Inatel

Instituto Nacional de Telecomunicações

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

DEVICE DRIVERS EM LINUX EMBARCADO

Priscila Pereira Apocalypse

Ramon Pereira dos Santos Chaib

Julho de 2011

DEVICE DRIVERS EM LINUX EMBARCADO

Priscila Pereira Apocalypse

Ramon Pereira dos Santos Chaib

Trabalho de conclusão de curso apresentado ao Instituto Nacional de Telecomunicações, como parte dos requisitos para obtenção do Título de Engenheiro da Computação.

ORIENTADOR: Prof. Me. Carlos Henrique Loureiro Feichas

Santa Rita do Sapucaí
2011

FOLHA DE APROVAÇÃO

Trabalho de Conclusão de Curso apresentado e aprovado em 18 / 06 / 2011 pela comissão julgadora:

Nome / Inatel – Orientador e Presidente da Comissão Julgadora

---

Nome / Inatel – Membro da Comissão Julgadora

Nome / Inatel – Membro da Comissão Julgadora

_____

Coordenador do Curso de Engenharia da Computação

"Importa, a todo custo, que façamos alguma coisa de nossa vida. Não o que os outros admiram, mas esse impulso que consiste em imprimir-lhe o infinito."

Emannuel Mounier

AGRADECIMENTOS

Agradecemos primeiramente aos nossos familiares, pois, em todos os momentos contamos com o apoio, a disciplina e os conselhos que nos fizeram vencer. Prestamos agradecimentos aos nossos amigos que foram companheiros em momentos difíceis de trabalho árduo. Agradecemos ao nosso orientador Carlos Henrique Loureiro Feichas que através de seu acompanhamento permitiu que esse trabalho consolidasse. Agradecemos também, em especial, ao tio da Priscila Apocalypse, Carlos Roberto da Silveira por guiar-nos com a luz do seu conhecimento durante esta etapa. E por fim, agradecemos, especialmente, a todos os professores e funcionários do INATEL que através dos anos, foram nossa fonte de conhecimento, companheirismo e dedicação, permitindo, assim, uma excelente formação acadêmica e profissional. Como já diz a sábia frase: “A mente que se abre a uma nova ideia jamais volta ao seu tamanho original.” — Albert Einstein.

RESUMO

Cresce a cada dia a quantidade de dispositivos eletrônicos que buscam melhorar o seu desempenho através do aprimoramento da comunicação entre seu *hardware* e *software*. Esta comunicação se deve por um conjunto de instruções que possibilitam que um ou mais dispositivos tenham acesso transparente ao sistema como se fosse nativo. O aprimoramento da maioria dos dispositivos atuais se dá através do desenvolvimento dos sistemas embarcados, sistema no qual, a capacidade computacional é encapsulada dentro de um circuito integrado, sendo este dedicado ao sistema que controla. Então, possuindo sistemas operacionais embarcados com um *set* de instruções reduzidos, estes conseguem desempenhar diversas funcionalidades para o qual foi projetado. Os *softwares* embarcados fazem a interface homem/máquina e estabelecem as funcionalidades para o sistema.

Os *Device Drivers* promovem a comunicação transparente desenvolvida na linguagem C que interagem diretamente com o *kernel* (núcleo do sistema operacional). O *kernel* estabelece quais funções e operações o *device* poderá utilizar. As chamadas destas funções são feitas através do *devices driver* e tem como principal objetivo facilitar e permitir o acesso a todos dispositivos possíveis. Os *devices drivers* oferecem um alto nível de abstração, por isso, é comum encontrá-los separados por módulos do restante do *kernel*. Exemplificando, seria como se os *devices drivers* fossem "caixas pretas" que escondem do *kernel* todas as informações dos dispositivos, como os detalhes do funcionamento e de sua implementação.
Sem os *devices drivers* não há a compatibilidade e interface com diversos dispositivos.

Portanto, a presente monografia tem como objetivo principal analisar a importância dos *devices drivers* em dispositivos externos, com a pretensão de apontar as suas vantagens e mostrar o seu desenvolvimento através da plataforma Linux, utilizando o microprocessador ARM. O modelo prático desta pesquisa consiste no desenvolvimento de um *device driver* na linguagem C que será embarcado no *kit* Friendly ARM.

Palavras-chaves: *Device Drivers*; Linux; ARM; Sistema Embarcado.

ABSTRACT

Every day are growing the number of devices that seek improves it performance by the improvement of communication between its hardware and software. This communication should be a set of instructions that enable one or more devices have transparent access to the system as if it were native. The improvement of systems occurred through the development of embedded systems that is a system in which computing power is encapsulated within an integrated circuit dedicated to the system it controls. Featuring embedded operating systems with a reduced instruction set, can make a variety of features for which it was designed. The embedded software makes the man / machine interface and provide the functionality to the system.

The Device Drivers are this transparent forms of communication, developed in C language that interacting directly with the kernel (core operating system). The kernel provides which functions and operations the device can use, the calls of these functions are done through the device driver and it's have principal objective to facilitate and enable access to all possible devices. It offer a high level of abstraction that is common to find it separated in modules of the rest of the kernel. For example, it would be if as the devices are like black boxes that hide of kernel all information's of the devices, such as details of the operation and implementation.

Without them we would not have this compatibility and interface with various devices. Therefore, this monograph have object of approach a study on embedded systems, ARM microprocessor, embedded operating system, more specifically Linux, show the importance of device drivers in external devices, showing it advantages and development. The practical part consists in development of a device driver in C language in a kit Friendly ARM for exemplify detailed its operation using the functions of the Linux kernel.

Keywords: Device Drivers; Linux; ARM; Embedded System.

ÍNDICE

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

TCC - Trabalho de Conclusão de Curso.

ARM *Advanced RISC Machine* – Máquina RISC Avançada.

GPIO - *General Purpose Input/Output* – Porta Geral de Entrada e Saída de Dados.

MIT - *Massachusetts Institute of Technology* – Instituto de Tecnologia de Massachusetts.

NAND - *Not And* – Porta Não-E. Memória Flash de Velocidade de Acesso Alta.

ULA - Unidade Lógica Aritmética.

PC - *Program Counter* – Registro que Guarda a Próxima Instrução.

RISC - *Reduced Instruction Set Computer* – Computador com um Conjunto Reduzido de Instruções.

KB - Kilobyte.

MMU - *Memory Management Unit* – Unidade de Gerenciamento de Memória.

CPU - *Central Processing Unit* – Unidade Central de Processamento.

MB - Megabyte.

GNU - *General Public License* – Licença Livre.

E/S - Entrada/Saída.

SBC - *Single-Board Computer* – Computador de placa única.

Cm - Centímetro.

Mm - Milímetro.

MHz - *Mega-Hertz* – Unidade de frequência.

RAM - *Random Access Memory* – Memória de Acesso Aleatório.

SDRAM - *Synchronous Dynamic RAM* – Memória *Síncrona* Dinâmica de Acesso Aleatório.

NOR - *Not-Or* – Porta Não-Ou. Memória Flash de Velocidade de Acesso Baixa.

BIOS - *Basic Input Output System* – Sistema Básico de Entrada/Saída.

EEPROM - *Electrically-Erasable Programmable Read-Only Memory* – Memória de armazenamento não volátil.

I2C - *Inter-Intergrated Circuit* – Circuito Inter-Integrado.

USB - Universal Serial Bus.

RTC - Real Time Clock.

LCD - *Liquid Crystal Display* – Display de cristal líquido.

STN - *Super-Twisted Nematic display* – Tipo de Display LCD de Cristal Monocromático.

TFT - *Thin film transistor liquid crystal display* – Variante do Visor de Cristal Líquido que usa a Tecnologia de Transistores.

VGA - *Video Graphics Array* – Hardware de Display.

A/D - Analógico/Digital.

LED - *Light-Emitting Diode* – Diodo Emissor de Luz.

JTAG - *Joint Test Action Group* – Protocolo Serial.

V - *Volts* – Unidade de Medida de Tensão Elétrica.

A - *Ampère* – Unidade de Medida de Intensidade de Corrente Elétrica.

GCC - *GNU Compiler Collection* – Compiladores com a Licença GNU.

DC - *Direct Current* – Corrente contínua.

CI - Circuito Integrado.

HTML - *HyperText Markup Language* – Linguagem de Marcação de Hipertexto.

CGI - *Common Gateway Interface* – Tecnologia que Permite Gerar Páginas Dinâmicas.

USB - *Universal Serial Bus* – Tipo de Barramento de Comunicação Entre Periféricos.

LISTA DE SÍMBOLOS

V *Volts* – Unidade de Medida de Tensão Elétrica.

A *Ampère* – Unidade de Medida de Intensidade de Corrente Elétrica.

MHz *Mega-Hertz* – Unidade de frequência.

CAPÍTULO 1. INTRODUÇÃO

Segundo Corbet, Rubini e Hartmank (2005, p.1-2), uma das muitas vantagens dos sistemas operacionais livres, como o Linux, é ser *open source*, ou seja, possui sua licença de modificação e distribuição gratuita. O sistema operacional, uma vez que era restrito a um pequeno número de programadores, agora pode ser examinado, compreendido e modificado por qualquer pessoa com as habilidades necessárias.

Os *devices drivers* assumem um papel especial no *kernel* do Linux. Eles são comparados as "caixas pretas", na qual fazem um determinado *hardware* responder a uma interface interna de programação já definida, escondendo completamente os detalhes de como funciona o equipamento. Atividades do usuário são executadas por meio de um conjunto de chamadas padronizadas que são independentes do *driver* específico; mapear essas chamadas para operações específicas do dispositivo que atuam em hardware real é então o papel do *device driver*.

Há uma série de razões para se interessar pela escrita de *device drivers* em Linux, como por exemplo, facilitar, permitir e otimizar a comunicação com o *hardware* externo ou interno, na qual necessita-se de uma comunicação.

Ainda segundo Corbet, Rubini e Hartmank (*Ibid.,* p.5), esta divisão dos módulos em diferentes tipos, ou classes, não é rígida, o programador pode optar por criar enormes módulos de execução de *drivers* diferentes em um único pedaço de código. Bons programadores, no entanto, costumam criar um módulo diferente para cada nova funcionalidade que desenvolver, pois a decomposição é um elemento-chave da escalabilidade e extensibilidade.

A linguagem predominante no código fonte dos *device drivers* é a linguagem C, como apresentado no modelo prático desta pesquisa.

1. MOTIVAÇÃO

Com o advento da era tecnológica, os computadores, bem como, o seu hardware e os seus softwares, estão desempenhando um papel vital em nossa vida diária. Podemos não ter

notado, mas aparelhos como máquinas de lavar, telefones, televisões, relógios e vários outros exemplos, estão tendo os seus componentes analógicos e mecânicos substituídos por processadores e softwares.

A indústria de sistemas embarcados está crescendo exponencialmente. Com um custo reduzido e controle melhorado, processadores e sistemas controlados por *softwares* oferecem um *design* compacto, manipulação flexível, ricos em recursos e de custo competitivo. As pessoas acreditavam que o "software nunca quebrava". Não há nenhuma restrição ambiental para operar o *software*, enquanto que o processador executado em *hardware* não pode operar. Além disso, o *software* não tem forma, cor, material e massa. Este não pode ser visto ou tocado, mas tem uma existência física e, é crucial para a funcionalidade do sistema. Sem ser provado o contrário, as pessoas otimistas pensam que uma vez que o *software* foi criado pode ser executado corretamente para sempre. Uma série de problemas e caos causados pelo *software* comprova que isso é errado. Por isso, temos que garantir a confiabilidade do sistema, fazendo com que sua comunicação seja otimizada, assim, garante-se a qualidade de acesso às suas informações, bem como, respostas adequadas a diversos ambientes diferentes de testes.

Portanto, vale ressaltar, a importância da utilização e garantia de confiabilidade dos *devices drivers* como meio de interface para outros *hardwares*, utilizando um sistema livre como é o Linux, que já possui muitas bibliotecas para facilitar seu desenvolvimento.

2. OBJETIVOS

A proposta deste projeto de conclusão de curso é apresentar a importância dos *devices drivers* em dispositivos externos. O modelo prático refere-se ao desenvolvimento de um *device driver* para o GPIO (General Purpose Input/Output) em um *kit* Friendly ARM Mini 2440, no qual é exemplificado detalhadamente seu funcionamento utilizando as funções do *kernel* do Linux.

Desta forma busca-se:

1. Aplicar e aprimorar o conhecimento adquirido durante o curso de Engenharia da Computação;

2. Estudar o comportamento dos *softwares* em sistemas embarcados;
    2.1. Compreender como são as funções e o comportamento do Linux embarcado;
    2.2. Estar apto para criar módulos no sistema;
3. Desenvolver um *device driver*;
    3.1. Conhecer o *kernel* Linux para utilizar suas funções e operações;
    3.2. Compreender como é sua estrutura básica;
    3.3. Conhecer como os sistemas operacionais criam suas interações com os *devices*;
    3.4. Propor um modelo prático para implementar no kit;
    3.5. Conhecer o método de codificação C otimizado;
    3.6. Propor uma comunicação via *wireless* com o projeto prático utilizando o *webserver* embarcado BOA;
    3.7. Analisar os resultados esperados do desenvolvimento do *device driver* em questão;

4. Identificar a importância da sua utilização nos sistemas atuais;
5. Comentar sobre as melhores práticas de desenvolvimento;
6. Comentar sobre a importância da otimização do *device driver* em relação sua codificação para evitarmos a sobrecarga no sistema.

