UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Trabalho de Conclusão de Curso

UMA AVALIAÇÃO DO PROCESSO DE PORTABILIDADE DO SISTEMA OPERACIONAL ANDROID PARA UMA PLATAFORMA EMBARCADA

Luiz Fernando Pereira do Prado

ORIENTADOR

Prof. Dr. Evandro Luís Linhari Rodrigues

LUIZ FERNANDO PEREIRA DO PRADO

UMA AVALIAÇÃO DO PROCESSO DE PORTABILIDADE DO SISTEMA OPERACIONAL ANDROID PARA UMA PLATAFORMA EMBARCADA

Trabalho de Conclusão de Curso
apresentado à Escola de Engenharia de São
Carlos, da Universidade de São Paulo

Curso de Engenharia de Computação com
ênfase em Sistemas Embarcados

ORIENTADOR: Prof. Dr. Evandro Luís Linhari Rodrigues

São Carlos

Dezembro - 2011

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

Prado, Luiz Fernando Pereira do.

P896a Uma avaliação do processo de portabilidade do sistema operacional Android para uma plataforma embarcada. / Luiz Fernando Pereira do Prado ; orientador Evandro Luís Linhari Rodrigues -- São Carlos, 2011.

Monografia (Graduação em Engenharia da Computação com ênfase em Sistemas Embarcados) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2011.

1. Sistemas embarcados. 2. Portabilidade. 3. Android. 4. U-Boot. 5. SAM9-L9260. 6. Sistemas operacionais. I. Titulo.

FOLHA DE APROVAÇÃO

Nome: Luiz Fernando Pereira do Prado

Título: "Uma Avaliação do Processo de Portabilidade do Sistema Operacional Android para uma Plataforma Embarcada"

Trabalho de Conclusão de Curso defendido e aprovado em 01 / 12 / 2011 ,

com NOTA 9,5 (NOVE , CINCO), **pela comissão julgadora:**

Marcelo Andrade da Costa Vieira

Prof. Dr. Marcelo Andrade da Costa Vieira - SEL/EESC/USP

Prof. Dr. Francisco José Monaco - ICMC/USP

Prof. Associado Evandro Luís Linhari Rodrigues
Coordenador pela EESC/USP do
Curso de Engenharia de Computação

“Muitos dos fracassos da vida ocorrem com as pessoas que não reconheceram o quão próximas elas estavam do sucesso quando desistiram.”

Thomas A. Edison, inventor norte-americano.

AGRADECIMENTOS

Dedico esta monografia a meus pais Helena Maria Pereira do Prado e Nero Luiz do Prado, pois eles são a razão pela qual ultrapassei as dificuldades e me encontro concluindo a graduação na melhor universidade do país. Agradeço pela dedicação, a confiança e o apoio incondicional nos momentos e nas decisões difíceis tomadas nestes anos.

Meu agradecimento especial se destina ao Prof. Dr. Evandro Luís Linhari Rodrigues, pela confiança e pelo apoio na realização deste trabalho.

Agradeço também à minha irmã Carise Pereira do Prado, estendendo a todos os meus familiares, pela torcida incessante a meu favor. Em especial, a meu avô Apparecido Antônio do Prado, pelo exemplo de perseverança e garra.

Aos amigos da cidade de General Salgado e também do curso de Engenharia de Computação, em especial meu primo Igor Araújo do Prado pela presença e pelos conselhos.

Por fim, agradeço aos docentes e funcionários da Universidade de São Paulo, pelos esforços para a formação de bons profissionais e cidadãos.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE SIGLAS

RAM – *Random Access Memory*

PC – *Personal Computer*

CPU – *Central Processing Unit*

ROM – *Read-Only Memory*

MP3 – *MPEG Audio Layer 3*

PDA – *Personal Digital Assistants*

CRAMFS – *Compressed ROM File System*

JFFS – *Journaling Flash File System*

DRAM (SDRAM) – *(Synchronous) Dynamic Random Access Memory*

TFTP – *Trivial File Transfer Protocol*

IDE – *Integrated Development Environment*

SCSI – *Small Computer System Interface*

USB - *Universal Serial Bus*

EXT2 (EXT3) – *Extended File System*

FAT – *File Allocation Table*

API – *Application Programming Interface*

NFS – Network File System

EPROM – *Erasable Programmable Read-Only Memory*

RISC – *Reduced Instruction Set Computer*

DSP – *Digital Signal Processor*

CISC - *Complex Instruction Set Computer*

MB / GB – *Mega Bytes / Giga Bytes*

RTOS – *Real-Time Operating System*

MAC – *Media Access Control*

USART – *Universal Synchronous Asynchronous Receiver Transmitter*

SPI – *Serial Peripheral Interface*

ULA – Unidade Lógica e Aritmética

CD – *Compact Disc*

SDK – *Software Development Kit*

RESUMO

Sistemas operacionais têm sido cada vez mais aplicados a sistemas embarcados, devido à crescente complexidade dos *software*s, ao aumento do número de periféricos envolvidos e à evolução dos componentes de *hardware*. A presença do sistema operacional Android contorna alguns destes obstáculos, oferecendo ao programador a possibilidade de criar *software* que tenha funcionalidade uniforme em várias plataformas, independentemente da arquitetura ou dos modelos de periféricos presentes. Estas características levaram o Android à condição de forte concorrente no mercado de sistemas embarcados em pouco tempo. O processo de portabilidade de um sistema operacional para uma plataforma embarcada é um processo de roteiro determinado, todavia a aplicação das etapas reserva uma série de dificuldades ao desenvolvedor. Utilizando um ambiente de desenvolvimento SAM9-L9260 da Olimex, este trabalho reúne os conceitos e desafios do processo de portabilidade, apresentando aplicativos e ferramentas capazes de solucionar incompatibilidades e discutindo as melhores configurações para as partes importantes do processo, em especial o U-Boot, *software* imbuído da tarefa de preparar e gerenciar a inicialização do sistema operacional.

Palavras-chave: Portabilidade, Android, U-Boot, SAM9-L9260, Sistemas Embarcados, Sistemas Operacionais.

ABSTRACT

Operating systems has been increasingly applied to embedded systems, due to the increasing complexity of the software, the number of peripherals involved and the evolution of hardware components. The presence of the Android operating system circumvents some of these obstacles offering to the programmer the ability of making software that has a uniform performance on several platforms, regardless of architecture or peripherals models present. These characteristics led to the condition of strong competitor in the embedded systems market to Android in a short time. The process of porting an operating system to an embedded platform is a determined script process; however the application of the steps reserves a series of difficulties to the developer. Using an Olimex SAM9-L9260 development environment, this work brings together concepts and challenges of the portability process, introducing applications and tools capable of solving inconsistences and discussing the best settings to the important parts of the process, especially the U-Boot, software imbued with the task of preparing and managing the operating system initialization.

Keywords: Portability, Android, U-Boot, SAM9-L9260, Embedded Systems, Operating Systems.

1. INTRODUÇÃO

O uso de sistemas embarcados para integração e controle de vários dispositivos, a fim de desempenhar uma tarefa específica, não é inovador. Entretanto, as aplicações vêm se tornando mais complexas, tendo a incumbência de fazer cada vez mais cálculos, gerenciar mais periféricos e lidar com mais informações. O desenvolvedor de sistemas embarcados passa a não apenas construir interfaces de comunicação entre os vários periféricos, como também se preocupar com a complexidade crescente das aplicações que os conectam. A incorporação dos sistemas operacionais como ferramentas para sistemas embarcados possibilita o desenvolvimento rápido de aplicações complexas. A presença do sistema operacional retira a integração dos componentes das incumbências do desenvolvedor, acelerando a produção de aplicações embarcadas.

O sistema operacional escolhido deve oferecer suporte a um amplo conjunto de componentes e proporcionar a produção de aplicativos com alto grau de independência do dispositivo em que se encontra. A escolha do sistema depende das configurações da plataforma disponível e dos resultados que são esperados pelo desenvolvedor.

O Android oferece uma grande variedade de *drivers* para os mais diversos periféricos, além de ser de código aberto, fato que, aliado ao suporte dado pelo Google[1] (principal empresa participante do desenvolvimento do Android[2]), auxilia sua rápida dispersão em número de dispositivos e, consequentemente, de usuários. Os desenvolvedores de aplicações e dispositivos são fortemente beneficiados pela disponibilidade do código fonte do sistema. Há a flexibilidade necessária para gerar um sistema operacional que atenda às necessidades impostas pelo *hardware* e respeite as limitações do mesmo. Ainda, pode-se usar o ambiente de desenvolvimento Android, facilitando a produção e o teste de *software*. Contudo, a portabilidade de um sistema operacional para determinado *hardware* é um assunto amplo. Embora existam ferramentas especializadas para auxiliar nas etapas do processo, ele ainda reserva muitas dificuldades. Erros inesperados e tratamento de incompatibilidades entre as ferramentas estão entre os maiores problemas.

---

[1] Grande empresa da área de informática, que teve como principal produto um site de buscas.

[2] Sistema operacional embarcado utilizado atualmente em celulares, *tablets* e *netbooks*. Mais informações na seção 2.1.

Em se tratando de um *software* de código aberto, há bastante suporte por parte da comunidade de desenvolvedores. Todavia, o conhecimento para a correção de problemas não está centralizado e não é simples de encontrar. É necessário entender o funcionamento do sistema operacional e do sistema embarcado para conseguir a orientação correta para determinado problema.

1.1 Delimitação

O tema a ser pesquisado é relacionado com a portabilidade de um sistema operacional baseado em Linux em uma determinada plataforma de desenvolvimento embarcada. São apresentados os conceitos necessários para a utilização dos métodos básicos de conversão do código fonte do sistema e compatibilidade com a plataforma. O detalhamento dos erros e dificuldades encontradas no processo de execução dos métodos citados e as formas de solução encontradas também são descritos.

1.2 Objetivos da pesquisa

O trabalho tem um objetivo geral e alguns objetivos específicos, que devem ser atingidos à medida que o projeto se aproxima de seu propósito inicial.

1.2.1 Objetivo Geral

O trabalho tem como objetivo avaliar a aplicabilidade dos métodos de portabilidade de sistemas operacionais em plataformas embarcadas, utilizando o sistema operacional Android, conhecido pela ampla disponibilidade de informações e pelo suporte a um grande número de periféricos.

1.2.2 Objetivos Específicos

Considerando o objetivo principal, os objetivos específicos são descobrir os pontos de dificuldade e de decisões importantes de projeto acerca do processo de portabilidade, identificar as limitações das ferramentas utilizadas e verificar as propriedades do Android.

1.3 Justificativa

Segundo Ableson, et al. (2011), o número de sistemas embarcados e a sua importância no nosso cotidiano têm crescido com o tempo, estando presentes em telefones celulares, tocadores de MP3, terminais bancários, carros, equipamentos médicos, entre outros. Um dispositivo com um microprocessador que realiza funções computacionais específicas (não de propósito geral) é considerado um sistema embarcado.

Com esta expansão, as aplicações embarcadas tornaram-se mais complexas e a demanda por plataformas que reunissem cada vez mais funcionalidades e gerisse um número cada vez maior de periféricos foi crescendo significativamente. Considerando a evolução nas plataformas de *hardware* capazes de suportar aplicações mais sofisticadas e que exigem mais recursos, passou a ser viável a implantação de sistemas operacionais, facilitando não só a integração dos componentes, como a padronização do desenvolvimento de aplicações.

Nota-se, então, que Android surgiu em um ambiente muito favorável. Ableson, et al. (2011) afirmam que o Android promete movimentar o mercado, considerando não apenas a funcionalidade do sistema, mas também a forma com que o sistema fez sua inserção no mesmo. Darcie e Conder (2010) ressaltam também que, apesar de relativamente jovem, o Android já tem alto grau de competitividade com sistemas já estabelecidos no mercado há algum tempo. Steele e To (2010, p.1) complementam: “A ideia de um sistema operacional de código aberto para sistemas embarcados não era nova, mas o apoio agressivo do Google definitivamente ajudou a empurrar o Android para o primeiro plano em apenas alguns anos”.

