Descobrindo o Linux Embarcado

EMBARCADOS

2023

Aviso legal

Este eBook foi escrito com fins didáticos, e com todos os esforço para que ele ficasse o mais claro e didático possível.

O objetivo deste eBook é educar. Os autores não garantem que as informações contidas neste eBook estão totalmente completas e/ou atualizadas e não deve ser responsável por quaisquer erros ou omissões.

Os autores não serão responsabilizados para com qualquer pessoa ou entidade com relação a qualquer perda, ou dano causado, ou alegado a ser causado direta, ou indiretamente por este eBook.

Os textos foram publicados originalmente no Embarcados com a licença CC-BY-SA e foram compilados nesse material.

Se achar algum erro, ou tiver sugestões de tópicos, ou melhorias, envie um e-mail: contato@embarcados.com.br

Sobre o Embarcados

O Portal Embarcados tem como foco inspirar qualidade e inovação tecnológica, disseminando o conhecimento da área de sistemas embarcados.

Atualmente, o Portal Embarcados é um grande site com conteúdo técnico sobre **hardware**, **firmware** e assunto relacionados a **Embarcados e IoT**, e boa parte desse conteúdo é criado pela comunidade, por pessoas que compartilham seu conhecimento na área de sistemas eletrônicos. Em novembro de 2022 cerca de 190 mil pessoas visualizam páginas do Embarcados, confirmando o site como uma fonte popular de referência nos assuntos referentes a sistemas embarcados, sistemas eletrônicos e IoT (Internet das Coisas).

Estamos formando uma grande base de conhecimento acessível a todos e viabilizando a troca de informações e experiências que ajudam a crescer e desenvolver nossa capacidade como profissionais da área de sistemas embarcados. O Portal convida a todos, incluindo makers e curiosos a consultarem os textos e a compartilharem seu conhecimento através de tutoriais, videos e projetos. Envie o seu texto pro Embarcados no link.

O Portal Embarcados também realiza **eventos** com a comunidade e também **webinars**, veiculados pelo menos 2 vezes por mês com o apoio de empresas e com pessoas de notório saber em assuntos relacionados a sistemas embarcados.

- Para acessar os Webinars do Embarcados, acesse o link.
- Para acessar o evento Seminário de Sistemas Embarcados e IoT 2020, acesse o link.
- Para acessar o evento Embarcados Experience 2020, acesse o link.
- Para acessar o evento Embarcados Experience 2022, acesse o link.

A plataforma Embarcados Contest recebe competições e programas que permitem com que pessoas e empresas desenvolvam seus projetos com um suporte e atenção especial das empresas envolvidas nesses desafios. Para acessar os concursos já realizados, acesse o link.

Canal do Youtube – Lives, Webinars, Videos Técnicos e Podcasts

Possuímos uma sessão com vídeos de reviews, lives, unboxing e ensino em sistemas embarcados, e onde concentramos os vídeos de nossos eventos e Webinars passados. Toda quarta-feira a tarde, às 17:00, temos o Café com Embarcados, com entrevistas com convidados especiais que trabalham na área. Toda sexta-feira à tarde, às 17:00, temos a Bancada do Embarcados, com assuntos técnicos, relacionados a Firmware, Hardware e ferramentas de nuvem. Para acessar o EmbarcadosTV clique aqui.

Siga o Embarcados na Redes Sociais

https://www.facebook.com/osembarcados/
https://www.instagram.com/portalembarcados/
https://www.youtube.com/embarcadostv/
https://www.linkedin.com/company/embarcados/
https://twitter.com/embarcados

Sumário

Como se tornar um especialista em Linux Embarcado

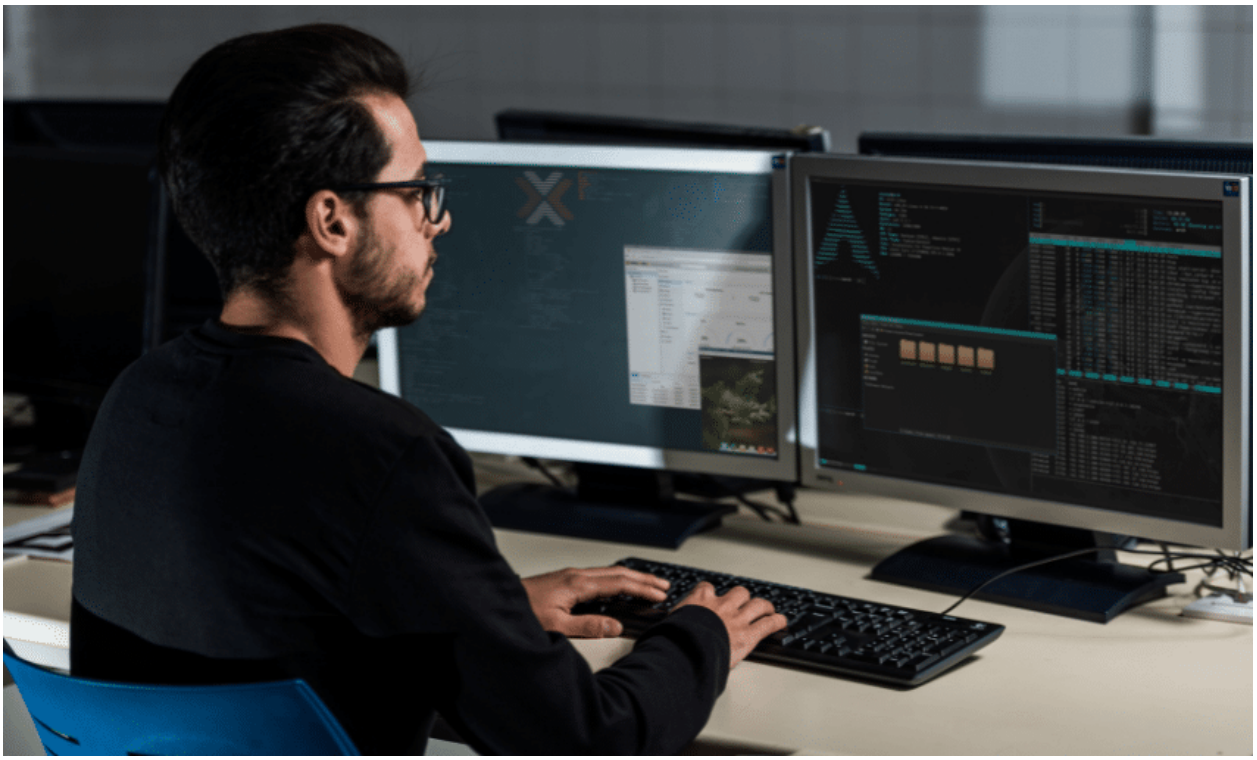
O objetivo deste artigo é apresentar uma visão pessoal do perfil ideal para trabalhar como um especialista em Linux Embarcado. Embora não sejam pré-requisitos obrigatórios, este guia pode ajudar na avaliação pessoal dos desafios envolvidos em projetos de Linux embarcado, que requerem habilidades e conhecimentos específicos. Então, vamos começar.

Conhecimentos técnicos necessários para o especialista em Linux Embarcado

Segue a lista de conhecimentos técnicos:

- Conhecimento em administração de sistemas Linux;
- Conhecimento em desenvolvimento de software, principalmente através de ferramentas de código aberto;
- Conhecimento em desenvolvimento de firmware. Em especial, microcontroladores;
- Conhecimentos em redes de computadores;

- Conhecimentos em projetos de hardware.



Especialista em Linux Embarcado

O profissional que possuir experiência em todos os aspectos mencionados anteriormente, certamente será um especialista em Linux Embarcado. No entanto, para aqueles que não possuem tal experiência, é possível adquiri-la ao trabalhar com Linux embarcado ao longo do tempo. É importante ressaltar que esses requisitos são indispensáveis para qualquer projeto. A seguir, detalharei cada um dos tópicos mencionados.

Administração de sistemas Linux

Em princípio, o conhecimento avançado de instalação, configuração e personalização do sistema operacional Linux em ambiente PC resolve este requisito. O ideal seria ter trabalhado com administração de sistema mesmo – seja em um departamento de escola ou alguma empresa – onde é preciso garantir que vários computadores estejam operando de maneira apropriada e organizada.

Uma boa administração de sistemas Linux envolve, obrigatoriamente, conhecimentos em modo terminal (creio que em Windows não seja diferente). É necessário não somente conhecer bem comandos em terminal, como a correta

localização dos arquivos nos diretórios, os serviços presentes no sistema operacional.