3. ESTRUTURA DO TRABALHO

O presente trabalho está dividido em cinco capítulos organizados da seguinte forma:

Capítulo 1: Introdução. Apresenta uma introdução ao assunto e objetivos gerais do Trabalho de Conclusão de Curso justificando a importância do tema.

Capítulo 2: Fundamentação Teórica. Embasamento teórico, no qual, são abordados os conceitos fundamentais para o desenvolvimento do projeto proposto, tais como: sistema embarcado, microprocessador, sistema operacional embarcado e *device drivers*. São pesquisados à luz das referências bibliográficas para maiores entendimentos e desenvolvimentos de *device drivers* na prática.

Capítulo 3: Metodologia. Neste capítulo são apresentados os métodos necessários e ferramentas utilizadas que viabilizam o desenvolvimento do projeto proposto.

Capítulo 4: Testes. São apresentados os testes executados, a estruturação do modelo e análise do desempenho.

Capítulo 5: Conclusão. São apresentadas as ideias principais sobre o projeto desenvolvido, baseados nos testes executados, bem como, tem a humilde pretensão de abrir uma perspectiva para estudos futuros sobre o assunto pesquisado.

CAPÍTULO 2. FUNDAMENTAÇÃO TEÓRICA

1. SISTEMA EMBARCADO

De acordo com Raghavan, Lad e Neelakandan (2006, p.1), um sistema embarcado é um sistema de computador com propósito específico, isto é, projetado por exclusividade. Eles são constituídos basicamente pelos mesmos componentes de um computador, pois possuem microprocessadores (ou microcontroladores), memórias, dispositivos de armazenamento, dispositivos de interface e outros componentes. Segundo Cunha (2011, p.1), o sistema embarcado é um sistema completo e independente, mas preparado para realizar apenas uma determinada tarefa para o qual foi projetado. Segundo Raghavan, Lad e Neelakandan (*Cf*., 2006, p. 1-2), hoje o sistema embarcado é utilizado para diversos fins, tais como:

- Equipamentos de redes: *switch* e roteadores;
- Equipamentos de consumo: MP3 *players*, telefones celulares, PDA's, câmera digital, filmadoras e sistemas de entretenimento doméstico;
- Eletrodomésticos: micro-ondas, máquinas de lavar, televisores;
- Sistemas de missão crítica:satélites e controles de voos, dentre outros.

1.1 CARACTERÍSTICAS DE SISTEMAS EMBARCADOS

Segundo Noergaard (*Cf*., 2005, p.11), são características primordiais de um sistema embarcado:

- Integridade do sistema (confiabilidade e segurança);
- Consumo de energia;
- Custo;
- Vida útil;
- Requisitos determinísticos.

1.2 MODOS DE FUNCIONAMENTO DE SISTEMAS EMBARCADOS

Os modos de funcionamento de sistemas embarcados determinam como será o comportamento deste diante da aplicação para o qual foi projetado. São basicamente dois os estes modos de funcionamento:

- Reativo;
- Controle em tempo real (*Soft Real Time e Hard Real Time*).

No desenvolvimento do projeto proposto nesta monografia, é fundamental a compreensão do funcionamento reativo. Segundo Cunha (2011, p.3-4), o funcionamento reativo se dá como resposta a eventos externos, que podem ser periódicos (caso de sistemas rotacionais ou de controles de loop) ou assíncronos (como, por exemplo, o pressionamento de um botão por parte do usuário). Ainda segundo o autor (*Ibid.*, p.4), há, então, uma necessidade de entrada de dados para que aconteçam as ações de funcionamento. O sistema capta as informações recebidas e as processa em um tempo adequado ao controle da aplicação.

Na figura 1, é possível observar a composição básica modular de um sistema embarcado que possui as três camadas:

- Camada de *software* de aplicação;
- Camada de *software* do sistema;
- Camada de *hardware*.

As camadas de *softwares* são opcionais, ao passo que a camada de *hardware* é obrigatória, ou seja, todos os modelos de sistemas embarcados devem ter pelo menos a camada de *hardware*. Segundo Noergaard (2005, p.12-13), a camada de *hardware* contém todos os principais componentes físicos localizados em uma placa embarcada. As camadas de *software* de sistemas e aplicações contêm todo o *software* necessário para ser processado pelo sistema.

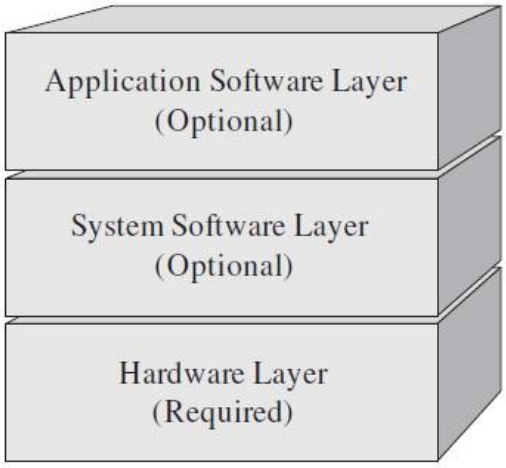


Figura 1. Modelo de sistema embarcado (NOERGAARD, 2005, p.12).

1.3 SOFTWARE EMBARCADO

O software embarcado é um conjunto de instruções em uma dada linguagem de programação, na qual foi projetado com um objetivo específico de realizar sua execução no sistema. Frequentemente as linguagens de programação utilizadas são as *assembler* e *C*. Também são utilizados *scripts* dentre outros métodos de programação.

2. MICROPROCESSADOR

2.1 BREVE DEFINIÇÃO

Segundo Andrew S. Tanenbaum (2006, p.29), microprocessador, também chamado de processador, é um circuito integrado que realiza as funções de cálculo e tomada de decisão de dispositivos eletrônicos. Todos os computadores e equipamentos eletrônicos baseiam-se nele para executar suas funções. Ainda segundo o autor (*Ibid.*, p.29), O processador é o "cérebro" do computador. Sua função é executar os programas armazenados na memória principal. O processador busca cada instrução na memória, examina-a e executa uma após outra.

2.2 COMPONENTES DE UM MICROPROCESSADOR

São componentes fundamentais de um microprocessador:

- Unidade de Controle;
- Unidade Aritmética Lógica (UAL);
- Registradores (*PC*, *IR* e outros registradores de propósito geral ou específico).

A figura 2 apresenta a estrutura dos componentes de um microprocessador com uma CPU e dois dispositivos de E/S.

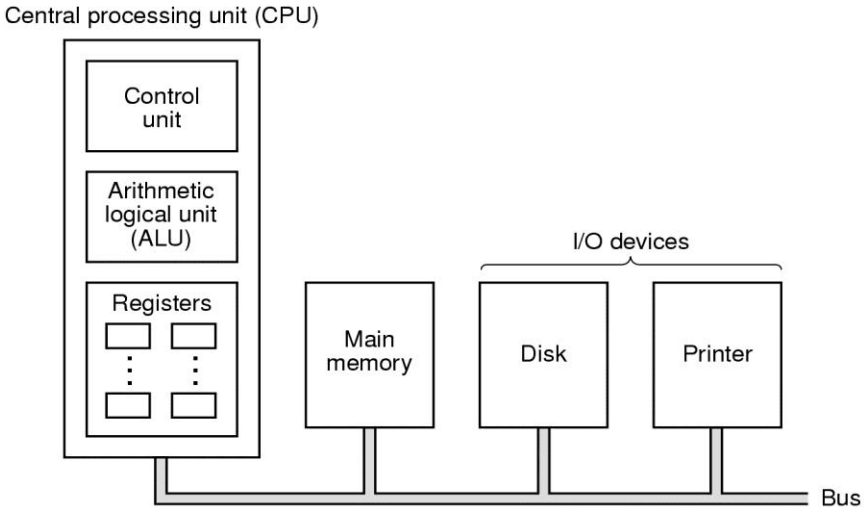


Figura 2. Organização de um computador simples (TANENBAUM, 2006, p.29).

2.3 MICROPROCESSADOR ARM

Segundo Steve Furber (2000, p.35-37), o ARM foi originalmente desenvolvido na Acorn Computers Limited de Cambridge, Inglaterra, entre 1983 e 1985. Foi o microprocessador RISC (*Reduced Instruction Set Computing* - computador com um conjunto reduzido de instruções) desenvolvido pela primeira vez para uso comercial e tem algumas diferenças significativas a partir de RISC de arquiteturas subsequentes. O conceito RISC foi originado nos programas de pesquisa em processadores nas universidades de Stanford e Berkeley em 1980.

Em 1990, a ARM Limited foi estabelecida como uma empresa separada especificamente para ampliar a exploração da tecnologia ARM. Desde licenciado (ARM) aos fabricantes de semicondutores ao redor do mundo, este se tornou um líder de mercado, devido o seu produto possuir baixa potência e custo acessível para aplicações embarcadas.

Nenhum processador é particularmente útil sem o suporte de ferramentas de desenvolvimento de *hardware* e *software*. O ARM é apoiado por um conjunto de ferramentas, que inclui um conjunto de instruções emulado para definir a modelagem de *hardware* e *software* de teste e de *benchmarking*, um *assembler*, compiladores C e C++, um *linker* e um depurador simbólico.

Neste trabalho o desenvolvimento foi realizado utilizando o ARM920T, que é o microprocessador do kit friendly ARM mini2440, segundo o ARM920T datasheet (*Cf.*, ARM920T Datasheet Home Page), e apresenta as seguintes características principais:

- Arquitetura cache Harvard, que é direcionado a aplicações de multiprogramação, na qual a gestão de memória cheia, alta performance e baixo consumo são importantes;

- Suporta tanto conjunto de instruções de 32 bits, quanto conjuntos de instruções de 16 bits, o que lhe permite "*trade off*" entre alto desempenho e alta densidade de código;

- Separação das instruções e dados de *cache* é de 16KB;

- Implementa uma arquitetura avançada v4 ARM MMU para fornecer serviços de tradução e verificações de permissão de acesso para endereços de instrução e dados;

- Utiliza a tecnologia ARM para *debug* (depuração) incluindo lógica para ajudar na depuração de *hardware* e *software*.

3. SISTEMA OPERACIONAL EMBARCADO

A evolução dos microprocessadores fez com que os sistemas operacionais fossem portados para os sistemas embarcados, permitindo um melhor controle do equipamento.

3.1 . DEFINIÇÃO

Sistema operacional embarcado foi especialmente concebido para computadores equipados com sistemas embarcados. É provavelmente o mais compacto e eficiente de todos os sistemas operacionais.

Dentre uma gama de características, suas principais são: fornecer o controle sobre acesso a memória, abstração no uso de periféricos e compartilhamento da CPU por mais de um processo (*multi thread*), além da confiabilidade no tempo de resposta. Na figura abaixo podemos ver três modelos de sistemas operacionais embarcados.

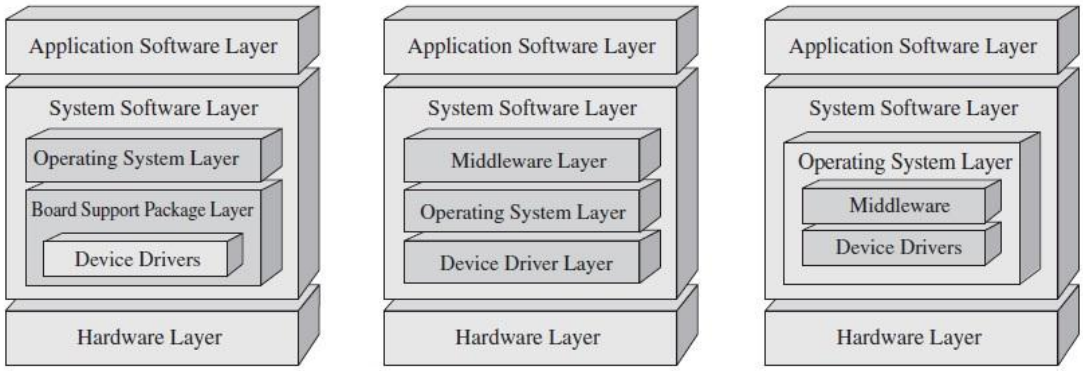


Figura 3. Modelos de SO embarcado (NOERGAARD, 2005, p.311).

3.2 . CARACTERÍSTICAS DETERMINANTES

As características determinantes na escolha de um bom sistema operacional embarcado são:

- Modularidade;

- Escalabilidade;
- Configurabilidade;
- Apresentar suporte de CPU;
- Desenvolvimento de *Device Drivers*.

3.3 . TIPOS DE SISTEMAS OPERACIONAIS EMBARCADOS

Existem diversos tipos de sistemas operacionais para sistemas embarcados. Alguns que merecem destaque e são amplamente utilizados são: Linux, Android,FreeDOS, RTOS LynxOS, Windows CE, ThreadX, SymbianOS e T2SDE.