Porém, a utilização de um sistema operacional embarcado ainda é limitada por ser uma área que está em seu desenvolvimento inicial. As ferramentas utilizadas são poucas e nem sempre funcionam de forma compatível entre elas. Também o suporte, tanto das plataformas de desenvolvimento quanto das ferramentas, não é abrangente, de forma que alguns erros precisam de extensa pesquisa junto à comunidade de desenvolvedores para serem solucionados.

Portanto, esta pesquisa tem como finalidade avaliar as dificuldades de implantação de um sistema operacional, o Android, em uma plataforma embarcada, mostrando as dificuldades e conceitos necessários para utilizar este tipo de tecnologia embarcada.

2. EMBASAMENTO TEÓRICO

Nesta seção, serão apresentados alguns conceitos necessários para o bom entendimento do projeto realizado.

2.1 Sistema Operacional Android

O Android foi o sistema operacional escolhido para exemplificar o processo de portabilidade pelas suas características, como disponibilidade de código e pluralidade de *drivers* para periféricos.

Segundo Darcey e Conder (2010, p.7): "Em 2007, um grupo de fabricantes de celulares, operadoras de telefonia móvel, e desenvolvedores de *software* [...] formaram a *Open Handset Alliance*, com o objetivo de desenvolver a próxima geração de plataforma sem fio". A Open Handset Alliance é apresentada por Ableson, et al. (2011) como uma aliança entre cerca de 50 empresas com objetivo de trazer ao mercado de telefonia móvel um produto melhor e de código aberto, proporcionando maior liberdade aos desenvolvedores.

Android foi lançado sob duas licenças diferentes de código aberto. A plataforma Android é licenciada pela *Apache Software License* (ASL), enquanto o *kernel* Linux contido no sistema operacional em questão está sob a *GNU General Public License* (GPL), o que é obrigatório para todo sistema operacional de código aberto. As duas licenças utilizadas tornam o sistema todo de código aberto. (ABLESON; et al., 2011)

Gargenta (2011, p.1) define o Android como

> [...] uma plataforma realmente aberta que separa o *hardware* do *software* que roda nele. Isto permite um número muito maior de dispositivos para executar as mesmas aplicações e cria um ecossistema muito mais rico para os desenvolvedores e consumidores.

Segundo Steele e To (2010, p.2), o Google, principal componente da *Open Handset Alliance*,

> [...] fornece assistência aos desenvolvedores de aplicativos (terceiros) em muitas formas como a Ferramenta de Desenvolvimento Android (ADT) na forma de *plugins* para o Eclipse [uma plataforma de desenvolvimento de código Java] [...], incluindo capacidades de registro em tempo real, um emulador realista que executa o código ARM nativo, e relatórios de erro de usuários para desenvolvedores de aplicativos para o *Android Market* [central de venda de aplicativos do Android].

O Android abstrai as diferenças de hardware, separando as características específicas dos dispositivos das aplicações; facilitando o desenvolvimento de aplicativos. Outras vantagens são a adaptação rápida aos métodos de programação e suporte a vários periféricos. Porém, há desafios conectados a essa ascensão para os desenvolvedores. Torna-se cada vez mais difícil o teste de aplicações, considerando a imprevisibilidade das características do aparelho do usuário. Algumas características podem mudar entre dispositivos, como tamanho de tela, teclados, sensores, versões do sistema operacional, taxas de comunicação, entre outras. (STEELE; TO, 2010)

Uma das maiores vantagens do Android é o ambiente de desenvolvimento, disponibilizado para o desenvolvedor sem custo. O emulador Android é uma das principais ferramentas inclusas neste ambiente, pois permite que o desenvolvedor seja iniciado na programação para Android sem o dispositivo ou que o mesmo possa testar configurações de outros dispositivos sem ter que possuí-los.

2.2 Linux

Sendo o Android construído embasado em um *kernel* Linux, é interessante apresentar as características principais deste sistema operacional.

O Android possui seu *kernel* baseado no sistema operacional Linux, uma rica interface de usuário, aplicativos para o usuário final, bibliotecas de código, suporte multimídia, entre outros.

Welsh et al. (1998, p.3) afirmam que o Linux

> [...] é uma versão gratuita do UNIX desenvolvida principalmente por Linus Torvalds na Universidade de Helsinki, na Finlândia, com a ajuda de muitos programadores UNIX e assistentes através da Internet. Qualquer pessoa com bastante conhecimento e bom senso pode desenvolver e mudar o sistema. [...] UNIX é um dos sistemas operacionais mais populares no mundo inteiro por causa de sua grande base de apoio e distribuição. Ele foi originalmente desenvolvido na AT&T [grande empresa de telefonia] como um sistema multitarefa para minicomputadores e *mainframes* na década de 1970, mas desde então cresceu e se tornou um dos sistemas operacionais mais amplamente usados [...].

Uma diferença do Linux para o UNIX comercial é que este último era desenvolvido sob uma política de qualidade rigorosa, utilizando sistemas de controle de código e revisão,

documentação e procedimentos para tratamento de erros. O modelo de *software* empregado pelo Linux descarta o conceito de desenvolvimento organizado, sendo feito de forma cooperativa por voluntários, usando a internet como forma de contato. (WELSH, et al., 1998)

Linux tem se tornado conhecido não só por ser um *software* livre e de fácil acesso (qualquer pessoa pode baixá-lo da internet gratuitamente), mas também por ser de boa qualidade, garantindo poucas falhas e até mais segurança que alguns sistemas comerciais. (SIEVER, et.al, 2009)

Silberchatz, et al. (2005, p.743) dividem o sistema operacional Linux em três partes principais de código:

> 1. *Kernel*. O *kernel* é responsável por manter todas as abstrações importantes do sistema operacional, incluindo funcionalidades como memória virtual e processos.
> 2. Bibliotecas do sistema. As bibliotecas do sistema definem um conjunto padrão de funções através do qual as aplicações podem interagir com o *kernel*. Estas funções executam grande parte da funcionalidade do sistema operacional que não necessita de totais privilégios de código do *kernel*.
> 3. Utilitários do sistema. Os utilitários de sistema são programas que executam tarefas de gerenciamento individuais e especializadas. Alguns utilitários do sistema podem ser invocados apenas uma vez para inicializar e configurar alguns aspectos do sistema, outros - conhecidos como *daemons* na terminologia UNIX - pode executar de forma permanente, fazendo desde tratamento de tarefas, como respondendo a conexões de rede de entrada, aceitando solicitações de *logon* dos terminais, e atualizando *log* de arquivos

2.3 O *kernel* Linux

O *kernel* é o núcleo do sistema operacional. O Android utiliza um *kernel* Linux com algumas alterações. Por isso, é fundamental o estudo do funcionamento do mesmo para a configuração de um *kernel* Android.

Segundo Love (2010), o *kernel* é o núcleo do Linux, provendo o embasamento para o sistema (como mostrado na figura 1), gerenciando o *hardware* e os recursos disponíveis. Alguns componentes comuns contidos no *kernel* são: sistema de tratamento de interrupções, escalonador para realizar a divisão de tempo de uso do processador por cada processo, gerenciador de memória e outros serviços do sistema, como comunicação entre processos e interface de rede.

O Linux é considerado um sistema bastante flexível, apesar de sua extensa base de código. Contribui para a afirmação anterior a possibilidade de personalização do código para um

ambiente específico, provendo uma grande chance de obter um sistema menor e mais rápido que se utilizada uma das distribuições Linux. (KROAH-HARTMANN, 2006)

De acordo com Love (2010, p.5, grifo do autor):

> Em sistemas modernos, com unidades de gerenciamento de memória protegida, o *kernel* normalmente reside em um estado de sistema elevado em comparação com aplicativos do usuário padrão. Isto inclui um espaço de memória protegido e acesso total ao *hardware*. Refere-se ao coletivo entre este estado do sistema e o espaço de memória como "espaço do *kernel*". Por outro lado, as aplicações do usuário executam no "espaço do usuário". Eles enxergam um subconjunto dos recursos disponíveis da máquina e podem executar determinadas funções do sistema [...]. Ao executar o código do *kernel*, o sistema está no "espaço do *kernel*", executando no modo *kernel*. Ao executar um processo comum, o sistema está no "espaço do usuário", executando no modo de usuário.

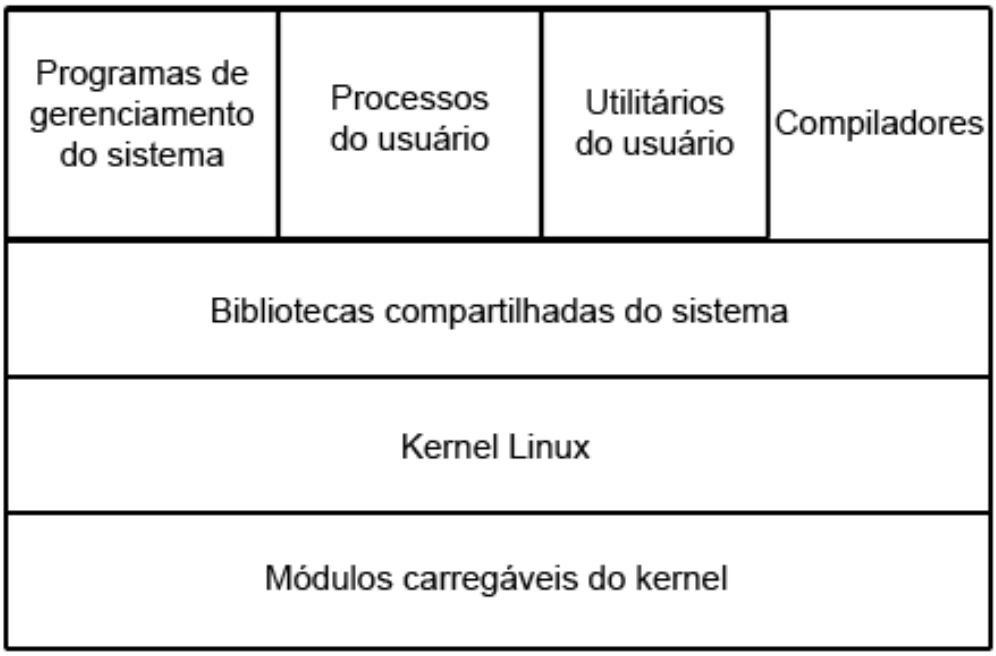

Figura 1 - Componentes do sistema Linux (adaptada de Silberchatz, et al., 2005, p. 743)

A interação dos aplicativos com o *kernel* ocorre por meio de chamadas de sistema (*system calls*). O aplicativo aciona funções de uma biblioteca, por exemplo, a biblioteca C, e esta, por sua vez, ao necessitar da intervenção do *kernel*, faz uma chamada de sistema. Os manipuladores de interrupção também são parte importante do sistema, utilizados pelo *kernel* para verificar e prover o tratamento adequado aos diferentes tipos de *hardware* existentes. As

interrupções são a forma de comunicação do *hardware* com o sistema operacional. (LOVE, 2010)

Segundo Silberchatz, et al. (2005, p.779): "O *kernel* Linux é implementado como um *kernel* monolítico tradicional por razões de desempenho, porém, é modular o bastante em seu projeto para permitir à maioria dos *drivers* serem carregados e descarregados dinamicamente em tempo de execução".

As diferenças entre o *kernel* Linux e os sistemas UNIX também são listadas por Love (2010, p.8):

> - Linux suporta o carregamento dinâmico de módulos do *kernel*. [..]
> - O Linux tem suporte a multiprocessador simétrico (SMP). Embora a maioria das variantes comerciais do Unix agora suportem SMP, as implementações mais tradicionais do Unix não suportam.
> - O *kernel* Linux é preemptivo. [...].
> - Linux tem uma abordagem interessante de suporte a *threads*: Não faz distinção entre *threads* e processos normais. [...]
> - Linux oferece um modelo de dispositivo orientado a objetos com classes de dispositivos, eventos que ocorrem sem a necessidade de reinício do sistema operacional [...].
>
> [...]
>
> - Linux é livre em todos os sentidos da palavra. O conjunto de recursos que o Linux implementa é o resultado da liberdade de modelo de desenvolvimento aberto do Linux. [...] o Linux tem adotado uma atitude elitista para mudanças: as modificações devem resolver um problema do mundo real específico, derivam de um design limpo, e ter uma implementação sólida.

