 1543
T: 0 ( 1165) P:99 I:1000 C:171597989 Min:  65 Act: 132 Avg: 116 Max: 247
```

Neste segundo Kit, que rodava o Linux com o *patch* RT, ocorreu o travamento do serviço de servidor SSH, entretanto, levando-se em consideração que foi pedido um ciclo de teste a cada 1ms, e que o teste anterior respeitou a proporção, o tempo deste teste pode ser calculado através do numero de cilos realizados:

```
"Tempo: 171597989 ciclos == 171597,989s == 2859m58s == 47h39m58s =~ 2 dias"
```

Como o cliente SSH registrou o resultado disponível após aproximadamente 48h de teste, considerou-se que o teste já havia registrado uma boa quantidade de dados, não houve preocupação com a origem do travamento, deixando este ser um tema de estudo e investigações futuros.

Os dados dos testes usando Cyclictest foram concentrados na tabela 4.5:

Tabela 4.5: Resultados do Teste Cyclictest

	Ambiente	Latência Mínima	Latência Média	Latência Maxima
A	Linux padrão	61 us	123 us	19178 us
B	Linux com o patch RT	65 us	116 us	247 us
	Comparativo			
	B/A	106,5%	94,3%	1,3%

Deste modo, pode-se comprovar, usando um teste considerado boa prática pela comunidade eLinux.ORG (2012), que o uso do *patch* RT mostrou resultados significativos em relação à previsibilidade de resposta do sistema, obtendo tempo de latência máxima de 247 us após quase 48h de teste.

Capítulo 5

Conclusão

Diante dos resultados obtidos no Capítulo 4, pode-se observar que os objetivos propostos no início do trabalho foram atingidos, obtendo uma plataforma de baixo custo, tempo de resposta com previsibilidade adequada e alta flexibilidade para a utilização em fins de automação.

Por meio de resultados obtidos nos testes, pode-se observar que a plataforma desenvolvida atenderia a sistemas de Tempo-Real que exijam menos de 500us de tempo de resposta, pois o tempo de latência máximo encontrado após quase 48h de teste foi menor que 250us. Esses valores são atrativos, levando-se em consideração que o hardware é bastante limitado.

Uma sugestão para trabalhos futuros é estudar o comportamento da plataforma em relação à necessidade de aplicações que utilizem Multi-Tarefas em Real-Time.

Outra possibilidade de estudo futuro é a análise da qualidade das tarefas que não são executadas em Real-Time, e como a possível perda de desempenho, em função do atendimento ao Real-Time, pode afetar a qualidade desses serviços.

Há boas estimativas de que, com componentes mais atuais e mais tempo de desenvolvimento do *patch* RT, a plataforma atinja boa conceituação e passe a ser considerada referência para automação em tempo real.

Referências

AT91SAM Community. **Real Time solutions on AT91SAM SoC**. Disponível em: <http://www.at91.com/linux4sam/bin/view/Linux4SAM/RealTime>. Acesso em 23 Oct. 2012.

eLinux.ORG. **Realtime Testing Best Practices**. Disponível em: <http://elinux.org/Realtime_Testing_Best_Practices>. Acesso em 23 Oct. 2012.

Free Software Foundation. **A Definição de Software Livre**. Disponível em: <http://www.gnu.org/philosophy/free-sw.html>. Acesso em 22 Oct. 2012.

Gartner Inc. **Worldwide Mobile Device Sales to End Users by Operating System in 2Q12**. Disponível em: <http://www.gartner.com/it/page.jsp?id=2120015>. Acesso em 21 Oct. 2012.

Kernel.ORG. **Linux Kernel Archives**. Disponível em: <https://kernel.org>. Acesso em 25 Oct. 2012.

Kernel.ORG. **Real Time Linux Patch**. Disponível em: <https://rt.wiki.kernel.org>. Acesso em 23 Oct. 2012.