3.4 . LINUX EMBARCADO

Segundo Raghavan (2006, p.2-7), Linus Benedict Torvalds na Universidade de Helsinki criou o sistema operacional Linux em 1991. Segundo Bovet (2005,p.1), foi um sistema desenvolvido inicialmente para computadores pessoais da IBM, baseado no microprocessador Intel 80386. Segundo Raghavan (2006, p.2-3), o modelo de desenvolvimento de código aberto e licença GNU, sob a qual o Linux é distribuído, atraíram contribuições de milhares de programadores do mundo todo. Esta licença permitiu que todos os códigos fontes do Linux fossem livremente distribuídos para uso pessoal ou uso comercial. Como o código fonte do Linux está disponível gratuitamente, isso fez com que muitos desenvolvedores contribuíssem com o seu *kernel*. Devido a oferta global de desenvolvedores, que temos um sistema operacional altamente confiável e robusto.

Ainda segundo Raghavan (*Ibid.*, p.2-7), no início de 1996, o Linux viu a sua chegada em sistemas embarcados de tempo real como um projeto de pesquisa de Michael Barabanov e Victor Barabanov Yodaiken. Em 1997, o projeto uClinux foi iniciado com o objetivo de utilizar Linux em microprocessadores e microcontroladores. Somente então em 1998 ele foi liberado para uso. Durante os anos de 1999 a 2004 o Linux era amplamente utilizado em sistemas embarcados. Atualmente, o Linux vem ganhando uma gama de usuários em diversos

tipos de hardware devido seu alto nível de modularidade, que é utilizado na implementação do *kernel*, tornando fácil sua portabilidade em diferentes plataformas.

São algumas vantagens da utilização do Linux embarcado:

- Baixo Custo;
- *Open Source*;
- Portabilidade e escalabilidade;
- *Kernel* reduzido;
- Vários aplicativos disponíveis.

3.5 WEBSERVER EMBARCADO: BOA

De acordo com a documentação do BOA (*Cf.*, BOA Home Page), ele é um servidor *web* de código aberto e pequeno, processamento que é conveniente para aplicações em sistemas embarcados. Originalmente escrito por Paul Phillips em 1991, é mantido agora por Larry Doolittle e Jon Nelson. Os objetivos do projeto principal de BOA são obter velocidade e segurança.

4. DEVICE DRIVERS

Segundo Corbet, Rubini e Hartmank (2005, p.1-2), os *devices drivers* assumem um papel essencial no *kernel* do Linux, pois oferecem um alto nível de abstração, razão pela qual é muito comum encontrá-los separados por módulos do restante do *kernel*. As atividades do usuário são executadas por meio de um conjunto de chamadas padronizadas que são independentes do *driver* específico. Mapear essas chamadas para operações específicas do dispositivo que atuam em *hardware* real é então o papel do *driver* de dispositivo. Sem eles não teríamos esta compatibilidade e interface com diversos dispositivos.

Segundo Corbet, Rubini e Hartmank (*Ibid.*,p.1), uma característica importante dos *device drivers* no Linux é a sua modularização, em outras palavras, um *device driver* pode ser anexado ao *kernel* desejado sem a necessidade de recompilá-lo todo ou mesmo reiniciar a

máquina. Segundo Baruchi (*Cf.,* "Desenvolvendo Device Drivers no Linux"), outro ponto interessante no desenvolvimento de *device drivers* é que não necessariamente um *device driver* é implementado para se comunicar com algum dispositivo de *hardware*; existem módulos construídos somente para coletar informações do próprio *kernel*. Um exemplo de módulo desse tipo é o que implementa o sistema de arquivos virtual /proc do Linux, que permite obter informações sobre a memória virtual, processador, escalonador, processos e outros mais, como se fossem arquivos.

4.1 DIVISÃO DOS DEVICES DRIVERS

De acordo com Corbet, Rubini e Hartmank (2005, p.5), a maneira que o Linux enxerga os devices distingue em três tipos fundamentais. Cada módulo geralmente implementa um destes tipos, e assim é classificado como: módulo *char device*, módulo *block device* e módulo *network device*. Tal divisão dos módulos em diferentes tipos, ou classes, não é rígida. O programador pode optar por criar enormes módulos de execução de *drivers* diferentes em um único pedaço de código. Bons programadores, no entanto, costumam criar um módulo diferente para cada nova funcionalidade que for implementada, pois a decomposição é um elemento-chave da escalabilidade e extensibilidade. Uma definição melhor de cada módulo é feita por Corbet, Rubini e Hartmank (*Cf.*, p.5-8):

- ***Char device*** – O dispositivo de caracteres é aquele que pode ser acessado com um fluxo de bytes, como exemplo, os *device drivers* das portas seriais (/dev/ttyS0), pois são representados pela captação do fluxo;

- ***Block device*** – Um dispositivo de bloco é aquele que pode hospedar um sistema de arquivos como, por exemplo, um disco rigido. Na maioria dos sistemas Unix, o *block device* só pode lidar com operações de E/S com transferências de blocos inteiros que geralmente possuem tamanho de 512 *bytes* permitindo que um aplicativo possa ler e escrever em um *block device*. A diferença entre um *char device* e um *block device* está apenas na forma como os dados são gerenciados internamente pelo *kernel*. O *block device* é acessado através de um nó do sistema de arquivos;
- ***Network device*** – Qualquer transação na rede é feita através de uma interface, ou seja, um dispositivo que é capaz de trocar dados com outros *hosts*. A

> interface de rede é responsável por enviar e receber pacotes de dados, guiado pelo subsistema de rede do *kernel*. Muitas conexões de rede (principalmente as que utilizam TCP) são orientadas a fluxo. Um *driver* de rede não sabe nada de conexões individuais ele somente manipula os pacotes. A comunicação entre o *driver* e o *kernel* é diferente do *block device* e *char device*, pois, ao invés de ler e escrever, o *kernel* chama funções relacionadas com a transmissão dos pacotes. No Unix, a interface de rede normalmente é chamada de eth0 para *ethernet*, wifi0 ou w0 para *wireless* e lo para interface de *loopback*.

4.2 IMPLEMENTAÇÃO

Segundo Baruchi (*Cf.*, "Desenvolvendo Device Drivers no Linux"), a linguagem predominante no código fonte do Linux é a linguagem C. Apenas algumas partes bem específicas (por exemplo, a parte do código que realiza o *bootstrap*) são escritas em *Assembly*.

Para o desenvolvimento de *device drivers* vários conceitos são relevantes como endereçamento de memória, *buffers*, lista ligadas, macros (pré-processador), conhecimento em ferramentas de desenvolvimento Linux GNU *make*, processos, *threads*, *system calls* (chamadas do sistema), conceito de funções E/S (funções de entrada e saída), *timers*, interrupções do sistema. Todo conhecimento da estrutura do *kernel* é fundamental para o bom desenvolvimento de um *device*.

Segundo Bovet (2005, p.2), para implementar as operações, o Linux 2.6 fornece algumas estruturas de dados e funções auxiliares que oferecem uma visão unificadora de todo o barramento, dispositivos e *drivers* no sistema, este quadro é chamado de modelo de *driver* de dispositivos.