2.4 Linux Embarcado

Embora seja o mesmo sistema operacional, o Linux embarcado tem algumas diferenças importantes em comparação com o Linux para PCs, ligadas principalmente à inicialização do sistema.

Segundo Raghavan, et al. (2006, p.1): "Um sistema embarcado é um sistema computacional de propósito específico que é projetado para executar conjuntos muito pequenos de atividades definidas [...]".

Um sistema embarcado é um sistema baseado em microprocessadores designado para uma tarefa específica. O objetivo de um sistema embarcado é controlar uma ou mais funcionalidades do sistema, ao contrário do que acontece com um PC, por exemplo, que tem

propósito geral. O usuário pode ajustar as funções realizadas, mas não pode mudar o propósito do sistema. (HEATH, 2003)

Raghavan, et al. (2006, p.1) apontam alguns fatores essenciais dos sistemas embarcados, como:

- Sistemas embarcados são geralmente sensíveis ao preço.
- A maioria dos sistemas embarcados tem restrições de tempo real
- Sistemas embarcados têm (e requerem) muito poucos recursos, em termos de memórias RAM, ROM ou outros dispositivos de entrada e saída, se comparado a um computador PC.
- O gerenciamento de energia é um aspecto importante na maioria dos sistemas embarcados.

Um PC não é em si um sistema de computação embarcada, mas é importante salientar que PCs podem ser utilizados para construir sistemas embarcados computacionais". (WOLF, 2008)

Segundo Raghavan, et al. (2006, p.1):

> Sistemas embarcados datam de cerca de 1960, onde foram utilizados para controle eletromecânico de centrais telefônicas. (…) Mais tarde, eles encontrariam seu espaço nas áreas militares, ciências médicas e nas indústrias aeroespaciais e automobilísticas. Hoje são amplamente utilizados para servir a diversos propósitos; seguem alguns exemplos:
>
> - Equipamentos de rede, como *firewalls*, roteadores, *switches* [...].
> - Equipamentos de consumo, como tocadores de MP3, telefones celulares, PDAs, câmeras digitais, gravadores de vídeo [...].
> - Eletrodomésticos como micro-ondas, máquinas de lavar, televisores, e assim por diante.
> - Sistemas de missão crítica, como satélites ou controladores de voo.

Nos primeiros sistemas embarcados, não eram usados sistemas operacionais. Os *software*s desenvolvidos apenas acionavam diretamente o *hardware*, com quase nenhuma interação com o usuário ou atividade multitarefa. Porém, com o tempo, mais funcionalidades foram sendo necessárias, fazendo as empresas iniciarem esforços a fim de construir sistemas operacionais para plataformas embarcadas. (RAGHAVAN, et al., 2006)

A qualidade e confiabilidade de código e o suporte de *hardware* são motivos para escolher o Linux como o sistema operacional para plataformas embarcadas. Qualidade e confiabilidade de código estão ligadas ao nível de confiança do *software* e englobam conceitos como estrutura, modularidade, legibilidade, extensibilidade, configurabilidade, previsibilidade, recuperação de falhas e longevidade (o programa deve conservar sua integridade por longos

períodos de tempo). Suporte de *hardware* significa que o Linux oferece suporte a diferentes tipos de plataformas de *hardware* e periféricos. Boa parte dos fabricantes ainda não fornece *drivers*, mas a comunidade Linux se encarrega de criar e manter estes *drivers*. É razoável esperar que, dada uma arquitetura, Linux execute nela ou alguém tenha feito um processo de portabilidade para ela. (YAGHMOUR, et al., 2008)

Pode-se destacar, além do suporte de *hardware*, o baixo custo e o código aberto. O custo de desenvolvimento é reduzido por geralmente não necessitar de pagamento de licenças ou ferramentas de desenvolvimento. No Linux embarcado, as bibliotecas e bons ambientes de desenvolvimento podem ser obtidos sem custo. O treinamento ou contratação de especialistas também é menor ou desnecessário, já que o modelo de programação UNIX é familiar a muitos engenheiros. Por fim, o Linux embarcado também é livre de *royalties* em termos de execução do sistema operacional embarcado. (RAGHAVAN, et al., 2006)

Algumas diferenças entre o Linux embarcado e o mesmo sistema operacional usado em PCs são listadas por Raghavan, et al. (2006, p.10):

> - O modo como o *kernel* Linux é configurado para sistemas embarcados difere do seu homólogo para *desktop*s. O conjunto de *drivers* de dispositivos que é necessário difere em ambos. Por exemplo, um sistema embarcado pode precisar de um *driver* para memória Flash e um sistema de arquivos para Flash (como CRAMFS ou JFFS2), enquanto estes não são necessários em um *desktop*.
> - Em Linux embarcado, há maior enfoque em ferramentas necessárias para desenvolvimento, depuração e monitoramento de recursos. O foco do Linux embarcado é em um conjunto de ferramentas de desenvolvimento cruzado que permitem aos desenvolvedores construir aplicações para seus dispositivos em ditos sistemas hospedeiros baseados em x86. Por outro lado, em Linux para *desktop*s é dado maior enfoque a um conjunto de ferramentas que são úteis para o usuário, como processadores de texto, gerenciadores de *e-mails*, leitores de notícias, e assim por diante.
> - Os utilitários que fazem parte de um Linux embarcado são diferentes dos similares em Linux para *desktop*s. *Ash*, *Tinylogin* e *Busybox* são considerados requisitos para usar com Linux embarcado. Mesmo as bibliotecas de aplicação uClib*c* são preferidas para aplicações embarcadas em contrapartida à sua similar para *desktop* Glibc. [...]
> - Dispositivos com Linux embarcado implantado, em sua maioria, executam em modo de único usuário com quase nenhuma capacidade de administração de sistema. Por outro lado, essa administração tem papel importante em *desktop*s.

De acordo com Yaghmour et al. (2008), quatro passos principais devem ser seguidos para criar um sistema Linux para determinado dispositivo: determinação dos componentes, configuração e compilação do *kernel*, montagem do sistema de arquivos e estabelecimento e configuração do *software* de *boot*. Normalmente, na estrutura do Linux embarcado, serviços de nível mais baixo executam funções dependentes da CPU ou da arquitetura presentes, por exemplo. Nos níveis mais altos, há componentes que fornecem as abstrações comuns aos sistemas UNIX, como processos e arquivos.

O processo de inicialização engloba os passos do sistema desde que ele inicia o processo de *boot* até a disponibilidade para o usuário (aparecimento da tela de autenticação, por exemplo). Conhecer o processo de inicialização é importante para o desenvolvimento, familiarizando-se com as peças mais básicas para a construção de um sistema Linux. (RAGHAVAN, et al., 2006)

Yaghmour et al. (2008, p.47) apresenta os componentes da inicialização do sistema (ilustrados na figura 2):

> [...] o *bootloader*, o *kernel*, e o processo *init*. O *bootloader* é o primeiro *software* a executar na inicialização e é altamente dependente do *hardware* do dispositivo final. [...] O *bootloader* executa uma inicialização de *hardware* de nível mais reduzido e então salta para o código de inicialização do *kernel*.

A fase de inicialização do *kernel* é responsável por fazer a inicialização específica para a plataforma utilizada, carrega os subsistemas do *kernel*, ativa a execução multitarefa e monta a raiz do sistema de arquivos. Logo após, é feita a inicialização do espaço de usuário (processo *init*), carrega os serviços do sistema operacional, inicializa os componentes de rede e então mostra a tela de autenticação do usuário. (RAGHAVAN, et al., 2006)

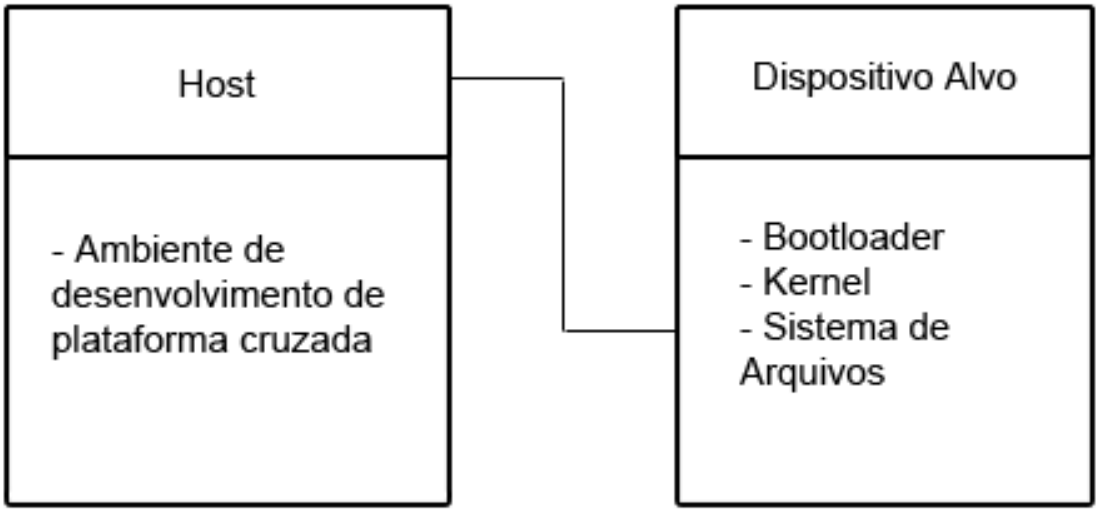

Figura 2 - Componentes do processo (adaptada de Yaghmour et al., 2008, p.39)

A forma com que estes componentes do sistema se organizam na memória é exemplificada pela figura 3.

2.5 *Bootloader*

O *bootloader* é o primeiro *software* que é carregado na inicialização. Dele é a responsabilidade de identificar o sistema e ajustar os parâmetros necessários para o funcionamento do sistema operacional.

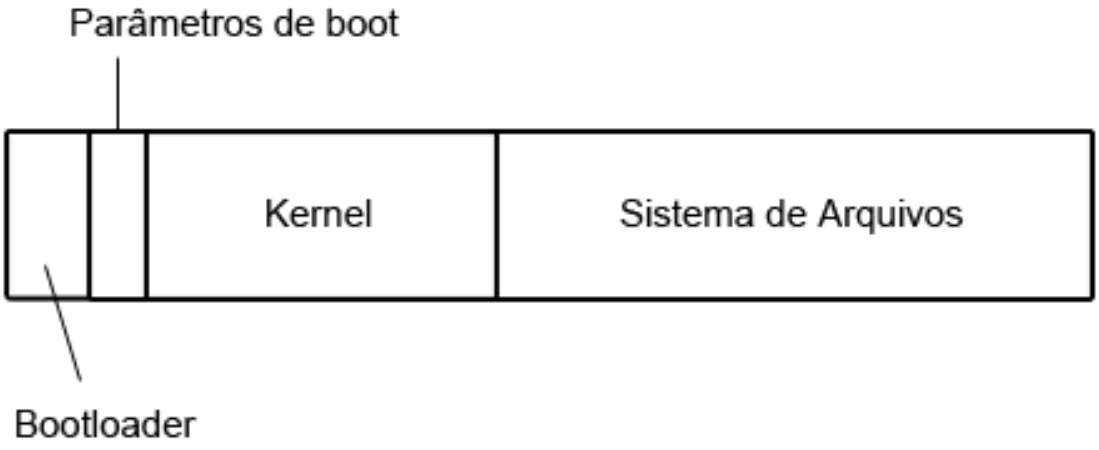

Figura 3 - Organização típica de um dispositivo de armazenamento (adaptado de Yaghmour et al., 2008, p.49)

Halinan (2006, p. 12) define com mais detalhes as funções do *bootloader*:

> - Inicializa componentes críticos de *hardware*, como o controlador da SDRAM, de entrada e saída e de componentes gráficos.
> - Inicializa a memória do sistema em preparação para a passagem do controle para o sistema operacional.
> - Aloca os recursos do sistema como memória e circuitos de interrupção para controladores periféricos, conforme necessário.
> - Fornece um mecanismo para localização e carregamento da [...] imagem do sistema operacional.
> - Carrega e entrega o controle para o sistema operacional, passando qualquer informação de inicialização que possa ser requerida, como tamanho total da memória, taxas do relógio do sistema, velocidade de portas seriais e outros dados específicos de configuração de *hardware* a nível mais baixo.