Dei o exemplo de trabalhar como administrador de sistemas com várias máquinas para que a pessoa já comece a se acostumar com os problemas em nível de sistema que ocorrem. Os problemas de funcionamento de sistema presentes em projetos Linux Embarcado são similares a problemas em computadores PC. A principal diferença – o que é melhor para o sistema embarcado — é que a probabilidade de mal uso é muito menor, porém, em geral, um sistema embarcado fica mais tempo em operação do que um computador PC.

Conheci casos de pessoas que dominam o sistema operacional só de usar em casa, para fins pessoais. Já é um excelente ponto de partida para se tornar um especialista em Linux Embarcado.

Desenvolvimento de software

É muito importante que o profissional que for trabalhar com projetos em Linux embarcado saiba desenvolver software para Linux. O desenvolvimento de software para Linux embarcado está mais próximo do desenvolvimento de software para um PC do que para um microcontrolador. Não somente os conhecimentos de programação, como também os conhecimentos das ferramentas para gerar o binário.

É extremamente importante que o profissional conheça bem o compilador GCC, as ferramentas MAKE e AUTOTOOLS. É preciso conhecer não só seu funcionamento como ser capaz de usar estas ferramentas sem a necessidade de um ambiente de desenvolvimento (IDE). A razão disso é a de que em alguns casos é preciso realizar modificações em projetos de código aberto existentes. E estes projetos raramente utilizam IDE para trabalhar (pode até ser que o desenvolvedor do projeto utilize durante o desenvolvimento, mas o pacote final no qual fazemos download não, e são raras as indicações de como integrar a um IDE). Mas uma coisa é certa: é totalmente possível fazer alterações e compilações através da linha de comando.

Não podemos deixar de mencionar aqui o conhecimento de sistemas de controle de versão. Acredito que não preciso dizer a razão para isso. Uma motivação extra está no fato de que as ferramentas mais novas de construção de sistemas Linux embarcado (mais especificamente, OpenEmbedded e Yocto) fazem a maior parte de seus downloads dos fontes a partir de seus repositórios. Assim, não só para

entender o que vai em seu projeto, como para criar um novo pacote de compilação, é fundamental o conhecimento de software de controle de versão.

Aqui neste tópico eu dispenso a necessidade de conhecimentos avançados – conhecimentos necessários para criar e configurar um repositório. Naturalmente os softwares mandatórios para se conhecer são o SUBVERSION e o GIT.

Desenvolvimento de firmware

Por que eu separei os dois casos?

Firmware não é um *software* também?

Sim, é. Porém, aqui eu não quero destacar os conhecimentos em programação comumente conhecida como 'baixo nível'.

Alguns componentes essenciais de projetos em Linux embarcado são componentes de baixo nível. São os casos dos *drivers* e do *bootloader*.

Os *drivers* em Linux devem respeitar a arquitetura de *drivers* do sistema operacional. Mas, na hora H de acessar ao recurso de *hardware*, a maneira de programar é similar a de um microcontrolador. O *bootloader* nada mais é do que um firmware – ele é executado sem a presença de um sistema operacional.

E estes componentes de *software* são amplamente presentes e alterados em projetos de sistemas Linux Embarcado.

Redes de computadores

Este tópico poderia ser, de uma forma simplista, inserida junto aos conhecimentos de administração de sistemas Linux.

Mas, na verdade, conhecer bem a forma de se projetar redes de computadores não é exatamente um conhecimento comum disponível por aí.

Hoje quase tudo é TCP/IP, quase tudo está ligado na Internet. Porém, preparar uma rede de forma a aproveitá-la ao máximo em desempenho não é exatamente uma tarefa que todos conhecem.

O mínimo do mínimo é ligar diversos computadores a um roteador que está ligado na Internet. É a configuração padrão de quase todas as residências, cafés e pequenas empresas. Eu não sei se todos sabem que em um roteador existe um switch.

É preciso conhecer a hierarquia de redes, sua organização. Atualmente as redes não são somente cabeadas. Existe o Wi-Fi, o GPRS, o 3G. É preciso conhecer protocolos como o PPP. Também é necessário saber como o Linux cuida de tudo isso. O Linux é o principal sistema operacional usado em equipamentos de rede. Podemos dizer que, quando se tem rede de computadores, existe um computador com Linux ali no meio. E com o Linux embarcado, essa situação aumentou muito.

Além dos conhecimentos na montagem da rede, é preciso conhecer bem os serviços necessários para um bom aproveitamento de rede: FIREWALL, SNMP, VPN, … Em sua grande maioria, são softwares já disponíveis para utilização, mas é preciso conhecê-los bem para a melhor implantação da solução.

Projeto de Hardware para sistemas com Linux Embarcado

Por fim, é preciso conhecer hardware também. Não é preciso ser projetista de hardware, mas pelo menos um conhecimento mínimo de leitura de esquema elétrico é fundamental.

Linux em PC roda em um computador, digamos, convencional. Um PC é uma plataforma de computadores consagrada, definida há muito tempo… onde concluo de forma simplista que qualquer PC tem o mesmo projeto de hardware para todos os casos. Aí a manutenção em software acaba sendo, digamos, menos complicado.

Em um sistema embarcado, cada projeto é de um jeito. São escolhas diferentes de componentes, que possuem características diferentes. E para situações específicas, o sistema operacional deve estar preparado para suportar os diversos componentes específicos presentes na placa. O Linux não vem preparado para atender a todas as necessidades de todos os projetos específicos. Para cada novo projeto de hardware, possivelmente uma nova personalização é necessária. E deixá-lo preparado implica em configurar drivers de dispositivos. Às vezes configurar não é possível, e um novo driver deverá ser desenvolvido. Com isso, conhecimento sobre as ligações dos componentes na placa é fundamental.

Uma importante etapa de um projeto de Linux embarcado é o “bring-up”.

Esta etapa é caracterizada pela inicialização do sistema operacional na nova placa projetada e prototipada. É quando carregamos o

bootloader, o kernel e um sistema de arquivos capaz de exercitar todos os dispositivos presentes na placa. Geralmente é uma etapa difícil e longa do projeto – e que ocorre nos momentos mais críticos do projeto.

E, nesta hora, o projetista Linux embarcado deve ser capaz de compreender o hardware de maneira a permitir a melhor interação possível com o projetista de hardware para que ele possa identificar problemas de projeto em hardware, quando for o caso.

Conclusão

Espero não ter desanimado ninguém com este artigo. Na verdade, a ideia é apresentar as ferramentas necessárias para que o engenheiro projetista possa desempenhar suas atividades de forma mais eficiente. Além disso, é interessante destacar que trabalhar com Linux embarcado pode ser muito empolgante e desafiador, uma vez que novos conhecimentos precisarão ser adquiridos. Então, vamos explorar juntos as possibilidades e aprender mais sobre esse fascinante mundo do Linux embarcado.

O que você indicaria para o profissional que deseja ser especialista em Linux Embarcado?

Saiba Mais

Ferramentas Linux que todo técnico e engenheiro deveria conhecer

Webinar gravado: Utilizando o Yocto Project para automatizar o desenvolvimento em Linux Embarcado

Webinar Gravado: Desenvolvendo com Linux Embarcado

Autor: Flavio de Castro Alves Filho

Sócio fundador da Phi Innovations e Professor das disciplinas de sistemas de tempo real e padrões e aplicações de sistemas operacionais do curso de pós graduação em eletrônica embarcada automotiva oferecido pelo SAE (Society of Automotive Engineers). Formado em Engenharia Elétrica pela UNICAMP, com especialização na Ecole Centrale de Lyon, na França, atua há mais de 10 anos em projetos de sistemas embarcados. Trabalhou com projetos de hardware e software embarcado para os setores de pagamento eletrônico, aeroespacial, defesa, segurança, equipamentos médicos, telecomunicações e energia. Atuou em projetos no Brasil e no exterior (França e

Alemanha). Começou suas atividades com Linux em 1996 e suas atividades empreendedoras em 2008. É um geek e apaixonado pelo Do-It Yourself (DIY).