Exemplificado por Corbet, Rubini e Hartmank, a figura 4 apresenta um exemplo simples de codificação em linguagem C.

```c
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Figura 4. Exemplo “Olá Mundo” (CORBET, RUBINI e HARTMANK, 2005, p.16).

No exemplo da figura 4, são apresentadas duas funções, uma para ser chamada quando o módulo é carregado (hello_init) e outra quando o módulo for terminado (hello_exit). O comando module_init e module_exit utilizam macros especiais do *kernel* para indicar o papel destas duas funções, ou seja, module_init para iniciar o módulo e module_exit para finalizar.

Outra macro especial é a module_license que é utilizada para informar o *kernel* que este módulo tem uma licença do tipo livre (GPL). Sem tal declaração, o *kernel* emite avisos quando o módulo é carregado.

A função *printk* é definida no kernel do Linux. Assim como *printf* imprime mensagem para usuário na linguagem C, o *kernel* tem a sua própria, como é o caso da *printk*.

A função KERN_ALERT que vai como parâmetro no *printk* é que define a prioridade da mensagem.

É possível testar o módulo com os utilitários *insmod* e *rmmod*, como mostrado na figura 5. Observe que somente o usuário *root* (que possui as prioridades) pode carregar e descarregar um módulo.

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M]  /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC      /home/ldd3/src/misc-modules/hello.mod.o
  LD [M]  /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Figura 5. Exemplo "Make" (CORBET, RUBINI e HARTMANK, 2005, p.17).

O comando *insmod* é para carregar o módulo no *kernel* e o *rmmod* para removê-lo.

CAPÍTULO 3. METODOLOGIA

Neste capítulo serão apresentados os conceitos e conhecimentos adquiridos para o desenvolvimento do projeto, incluindo as ferramentas utilizadas, detalhes sobre o projeto, uma apresentação detalhada sobre a biblioteca de funções, a implementação final e o funcionamento do sistema.

Exemplificaremos criando um *device driver* para a porta GPIO do *kit* FriendlyARM Mini 2440. Os GPIO são portas de entrada e saída de dados. São utilizadas para permitir a interface entre os periféricos.

1. KIT FRIENDLY ARM MINI2440

O *kit* escolhido foi o FriendlyARM Mini 2440 SBC. Ele possui um processador ARM9 Samsung S3C2440 de 400 MHz.

A placa possui dimensões de aproximadamente 10 cm x 10 cm, muito indicada para aprendizado sobre os sistemas ARM e para aprofundamento em Linux embarcado.

1.1 . ESPECIFICAÇÃO

De acordo com FriendlyARM (*Cf.,* FriendlyARM Home Page), são especificações do kit FriendlyARM Mini 2440 :

- Dimensão: 100 x 100 mm;
- CPU: 400 MHz Samsung S3C2440A ARM920T (freq. máx. 533 MHz);
- RAM: 64 MB SDRAM, 32 bit Bus Vida útil;
- Flash: 256 MB NAND Flash e 2 MB NOR Flash com BIOS;
- EEPROM: 1024 Byte (I2C);
- Memória Externa: SD-Card socket;
- Portas Seriais: 1x DB9 conector (RS232), total: 3x serial port connectors;
- USB: 1x USB-A Host 1.1, 1x USB-B Device 1.1;

- Saída de Áudio: 3.5 mm stereo Jack;
- Entrada de Áudio: Connector + Condenser microphone;
- Ethernet: RJ-45 10/100M (DM9000);
- RTC: Real Time Clock com bacteria;
- Beeper: PWM buzzer;
- Câmera: 20 pin interface de câmera (2.0 mm);
- LCD Interface;
- STN Displays;
- TFT Displays;
- 41 pin connector for FriendlyARM Displays (3.5" and 7") and VGA Board;
- Touch Panel: 4 wire resistive;
- Entrada do Usuário: 6x botões e 1x pot. A/D;
- Saídas do Usuário: 4x LED’s;
- Expansão: 40 pin System Bus, 34 pin GPIO, 10 pin Buttons (2.0 mm);
- Debug: 10 pin JTAG (2.0 mm);
- Alimentação: 5V;
- Consumo de energia: 0.3 A;
- Suporte de SO: Linux 2.6 , Windows CE 5 e 6 e Android.

Na Figura 6 é possível verificar a distribuição na placa:

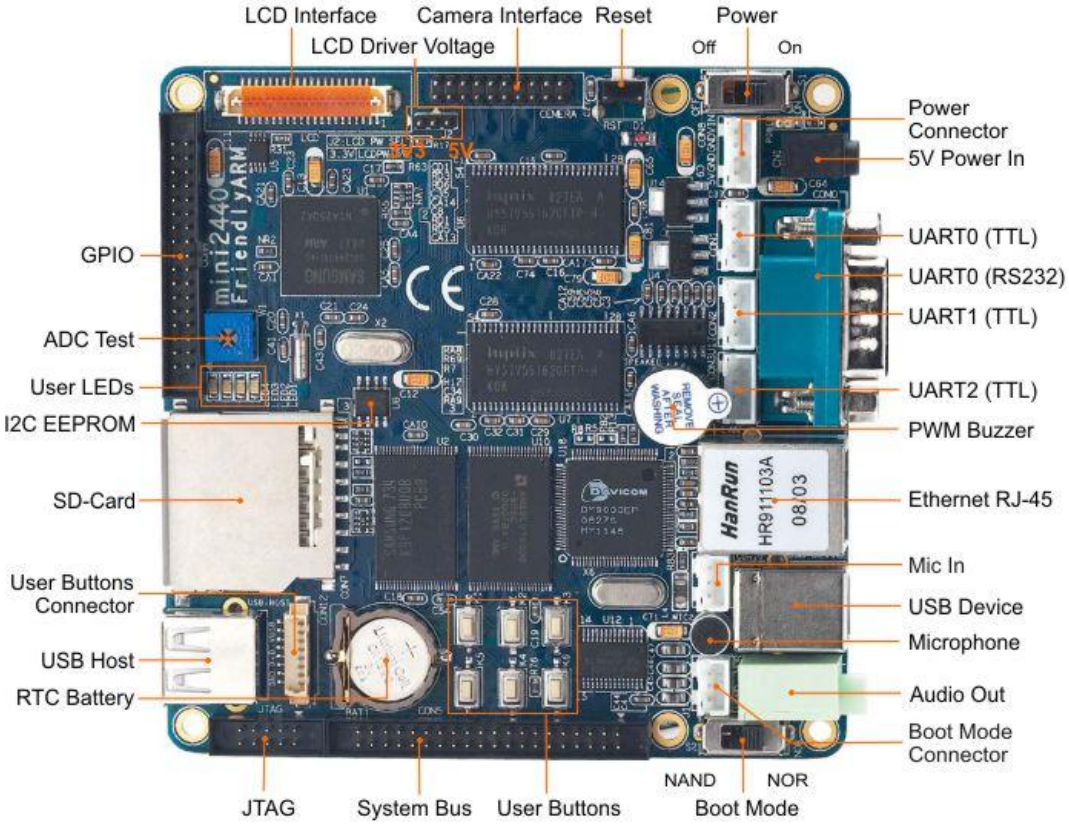


Figura 6. Hardware kit FriendlyARM Mini 2440 (*Cf.*,FriendlyARM Home Page).

2. FERRAMENTAS UTILIZADAS

Para desenvolver o *device driver* para a porta GPIO do *kit* foram utilizadas as seguintes ferramentas:

2.1 GEDIT

O Gedit é o editor oficial de texto sobre a plataforma Gnome. Dentre os vários recursos que esse editor oferece, dois deles são importantes para auxiliar no desenvolvimento desse projeto, tais como:

- Identação automática (a habilidade do editor de reconhecer estruturas de controle em seu código e de aplicar automaticamente a identação apropriada, quando se começa uma linha nova);

- *Syntax highlight* (Destacador de Sintaxe – para linguagens de programação).

2.2 CROSS-COMPILADOR ARM-LINUX GCC 4.4.3

Um *cross* compilador é capaz de gerar um programa executável para uma plataforma diferente daquela em que está sendo executado**.** Ferramentas de compilação *cross* são usadas para gerar executáveis para sistemas embarcados ou de múltiplas plataformas.

O compilador GCC foi originalmente escrito como o compilador para o sistema operacional GNU. O sistema GNU foi desenvolvido para ser 100% *software* livre, livre no sentido de que, este respeita a liberdade do usuário. Então o ARM-Linux GCC é um compilador C voltado para a arquitetura ARM.

Para compilar os códigos fontes desta pesquisa, foi utilizado o *cross* compilador na distribuição Linux Fedora, sendo possível a geração do código executável para a plataforma ARM.

2.3 . KERNEL DO FRIENDLYARM LINUX 2.6.32.2

Segundo o manual GNU Make (*Cf.*, GNU Make Manual Home Page), *make* é um programa para compilar automaticamente um código fonte. Ele utiliza instruções contidas num arquivo chamado "*Makefile*" que é capaz de resolver as dependências dos códigos fontes que se pretende compilar.

Foram utilizados também os seguintes comandos:

- yum: yum install kernel-package kernel-devel: Instalar o pacote do *kernel*;
- make: Como já foi explicado anteriormente o objetivo do make/makefile;
- insmod: Carregar o módulo no *kernel*.

2.4 BOA WEBSERVER

Nesta pesquisa foi utilizado o webserver (servidor web) BOA, pois garante velocidade e segurança. Segundo a documentação referente ao BOA (*Cf.*, BOA Home Page), testes já realizados mostraram que ele é capaz de lidar com milhares de acessos por segundo em um Pentium 300 MHz e dezenas de *clocks* por segundo em um 386/SX 20 MHz.

3. ARQUITETURA DO KERNEL

O *kernel* do Linux é modular e possibilita a carga de módulos em tempo de execução, através de comandos do sistema (insmod, rmmod, modprobe, etc). Isso significa que não é necessário gerar uma nova imagem para carregar um *driver* de dispositivo. Basta carregá-lo dinamicamente.

4. INTERFACE COM DRIVERS DE DISPOSITIVO

Em um sistema Linux assim como no Unix tudo é arquivo. Os *device drivers* ficam no diretório "/dev". Cada arquivo dentro de "/dev" provê acesso a um determinado *hardware*. Então, o acesso ao *hardware* é feito através da manipulação de arquivos.

A figura 7 representa a comunicação de um processo de usuário comunicando com o sistema de arquivos e esse com o *driver* do dispositivo.

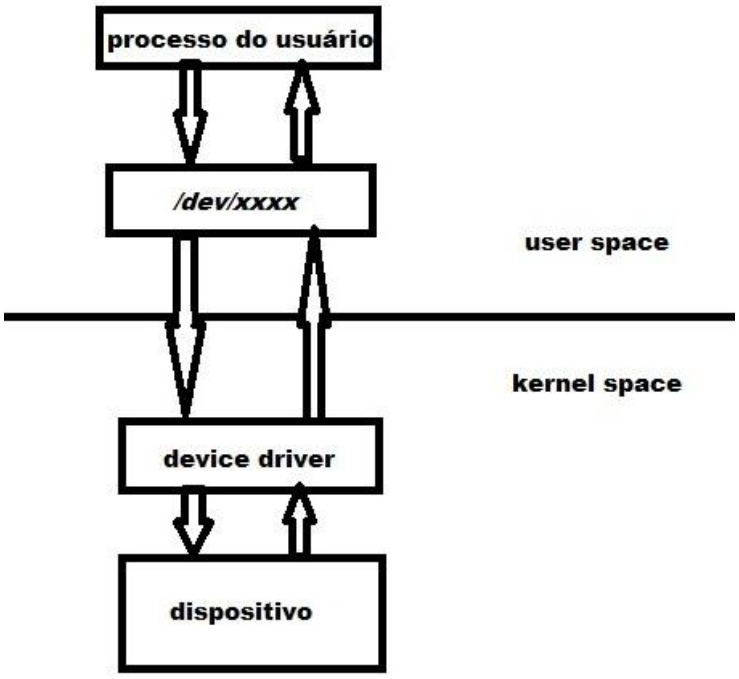


Figura 7. Interface de um processo do usuário com o *device driver*.

Dois valores são indicados para o *kernel*, para qual o dispositivo os *device drivers* estão mapeados. São eles: *major number*, que associa o arquivo ao *device drivers* e *minor number* que indica o número do seu dispositivo.

Conforme exemplificado na figura 8, "253" é o *major number* relativo ao *driver* do GPIO e (0, 1, 2, 3) são *minor numbers*, que indicam o número do dispositivo GPIO no qual o arquivo está mapeado.

Na listagem das permissões dos arquivos, o caractere "c", que é exibido antes de tais permissões, refere-se que é um *device drivers* do tipo char.

$ **ls -l ~/dev**
total 4
crw-rw-rw- 1 root root 253, 0 Jan 27 2010 /dev/gpio0
crw-rw-rw- 1 root root 253, 1 Jan 27 2010 /dev/gpio1
crw-rw-rw- 1 root root 253, 2 Jan 27 2010 /dev/gpio2
crw-rw-rw- 1 root root 253, 3 Jan 27 2010 /dev/gpio3

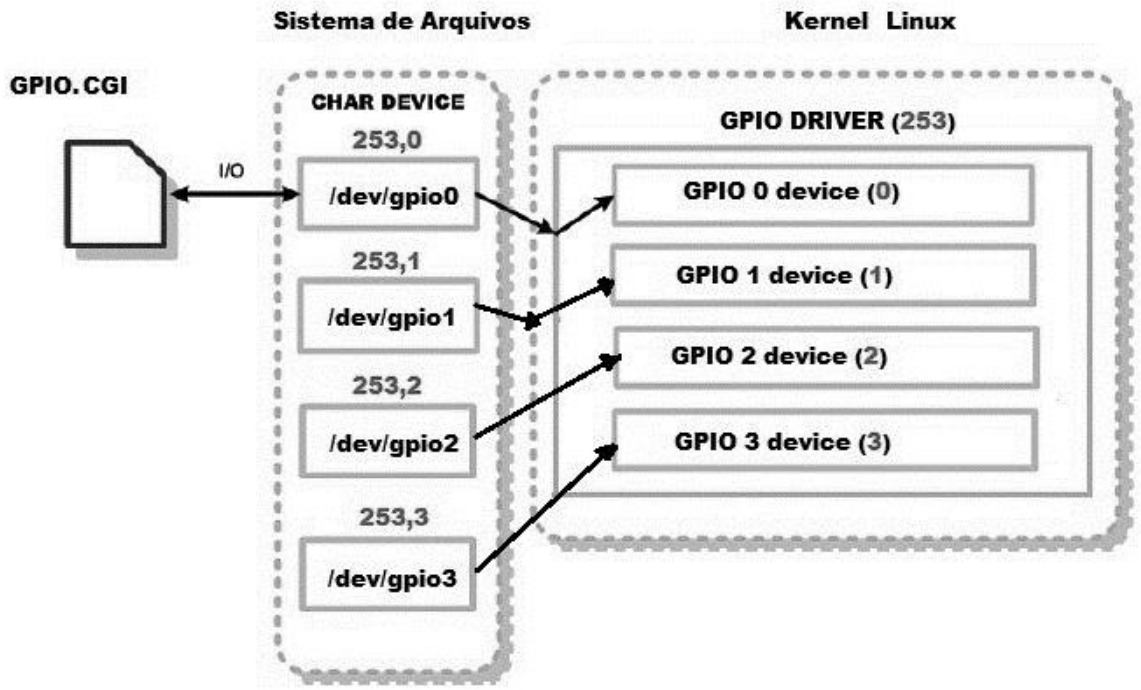


Figura 8. Números *major* e *minor* do *device* GPIO.

Neste projeto será desenvolvido um módulo *driver* para controlar o *status* das portas GPF0 à GPF7 (*Cf.*, Anexo II), de um ARM (S3C2440) que tem seus pinos disponibilizados no conector "CON4" (pinos 9 a 16) do kit mini2440 (FriendlyARM).

5. PROJETO : CODIFICAÇÃO

O módulo deve ser desenvolvido na linguagem C, pois será necessário incluir algumas bibliotecas do *kernel* em seu código:

```
#include <linux/module.h> //contem definições dos símbolos e funções necessárias por
                          //módulos que são "Carregáveis"
#include "linux/fs.h"  //funções para acesso a sistema de arquivos
#include "linux/cdev.h"  //funções para registrar dispositivos
#include "linux/device.h"  //funções para controle de dispositivos
#include "linux/kernel.h" //funçoes do kernel
#include "asm/uaccess.h" //funçoes de acesso do usuario
#include "asm/io.h" //funções de entrada e saída
#include "linux/slab.h" //funções de alocação de memória
```

Figura 9. Bibliotecas do kernel.

São variáveis importantes para o projeto:

```
#define DEVICE_NAME     "gpio" //nome do dispositivo
#define NUM_IOS        8 //numero de portas GPF0 a GPF7
#define GPIO_UP        1
#define GPIO_DOWN      0
#define GPF_BASE    0xFB000050 //endereço virtual referente a porta GPF
#define GPFCON      GPF_BASE //endereço de configuração
#define GPFDAT      GPF_BASE + 4 //endereço dos dados
#define CLEAR_PORT_MASK     0xFFFFFFFF  //mascara para utilização de todos
                                        //os endereços da porta
#define SET_WRITE_PORT_MASK 0x55555555 //mascara que configura todas as portas
                                       //como saída
```

Figura 10. Variáveis.

Na função de inicialização do *driver* deve ser definido o *major number* que estará associado a este.

Pode ser requisitado ao *kernel*, gerar um número dinamicamente com a função alloc_chrdev_region(&DEVICE_TYPE,MINOR_NUM_INIT,NUM_IOS, DEVICE_NAME). As outras definições são:

- DEVICE_TYPE: É o ponteiro que vai receber o endereço do qual foi alocado o *driver*. Este deve ser do tipo “dev_t” (device descriptor type) que foi definido em cdev.h;

- DEVICE_NAME: passa para o *kernel* o nome do dispositivo;

- MINOR_NUM_INIT: indica para o *kernel* qual vai ser o *minor number* do primeiro dispositivo;

- NUM_IOS: número de dispositivos a serem alocados.

Através da função cdev_init() associa-se a estrutura do *device* às funções de operação do arquivo. A função cdev_add() associa-se a estrutura do *device* ao *major* e *minor numbers*.

Desta forma, quando for realizada uma operação de leitura ou escrita no arquivo do dispositivo, o *kernel* acessará a estrutura cdev associada a este dispositivo (*major e minor numbers*)

Para que seja criado um nó (arquivo) no diretório “/dev” utiliza-se a função “device_create()”(“uaccess.h”) caso contrário, deve-se fazer manualmente através do comando “mknod” no terminal.