LI, Q.; YAO, C. **Real-Time Concepts for Embedded Systems**. San Francisco, CA: CPM Books, 2003. 294p.

LOCKE, C. Posix and linux application compatibility design rules. **system**, [S.l.], v.20, p.21, 2005.

MATTHEW, N.; STONES, R. **Beginning Linux® Programming 4th Ed.** Indianapolis, IN: CPM Books, 2008. 780p.

Olimex Ltd. **SAM9-L9260 development board User's Manual**. Disponível em: <http://www.opencore.eesc.usp.br/leonardo/Documentos/Manual.pdf> (Rev.D) e <https://www.olimex.com/Products/ARM/Atmel/SAM9-L9260/resources/SAM9-L9260.pdf> (Rev.B). Acesso em 23 Oct. 2012.

Open Source Initiative. **GNU General Public License, version 3 (GPL-3.0)**. Disponível em: <http://opensource.org/licenses/GPL-3.0>. Acesso em 02 Dec. 2012.

Open Source Initiative. **The BSD 3-Clause License**. Disponível em: <http://opensource.org/licenses/BSD-3-Clause>. Acesso em 02 Dec. 2012.

SIEVER, E.; WEBER, A.; FIGGINS, S.; LOVE, R.; ROBBINS, A. **Linux in a Nutshell**. [S.l.]: O'Reilly Media, Incorporated, 2009.

SILVA, G. M. da. **Guia Foca GNU/Linux**. Brasil: Focalinux.org, 2010. 850p.

SLOSS, A. N.; SYMES, D.; WRIGHT, C. **ARM System Developer's Guide**. San Francisco, CA: Elsevier, 2004. 689p.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. São Paulo, SP: Pearson, 2003. 673p.

Top500 Supercomputer Sites. **Top500 Statistics**: operating system family and performance share. Disponível em: <http://i.top500.org/stats>. Acesso em 23 Oct. 2012.

Apêndices

Apêndice A

Configuração do Kernel Linux

Para configurar a o *Kernel* Linux para funcionar no Kit SAM9-L9260 foram necessárias as seguintes configurações na etapa de *make xconfig* ou *make menuconfig*:

Em *General Setup*,

- Troque o *Kernel Compression* de Gzip para LZMA (melhor compactação),
- Desabilite *Support for paging of anonymous memory (swap)*,
- Desabilite *Initial RAM filesystem and RAM disk (initramfs/initrd) support*,
- Habilite *Optimize for size*,

Em *Kernel Features*,

- Habilite *High Resolution Timer Support*,
- Em *Preemption Model*, selecione *Fully Preemptible Kernel (RT)* (Essa opção habilita o patch RT. No teste que não foi usado o patch, bastou comentar essa opção),
- Habilite *Use the ARM EABI to compile the kernel* e sua sub-opção.

Em *Device Drivers*,

- Em *Memory Technology Device (MTD) support*, habilite a opção *NFTL (NAND Flash Translation Layer) support*,
- Em *Misc Devices*, habilite a opção *Atmel AT32/AT91 Timer/Counter Library* ,
- Em *Network device support*, desabilite *Wireless LAN* ,
- Em *GPIO Support*, habilite */sys/class/gpio/... (sysfs interface)*.

Apêndice B

Servidor TFTP

O servidor usado nesse trabalho roda a Distribuição Debian Squeeze, onde foi feita a instalação do serviço de servidor TFTP(*Trivial File Tranfer Protocol*). A instalação do servidor foi feita da maneira listada a seguir:

```
1 # Esses comandos precisam ser executados como Root ou Super-Usuário
2   apt-get install tftpd-hpa
3 # Nesse processo será criada a pasta /var/lib/tftpboot, que funcionará como
      seu servidor TFTP. Caso queira possibilitar acesso e escrita a todos os
      usuários, use o comando:
4   chmod 777 /var/lib/tftpboot
5 # Para simplificar, é possível criar um link para essa pasta no diretório
    raiz,
6   ln -s /var/lib/tftpboot /tftp
```

Para colocar os arquivos no servidor TFTP, por simplicidade foi usado o scp:

```
1   scp uImage usuario@10.235.0.130:/tftp/
2   #Onde uImage é o Linux modificado,
3   #10.235.0.130 é o servidor
4   # e /tftp/ é a pasta do servidor TFTP.
```

Apêndice C

Teste de Clock

O teste de clock escrito em C foi compilado a partir do Kit, com conta de root. Bastou-se realizar os seguintes comandos:

```
# Abrindo aquivo para edição,
 nano clock.c
```

Dentro do editor, foi escrito o seguinte:

```
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <poll.h>
#include <unistd.h>

//Seleciona o Pino de GPIO
static const char *gpio_out = "/sys/class/gpio/gpio74/value";/*PB10*/

int main (void)
{
  system("echo 74 > /sys/class/gpio/export");
  system("echo out > /sys/class/gipo/gpio74/direction");

    mlockall(MCL_CURRENT | MCL_FUTURE); //bloqueia toda a memoria mapeada
        nesse processo

  fd_output = open(gpio_out, O_RDWR); //Abre o File Descriptor
  if (fd_output < 0)    //Verifica se houve erro.
    printf("can't open gpio_out");

```

```
27      while (1)
28      {
29        write(fd_output,"1",2);       /* Escreve "1"*/
30        write(fd_output,"0",2);       /* Escreve "0"*/
31      }
32
33    munlockall();       // Desbloqueia a memória
34  }
```

Após a edição, usou-se *ctrl + o* para salvar e *ctrl + x* para sair.

Uma vez editado o arquivo, basta usar o comando a seguir para compilar o teste:

```
1 gcc clock.c -o c-clock
```

O teste de Clock escrito em Bash-Script foi feito da seguinte maneira:

```
1 nano bash-clock
```

Dentro do editor foi escrito o seguinte:

```
1 echo 73 > /sys/class/gpio/export
2 echo out > /sys/class/gpio/gpio73/direction
3 while true
4 do echo 1 >/sys/class/gpio/gpio73/value
5 echo 0 >/sys/class/gpio/gpio73/value
6 done
```

Após a edição, usou-se *ctrl + o* para salvar e *ctrl + x* para sair.

Para conferir permissão de execução para o *script* foi feito o seguinte:

```
1 chmod +x bash-clock
```

Apêndice D

Teste de GPIO

Para gerar o *clock* de entrada para o teste de GPIO foi usado o *Bash-Script* do apêndice C com tempos de espera entre as alterações de nível.

O teste de GPIO foi escrito em C e compilado a partir do Kit, com conta de root. O código aqui usado foi encontrado em [AT91SAM Community (2012)] e contou com algumas modificações.

Para obter o teste, bastou-se realizar os seguintes comandos:

```
# Abrindo aquivo para edição,
 nano "gpio-test.c"
```

Dentro do editor foi escrito o seguinte:

```
/* Small program to test gpio Linux
 *
 * Copyright (c) 2011, Free Electrons
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published
     by
 * the Free Software Foundation.
 *
 */