Publicada originalmente em:
https://embarcados.com.br/como-se-tornar-um-especialista-em-linux-embarcado/

Anatomia de um Sistema Linux embarcado

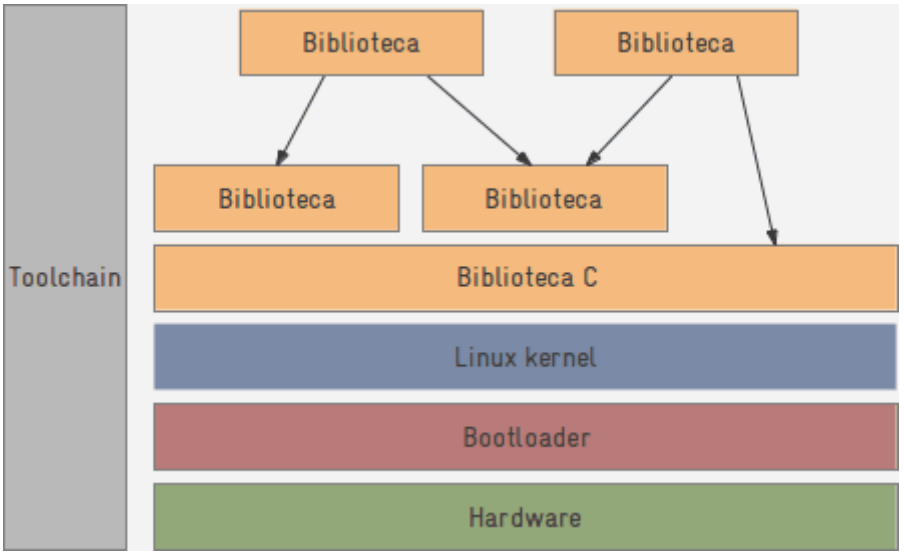
Um sistema Linux embarcado não se difere, no quesito sistêmico, de um sistema Linux desktop. A mesma estrutura e conceitos são aplicados em ambos os domínios. A principal diferença está nos requisitos de processamento, armazenamento, consumo de energia e confiabilidade. Na maioria dos sistemas Linux embarcado, os recursos disponíveis são limitados e muitas vezes a interface com usuário é bastante limitada ou simplesmente não existe.

Seguem alguns exemplos de sistemas embarcados que utilizam o Linux como sistema operacional:

- Roteador;
- Setup-box;
- Smart TV;
- Controlador Lógico Programável (CLP);
- Câmera Digital.

Arquitetura Básica

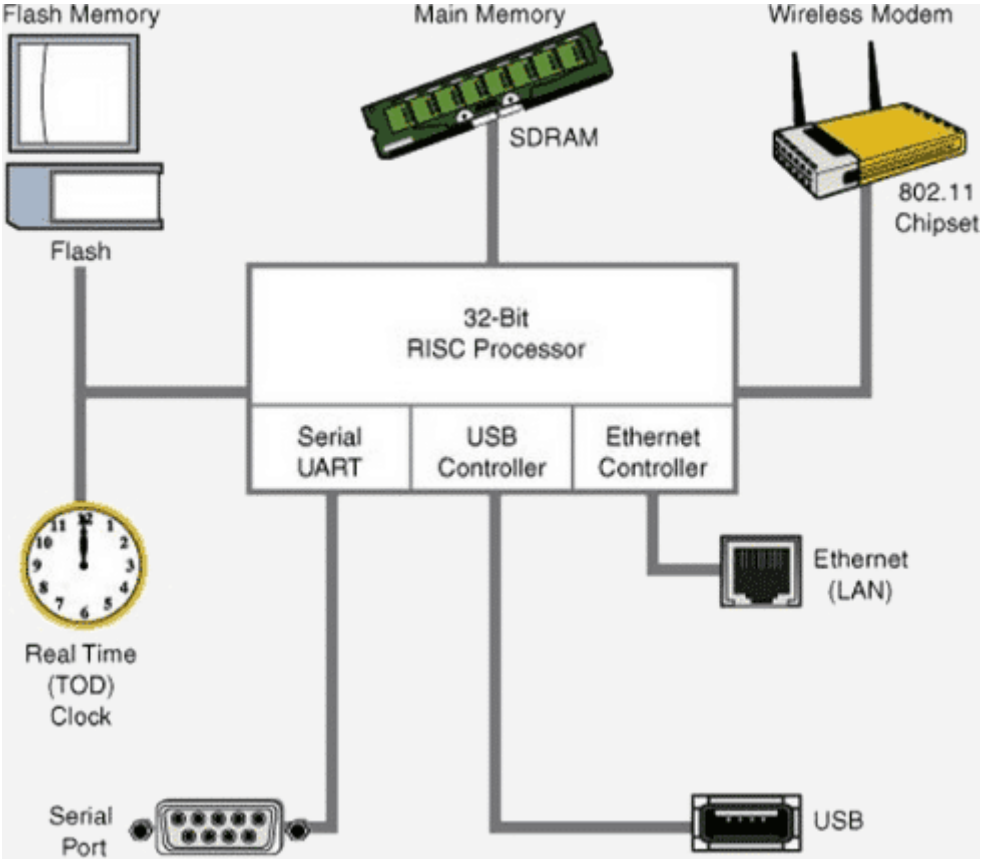
Na arquitetura básica de um sistema Linux podemos identificar cinco componentes básicos:In



1. **Hardware**: o seu produto;
2. **Bootloader**: iniciado pelo hardware, responsável pela inicialização básica, carregamento e execução do kernel Linux;
3. **Kernel Linux**: núcleo do sistema operacional. Gerencia CPU, memória e I/O, exportando serviços para as aplicações do usuário;
4. **Rootfs**: sistema de arquivos principal. Possui as bibliotecas do sistema para uso dos serviços exportados pelo kernel, além das bibliotecas e aplicações do usuário;
5. **Toolchain**: conjunto de ferramentas para gerar os artefatos de software do sistema.

Hardware

Vamos considerar um roteador Wi-Fi doméstico como exemplo de plataforma de hardware:



O Linux kernel é capaz de rodar em mais de 20 arquiteturas de CPU diferentes. Como por exemplo: x86, ia64, ARM, PowerPC, MIPS, SuperH, Blackfin, Coldfire, Microbalze.

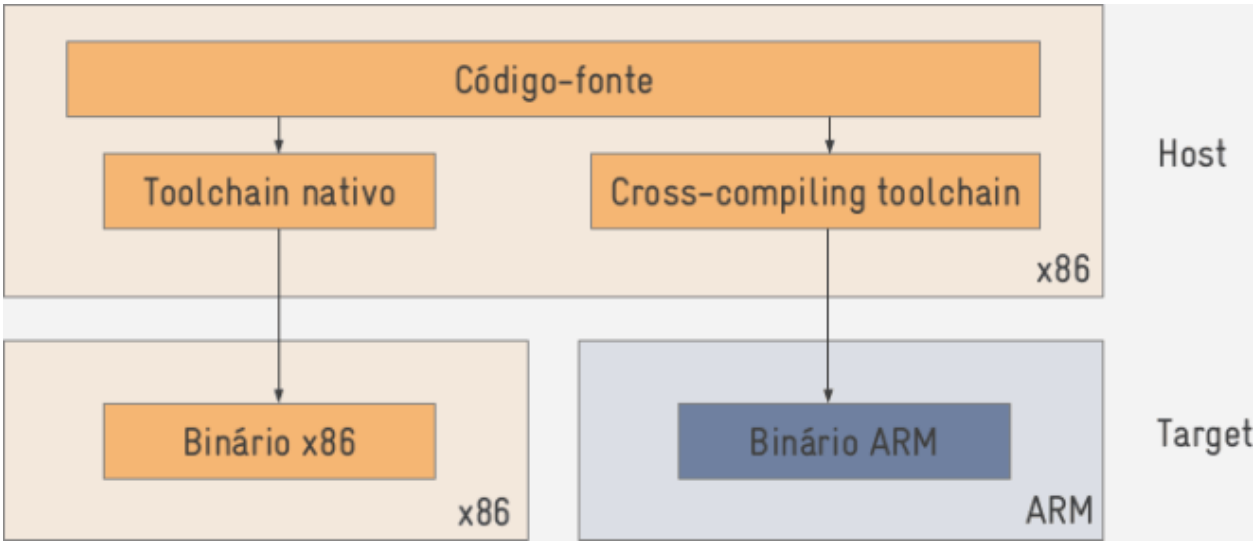
Tem suporte à arquiteturas de 32 e 64 bits e arquiteturas que não possuem MMU (Memory Management Unit).

Suporta armazenamento em memórias NAND, FLASH, MMC e hard disks.

No nosso exemplo, um sistema básico pode funcionar com até 8MB de RAM e 2MB de armazenamento, uma vez que, poucos drivers, funcionalidades do kernel e bibliotecas e aplicativos são necessárias para que o sistema desempenhe as funções desejadas.

Toolchain

Conjunto de ferramentas de programação usadas para gerar determinado produto, seja um software ou mesmo um sistema completo. Quando a plataforma de desenvolvimento (host) é diferente da plataforma alvo (target), chamamos o toolchain de cross-compiling toolchain.



Componentes principais:

- Compilador (gcc);
- Assembler e Linker (binutils);
- Biblioteca C padrão (glibc, uclibc, dietlibc, musl, etc).

Algumas opções de toolchains prontas;

- GNU Toolchain: Suporte para até 39 arquiteturas;
- Code Sourcery: ARM ;
- Linaro: ARM ;
- MIPS .