```
int __init gpio_init(void)
{
    int ret, i;

    /* request device major number */
    if ((ret = alloc_chrdev_region(&gpio_dev_number, 0, NUM_IOS, DEVICE_NAME) < 0)) {
        printk(KERN_DEBUG "Error registering device!\n");
        return ret;
    }

    /* init gpio GPIO port */
    initIoPort();

    /* init each io device */
    for (i = 0; i < NUM_IOS; i++) {

        /* init io status */
        gpio_dev[i].number = i;
        gpio_dev[i].status = GPIO_UP;

        /* connect file operations to this device */
        cdev_init(&gpio_dev[i].cdev, &gpio_fops);
        gpio_dev[i].cdev.owner = THIS_MODULE;

        /* connect major/minor numbers */
        if ((ret = cdev_add(&gpio_dev[i].cdev, (gpio_dev_number + i), 1))) {
            printk(KERN_DEBUG "Error adding device!\n");
            return ret;
        }

        /* init io status */
        changeIoStatus(gpio_dev[i].number, GPIO_UP);

    /* send uevent to udev to create /dev node */
        device_create(gpio_class, NULL, MKDEV(MAJOR(gpio_dev_number), i), NULL, "gpio%d", i);
    }
    printk("GPIO driver initialized.\n");
    return 0;
```

Figura 11. Função init do GPIO.

O módulo deve conter uma função para ser removido da memória, chamada função de "*cleanup*". Basicamente o processo inverso da função de inicialização.

A função "device_destroy()" remove os nós criados no sistema de arquivos, "/dev/gpio#", e a função "unregister_chrdev_region()" desaloca o dispositivo de seu *major number*.

```
int __exit gpio_exit(void)
{
    int i;

    /* release major number */
    unregister_chrdev_region(gpio_dev_number, NUM_IOS);

    /* delete devices */
    for (i = 0; i < NUM_IOS; i++) {
        device_destroy(gpio_class, MKDEV(MAJOR(gpio_dev_number), i));
        cdev_del(&gpio_dev[i].cdev);
    }

    /* destroy class */
    class_destroy(gpio_class);

    printk("Exiting gpio driver.\n");
    return 0;
}
```

Figura 12. Função exit do GPIO.

As funções “module_init(gpio_init)” e “module_exit(gpio_exit)” devem estar no código para indicar ao *kernel* quais são as funções de inicialização e “*cleanup*” , como podemos ver a init na figura 11 e a exit na figura 12.

Utilizando as macros __raw_readl(address) é possível ler uma variável do tipo inteiro de 4 bytes da memória virtual e __raw_writel(data, address) para escrever um inteiro de 4 bytes na memória virtual. Então, através delas criam-se as funções de configuração e de *status* da porta como pode ser visualizado na figura 13.

```c
/* initialize io port - GPF */
void initIoPort(void)
{
    void __iomem *base = (void __iomem *)GPFCON;

    u32 port = __raw_readl(base);

    port &= ~CLEAR_PORT_MASK;
    port |= SET_WRITE_PORT_MASK;

    __raw_writel(port, base);
}

/* change io status */
void changeIoStatus(int io_num, int status)
{
    void __iomem *base = (void __iomem *)GPFDAT;
    u32 mask, data;

    data = __raw_readl(base);

    printk("\nSeting port:%1d status to:%1d\n", io_num, status);

    mask = 0x01 << io_num;

    switch (status) {
        case GPIO_UP:
            mask = ~mask;
            data &= mask;
            break;
        case GPIO_DOWN:
            data |= mask;
            break;
    }
    __raw_writel(data, base);
}
```

Figura 13. Funções da porta E/S.

Para comunicar com o sistema de arquivos são utilizadas duas funções, “copy_to_user()“ e “copy_from_user()”, com estas, é possível enviar e receber informações da aplicação.

```c
/* read io status */
ssize_t gpio_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    struct gpio_device *gpio_devp = file->private_data;

    if (gpio_devp->status == GPIO_UP) {
        if (copy_to_user(buf, "1", 1))
            return -EIO;
    }
    else {
        if (copy_to_user(buf, "0", 1))
            return -EIO;
    }

    return 1;
}

ssize_t gpio_write(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    struct gpio_device *gpio_devp = file->private_data;
    char kbuf = 0;

    if (copy_from_user(&kbuf, buf, 1)) {
        return -EFAULT;
    }

    if (kbuf == '1') {
        changeIoStatus(gpio_devp->number, GPIO_UP);
        gpio_devp->status = GPIO_UP;
    }
    else if (kbuf == '0') {
        changeIoStatus(gpio_devp->number, GPIO_DOWN);
        gpio_devp->status = GPIO_DOWN;
    }

    return count;
}
```

Figura 14. Funções para enviar e receber informações da aplicação.

O código fonte completo do *device driver* GPIO.c encontra-se no Anexo I.

Para compilar o código fonte, este é o *Makefile*:

```
KDIR := /home/user/arm/linux-2.6.32.2/
PWD  := $(shell pwd)

obj-m += gpio.o

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
    @rm -Rf *.o *.ko *.mod.c modules.order Module.symvers
```

Figura 15. Makefile.

A variável KDIR recebe o diretório no qual o *kernel* do FriendlyARM foi extraído.

Nota-se que não é necessário configurar o compilador "arm-linux-gcc", pois o *kernel* já foi compilado com este. Mesmo assim, deve-se exportar o diretório no qual foi extraído o compilador para que este possa ser utilizado:

export PATH=/diretório/do/compilador:$PATH;
$make;

Obs: o *Makefile* e o código fonte do módulo devem estar no mesmo diretório.

Para verificar se o módulo foi corretamente compilado utiliza-se o comando "file":

$file gpio.ko

A mensagem abaixo deve ser exibida:

gpio.ko: ELF 32-bit LSB relocatable, ARM, version 1 (SYSV), not stripped;

Com o módulo *driver* compilado ele está pronto para ser copiado para o *kit* e inserido no *kernel* através do comando:

$insmod gpio.ko

A mensagem abaixo deve ser exibida:

Initializing GPIO *driver*.

Para configurar o *status* da porta basta inserir o caractere "0" ou "1" no arquivo referente à mesma. Isto pode ser feito por *software* ou com um simples comando "echo":

echo 1 > /dev/gpio0

A mensagem abaixo deve aparecer:

Seting port: 0 status to: 1

A seguir é apresentado o funcionamento da porta GPIO para controlar dois motores DC que movem um pequeno carro.

Para o controle dos motores é necessário um *driver* de potência (os pinos da porta GPIO não tem corrente nem tensão suficiente para alimentar motores) e um circuito de ponte H para controlar o sentido dos motores (horário/anti-horário). Felizmente já existem alguns CI's que fazem as duas coisas, neste caso, foi utilizado o L298N (*Cf.*, Anexo III), que controla dois motores DC de até 46 volts. O esquema elétrico pode ser visualizado na figura 16.

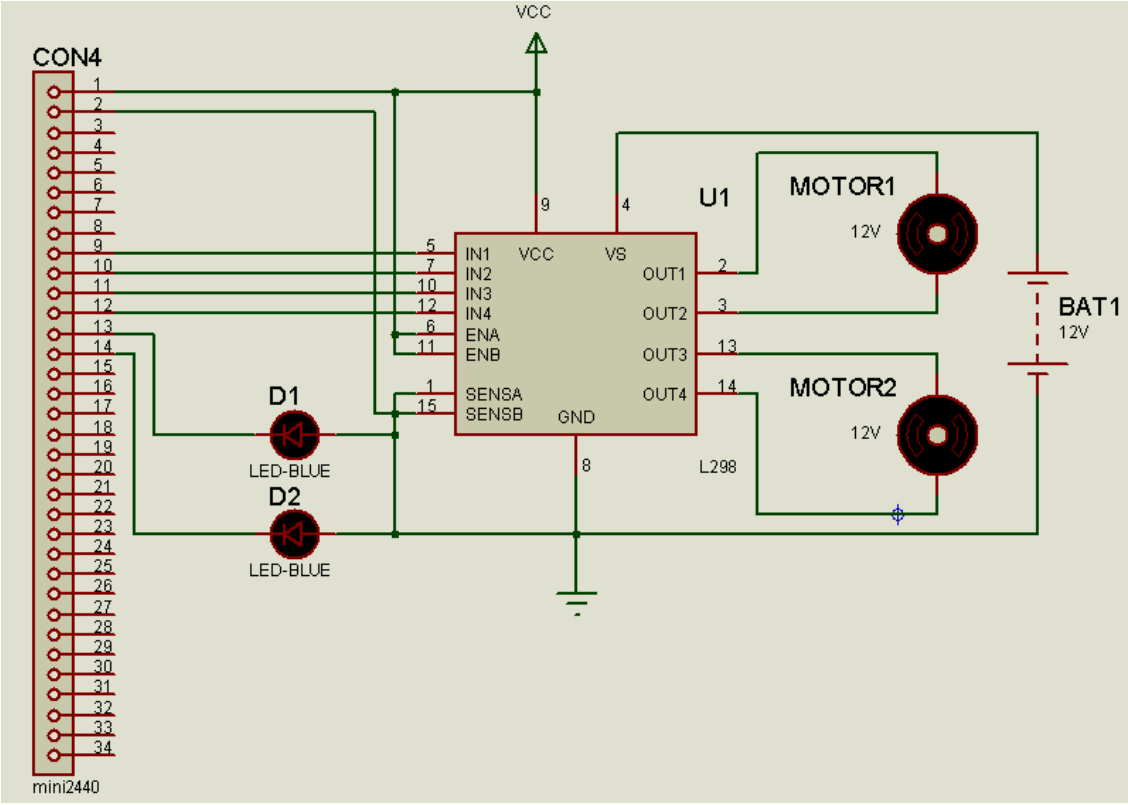


Figura 16. Esquema elétrico do projeto.

No *kit* foi carregado o *webserver* (BOA) no qual foi armazenado um código escrito em HTML com inserção de scripts em javascript, que utiliza um CGI com funções para

configurar o *status* da porta GPIO, dessa forma será possível controlar o carro por um navegador WEB.

A figura 17 mostra o código HTML desenvolvido.

```
<html>
<head>
<script type="text/javascript">
var last;
function callCGI(event) {
  var keyCode = event.which;
    if (keyCode == undefined)
      keyCode = event.keyCode;
  var xh = new XMLHttpRequest();
  xh.open("GET", "gpio.cgi?Var1="+keyCode, false);
  xh.send(null);
  last = keyCode;
}
function callCGIp(event) {
  var keyCode = event.which;
    if (keyCode == undefined)
      keyCode = event.keyCode;

  var xh = new XMLHttpRequest();
  xh.open("GET", "gpio.cgi?Var1="+keyCode*2, false);
  xh.send(null);

}
</script>

</head>

<body onkeydown="callCGI(event);" onkeyup="callCGIp(event);"/>

</body>

</html>
```

Figura 17. Index.html - HTML

No HTML da figura 17, foram utilizadas funções para acesso às setas direcionais do teclado e também à chamada do código CGI (Figura 18) com o evento da tecla pressionada.

```
#!/bin/sh

case $QUERY_STRING in
    *frente*)
        /bin/echo "1" > /dev/gpio0
        /bin/echo "0" > /dev/gpio1
        ;;
    *tras*)
        /bin/echo "0" > /dev/gpio0
        /bin/echo "1" > /dev/gpio1
        ;;
    *parar*)
        /bin/echo "1" > /dev/gpio0
        /bin/echo "1" > /dev/gpio1
        ;;
    *esquerda*)
        /bin/echo "1" > /dev/gpio2
        /bin/echo "0" > /dev/gpio3
        ;;
    *direita*)
        /bin/echo "0" > /dev/gpio2
        /bin/echo "1" > /dev/gpio3
        ;;
    *desvira*)
        /bin/echo "1" > /dev/gpio2
        /bin/echo "1" > /dev/gpio3
        ;;
esac