. O *bootloader* utilizado no projeto foi o U-Boot. Criado por Wolfgang Denk, da empresa *DENX Software Engeneering*, este *bootloader* suporta a maioria das plataformas de desenvolvimento; oferecendo opções de *boot* usando TFTP através de uma conexão de rede, discos IDE ou SCSI, USB ou uma variedade dispositivos Flash. O U-Boot pode ser considerado o mais rico, o mais flexível e o mais ativamente desenvolvido *bootloader* embarcado de código aberto disponível. U-Boot suporta vários sistemas de arquivos, como CRAMFS, EXT2, JFFS2[3] ou FAT[4] e, apesar de ter um conjunto grande de comandos, que são configuráveis, apresenta uma boa documentação. Nas plataformas de desenvolvimento atuais com suporte a Linux, há uma grande chance do U-Boot ser o *bootloader* escolhido. (YAGHMOUR, et al., 2008)

Halinan (2006) ressalta que o U-Boot tem suporte para várias arquiteturas e possui um grande número de desenvolvedores de sistemas embarcados e fabricantes de *hardware* que o adotaram para uso em seus projetos.

---

[3] Mais informações sobre o sistema de arquivos na seção 3.7.

[4] Sistema de arquivos cuja principal característica é a presença de uma tabela que reúne os blocos e informações sobre a disponibilidade deles para uso pelo sistema operacional.

2.6 Montagem do *kernel*

O processo de construção da imagem do *kernel* requer algumas ferramentas e configurações específicas, de acordo com a plataforma de desenvolvimento e os periféricos utilizados.

Um dos primeiros passos ao construir um Linux embarcado é configurar as *toolchains* para montar o *kernel* e as aplicações. Este *toolchain* é conhecido como *toolchain* de plataforma cruzada (*cross-platform toolchain*). Essa técnica é usada pelo fato de muitos sistemas embarcados não possuem memória ou espaço em disco suficiente para abrigar as ferramentas necessárias. (RAGHAVAN, et al., 2006)

Yaghmour et al. (2008) explica que um *toolchain* é um conjunto de ferramentas necessárias para construir o *software* do computador. Incluem alguns componentes, como *linker*, *assembler* e compilador (na maioria dos casos, de linguagem C). Também define um *toolchain* de plataforma cruzada como *software*s que são feitas para rodar em uma plataforma, porém criam programas para outra.

Raghavan, et al. (2006, p.49) também apresentam as partes que compõem um *toolchain* de plataforma cruzada:

- *Binutils*: *Binutils* é um conjunto de programas necessários para compilação/*linking*/montagem e outras operações de depuração.
- Compilador GNU C: O compilador C básico usado para gerar código objeto (tanto *kernel* quanto aplicações).
- Biblioteca GNU C: Essa biblioteca implementa as APIs das chamadas de sistema como *open*, *read* e assim por diante, e outras funções de suporte. Todas as aplicações que são desenvolvidas precisam sofrer *linking* em comparação a esta biblioteca base.

O sistema de montagem do *kernel* (*kbuild*) é bastante e simples em termos de customização e é baseado no comando *make*. É feito em quatro etapas. Primeiro é configurado o ambiente de desenvolvimento cruzado, no qual são fornecidas informações ao processo de montagem acerca da plataforma para a qual a imagem do *kernel* e os módulos estão sendo construídos. Após esta fase, inicia-se a segunda etapa (mostrada na figura 4), o processo de configuração, selecionando que componentes (escolha do processador, *drivers*, opções do *kernel*) serão incorporados ao sistema, ou construídos como módulos. O resultado é armazenado em arquivos. A terceira etapa é a montagem dos arquivos objeto e o *linking* deles para construir a imagem do *kernel*. Por fim, a última fase é a montagem dinâmica de módulos carregáveis. (RAGHAVAN, et al., 2006)

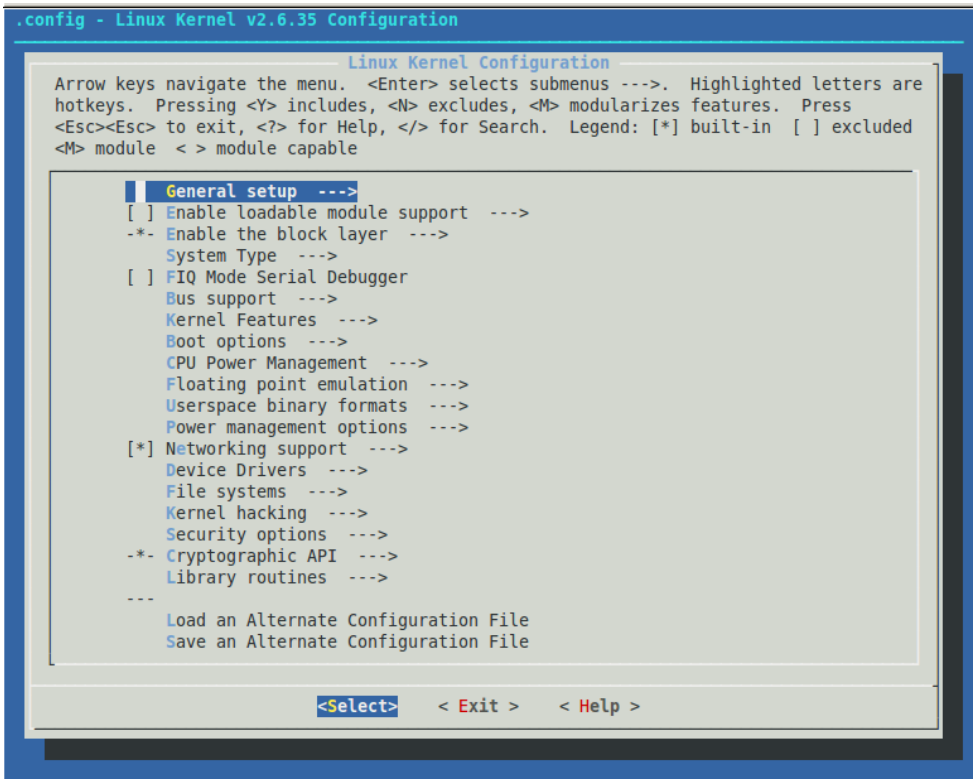

Figura 4 - Tela de configuração do kernel

Yaghmour et al. (2008) ressalta que:

> [...] Independente do método de configuração usado ou das opções de configurações escolhidas no momento, o *kernel* irá gerar um arquivo *.config* ao fim da configuração e irá gerar alguns *links* simbólicos e cabeçalhos de arquivos que serão usados pelo restante do processo de montagem.

2.7 Sistemas de arquivos

Os sistemas de arquivos são parte fundamental para o funcionamento de um sistema operacional, pois concentram os aplicativos, principal forma de interação entre o usuário e o *software* embarcado.

Algumas das formas de disponibilizar os sistemas de arquivos suportados pelo U-Boot são *Ramdisk*, CRAMFS, JFFS (ou JFFS2), e NFS. *Ramdisk* é um disco rígido emulado pelo Linux na memória e deve ser usado apenas quando não se dispõe de um dispositivo de armazenamento. CRAMFS é um sistema para armazenamento em Flash comprimido (usa *zlib*) e "somente leitura". NFS é usado para montar um sistema de arquivos pela rede, usando um PC para guardar o sistema EXT2 ou EXT3 e acessá-lo. É mais indicado para desenvolvimento e não tem tantas restrições de tamanho. (RAGHAVAN, et al., 2006)

Sistemas de arquivos EXT2 permitem escritas e são persistentes, feitos para dispositivos separados em blocos; porém, não suportam compressão ou confiabilidade em caso de queda de energia. (YAGHMOUR, et al., 2008)

Alguns sistemas são criados para memórias Flash. Heath (2003, p.79) explica que:

> Memória Flash é uma memória não volátil que é apagada eletricamente e oferece tempos e acesso e densidades similares às da DRAM. [...] Flash tem se posicionado e ganhado terreno como uma substituta das EPROMs, mas não obteve sucesso ao substituir discos rígidos como forma de armazenamento em massa de propósito geral. Isto é devido aos avanços na tecnologia de discos, o tempo de escrita relativamente lento e o mecanismo de degradação com excesso de uso que limita o número de escritas que podem ser realizadas.

Arquivos pequenos são um problema para o sistema de arquivos em Flash, pois esta deve ser apagada de um bloco inteiro por vez, o que pode comportar muitos desses arquivos. A memória Flash deve reescrever os arquivos restantes do bloco todo em outro bloco vazio para apagar o bloco anterior, o que pode consumir bastante tempo. Por essas escritas serem lentas, aumenta o risco de dados serem corrompidos por repentinas perdas de energia, comuns em sistemas embarcados. (HALINAN, 2006)

A Axis Communications apresentou o JFFS para o *kernel* Linux 2.0 em 1999. A falta de suporte à compressão iniciou o projeto JFFS2, lançado para a versão 2.4 do *kernel*. Em tanto JFFS, como JFFS2, uma mudança a um arquivo é gravada como um *log* (nó), que é diretamente armazenado na memória Flash. Numa escrita, um *log* guarda o deslocamento no arquivo para onde os dados estão escritos e o tamanho dos dados escritos junto com os dados atuais. Na leitura, os *logs* são reproduzidos e o arquivo é recriado. Com o tempo, alguns logs tornam-se obsoletos parcialmente ou totalmente, sendo capturados pelo 'coletor de lixo' (*garbage collector*), que identifica e apaga esses blocos, colocando na lista de blocos disponíveis. (RAGHAVAN, et al., 2006)

JFFS foi feito conhecendo-se a arquitetura e as limitações das memórias Flash. JFFS2 também incorpora uma técnica de nivelamento de uso dos blocos disponíveis, prolongando a vida útil do dispositivo. (HALINAN, 2006)

Raghavan, et al. (2006, p.118) mostram com mais detalhes as principais características do JFFS2:

> Gerenciamento de blocos de apagamento: No JFFS2, os blocos de apagamento estão em três listas: limpos, sujos e livres. A lista de limpos contém apenas *logs* válidos [...]. A lista de sujos contém um ou mais *logs* que estão obsoletos e, portanto, podem ser apagados quando o coletor de lixo for

> chamado. A lista de livres não contém *logs* e, portanto, pode ser utilizada para armazenar novos *logs*.
>
> 'Coleta de lixo': A 'coleta de lixo' no JFFS2 ocorre no contexto de um *thread* separado, que inicia quando um sistema de arquivos JFFS2 é montado. Para cada 99 de 100 vezes este *thread* pegará um bloco da lista de sujos e para a outra vez das 100, pegará um da lista de limpos para garantir balanceamento de uso. JFFS2 reserva cinco blocos para fazer coleta de lixo [...].
>
> Compressão: O fator de distinção entre JFFS e JFFS2 é que JFFS2 fornece uma série de compressões, incluindo *zlib* e *rubin*.

Outro importante sistema de arquivos é o YAFFS2. De acordo com Yaghmour et al. (2008), o *Yet Another Flash File System*, versão 2, é um sistema de arquivos para dispositivos Flash que permite escrita, persistente e confiável em termos de queda de energia. Tem nivelamento de uso do dispositivo e é otimizado para dispositivos NAND Flash. Porém, YAFFS2 requer a obtenção de códigos para atualização do *kernel*, por não fazer parte da árvore principal do *kernel* Linux.

As justificativas principais para o uso do YAFFS2 é o menor uso de memória RAM em tempo de execução e um algoritmo mais rápido de 'coleta de lixo' em comparação ao JFFS2. Por outro lado, YAFFS não suporta compressão, sendo o JFFS2 mais adequado para sistemas com partições pequenas de memória Flash.