Artefatos de software de um Sistema Linux embarcado

Os artefatos de software principais de um sistema Linux embarcado são: Bootloader, Kernel e Rootfs. Tomando-se como base o hardware proposto, estes artefatos estão organizados na memória flash da seguinte maneira:

Bootloader

Toda CPU possui um mecanismo de inicialização, que é responsável por carregar e executar o bootloader. Em algumas arquiteturas de CPU é necessário o uso de um bootloader de primeiro estágio, no qual é carregado pelo processador e executado a partir de sua memória interna. Em alguns casos esse bootloader de primeiro estágio está concatenado junto ao bootaloader.

As principais funcionalidades do bootloader são:

- Inicializar o hardware antes de executar o kernel como, por exemplo, configurar a controladora de SDRAM;
- Passar parâmetros para o kernel;
- Prover mecanismos para carregar e gravar o kernel e o sistema de arquivos na memória flash ou cartão SD;
- Inicializar via rede ou pelo cartão SD;
- Rotinas de diagnóstico de hardware.

Principais bootloaders utilizados em sistemas embarcados:

- x86:
  - LILO;
  - Grub;
  - Syslinux.
- ARM, MIPS, PPC e outras arquiteturas:
  - U-Boot;
  - Barebox;
  - Redboot.

Kernel

O Kernel é o coração do nosso sistema. No boot ele é responsável por:

- Inicializar CPU, memória e barramentos;
- Configurar a memória virtual (se tiver MMU);
- Inicializar os device drivers;
- Iniciar o escalonador de tarefas;
- Iniciar threads do kernel;
- Montar sistema de arquivos principal (rootfs) e chamar o processo init.

Como principais características podemos apontar:

- Gerencia execução de processos e controla acesso à memória e I/O;
- Gerenciamento do kernel space X user space;
- Interface de user space com kernel *space* via chamadas do sistema (system calls);
- Acesso ao hardware via arquivos de dispositivo;
- Gerenciamento dinâmico dos módulos do kernel.

Rootfs

Após ter sido montado pelo kernel e ter o processo init (PID = 1) iniciado, o rootfs utiliza seu mecanismo de inicialização (ex.: SysVinit, Systemd etc) para inicializar os processos e aplicações do sistema. É responsável por prover as bibliotecas de sistema e de usuário.

Alguns exemplos de aplicações para sistemas embarcados:

- Dropbear: cliente e servidor SSH;

- Thttpd: servidor web;
- DirectFB: biblioteca gráfica;
- SQLite: banco de dados;
- Busybox: o canivete suíço de sistemas embarcados com Linux.

Referência

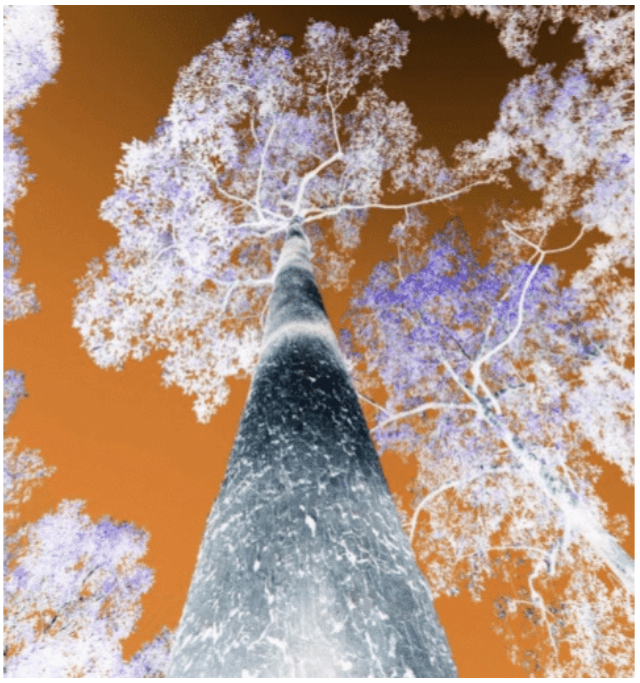
Material do Treinamento Desenvolvendo Sistemas Linux Embarcado da Embedded Labworks.

Autor: Diego Sueiro

Formado em Engenharia de Controle e Automação pela UNIFEI e CEO do Embarcados, atualmente trabalho na Inglaterra com Desenvolvimento de Sistemas Linux Embarcado Real Time. Sou consultor e desenvolvedor de software para Sistemas Embarcados com vasta experiência em projetos com processadores de 8bits a 32bits, sistemas bare metal, RTOS, Linux Embarcado e Android Embarcado. Um apaixonado por Pink Floyd e Empreendedorismo.

Publicado originalmente em:
https://embarcados.com.br/anatomia-de-um-sistema-linux-embarcado/

Introdução ao uso de Device Tree e Device Tree Overlay em Sistemas Linux Embarcado



Em várias situações do nosso dia-a-dia temos que usar documentos ou ferramentas que nos ajudam a descrever certas situações, funcionalidades e até mesmo direções. Manuais de usuário nos auxiliam a entender as funcionalidades de um produto, e um aparelho de GPS (Global Positioning System) nos auxilia descrevendo todo o trajeto que devemos traçar para chegar a um determinado local. Device Tree e Device Tree Overlay em sistemas Linux Embarcado compartilham dessa mesma particularidade, e servem para descrever com precisão como será configurado e utilizado o hardware em um sistema Linux embarcado.

Imagine, por exemplo, que você deseje que um determinado pino de sua placa seja habilitado para usar um pull-up interno para detecção de versão de hardware de sua placa auxiliar. Ou ainda necessite descrever como uma UART se comportará com relação ao sistema, quais pinos ela utilizará, qual device que deve ser habilitado e, por fim, que driver deve ser carregado. Essa descrição será possível por meio do uso de Device Tree e ajudará muito nas tarefas diárias de desenvolvimento ou hacking.

Hardware utilizado e Pré-Requisitos

A fim de tornar a explicação um pouco mais prática, faremos alguns exemplos utilizando a BeagleBone Black (Revisão C). Para isso a placa precisa estar, no mínimo, com a versão de kernel 3.8 (por conta de questões históricas que serão abordadas em seguida). Essa série de artigos visa explicar alguns conceitos e dar exemplos de como podemos e iremos utilizar esse recurso no dia-a-dia.

Histórico de Device Tree

Por conta dos inúmeros sistemas ARM que surgiram nos últimos anos, existiu muita confusão no desenvolvimento do kernel do Linux, o que forçou Linus Torvalds a recusar a aceitação de novos *ARM board files* na *mainline* do kernel e assim as fabricantes de placas ARM passaram a utilizar **Device Tree´s (DT)**.

A implementação de Device Tree também foi necessária para a fabricante da BeagleBone Black, e após a correção de um problema (que impedia que as modificações do Device Tree fossem aplicadas em tempo de execução e a partir do user-space), essa ferramenta está disponível para uso de desenvolvedores e makers.

Device Tree Overlays

Vamos estudar agora um simples e pequeno **Device Tree Overlay (DTO)**, e entender cada uma das seções.

Abaixo está o *Device Tree Overlay* para o **pino 16** do header **P8** da BeagleBone Black. Ele diz ao Kernel tudo o que ele precisa saber para que esse pino funcione de acordo com a nossa necessidade, que para este exemplo será o de **habilitar o pino como GPIO**, colocando um *pull-up interno* para que seja possível detectar versões de placa diferentes por meio desse pino.

Lembrete: Use sempre a última versão disponível da documentação de hardware da placa que estiver utilizando pois será necessário para saber o offset dos registradores, bem como os modos de *pin-mux*.

```
/*
*  Virtual cape for pin P8.16
*
*  This program is free software; you can redistribute it and/or modify
*  it under the terms of the GNU General Public License version 2 as
*  published by the Free Software Foundation.
*/

/dts-v1/;
/plugin/;

/{
    compatible = "ti,beaglebone", "ti,beaglebone-black";
    part_number = "BS_PINMODE_P8_16_0x17";

    exclusive-use =
        "P8.16",
        "gpio1_14";

    fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            bs_pinmode_P8_16_0x17: pinmux_bs_pinmode_P8_16_0x17 {
                pinctrl-single,pins = <0x038 0x17>;
            };
        };
    };

    fragment@1 {
        target = <&ocp>;
        __overlay__ {
            bs_pinmode_P8_16_0x17_pinmux {
                compatible = "bone-pinmux-helper";
                status = "okay";
                pinctrl-names = "default";
                pinctrl-0 = <&bs_pinmode_P8_16_0x17>;
            };
        };
    };
};
```

Nós podemos separar o arquivo acima em algumas partes para poder entendê-lo melhor.