echo "Content-type: text/html;"
echo
/bin/cat index.html

exit 0
```

Figura 18. gpio.cgi - CGI

Conforme já mencionado (*Cf.*, acima, p.38), foram projetados quatro nós ao dispositivo GPIO, cada qual mapeado conforme sua funcionalidade específica. Através do CGI foram enviadas combinações de “0” e “1” para a porta GPIO, que por sua vez, as interpretou da forma na qual está mapeada.

A figura 19 e a Figura 20 apresentam as fotos com todo o projeto desenvolvido.

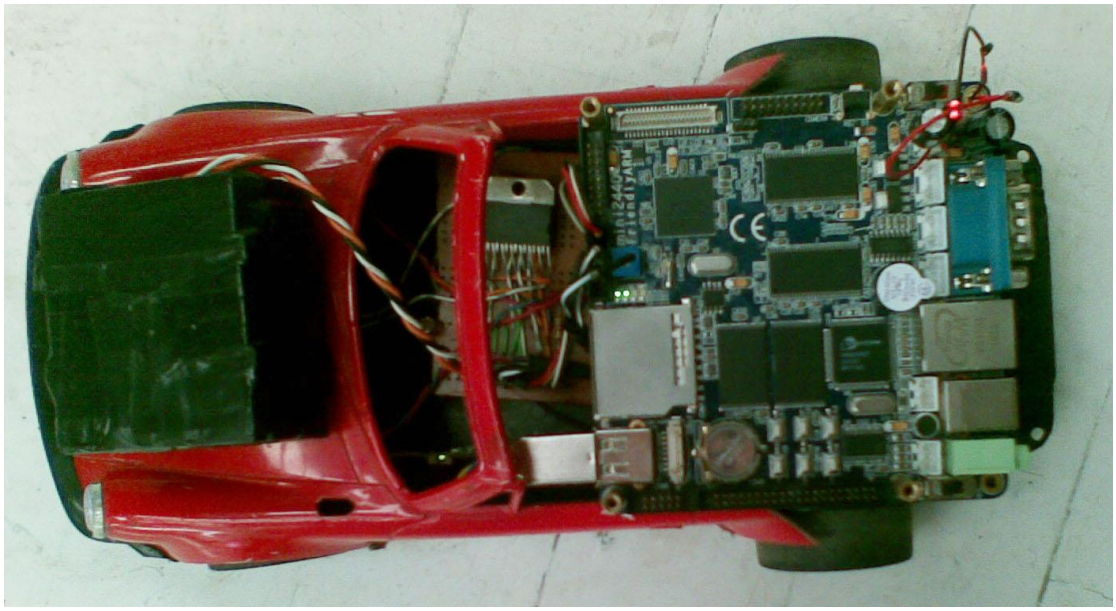


Figura 19. Projeto utilizando o kit FriendlyARM Mini2440 controlado via webserver.

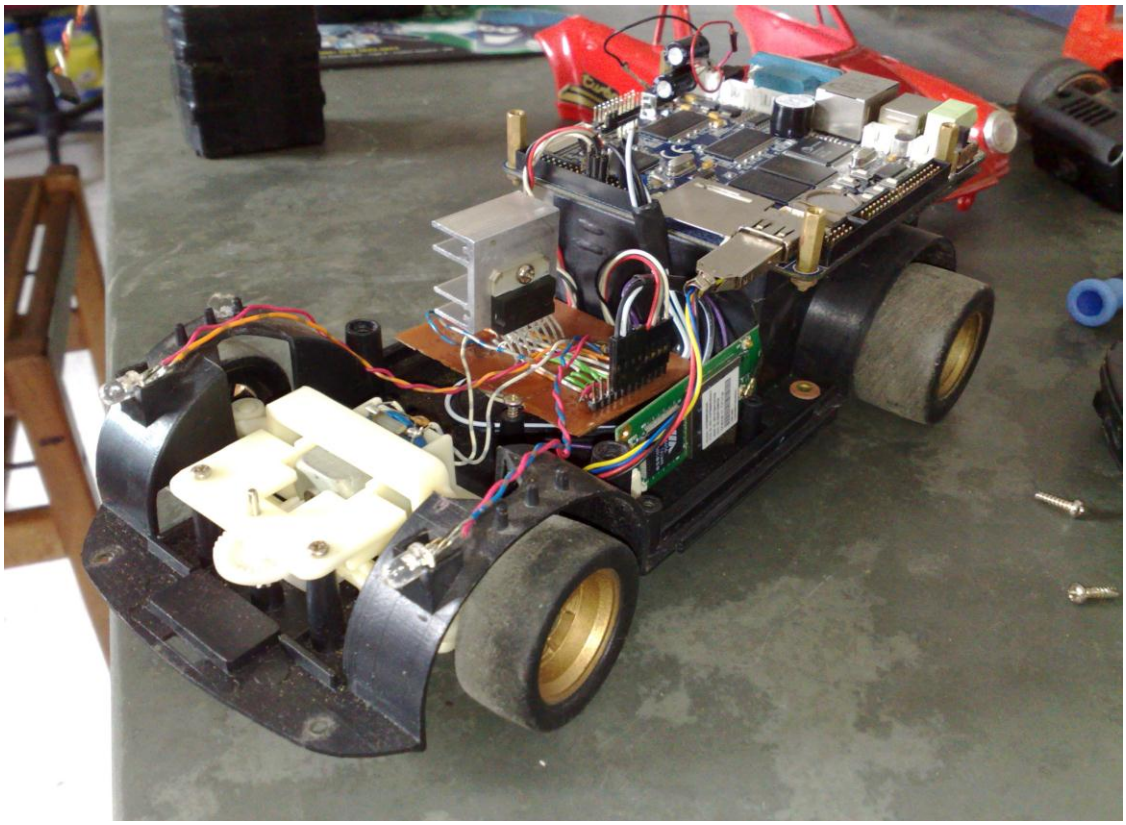


Figura 20. Projeto utilizando o kit FriendlyARM Mini2440.

CAPÍTULO 4. TESTES

Com o objetivo de constatar as funcionalidades do *hardware* e do *software* desenvolvidos, diversos testes foram realizados. O carro criado foi controlado via rede sem fio (*wireless*) e funcionou perfeitamente.

Utilizando *device drivers* em Linux embarcado com um kit de desenvolvimento, percebeu-se que o acesso aos dispositivos via *device driver* é rápido e é possível garantir a confiabilidade do sistema. Além disso, a vantagem de utilizar o Linux embarcado foi observada no seu desempenho, tempo de resposta na chamada das funções, tamanho do *kernel* e as bibliotecas disponíveis. Por ser um sistema operacional livre, também possui muitos contribuintes, sendo que várias atualizações sempre são realizadas, acrescentando novas bibliotecas e funções, facilitando o caminho para o desenvolvimento de novos projetos.

O *kit* Friendly ARM utilizado disponibiliza várias ferramentas, dentre elas, o depurador de compilação ARM-GCC.

Com relação às vantagens de se utilizar o *kit*, este apresentou através da facilidade em desenvolver novas aplicações que necessitam de maior poder de processamento, a possibilidade de acrescentar novos periféricos e modificar o *hardware* dos existentes, para adequá-los melhor a cada aplicação desejada. A gravação da imagem no *kit*, bem como, a inserção de novos *device drivers* é feito de uma forma bem mais ágil. Depois de compilado com o ARM-GCC, pode-se inseri-lo na placa via USB. Isso facilita muito o processo, não precisando regravar todo o *kernel* caso seja necessário modificar algo do *device drivers*, do HTML ou CGI embarcados.

Para comunicação via *wireless*, foram utilizados códigos escritos em HTML e CGI. Observou-se que a utilização do Linux traz vantagens na comunicação, uma vez que foram utilizados no CGI, comandos diretos para o *device driver* (GPIO). Como exemplo, tem-se o comando /bin/echo "1" > /dev/gpio0 que escreve o valor “1” para o device gpio0, no qual, cada valor escrito (0 e 1) tem uma funcionalidade, possibilitando que o carro possa seguir para frente, para trás, parar, virar a direita ou a esquerda.

CAPÍTULO 5. CONCLUSÃO

Atualmente como consequência da demanda, a evolução tecnológica cresce cada vez mais. Daí, a necessidade de criar projetos embarcados otimizados que ofereçam altos níveis de abstração para o usuário final. Por isso, busca-se manter a simplicidade em sua codificação (quando possível), para gerar as melhorias necessárias a tais projetos.

Os *device drivers* tem um papel essencial na comunicação do *hardware* externo ou interno com os demais dispositivos periféricos, por isso, é de suma importância o seu desenvolvimento para os sistemas embarcados.

Foi observado durante o desenvolvimento deste projeto, que a velocidade de acesso, bem como, o tempo de respostas de comandos foram eficazes. As funções de escrita e leitura para interface com o usuário, foram criadas naturalmente no *device drivers*. Quanto às bibliotecas, muitas já possuem uma série de funcionalidades e variáveis organizadas. Cabe ao desenvolvedor aproveita-las para facilitar e permitir uma padronização em sua estrutura. Todo *device driver* possui padronização de suas funções como: inicialização, finalização, leitura, escrita, nome do *device*, *major* e *minor numbers* e algumas outras definições opcionais como, por exemplo, o tipo da licença.

De acordo com esta pesquisa, ao utilizar o Linux garantiu-se uma série de benefícios como a portabilidade, a escalabilidade e *kernel* reduzido. Por ser um sistema operacional de licença livre e possuindo uma gama de bibliotecas já desenvolvidas pela comunidade de *software* livre, proporcionou a este estudo, a liberdade de aprofundar na esfera do funcionamento de um programa e, de adaptá-lo ao objetivo deste trabalho. De forma geral, espera-se que esta pesquisa possa contribuir de alguma forma para os estudos futuros sobre *device drivers* em Linux embarcado.

REFERÊNCIAS BIBLIOGRÁFICAS

ARM920T Datasheet Home Page, “Datasheet ARM920T.,”. *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0151c/I1004722.html*. Acesso em: 4 abril 2011.

BOA Home Page, “BOA webserver .,”. *http://www.boa.org/documentation/,Fev.2000.* Acesso em: 17 março 2011.

BOVET, Daniel P., CESATI, M. *Understanding the Linux Kernel*, O’Reilly, 3nd edition, 2005.

BARUCHI, Arthur. “*Desenvolvendo Device Drivers no Linux*”, IBM developerWorks *http://www.ibm.com/developerworks/br/local/linux/l-device-drivers/?ca=drs-#N101AF,Dez.2010.* Acesso em: 4 abril 2011.

CORBET, J., RUBINI, A. and KROAH-HARTMANK, G. *Linux Device Drivers*, O’Reilly, 3nd edition, 2005.

CUNHA,ALESSANDRO, “O que são sistemas embarcados.,” *http://www.techtraining.eng.br/conteudo/ARTIGO-SIST-EMB.pdf.* Acesso em: 11 abril 2011.

FriendlyARM Home Page, “FriendlyARM Mini2440.,”. *http://www.friendlyarm.net/products/mini2440?lang=en.* Acesso em: 4 março 2011.

FURBER, Steve. *ARM System-on-Chip Architecture (2nd Edition)*, Addison-Wesley Professional, 2nd edition, 2000.

GNU Make Manual Home Page, “Comando Make.,”. *http://www.gnu.org/software/make/manual/make.html#Overview.*

KERRISK, Michael. *The Linux Programming Interface,* No Starch Press, 1nd edition, 2010.

NOERGAARD, Tammy. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers,* Newnes, 2005.

RAGHAVAN, P.; LAD, Amol; NEELAKANDAN, Sriram, *Embedded Linux System Design and Development*, Auerbach Publications, 2006

TANENBAUM, ANDREW S. *Organização Estruturada de Computadores*, Addison-Wesley Professional, 5nd edition, 2006.

ANEXO I

DEVICE DRIVER GPIO.C

```
//Bibliotecas
#include "linux/fs.h"
#include "linux/cdev.h"
#include "linux/module.h"
#include "linux/kernel.h"
#include "linux/device.h"
#include "asm/uaccess.h"
#include "asm/io.h"
#include "linux/slab.h"
//Variáveis
#define DEVICE_NAME     "gpio"
#define NUM_IOS        8
#define GPIO_UP        1
#define GPIO_DOWN      0
#define GPF_BASE    0xFB000050
#define GPFCON      GPF_BASE
#define GPFDAT      GPF_BASE + 4
#define CLEAR_PORT_MASK     0xFFFFFFFF
#define SET_WRITE_PORT_MASK 0x55555555

/* Protótipos */
int gpio_open(struct inode *inode, struct file *file);
ssize_t gpio_read(struct file *file, char *buf, size_t count, loff_t *ppos);
ssize_t gpio_write(struct file *file, const char *buf, size_t count, loff_t *ppos);
int gpio_release(struct inode *inode, struct file *file);

/* Estrutura E/S device */
struct gpio_device {
   int number;
   int status;
   struct cdev cdev;
} gpio_dev[NUM_IOS];
```

```
/* Operações */
static struct file_operations gpio_fops = {
    .owner      = THIS_MODULE,
    .open       = gpio_open,
    .release    = gpio_release,
    .read       = gpio_read,
    .write      = gpio_write,
};

/* GPIO driver major number */
static dev_t gpio_dev_number;

/* Inicializar porta E/S - GPF */
void initIoPort(void)
{
    void __iomem *base = (void __iomem *)GPFCON;
    u32 port = __raw_readl(base);
    port &= ~CLEAR_PORT_MASK;
    port |= SET_WRITE_PORT_MASK;

    __raw_writel(port, base);
}

/* Mudar status da porta E/S */
void changeIoStatus(int io_num, int status)
{
    void __iomem *base = (void __iomem *)GPFDAT;
    u32 mask, data;
    data = __raw_readl(base);
    printk("\nSeting port:%1d status to:%1d\n", io_num, status);
    mask = 0x01 << io_num;
```

```
        switch (status) {
            case GPIO_UP:
                mask = ~mask;
                data &= mask;
                break;

            case GPIO_DOWN:
                data |= mask;
                break;
        }
        __raw_writel(data, base);
    }

    struct class *gpio_class;

    /* Inicialização do driver GPIO*/
    int __init gpio_init(void)
    {
        int ret, i;

        /* request device major number */
        if ((ret = alloc_chrdev_region(&gpio_dev_number, 0, NUM_IOS, DEVICE_NAME)
< 0)) {
            printk(KERN_DEBUG "Error registering device!\n");
            return ret;
        }

        /* create /sys entry */
        gpio_class = class_create(THIS_MODULE, DEVICE_NAME);

        /* Inicializar porta GPIO */
        initIoPort();
```

```
    /* init each io device */
    for (i = 0; i < NUM_IOS; i++) {

        /* init io status */
        gpio_dev[i].number = i;
        gpio_dev[i].status = GPIO_UP;

        /* connect file operations to this device */
        cdev_init(&gpio_dev[i].cdev, &gpio_fops);
        gpio_dev[i].cdev.owner = THIS_MODULE;