2.8 Microprocessadores

Sistemas embarcados são embasados em plataformas microprocessadas. Os microprocessadores são designados para importantes funções, provendo ao sistema a execução das operações com alto desempenho.

Wolf (2008) define um microprocessador como uma CPU de *chip* único. Microprocessadores apresentam vários níveis de sofisticação, geralmente classificados pelo tamanho da palavra. Os de palavra de oito *bits* são usados em aplicações de baixo custo e incluem memória interna e dispositivos de entrada e saída; já os de 16 *bits* são para aplicações mais complexas, que necessitem de memória ou dispositivos de entrada e saída externos. Por fim, os de 32 *bits* RISC oferecem alto desempenho para aplicações de computação intensa. Os microprocessadores provêm aos sistemas embarcados exercer suas funcionalidades, como execução de algoritmos complexos e apresentação de interfaces de usuário, sem esquecer os compromissos com requisitos de tempo real, comportar serviços que necessitam de diferentes prazos e manter os custos de produção reduzidos.

2.9 Microprocessadores ARM

ARM é uma família de microprocessadores amplamente empregada atualmente em dispositivos embarcados. Sua estrutura agrega *pipelines* e um número reduzido de instruções, o que permite uma alta frequência de *clock*, melhorando a velocidade de execução.

Yaghmour et al. (2008, p.57) define:

> ARM, sigla para *Advanced* RISC *Machine*, é uma família de processadores mantida e promovida pela ARM *Holdings Ltd*. Em contraste com outras fabricantes de chips como IBM, *Freescale* e *Intel*, ARM *Holdings* não fabrica seus próprios processadores. Ao invés disso, ela projeta núcleos de CPU para seus consumidores baseado no núcleo ARM, cobrando dos consumidores taxas de licenciamento do projeto, e permite a eles fabricarem o *chip* onde eles acharem adequado. Isso oferece muitas vantagens às partes envolvidas, porém cria certa confusão ao desenvolvedor na que aborda essa tecnologia pela primeira vez, por não ter um produtor central de *chips* ARM. [...]

As versões são diferentes dos processadores ARM indicadas por números, porém essas alterações são invisíveis ao programador *assembly*, exceto por questões de desempenho, por compartilharem um mesmo conjunto básico de instruções. Microprocessadores ARM padrão tem palavras de 32 *bits* que podem ser configurados na inicialização tanto no modo *little-endian* (o *byte* de menor ordem é o menos significativo) como *big-endian* (o *byte* de menor ordem é o mais significativo). O funcionamento do processador é do tipo *load-store*, ou seja, os operandos precisam primeiro ser carregados para a realização da operação, e então são armazenados de volta na memória. (WOLF, 2008)

Embora todas as ferramentas de desenvolvimento e montagem necessárias para sistemas embarcados Linux gratuitamente disponíveis, as tarefas de integração, montagem e teste requerem tempo e vem com uma curva de aprendizado para o recém-iniciado desenvolvedor de Linux embarcado. (YAGHMOUR, et al., 2008)

O núcleo ARM usa um conjunto de instruções RISC, focado em desenvolver instruções simples e poderosas, que executem em um ciclo apenas com alta velocidade de *clock*. RISC transfere a complexidade para o *software* (compilador). Além do conjunto de instruções reduzido, estas também podem ser quebradas e executadas em *pipelines*. (SLOSS, et al., 2004)

Instruções RISC tornam o processador mais simples, trazendo vantagens como menor área de superfície, tempo de desenvolvimento reduzido e maior desempenho. Em contrapartida, RISC tem densidade de código pior se comparado ao CISC e também não consegue executar código x86. (FURBER, 2000)

ARM é apenas uma família de microprocessadores, podendo apresentar diferentes arquiteturas, de acordo com as características da versão sob a qual foram desenvolvidos. Microprocessadores ARM7, por exemplo, têm arquitetura do tipo Von Neumann, enquanto ARM9 e ARM11 possuem arquitetura do tipo Harvard.

O núcleo ARM não é constituído puramente com instruções do tipo RISC por causa dos sistemas embarcados, primando não só velocidade, como também desempenho efetivo e baixo consumo de energia. As características que fazem o processador ARM não ser considerado RISC puro são: o ciclo de instrução variável para algumas instruções (algumas instruções de carregamento e armazenamento múltiplos variam no número de ciclos de execução), *inline barrel shifter* (*hardware* que pré-processa um dos registradores de entrada, leva a instruções mais complexas), conjunto de instruções de 16 *bits* (*Thumb*, aumenta a densidade de código), execução condicional e instruções aprimoradas (substituindo algumas funções de DSPs). (SLOSS, et al., 2004)

Ainda de acordo com Furber (2000, p.41) há três tipos de instruções ARM:

- Instruções de processamento de dados. Estas usam e mudam apenas valores de registradores. Por exemplo, uma instrução pode somar dois registradores e colocar o resultado em um registrador.
- Instruções de transferência de dados. Estas copiam valores de memória para registradores (instruções de carregamento) ou copiam valores de registradores para a memória (instruções de armazenamento). [...]
- Instruções de fluxo de controle. A execução normal de instruções utiliza as instruções armazenadas em endereços de memória consecutivos. Instruções de fluxo de controle causam uma mudança da execução para um endereço diferente, seja permanentemente (instruções de desvio) ou salvando o endereço de retorno para continuar a sequência original (instruções de desvio e *link*) ou prender no código do sistema (chamadas de supervisor).

2.10 SAM9-L9620

A plataforma de desenvolvimento utilizada para o processo de portabilidade é o *kit* SAM9-L9260 da Olimex, ilustrado na figura 5. Esta plataforma já é própria para elaboração de aplicações envolvendo sistemas embarcados, embora tenha configurações limitadas em comparação com as novas tecnologias existentes.

SAM9-L9260 é uma plataforma de desenvolvimento de baixo custo com microcontrolador ARM9, 64MB de SDRAM, 2 MB de DATA Flash e 512 MB de NAND Flash. A plataforma executa Linux, WindowsCE e outros RTOS nativamente. (OLIMEX, 2009)

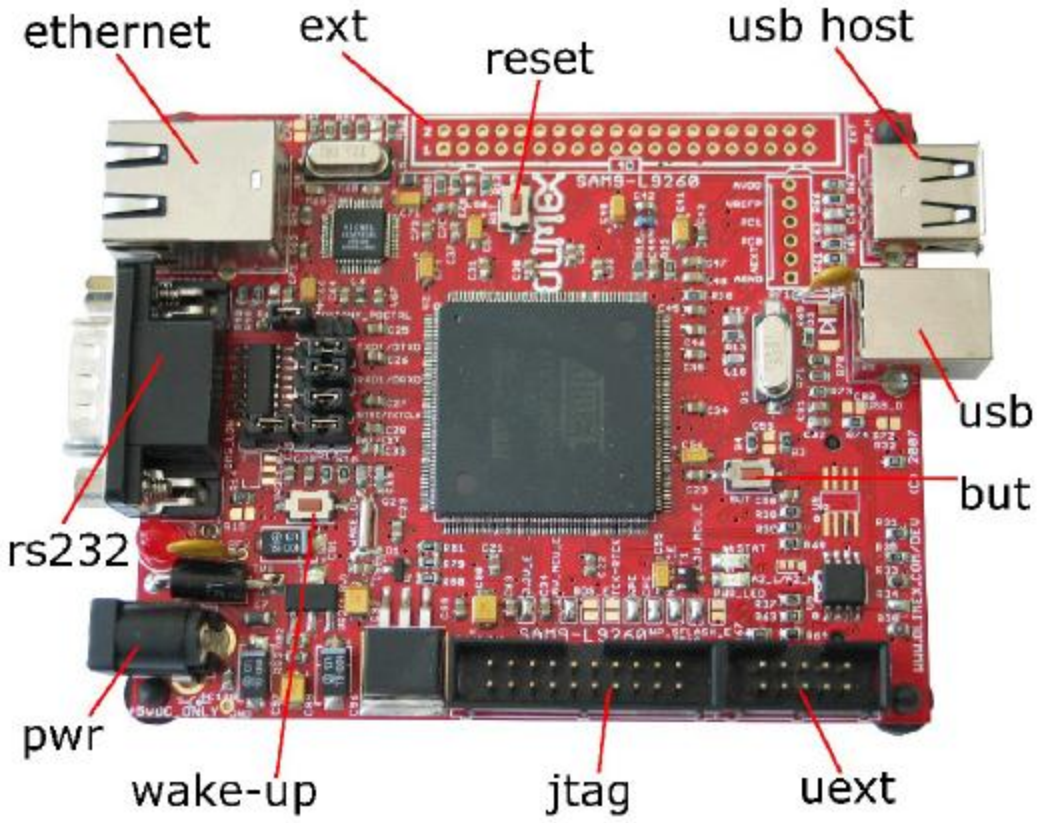

Figura 5 - Placa de desenvolvimento SAM9-L9260 (retirado de Olimex, 2009, p.2)

SAM9260 tem como base o processador ARM926EJ-S, integrado com memórias RAM e ROM, MAC Ethernet, porta de dispositivos USB, controlador host USB, USART, SPI, contadores/temporizadores e interfaces para cartão de memória. O processador ARM926EJ-S faz parte da família ARM9. Funciona em três estados: ARM (32 *bits*), *Thumb* (16 *bits*) e Jazelle (tamanho variável, usado para execução eficiente de aplicações Java). O processador contém 37 registradores de 32 *bits* cada, sendo 31 de propósitos gerais e seis registradores de *status*. Utiliza *pipeline* de cinco estágios, como mostrado na figura 6, para ARM e *Thumb* (seis estágios para Jazelle – decodificação leva dois estágios); o acesso à memória é feito por *byte*, meia palavra e palavra inteira (32 *bits*). (ATMEL, 2011)

Furber (2000, p.79) cita os estágios do pipeline e suas funções:

> - Busca. A instrução é buscada da memória e colocada no pipeline de instruções.
> - Decodificação. A instrução é decodificada e os registradores de operandos lidos do banco de registradores. [...]

- Execução. Um operando é deslocado e o resultado da ULA gerado. Se a instrução for de carregamento ou armazenamento, o endereço da memória é computado na ULA.
- *Buffer*/dados. A memória de dados é acessada, se preciso. Caso contrário a ULA é simplesmente armazenada por um ciclo de *clock* para dar o mesmo fluxo de *pipeline* a todas as instruções.
- Escrita. Os resultados gerados pela instrução são escritos de volta ao banco de registradores, incluindo qualquer dado carregado da memória.

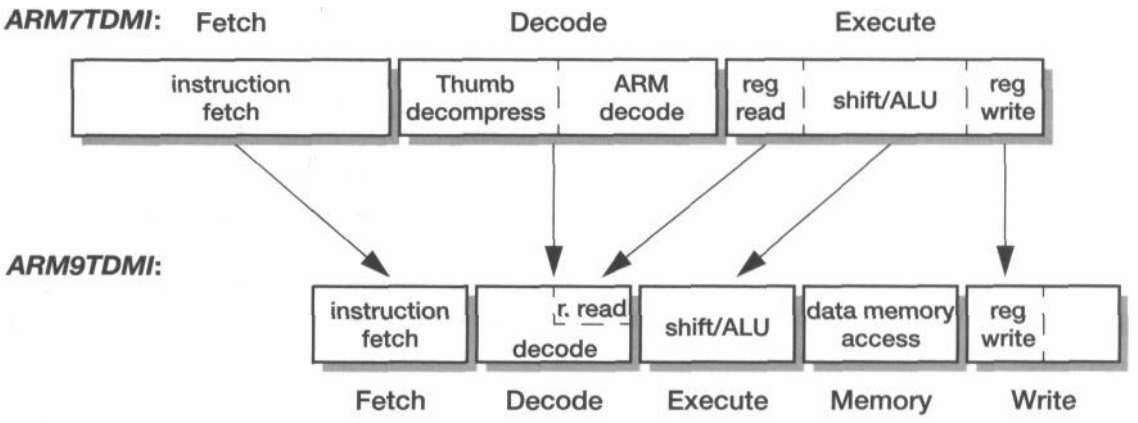

Figura 6 - Pipeline de cinco estágios do ARM9 (retirado de Furber, 2000, p.261)

As instruções presentes no processador ARM926EJ-S são dos tipos: desvio, processamento de dados, transferência de registrador *status*, carregamento/armazenamento, coprocessador e geradoras de exceção. (ATMEL, 2011)

3. METODOLOGIA

O trabalho consiste na aplicação das técnicas de portabilidade de sistemas operacionais disponíveis para transporte do Android em uma plataforma de desenvolvimento SAM9-L9260. O roteiro utilizado foi desenvolvido em meio a pesquisas acerca da comunidade de desenvolvimento do Android, levantando informações que auxiliem na decisão das etapas mais adequadas para a realização do projeto.