A primeira parte são apenas alguns comentários e são opcionais, mas normalmente ajudam as demais pessoas a entenderem qual o objetivo desse arquivo, caso seja necessária uma alteração futura.

```
/*
*  Virtual cape for pin P8.16
*
*  This program is free software; you can
redistribute it and/on modify
*  it under the terms of the GNU General Public
License version 2 as
*  published by the Free Software Foundation.
*/
```

As próximas duas linhas são símbolos que servem para indicar a versão do *dts* e que o mesmo é um plug-in.

```
/dts-v1/;
/plugin/;
```

A próxima linha indica o início do nó principal do Device Tree:

```
/{
```

A linha **"compatible"** descreve qual plataforma para a qual o Device Tree foi feito para funcionar. E nela existe uma regra: vai da mais compatível, para a menos compatível. É importante mencionar todas as plataformas para as quais se deseja suporte, pois falhas acontecerão nas plataformas que não forem mencionadas.

```
compatible = "ti,beaglebone",
"ti,beaglebone-black";
```

Na próxima parte, o **part number** e **version** são proteções para assegurar que o Device Tree Overlay apropriado seja carregado. Eles também devem ser usados para nomear o arquivo .dts no formato -.dts. A versão, até a publicação deste artigo, deve ser **00A0** para a BeagleBone Black.

```
part_number =
"BS_PINMODE_P8_16_0x17";
version = "00A0"
```

A propriedade **exclusive-use** permite que os overlays descrevam quais recursos eles precisam e assim evitam que qualquer outro overlay use os mesmos recursos. No nosso caso estamos usando o **pino P8.16** e sua funcionalidade como **gpio 1_14**.

```
exclusive-use =
        "P8.16",
        "gpio1_14";
```

A próxima parte são os fragmentos do device tree. Eles irão descrever qual dispositivo será sobreposto, e a partir daí cada fragmento irá customizar os **pin-muxes** e/ou habilitar devices.

O fragmento abaixo parece ser um pouco complicado mas não é tanto. A primeira coisa é que estamos setando qual o target que será sobreposto para esse fragmento. No nosso caso, é o **am33xx_pinmux**, que é compatível com o driver **pinctrl-single**.

A próxima linha é o próprio nó do **__overlay__**. A primeira propriedade dentro desse nó será usada no próximo fragmento (**bs_pinmode_P8_16_0x17**) e contém as definições para realizar o mux dos pinos, que por sua vez são tratados pelo driver **pinctrl-single**. Você pode encontrar como realizar essa configuração procurando a página de documentação da placa que está utilizando.

Dentro do *pinctrl-single*, temos o bloco onde inserimos os valores dos pinos e as duas colunas de valores estão configurando o pino para funcionar como GPIO. No nosso caso, o pino P8.16 do header P8 configurado no modo 7.

```
fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            bs_pinmode_P8_16_0x17:
pinmux_bs_pinmode_P8_16_0x17 {
                pinctrl-single,pins = <0x038 0x17>;
            };
        };
    };
```

O último fragmento habilita o pino da forma como configuramos anteriormente. O target para o overlay nesse caso é o **ocp**. Há também uma referência à

propriedade do fragmento anterior (**bs_pinmode_P8_16_0x17**) para mapear os pinos habilitados pelo driver do pinctrl para o GPIO em questão.

A linha *compatible* faz a configuração para que esse fragmento seja compatível com o **bone-pinmux-helper** presente na placa utilizada.

```
fragment@1 {
        target = <&ocp>;
        __overlay__ {
            bs_pinmode_P8_16_0x17_pinmux {
                compatible = "bone-pinmux-helper";
                status = "okay";
                pinctrl-names = "default";
                pinctrl-0 = <&bs_pinmode_P8_16_0x17>;
            };
        };
    };
};
```

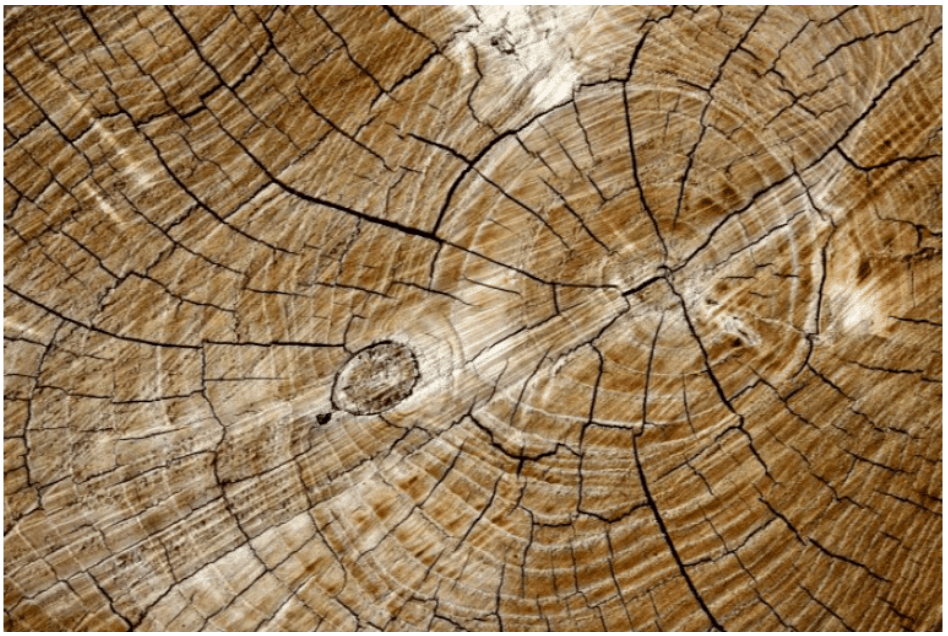
Isto parece ser um pouco obscuro e, na verdade é em partes. A maneira mais fácil de criar um novo overlay para se adequar às suas necessidades é criar o seu próprio a partir de um que já esteja pronto e próximo daquilo que se deseja. O melhor lugar para procurar por isso é no diretório **/lib/firmware** (on Debian), e examinar os overlays já criados para ver como funcionam.

Autor: Mateus Gagliardi

Formado em Matemática pela UNICAMP com experiência no desenvolvimento de firmware para sistemas embarcados. Apaixonado por tecnologia e atuante em soluções que envolvem sistemas bare metal e Linux Embarcado.

Publicado originalmente em: https://embarcados.com.br/device-tree-linux-embarcado/

Processo de Exportação de Device Tree Overlay



No texto anterior pudemos entender um pouco mais sobre o histórico que envolve os **Device Tree´s**, e também estudamos detalhadamente cada uma das suas partes e suas particularidades. Mas apenas "escrever" o **Device Tree Overlay** não é suficiente para chegarmos ao nosso objetivo, que é descrevermos o hardware de forma precisa. Então neste artigo entenderemos um pouco mais a fundo o processo de exportação de um Device Tree Overlay.

Os exemplos a seguir são demonstrados usando a distribuição Debian padrão que atualmente vem gravada nas placas, lembrando que estamos usando uma BeagleBone Black – Revisão C.

Exportando e "Des- Exportando" um Device Tree Overlay

Para começar, navegamos até a pasta **/lib/firmware** e alí podemos ver todos os *device tree overlays* disponíveis por padrão.

```
root@beaglebone:/lib/firmware# ls -ltr
total 1660
...
```

Será possível ver uma quantidade razoável de device tree overlays disponíveis. O arquivo fonte (dts) e o compilado (dto) estão no mesmo diretório. Se desejar, abra qualquer um dos arquivos para verificar o conteúdo e até mesmo para fazer uma verificação do que foi explicado no artigo anterior. Cada device tree overlay, por si só, é bem descritivo e, geralmente, bem comentado.

Nós poderíamos utilizar o *.dts* que produzimos no artigo anterior, mas para que a explicação fique mais didática e ilustrada, vamos aproveitar o fato de termos alguns *.dto's* já compilados disponíveis. Você poderá aplicar os mesmos passos que veremos a seguir com o *.dts* que produzimos anteriormente.

Dito isso, vamos usar como base daqui pra frente o arquivo **BB-UART1-00A0.dts**.

O primeiro passo será navegar até o diretório onde é possível ver quais device tree overlays estão habilitados pelo *bone cape manager*.

```
root@beaglebone:/lib/firmware# cd /sys/devices/bone_capemgr.*
```

Note na linha acima que o * é necessário porque nós não conhecemos qual número deve ser, porque isso depende da ordem de boot. Você terá que descobrir qual é o seu caminho particular. No caso da placa e da distribuição que estamos utilizando para escrever o artigo podemos ver o caminho como */sys/devices/bone_capemgr.9/slots*.