        /* connect major/minor numbers */
        if ((ret = cdev_add(&gpio_dev[i].cdev, (gpio_dev_number + i), 1))) {
            printk(KERN_DEBUG "Error adding device!\n");
            return ret;
        }

        /* init io status */
        changeIoStatus(gpio_dev[i].number, GPIO_UP);

      /* send uevent to udev to create /dev node */
        device_create(gpio_class, NULL, MKDEV(MAJOR(gpio_dev_number), i),
NULL, "gpio%d", i);

    }
    printk("GPIO driver initialized.\n");
    return 0;
}

/* Finalização do driver*/
int __exit gpio_exit(void)
{
    int i;
```

```
    /* release major number */
    unregister_chrdev_region(gpio_dev_number, NUM_IOS);

    /* delete devices */
    for (i = 0; i < NUM_IOS; i++) {
        device_destroy(gpio_class, MKDEV(MAJOR(gpio_dev_number), i));
        cdev_del(&gpio_dev[i].cdev);
    }

    /* destroy class */
    class_destroy(gpio_class);

    printk("Exiting gpio driver.\n");
    return 0;
}

/* open io file */
int gpio_open(struct inode *inode, struct file *file)
{
    struct gpio_device *gpio_devp;

    /* get cdev struct */
    gpio_devp = container_of(inode->i_cdev, struct gpio_device, cdev);

    /* save cdev pointer */
    file->private_data = gpio_devp;

    /* return OK */
    return 0;
}

/* close io file */
int gpio_release(struct inode *inode, struct file *file)
```

```
{
    /* return OK */
    return 0;
}

/* read io status */
ssize_t gpio_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    struct gpio_device *gpio_devp = file->private_data;

    if (gpio_devp->status == GPIO_UP) {
        if (copy_to_user(buf, "1", 1))
            return -EIO;
    }
    else {
        if (copy_to_user(buf, "0", 1))
            return -EIO;
    }

    return 1;
}

ssize_t gpio_write(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    struct gpio_device *gpio_devp = file->private_data;
    char kbuf = 0;
    if (copy_from_user(&kbuf, buf, 1)) {
        return -EFAULT;
    }

    if (kbuf == '1') {
        changeIoStatus(gpio_devp->number, GPIO_UP);
        gpio_devp->status = GPIO_UP;
```

```
    }
    else if (kbuf == '0') {
        changeIoStatus(gpio_devp->number, GPIO_DOWN);
        gpio_devp->status = GPIO_DOWN;
    }

    return count;
}

module_init(gpio_init);
module_exit(gpio_exit);