A análise resultou na elaboração das seguintes etapas:

- Obtenção e configuração do sistema Android
- Obtenção e configuração do *compilador cruzado*
- Construção da imagem do *kernel* Android
- Obtenção do sistema de arquivos
- Configuração do U-Boot

Na etapa inicial, foi feita uma busca nos repositórios existentes, identificando qual dos códigos fonte encontrados era o mais adequado para o processo. Também foi escolhido o melhor conjunto de configurações e *drivers* para a compilação do *kernel*.

Nova pesquisa foi realizada, analisando os compiladores cruzados disponíveis. A presença de um *compilador cruzado* compatível é essencial para a geração de código para a plataforma ARM. A construção da imagem foi feita, identificando os obstáculos presentes. Em várias ocasiões, houve a necessidade de reconfiguração do código, para contornar problemas encontrados durante o processo de montagem.

O sistema de arquivos, por sua vez, pode ser obtido extraindo os dados do emulador Android, disponibilizado junto de seu ambiente de desenvolvimento. Há algumas formas diferentes de obtenção dos arquivos, cada uma com suas restrições e dificuldades de execução.

Por fim, a configuração do U-Boot engloba a avaliação do conjunto de opções oferecidas, que é bastante numeroso. A escolha da forma de inicialização do sistema leva em consideração as limitações tanto do sistema operacional, quanto do sistema embarcado. Este último apresenta inúmeras restrições relacionadas com seus componentes.

No caso do SAM9-L9260, a utilização do U-Boot deve conviver com a escassez (ou incompatibilidade) de memória Flash para inicialização do sistema, além da quantidade reduzida de memória RAM. Por fim, a incompatibilidade do Android com o sistema de arquivos

do U-Boot também exige a adoção de diferentes abordagens, explorando as funcionalidades disponibilizadas pelo U-Boot.

4. RESULTADOS

A portabilidade de um sistema operacional para determinada plataforma embarcada, embora existam variadas ferramentas de auxílio ao desenvolvedor, não é tarefa simples. Muitos ainda são os erros e incompatibilidades existentes, dificultando os processos de construção e execução do sistema. Os obstáculos se estendem desde a preparação do *kernel* Android até a compatibilidade do sistema com o *hardware*.

Primeiramente, foi feita uma análise das fontes de código existentes na rede. A obtenção do código Android pode ser feita por meio do repositório fornecido pelo Google. Da mesma forma, também se encontra disponível o código fonte apenas do *kernel* do Android.

Para *download* do código fonte completo devem ser utilizados os seguintes comandos[5] (em Linux), assegurando-se da existência do pacote Git instalado:

```
mkdir ~/bin
PATH = ~/bin:$PATH
chmod a+x ~/bin/repo
mkdir WORKING_DIR
cd WORKING_DIR
repo init –u https://android.googlesource.com/platform/manifest/
repo sync
```

Todavia, há algumas desvantagens em obter o código fonte completo, como a ocupação de uma quantidade grande de espaço em disco, chegando a ocupar mais de 4GB, o que acarreta também um maior tempo de transferência dos arquivos pela rede.

A alternativa para desenvolvimento de sistemas operacionais embarcados é o *download* apenas do *kernel* do sistema, suficiente para as próximas etapas do desenvolvimento, já que alterações de aplicativos voltados ao espaço de usuário não fazem parte do processo básico de portabilidade de um sistema operacional. O *kernel* Android ocupa uma quantidade bem menor de espaço em disco (no máximo 2GB) e, por isso, seu *download* também leva menos tempo. Os comandos utilizados para a obtenção do código fonte apenas do *kernel* foram:

```
cd MY_FOLDER
git clone git://android.git.kernel.org/kernel/common.git android-kernel
```

[5] Instruções em http://source.android.com/source/downloading.html

Com a obtenção do código concluída, é necessário que o ambiente de compilação do *kernel* seja preparado. A plataforma embarcada possui processador ARM, o que significa que o ambiente de desenvolvimento padrão contido em um Linux para PCs, de arquitetura *x86*, não é adequado para gerar uma imagem de *kernel* compatível com o sistema final. Dessa forma, deve-se encontrar um *toolchain* de compilação cruzada (*cross compiler*) que faça a conversão do código do *kernel* para a arquitetura do dispositivo alvo.

O código fonte do sistema Android completo já inclui em seus arquivos um *toolchain* do compilador cruzado ARM-EABI[6] pré-montado. Por este motivo, este foi o compilador escolhido para a montagem do *kernel* Android.

A utilização do compilador cruzado é simples. Os arquivos estão localizados dentro do diretório de extração do *toolchain*, na pasta *bin*. Para configurar o código fonte, basta alterar o arquivo *Makefile*, presente na pasta com o código do *kernel*. Devem-se procurar as linhas das variáveis ARCH e CROSS_COMPILE e alterá-las para:

```
ARCH ?= arm
CROSS_COMPILE ?= PASTA_DE_EXTRAÇÃO/bin/arm-none-eabi-
```

Desta forma, todas as ferramentas usadas na compilação irão gerar resultados próprios para execução em plataforma ARM. Também é possível utilizar as ferramentas encontradas no código fonte completo do sistema, dentro da pasta de extração do código fonte, em *prebuilt/linux-x86/toolchain/arm-eabi-VERSION/bin*.

Porém, ao aplicar o *toolchain* fornecido no código completo no código apenas relativo ao *kernel*, o primeiro apresentou alguns problemas no processo de montagem. Os erros encontrados eram vinculados ao processo de integração do código objeto gerado pelo compilador para formar a imagem do *kernel*. A correção destes erros dependia de um amplo conhecimento de todo o código fonte, que é bastante extenso, inviabilizando assim o uso para esse determinado cenário. Deve-se ressaltar que estas incompatibilidades também são relacionadas às características do código fonte, podendo funcionar corretamente em variações de código obtidas de outro repositório, por exemplo.

Com o ARM-EABI obtido diretamente da página da rede também houve problemas no processo de compilação. Porém, foram de fácil solução ou relacionados a certos *drivers* que não são essenciais para o funcionamento do sistema, podendo ser retirados sem prejudicar a portabilidade do sistema operacional.

---

[6] Disponível em https://sourcery.mentor.com/sgpp/lite/arm/portal/subscription3053.

O código fonte do *kernel* obtido apresenta erros, alguns até de digitação. Não é incomum encontrar, em algum repositório, uma versão do *kernel* com o erro mostrado na figura 7. Em contrapartida, há um número significativo de desenvolvedores que identificam esses problemas, os resolvem e disponibilizam as soluções para que outros desenvolvedores possam transpor esses obstáculos de forma mais rápida.

Os outros erros encontrados durante a compilação remetem a códigos de *drivers* que contém incompatibilidades com o *toolchain* ou com variáveis do próprio código fonte. Contudo, os códigos que apresentam esse tipo de erro pertencem a *drivers* que podem ser retirados do sistema. O *driver* pode ser retirado de duas formas, na tela de configuração do *kernel*, ou alterando manualmente o arquivo de configuração gerado.

cpuset.c

```
}

/* Protected by cgroup_lock */
static cpumask_var_t cpus_attach;

/* Called by cgroups to determine if a cpuset is usable; cgroup_mutex held */
static int cpuset_can_attach(struct cgroup_subsys *ss, struct cgroup *cont,
                             struct task_struct *tsk, bool threadgroup)
{
        int ret;
        struct cpuset *cs = cgroup_cs(cont);

        if ((current != task) && (!capable(CAP_SYS_ADMIN))) {
                const struct cred *cred = current_cred(), *tcred;

                if (cred->euid != tcred->uid && cred->euid != tcred->suid)
                        return -EPERM;
        }

        if (cpumask_empty(cs->cpus_allowed) || nodes_empty(cs->mems_allowed))
                return -ENOSPC;
```

Figura 7 - Erro de código (seria 'tsk', não 'task')

A configuração do *kernel*, exemplificada na figura 8, é feita abrindo um terminal, movendo-se até a pasta onde se encontram os arquivos extraídos com o código fonte e utilizando o comando *make menuconfig*. Desta forma, uma tela em forma de *menu* aparecerá, permitindo ao usuário escolher, separados em categorias, quais configurações e *drivers* estarão presentes na imagem do *kernel* resultante.

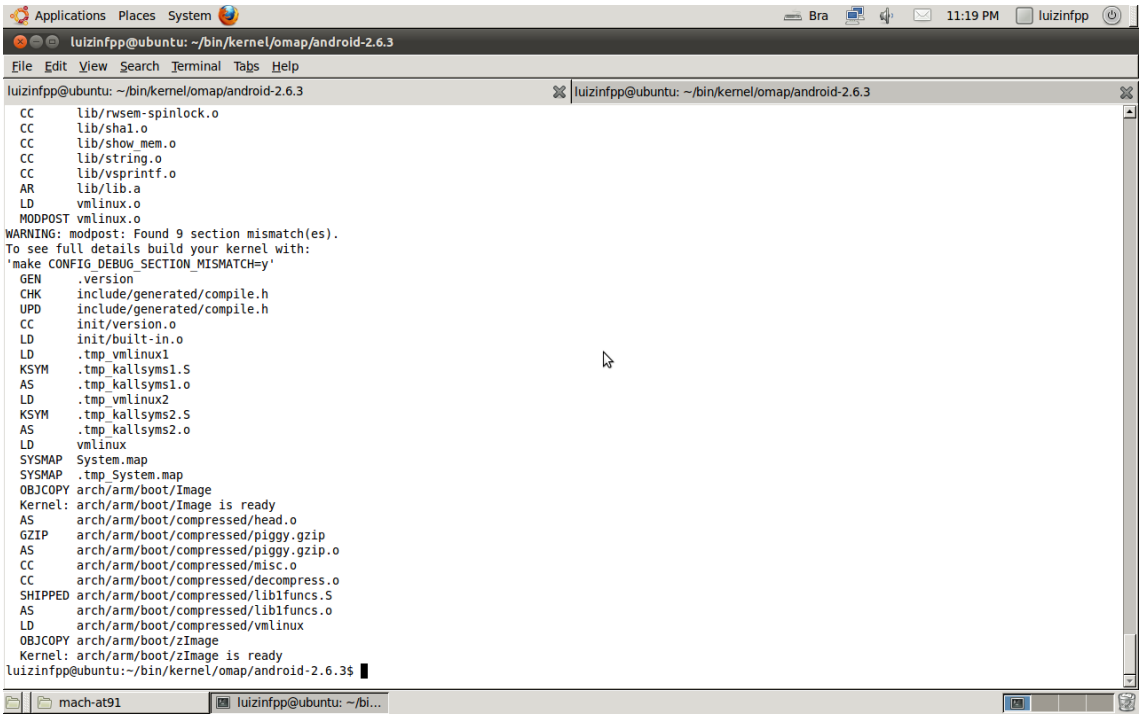

Figura 8 - Compilação do Android

No entanto, há uma forma mais simples de configurar o *kernel*, selecionando apenas as características necessárias para o funcionamento de cada plataforma. Utiliza-se o comando *make*, acompanhado de uma sequência de caracteres formados pelo nome da placa utilizada concatenada com o sufixo *'_defconfig'*. Desta forma, para a plataforma de desenvolvimento SAM9-L9260, o comando é *make sam9_l9260_defconfig*.