Agora vamos verificar o conteúdo do arquivo slots:

```
root@beaglebone:/sys/devices/bone_capemgr.9# cat slots
```

O arquivo deve estar muito parecido com o que você verá abaixo, considerando que você não tenha feito grandes customizações da distribuição Debian padrão. \

```
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas
Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas
Instrument,BB-BONELT-HDMI
```

De acordo com a documentação da BeagleBone, os três primeiros slots são designados para os EEPROM IDs das capes. Os próximos 2 são overlays carregados no boot. O número 4 é a memória eMMC da placa, de onde, muito

provavelmente, você está rodando sua distribuição Debian. O quinto device tree overlay é para habilitar o HDMI da placa.

Se exportamos agora um novo device tree overlay, como o que estamos usando neste artigo (UART1 overlay), você verá uma nova opção surgir listada como um novo número. Vamos fazer esse teste exportando o arquivo dtbo da UART1:

```
root@beaglebone:/sys/devices/bone_capemgr.9# echo BB-UART1 > slots
```

Com o comando acima nós obtemos a saída do comando echo, "BB-UART1", e a escrevemos no arquivo de slots a fim de habilitar os drivers e o device para a UART1 usando o overlay. Agora vamos conferir se o device tree overlay foi carregado adequadamente:

```
root@beaglebone:/sys/devices/bone_capemgr.9# cat slots
```

E assim conseguimos ver nesse ponto a UART1 carregada e pronta para ser utilizada:

```
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
7: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-UART1
```

Agora vamos imaginar que por uma outra necessidade durante seu processo de desenvolvimento você não precisa mais utilizar a UART1 e deseja utilizar os pinos para outra necessidade qualquer. Um dos caminhos para remover o overlay carregado é reinicializar a BeagleBone. A outra forma é "des-exportar", e você pode fazer isso executando o seguinte comando:

```
root@beaglebone:/sys/devices/bone_capemgr.9# echo -7 > slots
```

Nós utilizamos a sétima opção listada, adicionamos um '-' antes, e escrevemos isso no arquivo de slots.

Todavia, há uma observação importante para se fazer. Na versão mais recente do Debian (bone-debian-7.8-lxde-4gb-armhf-2015-03-01-4gb.img) que foi utilizada durante a escrita deste artigo e também em algumas versões da distribuição Angstrom (a partir da 6-20-2013), descarregar vários overlays pode causar um kernel panic, causando a perda da sessão ssh, e deixando o cape manager com

um comportamento imprevisível. A recomendação é para que se faça a reinicialização do sistema para descarregar os overlays, até que esse problema seja corrigido.

Mas agora que sabemos que a reinicialização do sistema gera o descarregamento do overlay, como fazer para que ele seja carregado junto com o boot do sistema?

Isso é bem simples de ser feito. Tudo o que você precisa fazer é referenciá-lo no arquivo **uEnv.txt**, que está localizado na pequena partição FAT da sua BeagleBone Black.

Os passos a seguir ilustram como fazer isso para o device tree overlay da UART1:

```
mkdir /mnt/boot
mount /dev/mmcblk0p1 /mnt/boot
nano /mnt/boot/uEnv.txt
#adicione isto ao fim de uma linha do uEnv.txt (não
numa nova linha) :
capemgr.enable_partno=BB-UART1
```

Autor: Mateus Gagliardi

Formado em Matemática pela UNICAMP com experiência no desenvolvimento de firmware para sistemas embarcados. Apaixonado por tecnologia e atuante em soluções que envolvem sistemas bare metal e Linux Embarcado.

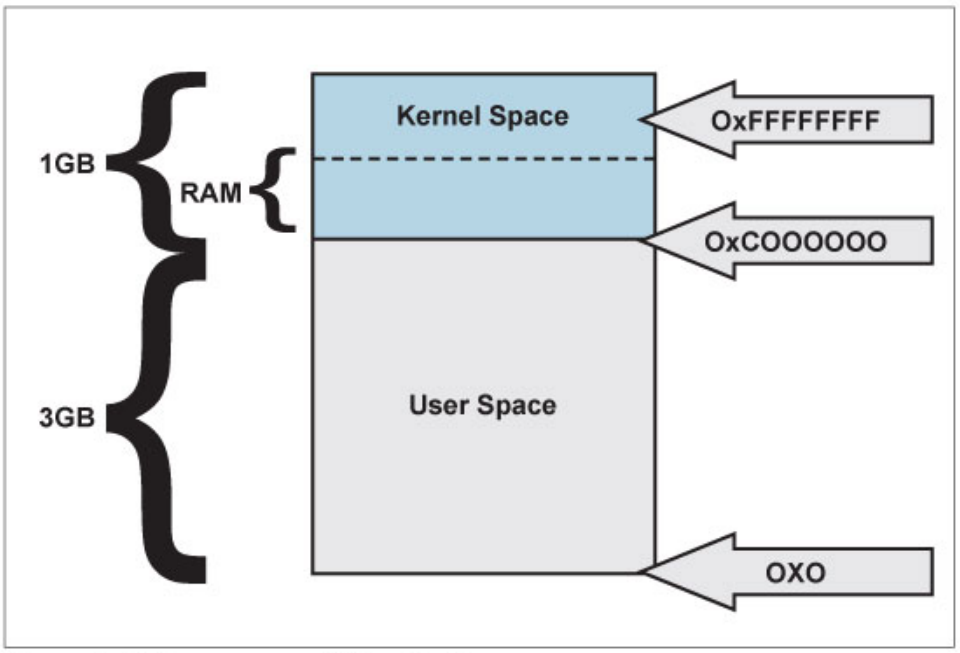
Publicado originalmente em:
https://embarcados.com.br/exportacao-de-device-tree-overlay/

Device Drivers para Linux Embarcado – Introdução

Neste artigo eu irei falar sobre desenvolvimento de *device drivers* para Linux embarcado. Há certa carência de documentação ou tutoriais sobre o assunto. Espero com este artigo preencher um pouco essa lacuna.

Introdução

Aplicações em espaço de usúario não podem se comunicar diretamente com o hardware porque o Linux não permite. O Linux divide a memória RAM em duas regiões: espaço de kernel (*kernel space*) e espaço de usúario (*user space*). O kernel space é onde o Linux executa e provêm seus serviços, e onde os *device drivers* residem. Já o *user space* é a região de memória onde os processos de usuários são executados. O *kernel space* pode ser acessado por processos de usuário somente através do uso de *system calls* (chamadas de sistema).



Desta forma, preferencialmente processos em espaço de kernel podem se comunicar com o hardware e acessar os periféricos. Você também pode usar *drivers* em espaço de usuário para acessar o hardware. Esse mecanismo permite aos desenvolvedores escreverem o código do software sem ter que se preocupar com detalhes de hardware. E protege o usúario de, inadvertidamente, acessar os dispositivos e de alguma maneira danificá-los. Isso tem funcionando bem até agora.

O que é Device Driver?

É um programa ou processo executado em uma região especial de memória e através do qual o usuário pode acessar um dispositivo ou recurso de um processador ou sistema computacional. Ele serve como um mediador entre o software e o hardware.

Por exemplo, quando você tira uma foto em seu celular Android, o *driver* da câmera interage com o software e passa informações a ele referentes às imagens capturadas e outras informações. O resultado final é a foto. Ou quando você quer jogar seu jogo preferido, mas não é possível sem o *driver* da placa de vídeo, certo? Por meio do *driver* o jogo acessa recursos da placa de vídeo que permitem a experiência incrível que os jogos proporcionam.

Então, a ideia é justamente essa. Servir como mediador entre o usuário e o hardware ou entre o software e o hardware.

Drivers como módulos ou embutidos no kernel

Os *drivers* no kernel Linux podem ser compilados como módulos que podem ser carregados em tempo de execução ou embutidos no próprio kernel.

Os *drivers* em módulos são conhecidos como *Loadable Kernel Module* (Módulo de Kernel Carregável) e se comportam de forma semelhante às DLLs do Windows. Um LKM (Loadable Kernel Module) se constitui de um único arquivo de objeto ELF, normalmente nomeado como "serial.o" para o kernel 2.4.x ou "serial.ko" para o kernel 2.6.x e versões posteriores. Uma grande vantagem é que ele pode ser carregado na memória em tempo de execução, bastando para isso um simples comando. E outra vantagem é que quando você altera o código do LKM, não é necessário compilar todo o kernel, basta compilar somente o LKM.

Os *drivers* embutidos ou monolíticos são módulos que são construídos como parte do kernel. Eles formam, juntamente com subsistemas do Linux e outros componentes, uma imagem única, cujo resultado final é o kernel propriamente dito. Vantagens de usar *drivers* monolíticos:

- uma vez aceito no kernel Linux oficial, ele será mantido pelos desenvolvedores;
- com custo de manutenção grátis, conserto de falhas de segurança e melhorias;
- fácil acesso ao código fonte pelos usuários.