MODULE_LICENSE("GPL");
```

ANEXO II

S3C2440(ARM) DATASHEET

1 PRODUCT OVERVIEW

INTRODUCTION

This user's manual describes SAMSUNG's S3C2440A 16/32-bit RISC microprocessor. SAMSUNG's S3C2440A is designed to provide hand-held devices and general applications with low-power, and high-performance micro-controller solution in small die size. To reduce total system cost, the S3C2440A includes the following components.

The S3C2440A is developed with ARM920T core, 0.13um CMOS standard cells and a memory complier. Its low-power, simple, elegant and fully static design is particularly suitable for cost- and power-sensitive applications. It adopts a new bus architecture known as Advanced Micro controller Bus Architecture (AMBA).

The S3C2440A offers outstanding features with its CPU core, a 16/32-bit ARM920T RISC processor designed by Advanced RISC Machines, Ltd. The ARM920T implements MMU, AMBA BUS, and Harvard cache architecture with separate 16KB instruction and 16KB data caches, each with an 8-word line length.

By providing a complete set of common system peripherals, the S3C2440A minimizes overall system costs and eliminates the need to configure additional components. The integrated on-chip functions that are described in this document include:

- Around 1.2V internal, 1.8V/2.5V/3.3V memory, 3.3V external I/O microprocessor with 16KB I-Cache/16KB D-Cache/MMU
- External memory controller (SDRAM Control and Chip Select logic)
- LCD controller (up to 4K color STN and 256K color TFT) with LCD-dedicated DMA
- 4-ch DMA controllers with external request pins
- 3-ch UARTs (IrDA1.0, 64-Byte Tx FIFO, and 64-Byte Rx FIFO)
- 2-ch SPIs
- IIC bus interface (multi-master support)
- IIS Audio CODEC interface
- AC'97 CODEC interface
- SD Host interface version 1.0 & MMC Protocol version 2.11 compatible
- 2-ch USB Host controller / 1-ch USB Device controller (ver 1.1)
- 4-ch PWM timers / 1-ch Internal timer / Watch Dog Timer
- 8-ch 10-bit ADC and Touch screen interface
- RTC with calendar function
- Camera interface (Max. 4096 x 4096 pixels input support. 2048 x 2048 pixel input support for scaling)
- 130 General Purpose I/O ports / 24-ch external interrupt source
- Power control: Normal, Slow, Idle and Sleep mode
- On-chip clock generator with PLL

BLOCK DIAGRAM

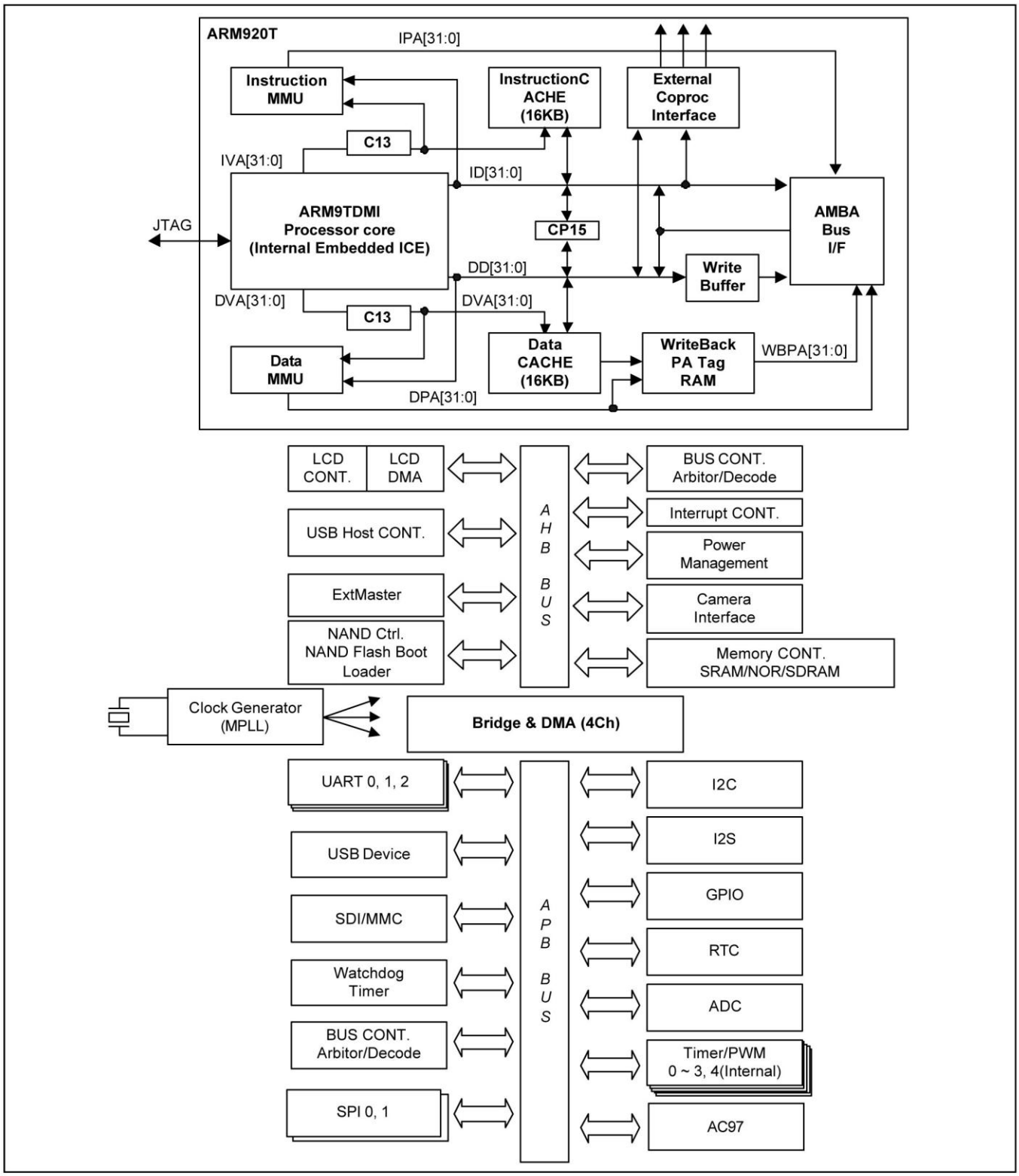


Figure 1-1. S3C2440A Block Diagram

9 I/O PORTS

OVERVIEW

S3C2440A has 130 multi-functional input/output port pins and there are eight ports as shown below:

— Port A(GPA): 25-output port
— Port B(GPB): 11-input/out port
— Port C(GPC): 16-input/output port
— Port D(GPD): 16-input/output port
— Port E(GPE): 16-input/output port
— Port F(GPF): 8-input/output port
— Port G(GPG): 16-input/output port
— Port H(GPH): 9-input/output port
— Port J(GPJ): 13-input/output port

Each port can be easily configured by software to meet various system configurations and design requirements. You have to define which function of each pin is used before starting the main program. If a pin is not used for multiplexed functions, the pin can be configured as I/O ports.

Initial pin states are configured seamlessly to avoid problems.

PORT F CONTROL REGISTERS (GPFCON, GPFDAT)

If GPF0–GPF7 will be used for wake-up signals at power down mode, the ports will be set in interrupt mode.

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configures the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undef.
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x000
Reserved	0x5600005c	–	–	–

GPFCON	Bit	Description
GPF7	[15:14]	00 = Input 01 = Output 10 = EINT[7] 11 = Reserved
GPF6	[13:12]	00 = Input 01 = Output 10 = EINT[6] 11 = Reserved
GPF5	[11:10]	00 = Input 01 = Output 10 = EINT[5] 11 = Reserved
GPF4	[9:8]	00 = Input 01 = Output 10 = EINT[4] 11 = Reserved
GPF3	[7:6]	00 = Input 01 = Output 10 = EINT[3] 11 = Reserved
GPF2	[5:4]	00 = Input 01 = Output 10 = EINT2] 11 = Reserved
GPF1	[3:2]	00 = Input 01 = Output 10 = EINT[1] 11 = Reserved
GPF0	[1:0]	00 = Input 01 = Output 10 = EINT[0] 11 = Reserved

GPFDAT	Bit	Description
GPF[7:0]	[7:0]	When the port is configured as an input port, the corresponding bit is the pin state. When the port is configured as an output port, the pin state is the same as the corresponding bit. When the port is configured as functional pin, the undefined value will be read.

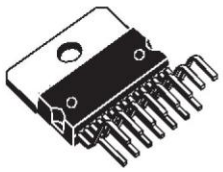
GPFUP	Bit	Description
GPF[7:0]	[7:0]	0: The pull up function attached to the corresponding port pin is enabled. 1: The pull up function is disabled.

ANEXO III

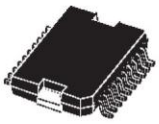
L298N DATASHEET

DUAL FULL-BRIDGE DRIVER

- OPERATING SUPPLY VOLTAGE UP TO 46 V
- TOTAL DC CURRENT UP TO 4 A
- LOW SATURATION VOLTAGE
- OVERTEMPERATURE PROTECTION
- LOGICAL "0" INPUT VOLTAGE UP TO 1.5 V (HIGH NOISE IMMUNITY)



Multiwatt15



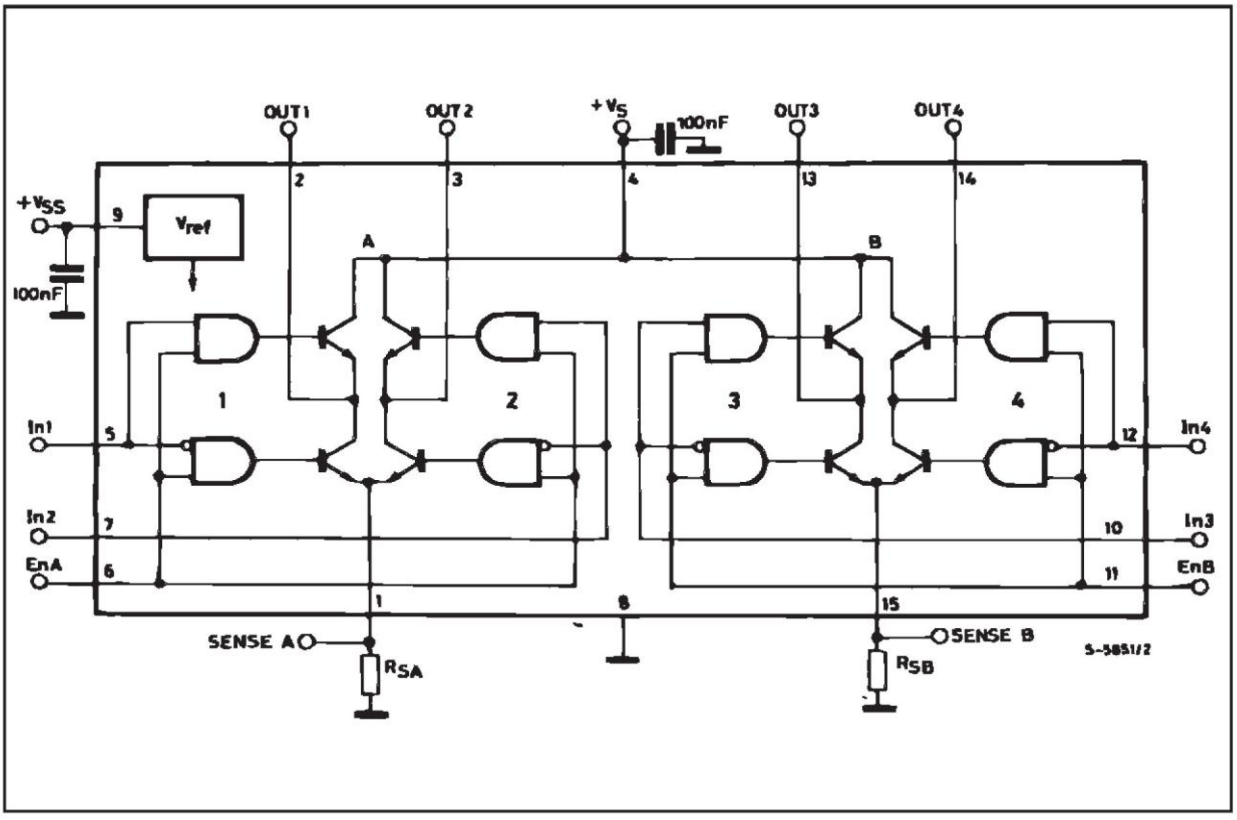
PowerSO20

ORDERING NUMBERS : L298N (Multiwatt Vert.)
L298HN (Multiwatt Horiz.)
L298P (PowerSO20)

DESCRIPTION

The L298 is an integrated monolithic circuit in a 15-lead Multiwatt and PowerSO20 packages. It is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepping motors. Two enable inputs are provided to enable or disable the device independently of the input signals. The emitters of the lower transistors of each bridge are connected together and the corresponding external terminal can be used for the connection of an external sensing resistor. An additional supply input is provided so that the logic works at a lower voltage.

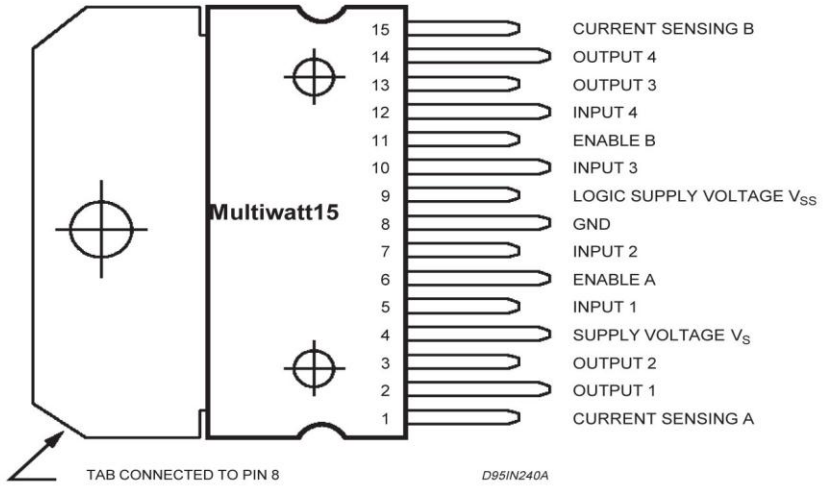
BLOCK DIAGRAM

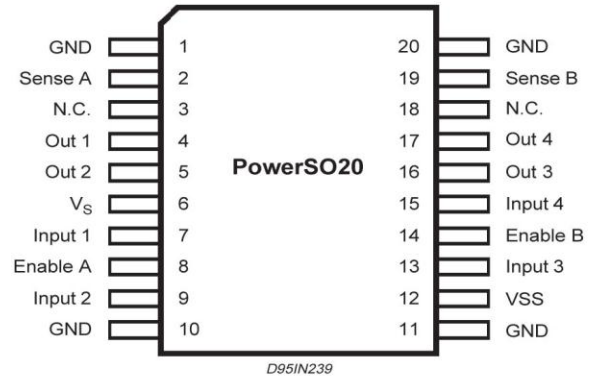


ABSOLUTE MAXIMUM RATINGS

Symbol	Parameter	Value	Unit
V_S	Power Supply	50	V
V_{SS}	Logic Supply Voltage	7	V
V_I, V_{en}	Input and Enable Voltage	–0.3 to 7	V
I_O	Peak Output Current (each Channel) – Non Repetitive (t = 100µs) –Repetitive (80% on –20% off; t_{on} = 10ms) –DC Operation	 3 2.5 2	 A A A
V_{sens}	Sensing Voltage	–1 to 2.3	V
P_{tot}	Total Power Dissipation (T_{case} = 75°C)	25	W
T_{op}	Junction Operating Temperature	–25 to 130	°C
T_{stg}, T_j	Storage and Junction Temperature	–40 to 150	°C

PIN CONNECTIONS (top view)





THERMAL DATA

Symbol	Parameter		PowerSO20	Multiwatt15	Unit
$R_{th\ j\text{-}case}$	Thermal Resistance Junction-case	Max.	–	3	°C/W
$R_{th\ j\text{-}amb}$	Thermal Resistance Junction-ambient	Max.	13 (*)	35	°C/W

(*) Mounted on aluminum substrate

PIN FUNCTIONS (refer to the block diagram)

MW.15	PowerSO	Name	Function
1;15	2;19	Sense A; Sense B	Between this pin and ground is connected the sense resistor to control the current of the load.
2;3	4;5	Out 1; Out 2	Outputs of the Bridge A; the current that flows through the load connected between these two pins is monitored at pin 1.
4	6	V_S	Supply Voltage for the Power Output Stages. A non-inductive 100nF capacitor must be connected between this pin and ground.
5;7	7;9	Input 1; Input 2	TTL Compatible Inputs of the Bridge A.
6;11	8;14	Enable A; Enable B	TTL Compatible Enable Input: the L state disables the bridge A (enable A) and/or the bridge B (enable B).
8	1,10,11,20	GND	Ground.
9	12	VSS	Supply Voltage for the Logic Blocks. A100nF capacitor must be connected between this pin and ground.
10; 12	13;15	Input 3; Input 4	TTL Compatible Inputs of the Bridge B.
13; 14	16;17	Out 3; Out 4	Outputs of the Bridge B. The current that flows through the load connected between these two pins is monitored at pin 15.
–	3;18	N.C.	Not Connected

ELECTRICAL CHARACTERISTICS (V_S = 42V; V_{SS} = 5V, T_j = 25°C; unless otherwise specified)

Symbol	Parameter	Test Conditions	Min.	Typ.	Max.	Unit
V_S	Supply Voltage (pin 4)	Operative Condition	V_{IH} +2.5		46	V
V_{SS}	Logic Supply Voltage (pin 9)		4.5	5	7	V
I_S	Quiescent Supply Current (pin 4)	V_{en} = H; I_L = 0 V_i = L V_i = H		13 50	22 70	mA mA
		V_{en} = L V_i = X			4	mA
I_{SS}	Quiescent Current from V_{SS} (pin 9)	V_{en} = H; I_L = 0 V_i = L V_i = H		24 7	36 12	mA mA
		V_{en} = L V_i = X			6	mA
V_{iL}	Input Low Voltage (pins 5, 7, 10, 12)		–0.3		1.5	V
V_{iH}	Input High Voltage (pins 5, 7, 10, 12)		2.3		VSS	V
I_{iL}	Low Voltage Input Current (pins 5, 7, 10, 12)	V_i = L			–10	µA
I_{iH}	High Voltage Input Current (pins 5, 7, 10, 12)	Vi = H ≤ V_{SS} –0.6V		30	100	µA
V_{en} = L	Enable Low Voltage (pins 6, 11)		–0.3		1.5	V
V_{en} = H	Enable High Voltage (pins 6, 11)		2.3		V_{SS}	V
I_{en} = L	Low Voltage Enable Current (pins 6, 11)	V_{en} = L			–10	µA
I_{en} = H	High Voltage Enable Current (pins 6, 11)	V_{en} = H ≤ V_{SS} –0.6V		30	100	µA
$V_{CEsat(H)}$	Source Saturation Voltage	I_L = 1A I_L = 2A	0.95	1.35 2	1.7 2.7	V V
$V_{CEsat(L)}$	Sink Saturation Voltage	I_L = 1A (5) I_L = 2A (5)	0.85	1.2 1.7	1.6 2.3	V V
V_{CEsat}	Total Drop	I_L = 1A (5) I_L = 2A (5)	1.80		3.2 4.9	V V
V_{sens}	Sensing Voltage (pins 1, 15)		–1 (1)		2	V

ELECTRICAL CHARACTERISTICS (continued)

Symbol	Parameter	Test Conditions	Min.	Typ.	Max.	Unit
T_1 (V_i)	Source Current Turn-off Delay	0.5 V_i to 0.9 I_L (2); (4)		1.5		µs
T_2 (V_i)	Source Current Fall Time	0.9 I_L to 0.1 I_L (2); (4)		0.2		µs
T_3 (V_i)	Source Current Turn-on Delay	0.5 V_i to 0.1 I_L (2); (4)		2		µs
T_4 (V_i)	Source Current Rise Time	0.1 I_L to 0.9 I_L (2); (4)		0.7		µs
T_5 (V_i)	Sink Current Turn-off Delay	0.5 V_i to 0.9 I_L (3); (4)		0.7		µs
T_6 (V_i)	Sink Current Fall Time	0.9 I_L to 0.1 I_L (3); (4)		0.25		µs
T_7 (V_i)	Sink Current Turn-on Delay	0.5 V_i to 0.9 I_L (3); (4)		1.6		µs
T_8 (V_i)	Sink Current Rise Time	0.1 I_L to 0.9 I_L (3); (4)		0.2		µs
fc (V_i)	Commutation Frequency	I_L = 2A		25	40	KHz
T_1 (V_{en})	Source Current Turn-off Delay	0.5 V_{en} to 0.9 I_L (2); (4)		3		µs
T_2 (V_{en})	Source Current Fall Time	0.9 I_L to 0.1 I_L (2); (4)		1		µs
T_3 (V_{en})	Source Current Turn-on Delay	0.5 V_{en} to 0.1 I_L (2); (4)		0.3		µs
T_4 (V_{en})	Source Current Rise Time	0.1 I_L to 0.9 I_L (2); (4)		0.4		µs
T_5 (V_{en})	Sink Current Turn-off Delay	0.5 V_{en} to 0.9 I_L (3); (4)		2.2		µs
T_6 (V_{en})	Sink Current Fall Time	0.9 I_L to 0.1 I_L (3); (4)		0.35		µs
T_7 (V_{en})	Sink Current Turn-on Delay	0.5 V_{en} to 0.9 I_L (3); (4)		0.25		µs
T_8 (V_{en})	Sink Current Rise Time	0.1 I_L to 0.9 I_L (3); (4)		0.1		µs

1) 1)Sensing voltage can be –1 V for t ≤ 50 µsec; in steady state V_{sens} min ≥ –0.5 V.
2) See fig. 2.
3) See fig. 4.
4) The load must be a pure resistor.

Figure 1 : Typical Saturation Voltage vs. Output Current.

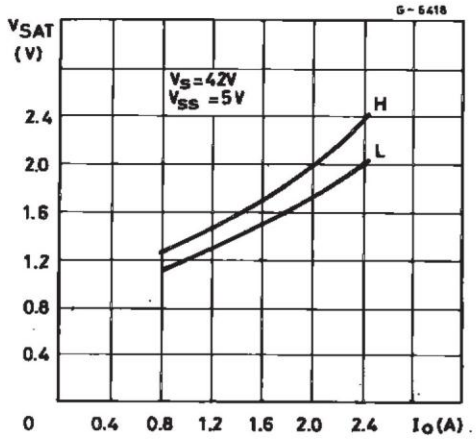
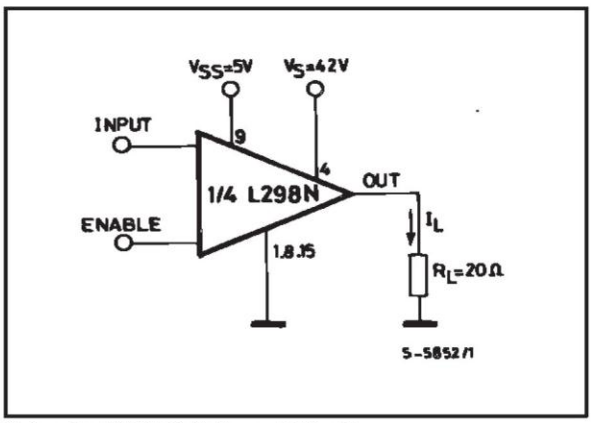


Figure 2 : Switching Times Test Circuits.



Note : For INPUT Switching, set EN = H
For ENABLE Switching, set IN = H