No caso da ocorrência de algum erro em determinado *driver* durante a compilação, é recomendado apenas desabilitar o *driver* em questão e prosseguir com o processo. Pode ser mais simples realizar esse procedimento de forma manual, procurando pelo nome do *driver* em questão no arquivo *'.config'* localizado na pasta com os arquivos do código fonte e editando o mesmo. A simples alteração do estado do *driver* de 'y' ou 'm' para 'n' já o retira da montagem do sistema.

Também é possível ocorrerem problemas no processo de incorporação dos módulos na imagem do *kernel* em questão. Nesse caso, o problema é resolvido apenas trocando a opção de montagem em módulos para a incorporação direta no *kernel*. Essa opção retira a flexibilidade obtida com o uso de módulos, porém esta propriedade, considerando o propósito de portabilidade de sistemas, apresenta-se supérflua.

Mesmo com a utilização da configuração própria para a plataforma de desenvolvimento SAM9-L9260, alguns erros foram encontrados. Por exemplo, em um dos códigos obtidos para o kernel, os *drivers* de vídeo DMM e TILER apresentaram problemas de compilação e precisaram ser retirados da montagem. O impacto disso é quase nulo, já que a placa não fornece interface

com dispositivos de vídeo. Há um erro também devido a uma incompatibilidade na formação da imagem, que é resolvida adicionando um *driver* que até então não seria incorporado, o SWITCH (através da edição do arquivo *'.config'*, retirando o comentário da linha CONFIG_SWITCH e adicionando '=y' ao fim da mesma, caso haja o pedido para incorporar mais *drivers* de SWITCH, os mesmos devem ser ativados). É importante lembrar que variantes de código ou do *toolchain* podem apresentar conjuntos de erros distintos.

Após as alterações, o processo alcança seu término com sucesso e duas imagens são geradas. Uma imagem de formato *Image* e outra comprimida, de formato *zImage*. É necessária uma conversão, já que, para a utilização no *bootloader* escolhido (U-Boot), a imagem deve estar em formato *uImage*.

A descompressão feita pelo U-Boot não foi bem sucedida quando da utilização de uma imagem *zImage* convertida para *uImage*. Por esse motivo, é aconselhável utilizar uma imagem descomprimida (*Image*) e convertê-la para formato *uImage* utilizando um aplicativo para Linux chamado *mkimage*, facilmente encontrado no repositório (usando *apt*, para usuários da distribuição Ubuntu).

O comando utilizado para fazer a conversão para um arquivo uImage foi:

```
mkimage –A arm –O linux –T kernel –C none –a 0x20008000 –e
0x20008000 –d Image uImage
```

Com a imagem do *kernel* já compilada e formatada adequadamente, a etapa seguinte consiste na implantação do sistema operacional no dispositivo alvo. Para este processo, será utilizada a estrutura fornecida pela Olimex para configuração e gravação dos dados para a plataforma SAM9-L9260.

No CD que acompanha a placa são fornecidos arquivos que correspondem às partes essenciais do *boot* de um sistema operacional: o *bootloader*, o *kernel* e a raiz do sistema de arquivos. Neste caso, o fabricante já fornece um *kernel* Linux adaptado para o desenvolvimento de aplicações embarcadas.

A execução dos comandos de gravação e configuração deve ser feita em ambiente Windows, após a instalação do *software* SAM-BA[7], responsável não só pelo fornecimento dos *drivers* necessários para comunicação do sistema com o PC, como pelo próprio processo de gravação dos arquivos e configurações nas memórias Flash da plataforma.

[7] Disponível em http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3883.

Entre os arquivos fornecidos no CD estão um arquivo com a imagem do *kernel* (*uImage*), um com o *bootloader* (U-Boot.bin), o sistema de arquivos (sam9-l9260-rootfs.jffs2), um arquivo de *batch* do Windows que coordena a execução do SAM-BA (at91sam9260_demo_linux_dataFlash.bat) e um arquivo contendo todas as informações de configuração para a gravação dos dados (at91sam9260_demo_linux_dataFlash.tcl).

Neste último arquivo citado, encontram-se informações importantes, como em qual dispositivo cada um dos arquivos presentes na pasta será gravado. Também indica o endereço de gravação e identifica os arquivos pelo nome, os associando às suas funções, facilitando a atribuição dos endereços.

A opção de *boot* padrão configurada para a placa utiliza a DATA Flash para armazenamento dos arquivos essenciais para o processo de inicialização. Há a opção de utilizar a NAND Flash para o mesmo propósito, porém, para esta plataforma especificamente, o carregamento do sistema desta última forma citada pode não funcionar, segundo o próprio fabricante.

Feita a opção de *boot* pela DATA Flash, devem estar contidos na mesma o U-Boot, um espaço para armazenar as variáveis iniciais utilizadas pelo U-Boot (também inseridas sob o controle dos comandos do arquivo '.tcl') e um aplicativo inicial de *boot* (aqui chamado de *bootloader*, corresponde a uma etapa anterior, entregando a execução para o U-Boot). Já na NAND Flash fica armazenado apenas o sistema de arquivos.

Inicialmente, a simples troca de um arquivo de *kernel* pelo outro poderia ser suficiente para ao menos fazer uma execução simples do sistema Android. Porém, o espaço destinado ao *kernel* na DATA Flash é bastante reduzido (menos de 2MB), o que não é suficiente para armazenar o *kernel* Android, mesmo estando desprovido de uma grande parte dos *drivers* que estão à disposição do desenvolvedor.

A alternativa para contornar a escassez de recursos, característica corriqueira em sistemas embarcados, foi explorar a multiplicidade de opções oferecidas pelo programa U-Boot. Ao conhecer as configurações e funcionalidades deste *software*, o entendimento e a criação de procedimentos de inicialização do sistema tornaram-se tarefas factíveis.

As configurações de ambiente do U-Boot, armazenadas na DATA Flash, revelam, por exemplo, que o *boot* ocorre na memória RAM. Independente do local onde o *kernel* esteja armazenado, ele deve ser primeiro carregado para a memória RAM e, após isso, é iniciado o processo de *boot* utilizando o código presente na memória RAM.

Também foi possível conhecer outras formas de obtenção da imagem do *kernel*, além das memórias Flash presentes no sistema, que não ofereciam problemas quanto à utilização de imagens de *kernel* com tamanho maior que 2MB. Os arquivos podem ser transferidos via NFS, TFTP ou por um dispositivo USB. O fato do U-Boot reconhecer o sistema de arquivos FAT

também contribuiu para que a forma de transferência do *kernel* escolhida fosse através de um dispositivo USB.

Com a transferência do arquivo pela USB, o tamanho do arquivo já não é tão crítico, pois elimina o intermediário limitante (DATA Flash), ao possibilitar a transferência direta da USB para a memória RAM. Permitindo o uso de todo o espaço da memória (64MB), o tamanho final da imagem do *kernel* (cerca de 4MB) não será mais problema para o processo de portabilidade.

Ao término deste procedimento, enfim o *kernel* estará pronto para execução no sistema. Ao realizar a operação de *boot* da memória RAM, o *kernel* reconhece o sistema e os periféricos e começa a execução. A portabilidade do *kernel* do sistema operacional está completada.

Porém, o sistema de arquivos do Android tem algumas diferenças em relação a um sistema Linux padrão, não só relacionados aos aplicativos Android exclusivos, mas também a organização do sistema é diferente. Por exemplo, alguns aplicativos comuns em Linux não estão presentes, como o 'cp'. Também a pasta *bin*, responsável por conter os aplicativos base do sistema, aparece dentro de uma pasta chamada *system*, diferentemente do sistema de arquivos básico Linux.

Assim, é necessário obter também o sistema de arquivos do Android, pois a adaptação de um sistema Linux disponível apresenta-se uma tarefa complicada. Todavia, o acesso a este sistema de arquivos não é tão simples como ao código *kernel*, por exemplo.

A forma mais simples de obtenção do sistema de arquivos é a extração por meio do emulador[8] Android, ilustrado na figura 9. O emulador (que se encontra na pasta de extração do SDK Android, dentro da pasta *tools*) oferece uma série de ferramentas, inicialmente projetadas para o desenvolvedor de *software* para Android.

Por outro lado, o emulador oferece bom suporte ao desenvolvedor de plataformas embarcadas, por permitir a emulação de um *kernel* de configuração personalizada ou também o acesso ao shell do sistema Android emulado (como na figura 10), estabelecendo uma comunicação, com possibilidade de transferência de arquivos, entre o Linux que hospeda o ambiente de desenvolvimento e o Android, podendo tanto empreender a instalação de *software*s no Android, como retirar dados do sistema emulado para o Linux.

[8] Disponível em http://developer.android.com/sdk/index.html

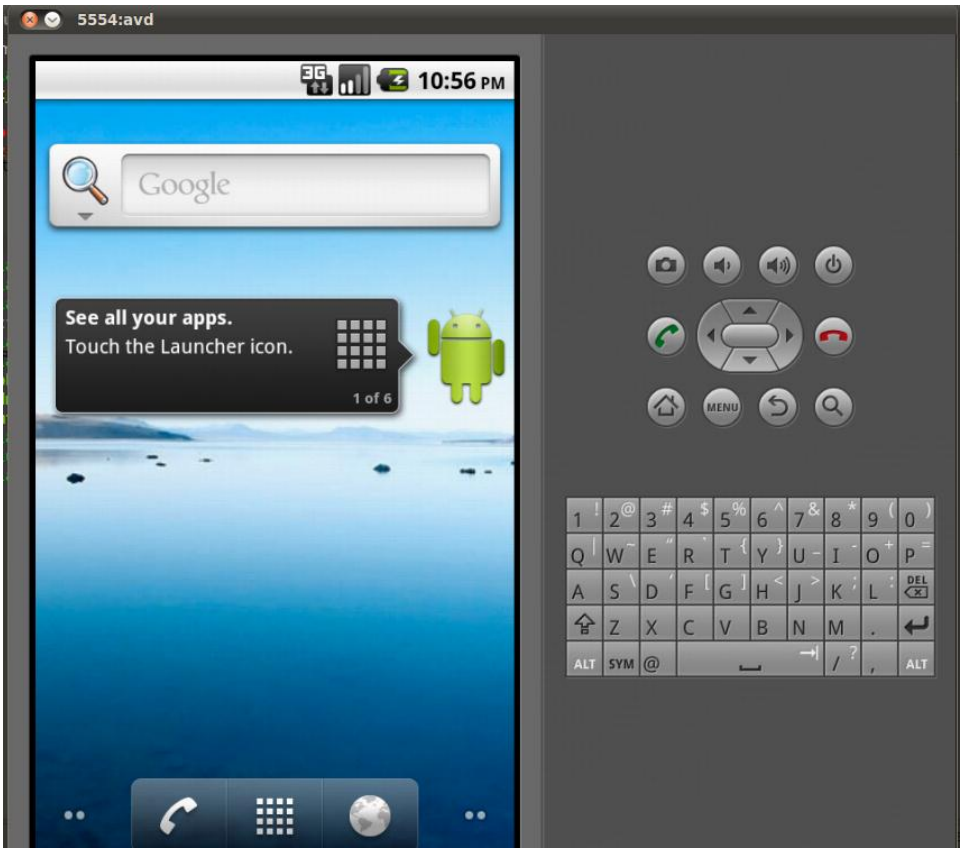

Figura 9 - Emulador Android

A transferência de arquivos pode seguir ambas as direções e é realizada pelo aplicativo ‘adb’, presente na pasta *platform-tools* do SDK. O sistema de arquivos é retirado em três partes. É utilizado como base da construção o arquivo *ramdisk.img*, localizado em *platform/android-VERSION/images*, dentro da pasta do SDK. A sequência de comandos necessária para extrair aos arquivos desta imagem é:

```
cp ramdisk.img ramdisk.gz
gunzip ramdisk.gz
cpio –iv < ../ramdisk
```