Device Tree

Device Tree é uma estrutura de dados para descrever hardware. Em vez de incluir código de hardware no sistema operacional, muitos aspectos do hardware podem ser descritos em uma estrutura de dados, que é passada ao kernel na hora do boot.

O *Device Tree* era usado até pouco tempo apenas em sistemas com processadores PowerPC e SPARC. Mas foi recentemente portado para processadores ARM no kernel 3.1 e seu uso é agora obrigatório no desenvolvimento de novos drivers. É uma grande vantagem para sistemas

embarcados, porque podemos facilmente descrever diferentes tipos de placas que usam o mesmo processador ou ainda processadores diferentes.

A estrutura de dados em si é uma simples árvore de nós e propriedades com nomes. Os nós contêm propriedades e nós filhos. Propriedades são um par de valor-nome. Exemplo de descrição de um controlador SPI:

```
spi@10115000 {
    compatible = "arm,pl022";
    reg = ;
};
```

Com o *Device Tree*, é possível dar boot em qualquer placa com o mesmo kernel! Basta que os *drivers* do processador escolhido tenham suporte a *Device Tree*.

Toolchain

Caso você não saiba, da mesma forma que é necessário um compilador cruzado (*cross compiler*) ou *toolchain* para gerar executáveis para uma dada plataforma, você precisa de um compilador cruzado para gerar *device drivers*. Hoje em dia diversas distribuições Linux disponibilizam binários pré-compilados de compiladores cruzados e ferramentas de fácil instalação para uma variedade de arquiteturas. Você pode também compilar um manualmente se preferir.

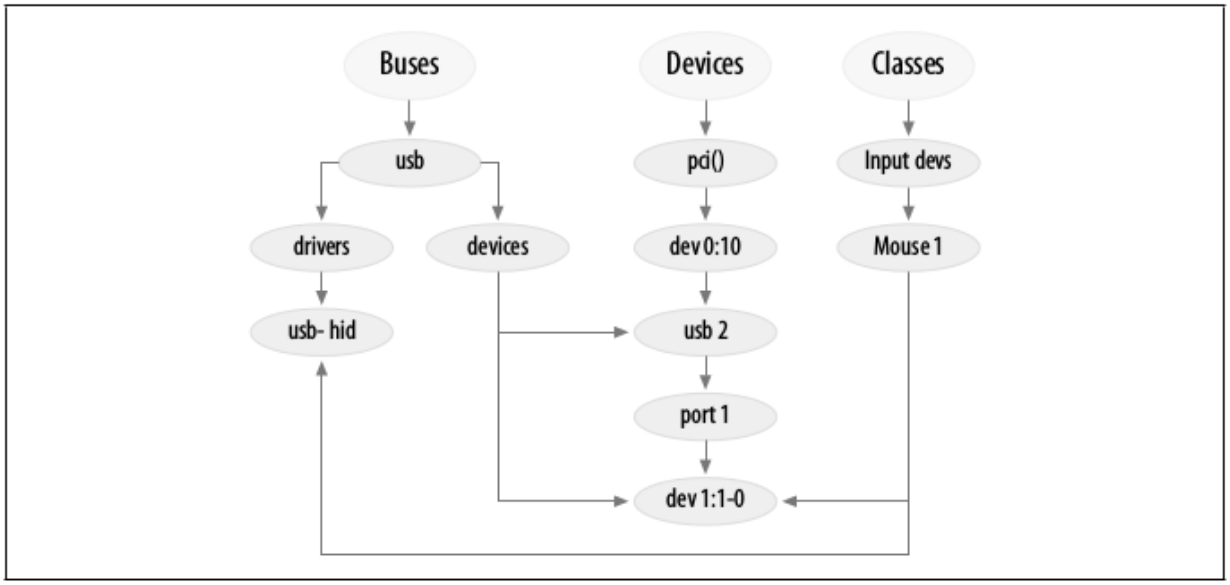
O compilador usado vai depender da arquitetura do SoC (*System-on-a-chip*) escolhido. As plataformas ou arquiteturas mais utilizadas em Linux Embarcado atualmente são ARM, PowerPC e MIPS. Também é usado às vezes o x86 da Intel. Você precisa compilar ou baixar uma variante do gcc para o alvo especificado. É o que chamamos de compilador cruzado ou *toolchain*. Então para ARM nós teríamos algo como arm-linux-gcc. Para MIPS, mips-linux-gcc. E assim por diante.

Versão do kernel

“The 2.5 kernel implements a unified device driver model that will make driver development for 2.6 easier.”

A versão do kernel é muito importante quando se programa Linux device drivers. A API do kernel Linux muda constamente com o tempo. Um driver I2C escrito para a versão 2.6.30 do kernel não funciona para a versão 2.6.36 ou maior. E as diferenças entre a versão 2.4 e 2.6 são ainda maiores.

Na versão 2.5 do Linux os desenvolvedores criaram um Modelo Unificado de Device Drivers (*Unified Device Driver Model*) e que passou a vigorar na versão 2.6 do kernel. Ele consiste de um número de estruturas e funções comuns a todos os *device drivers*. E inclui suporte a gerenciamento de energia, comunicação com o espaço de usuário, dispositivos hotplug, classes de dispositivos, e outros mais. O *Linux Device Driver Model* é uma complexa estrutura de dados que reside na pasta /sys.



Fonte: Livro "Linux Device Drivers, 3º Edição" – Figura 14-1 (A small piece of the device model)

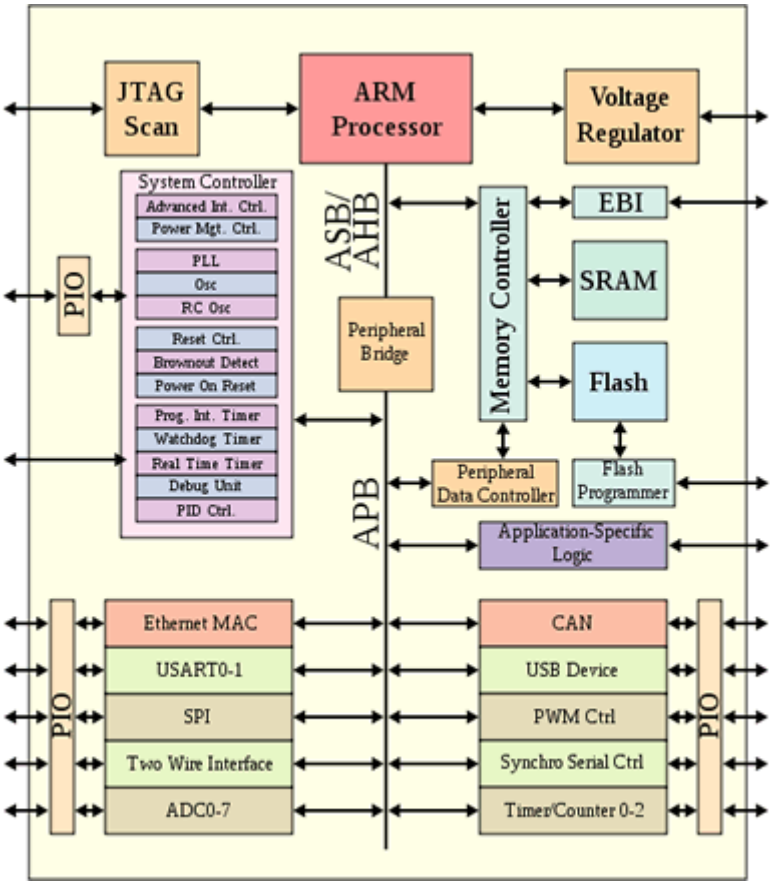
E sempre há mudanças na API do kernel ao longo do tempo. Por que a API do kernel muda constamente? Leia este documento para mais detalhes: https://www.kernel.org/doc/Documentation/stable_api_nonsense.txt

Uma coisa interessante sobre a evolução do kernel Linux é que as novas versões sempre adicionam novos atributos e nos anos recentes muitos atributos especifícos para sistemas embarcados tem sido adicionados. Um bom exemplo é o subsistema IIO (*Industrial I/O*). Ele foi designado para dar suporte a conversores A/D e D/A, acelerômetro, sensor de luz, sensor de proximidade, magnetrômetro, etc.

Para quem escreve código para software não importa a versão do kernel. Uma aplicação escrita para a versão 2.4.x roda nas versões 2.6.x, 3.x e possivelmente posteriores. Isso porque o Linux sempre mantém compatibilidade com versões anteriores em nível de aplicação.

Periféricos em Sistemas Embarcados

De uma perspectiva de *device driver*, desenvolvedores de software embarcado (firmware) frequentemente lidam com dispositivos que não são comumente encontrados em computadores convencionais. Exemplos de tais dispositivos: modem GSM, SPI, I2C, ADC, memória NAND, rádio, GPS.