#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <poll.h>
```

```
22
23 //Pinos Usados
24 static const char *gpio_in = "/sys/class/gpio/gpio75/value";
25 static const char *gpio_out = "/sys/class/gpio/gpio73/value";
26 #define MAX_BUF 64
27
28 //Função de retorno de erro.
29 static void pabort(const char *s)
30 {
31   perror(s);
32   abort();
33 }
34
35 #define POLL_TIMEOUT -1 /* No timeout */
36
37 int main (void)
38 {
39   //Comandos para preparar o sistema para o teste.
40   system("echo 73 > /sys/class/gpio/export");
41   system("echo out > /sys/class/gipo/gpio73/direction");
42   system("echo 75 > /sys/class/gpio/export");
43   system("echo in > /sys/class/gipo/gpio74/direction");
44   system("echo both > /sys/class/gipo/gpio74/edge");
45
46   int ret = 0;
47   int fd, fd_output;
48   struct pollfd fdset;
49   char *buf[MAX_BUF];
50   long count = 0;
51
52   //Bloqueando memória
53   mlockall(MCL_CURRENT | MCL_FUTURE);
54
55   //Abrindo File-descriptors
56   fd = open(gpio_in, O_RDONLY | O_NONBLOCK );
57   if (fd < 0)
58     pabort("can't open gpio_in");
59
60   fd_output = open(gpio_out, O_RDWR);
61
62   if (fd_output < 0)
63     pabort("can't open gpio_out");
64
65   fflush(stdout);
66   while (1)
67   {
68     fdset.fd = fd;
```

```
69      fdset.events = POLLPRI;
70      fdset.revents = 0;
71      ret = poll(&fdset, 1, POLL_TIMEOUT);
72
73      if (ret < 0) {
74        printf("\npoll() failed!\n");
75        return -1;
76      }
77
78      if (ret == 0) {
79        /* Must not appear*/
80        printf("Timeout .");
81      }
82
83      //Gerando o pulso
84      if (fdset.revents & POLLPRI) {
85        /* Write "1"*/
86        write(fd_output,"1",2);
87        /* Write "0"*/
88        write(fd_output,"0",2);
89        read(fd, buf, MAX_BUF);
90        count++;
91      }
92
93      fflush(stdout);
94    }
95  }
```

Após a edição, usou-se *ctrl + o* para salvar e *ctrl + x* para sair.

Para compilar o código editado

```
1 gcc "gpio-test.c" -o "gpio-test"
```

Apêndice E

Aquisição e Compilação do CyclicTest

Dentro da placa, foram realizados os seguintes comandos para aquisição e compilação do CyclicTest:

```
# Adquirindo o Software GIT
  apt-get install git
# Adquirindo o código fonte do CyclicTest pelo repositório Git
  git clone git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-
      tests.git
# Entrando na pasta e compilando o código
  cd rt-tests
  make all
# Copiando o executável para um caminho padrão, a partir de onde ele pode
    ser executado de qualquer lugar.
  cp ./cyclictest /usr/bin/
# Para obter ajuda sobre os parâmetros:
  cyclictest --help
```

Cabe lembrar que o código é Software Livre e está disponível nos servidores do *Kernel.org*.

Apêndice F

Algoritmo de Geração de Carga do Sistema

O *script* de geração de carga foi adaptado de um encontrado em [AT91SAM Community (2012)]. Inicialmente é necessário gerar um arquivo grande o suficiente para provocar carga na leitura. Isso pode ser feito encontrando um arquivo de *changelog* já presente no sistema.

```
zcat /usr/share/doc/gcc-4.6-base/changelog.Debian.gz > changelog.Debian
```

Após isso, o arquivo doload.sh foi editado como segue:

```
nano doload.sh
```

Dentro do arquivo foi escrito:

```
#!/bin/sh
#Geração de output (stdout) e uso de socket server.
(while true; do cat changelog.Debian; sleep 7; done | netcat -vv -l -p 5566
    ) &
a=$!
#Gera tráfego de leitura e escrita.
dd if=/dev/zero of=/dev/null &
b=$!
#Executa e mata um teste de performance
while true; do killall hackbench > /dev/null 2>&1; sleep 5; done &
d=$!
while true; do /bin/hackbench 1 > /dev/null 2>&1; done &
e=$!
#Trafego gerado por leitura do sistema de arquivos.
while true; do ls -lR / > /dev/null 2>&1; done &
f=$!
sleep $1
#Mata os processos anteiores
kill $a $b $c $d $e $f
```

Após a edição, usou-se *ctrl + o* para salvar e *ctrl + x* para sair.

Para conferir permissão de execução para o script foi feito o seguinte:

```
1 chmod +x doload.sh
```