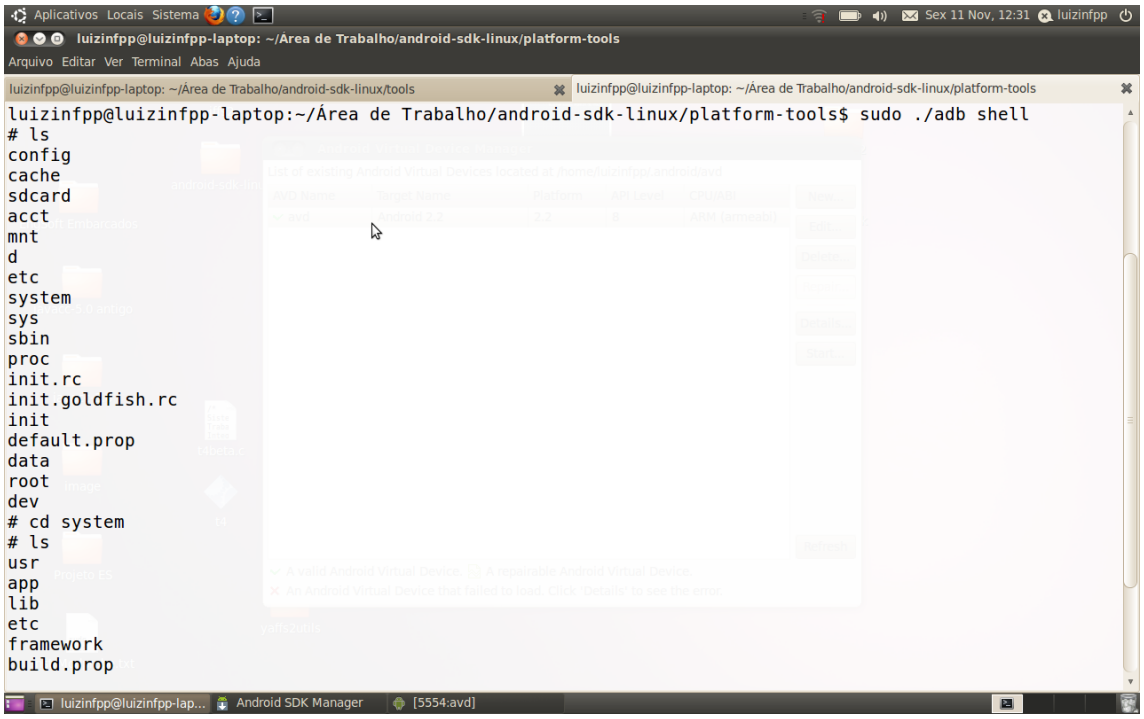

Figura 10 - Acesso ao shell do emulador

A base obtida do arquivo *ramdisk.img* corresponde ao sistema de arquivos do Android, sem o conteúdo das pastas *data* e *system*. Estes dois devem ser retirados do emulador utilizando o comando adb. Junto ao arquivo ramdisk.img existem dois outros arquivos system.img e userdata.img, entretanto estes são imagens YAFFS2, que não são descompactadas pelo programa fornecido pelos desenvolvedores do Android (*unyaffs2*). Aparentemente, o motivo seria que o *unyaffs2* consegue descompactar apenas arquivos gerados pelo programa *mkyaffs2*, consequentemente a utilização de outro método para a criação da imagem inviabilizaria o processo inverso. Um comando adb que pode ser utilizado para retirar as pastas do emulador é:

```
sudo ./adb pull FOLDER_AND [FOLDER_LINUX]
```

Com o sistema de arquivos pronto, resta apenas convertê-lo para YAFFS2, o sistema de arquivos adotado pelo Android. Para fazer a conversão do sistema para YAFFS2, deve-se usar o *software mkyaffs2*[9]. O comando utilizado é:

```
./mkyaffs2 FOLDER IMAGE.yaffs2
```

[9] Disponível em http://code.google.com/p/yaffs2utils/downloads/list.

O Android não reconhece o sistema de arquivos JFFS2. Na tentativa da montagem pelo *kernel*, o sistema de arquivos não é inicializado e retorna um erro.

Mesmo com todos os componentes disponíveis o sistema ainda não reconhece o sistema de arquivos YAFFS2. O próprio U-Boot não tem suporte a esse sistema de arquivos e não há muitas soluções disponíveis. O funcionamento do sistema de arquivos teria função complementar em relação ao sistema já executado na nova plataforma, permitindo o uso de aplicativos e a realização de testes.

5. CONCLUSÕES

Este trabalho teve como objetivo analisar o processo de portabilidade do sistema operacional Android para uma plataforma de desenvolvimento, no caso a SAM9-L9260 da Olimex. Vários fatores contribuíram favoravelmente à escolha do Android, como a presença de um *kernel* Linux, a existência de uma comunidade de desenvolvimento, o número expressivo de *drivers* disponíveis, permitindo suporte a vários componentes, entre outros.

Foi possível comprovar a disponibilidade e facilidade de uso do código fonte Android. Não é necessário muito esforço para encontrar bons tutoriais e repositórios com soluções para o sistema operacional. Porém, especificamente para o caso do código fonte somente do *kernel*, erros de programação foram encontrados, o que pode indicar uma falta de supervisão do produto final.

No que tange à configuração e compilação do *kernel*, o método indicado funciona relativamente bem. O *compilador cruzado* não é difícil de ser obtido e configurado. Quanto ao *kernel*, a configuração também é simples, por meio de um *menu* bastante intuitivo, é necessário determinar as diretrizes principais e os *drivers* a serem incluídos no sistema. Alguns *drivers* apresentam problemas de compilação, porém este fato não foi crítico para o comprometimento da montagem do *kernel*.

O processo de integração da imagem do *kernel* Android com o *hardware* da plataforma SAM9-L9260 mostrou-se o principal obstáculo do trabalho. A limitação da placa em termos da memória Flash onde se mantinha o *kernel* (DATA Flash) tornou compulsória a obtenção de mais informações sobre o funcionamento e as alternativas oferecidas pelo *software* U-Boot.

O U-Boot mostrou ser o principal responsável pelas ações essenciais ao processo de portabilidade. Várias formas diferentes de comunicação e transferência dos dados necessários para o funcionamento do sistema operacional, com uma prévia configuração de alguns periféricos, são algumas das vantagens descobertas para este *software*, dando ao programador a flexibilidade e facilidade de testes na fase de desenvolvimento e, com outro conjunto de opções, oferecendo a solidez necessária na comercialização de um produto.

Todavia, a combinação entre U-Boot e Android causa um conflito de compatibilidades quanto ao sistema de arquivos. O Android adota o YAFFS2 como formato padrão de seus sistemas de arquivos. O U-Boot não reconhece este sistema de arquivos, apenas outros como EXT2, JFFS2 e FAT. A simples substituição do sistema JFFS2 para YAFFS2 na execução do U-Boot não é suficiente, pois o *software* tenta realizar uma pré-montagem do dispositivo antes de iniciar o *kernel*.

Desta forma, o *kernel* é iniciado, os serviços executam, mas os aplicativos do sistema não, pois o sistema de arquivos é dado como inexistente. O problema persiste mesmo com

tentativas com formato EXT2, seja por comunicação USB ou por NFS. Segundo a configuração do *kernel* Android a inserção de um *driver* para reconhecimento de sistemas JFFS2 é possível, contudo, o sistema de arquivos neste formato não é reconhecido, mesmo com o *driver* instalado.

A obtenção do sistema de arquivos também apresenta algumas dificuldades. Os arquivos *system.img* e *userdata.img* são imagens codificadas em YAFFS2 geradas pelo emulador Android, porém o programa para desfazer a codificação, oferecido pelos desenvolvedores do sistema operacional, não consegue realizar a operação. Assume-se que este tipo de incompatibilidade não ocorreria, por serem ferramentas fornecidas pelos desenvolvedores do Android.

De qualquer modo, é importante ressaltar que a incompatibilidade se resume apenas à combinação U-Boot e Android, impedindo que o sistema atinja o fim de sua inicialização. Todavia, o processo de compilação, transferência e execução do Android para a plataforma de desenvolvimento foi realizado com sucesso.

Em suma, pode-se concluir acerca do projeto que:

- O método proposto para a portabilidade do sistema operacional Linux para uma plataforma específica funciona. Sua aplicação pode encontrar dificuldades, que são relacionadas a características específicas do sistema operacional ou do *hardware*.
- O sistema de arquivos YAFFS2 não é reconhecido pelo U-Boot. Na execução final do projeto, o *kernel* não consegue executar o sistema de arquivos.
- A aplicação do processo pode encontrar dificuldades. Tanto os problemas contornados, quanto o que não foram solucionados, são relacionados a características específicas do sistema operacional ou do *hardware*. Diferentes podem resultar em novos problemas ou em um sistema totalmente operante.
- Os problemas de compilação são exclusivos do código Android obtido e do *toolchain*. Outro sistema baseado em Linux ou mesmo um código de outro repositório combinados a outros *toolchains* podem apresentar diferentes resultados. São independentes da plataforma escolhida.
- A geração de código pelo *toolchain* é padronizada e suporta a grande maioria das plataformas de desenvolvimento existentes. A mudança da placa não inviabiliza a compilação.
- O conjunto de configurações do U-Boot (ou outro *bootloader*), por sua vez, tem interferência direta da plataforma escolhida. A mudança do *bootloader* ou da plataforma pode alterar a estratégia utilizada para a inicialização do sistema.

Por fim, pode-se considerar o objetivo de estudo da portabilidade de sistemas operacionais, no caso Android, em plataformas embarcadas, atingido; elucidando em cada uma das etapas as dificuldades e apresentando conceitos fundamentais para a repetição do processo em um sistema operacional baseado em Linux para uma determinada plataforma.

REFERÊNCIAS BIBLIOGRÁFICAS

ABLESON, W. F.; SEN, R.; KING, C. **Android in Action**. 2 ed. Stamford, CT, EUA: Manning 2011

ATMEL, **ARM Architecture Reference Manual**, 2000.

DARCEY, L.; CONDER, S. **Sams Teach Yourself Android Application Development in 24 Hours**. Indianapolis, Indiana, EUA: Sams 2010.

FURBER, S. **ARM System-on-chip architecture**. 2 ed., Addison-Wesley 2000.

GARGENTA, M. **Learning Android**. Sebastopol, CA, EUA: O'Reilly 2011

HALLINAN, C. Embedded Linux Primer: A Real-World Approach. . Crawfordsville, Indiana, EUA : Pearson 2006.

HEATH, S. **Embedded System Design**. 2 ed. Burlington, MA, EUA: Newnes 2003.

KROAH-HARTMAN, G. **Linux *Kernel* in a Nutshell**, Sebastopol, CA, EUA: O'Reilly 2006.

LOVE, R. **Linux *Kernel* Development**. 3 ed. Crawfordsville, Indiana, EUA : Pearson 2010.

MURPHY, M. **Beginning Android 3**. New York, NY, EUA: Apress 2011.

OLIMEX, SAM9-L9260 development board. **Users Manual.** 2009.

RAGHAVAN, P.; LAD, A.; NEELAKANDAN, S. **Embedded Linux System Design and Development**. Boca Raton, FL, EUA: Auerbach 2006.

SIEVER, E. et al. **Linux in a Nutshell**. 6 ed. Sebastopol, CA, EUA: O'Reilly 2009.

SILBERCHATZ, A.; GAGNE, G.; GALVIN, P. B. **Operating System Concepts**. 7 ed. Danvers, MA, EUA: Wiley 2005.

SLOSS, A. N.; SYMES, D.; WRIGHT, C. **ARM System Developer's Guide**. Designing and Optimizing System *Software*. San Francisco, CA, EUA: Elsevier 2040.

STEELE, J.; TO, N. **The Android Developer's Cookbook.** Building Aplications with the Android SDK. Crawfordsville, Indiana, EUA : Addison-Wesley 2011.

WELSH, M. et al. **Linux Installation and Getting Started**. Seattle, WA, EUA: SSC 1998 .

WOLF, W. **Computers as Components**. Principles of Embedded Computing System Design. 2 ed. Burlington, MA, EUA: Elsevier 2008.

YAGHMOUR, K. et al. **Building Embedded Linux Systems**. 2 ed. Sebastopol, CA, EUA: O'Reilly 2008.