Sob o Linux, existem essencialmente três tipos de dispositivos: dispositivos de rede, dispositivos de bloco e dispositivos de caractere. A maioria dos dispositivos se enquadram na categoria de dispositivos de caractere. Porém, hoje em dia, muitos *device drivers* não são implementados diretamente como dispositivos de caractere. Eles são desenvolvidos sob um framework, específico para um dado dispositivo. Exemplos de frameworks: framebuffer (gráficos), V4L2 (captura de vídeo), IIO (Industrial I/O), etc.

Por exemplo, se você deseja usar uma câmera para a qual não há suporte no Linux, você precisa escrever um *device driver* para essa câmera usando o framework Video4Linux2.

Interrupções

Se você já tem algum conhecimento ou experiência com sistemas embarcados, provavelmente sabe que interrupções é uma parte importante no desenvolvimento de firmwares. No entanto, infelizmente não é tão simples usar interrupções em Linux Embarcado. Só é possível usar interruções em *drivers*, ou seja, em espaço de kernel ou através de mecanismos de *poll/select* em espaço de usuário. Porém o uso de *poll/select* oferece poucos recursos.

A forma de interrupção que estamos acostumados a usar em sistemas embarcados só é possível em *device drivers*. Não se pode criar uma rotina ou função de tratamento de interrupção em um software. Outra alternativa é usar *drivers* em espaço de usuário.

Como em muitos outros sistemas operacionais, as interrupções são associadas a números e intervalos de endereços de entrada e saída. A programação de interrupções em Linux embarcado é semelhante a como é feito para Linux Desktop.

Escrevendo Device Drivers Portáveis

"Follow the kernel team's rules to make your drivers work on all architectures."

Quando você trabalha com sistemas embarcados há uma boa chance de que use múltiplos tipos de processadores ao longo de sua carreira. É importante manter em mente que você deveria sempre que possível escrever *device drivers* portáveis. Quase todos os *device drivers* do Linux funcionam em mais de um tipo de processador. Para alcançar este objetivo, é necessario conhecer alguns conceitos como tipos de variavéis apropriadas, tamanhos de página de memória, problemas com ordem de byte, alinhamento de dados apropriado, e muitos outros.

Por exemplo, os processadores ARM dispõem de páginas de memória de 4K, 16K ou 32K, mas o i386 da Intel dispõe apenas de páginas de memória de 4K.

Mapeamento de Memória

Alguns processadores usam o método de *Port-Mapped I/O* (Entrada e Saída mapeados em porta) e tem instruções especiais para acessar portas e periféricos, como o x86 da Intel. Outros processadores usam o metodo de *Memory-Mapped I/O* (Entrada e Saída mapeados em memória), como o ARM,

onde o acesso a portas e periféricos é feito diretamente através da memória RAM.

Para qualquer um dos métodos, MMIO ou PMIO, para poder acessar a memória RAM ou as portas de E/S no Linux, é necessário mapear a memória física para uma memória virtual. Para acessar a memória ou portas de maneira portável, você precisa chamar *ioremap()* para acessar uma região de memória e *iounmap()* para liberar uma região alocada anteriormente. No entanto essas funções estão sendo substituídas por *devm_ioremap_resource()* e futuramente se tornarão obsoletas. Se a alocação da região de memória não retornar um erro, pode-se então usar a família de funções *read()/write()* para ler e escrever a memória mapeada. Esse é o método típico para acessar os registradores de um processador que usa o método de MMIO.

Conclusão

Neste artigo procurei abordar os tópicos mais relevantes no desenvolvimento de *drivers* para Linux Embarcado e que apresentam diferenças entre a programação de *drivers* para Linux Desktop e para Linux Embarcado. Essas diferenças são mais perceptíveis quando se programa *drivers* de plataforma.

Referências

- Linux Kernel and Driver Development Training – Free Electrons
- Essential Linux Device Drivers
- Linux Device Drivers – capitulo 14 (The Linux Device Model)
- https://www.linuxjournal.com/article/5783
- https://www.linuxjournal.com/article/6717
- https://lwn.net/Articles/232575/
- https://linux.die.net/man/2/poll
- https://linuxembeded.blogspot.com.br/2014/04/embedded-linux-drivers.html

Autor: Vinicius Maciel

Cursando Tecnologia em Telemática no IFCE. Trabalho com programação de Sistemas Embarcados desde 2009. Tenho experiência em desenvolvimento de software para Linux embarcado em C/C++. Criação ou modificação de devices drivers para o sistema operacional Linux. Uso de ferramentas open source para desenvolvimento e debug incluindo gcc, gdb, eclipse.

Publicado originalmente em:
https://embarcados.com.br/device-driver-para-linux-embarcado-intro/

Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo

O tema em foco para este artigo será: Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo. Será mostrado as diferenças e semelhanças entre eles, além de exemplos.

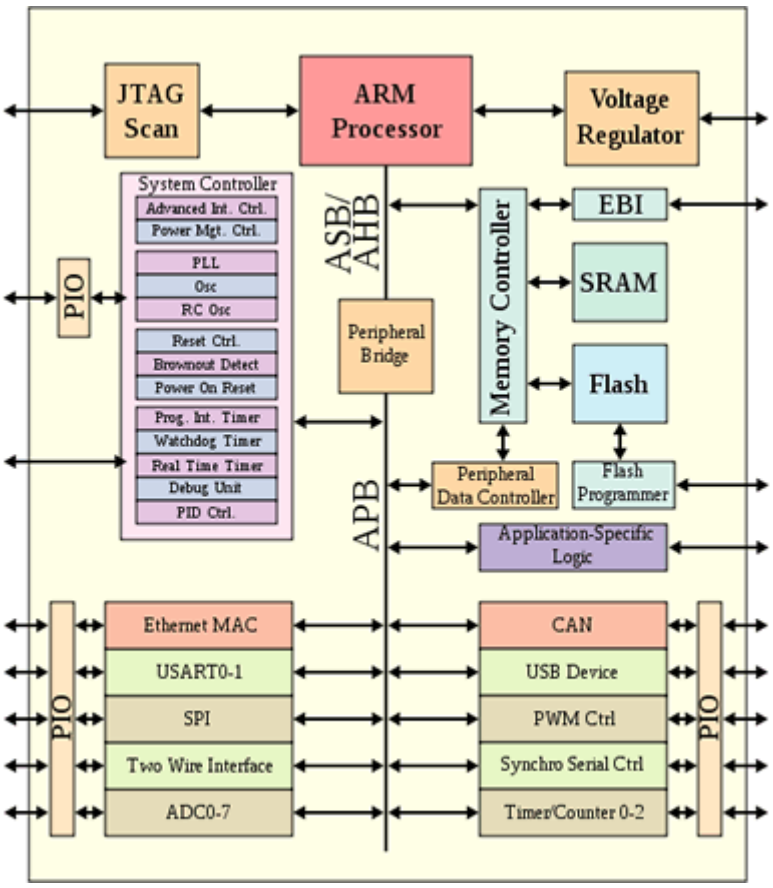
Um conceito muito importante quando se inicia na programação de device drivers para Linux é saber a diferença entre drivers de plataforma e drivers de dispositivo. Muitas pessoas que iniciam o estudo de linux device drivers, começam fazendo ou estudando drivers do tipo UART, I2C ou SPI. Porém, definitivamente você deveria primeiro saber essa diferença entre driver de plataforma e driver de dispositivo, porque isso te daria uma visão bem ampla de como funciona o drivers no Linux. Será apresentado neste artigo quais são essas diferenças e como eles interagem.

Linux Device Drivers: O que é um Driver de Plataforma

De acordo com a documentação do kernel, existem dois conceitos distintos: Platform devices e Device drivers. Platform devices são dispositivos que aparecem como entidades autônomas no sistema e contêm informações de hardware. Estão associados a controladores de barramento como: SPI, I2C e USB. Device Driver proporciona um modelo de driver padrão, com funções como

probe e remove, usadas por todos os drivers. Estão associados a dispositivos externos ao processador e ligados através de barramentos como os citados acima.

Então, em termos práticos, driver de plataforma é uma abstração para representar um controlador ou adaptador interno de um SoC, como por exemplo, controlador I2C, controlador SPI ou controlador USB. E também outros periféricos mais complexos como controlador de Interrupções e controlador de DMA. Ele também pode ser visto como um pseudo-barramento que conecta o CPU a um controlador interno a ele, isto é, no mesmo chip. Na figura abaixo vemos vários controladores como Ethernet MAC, USART0-1, SPI, I2C (Two Wire Interface), CAN e outros. Todos conectados ao CPU ARM (ARM processor) através do barramento APB.



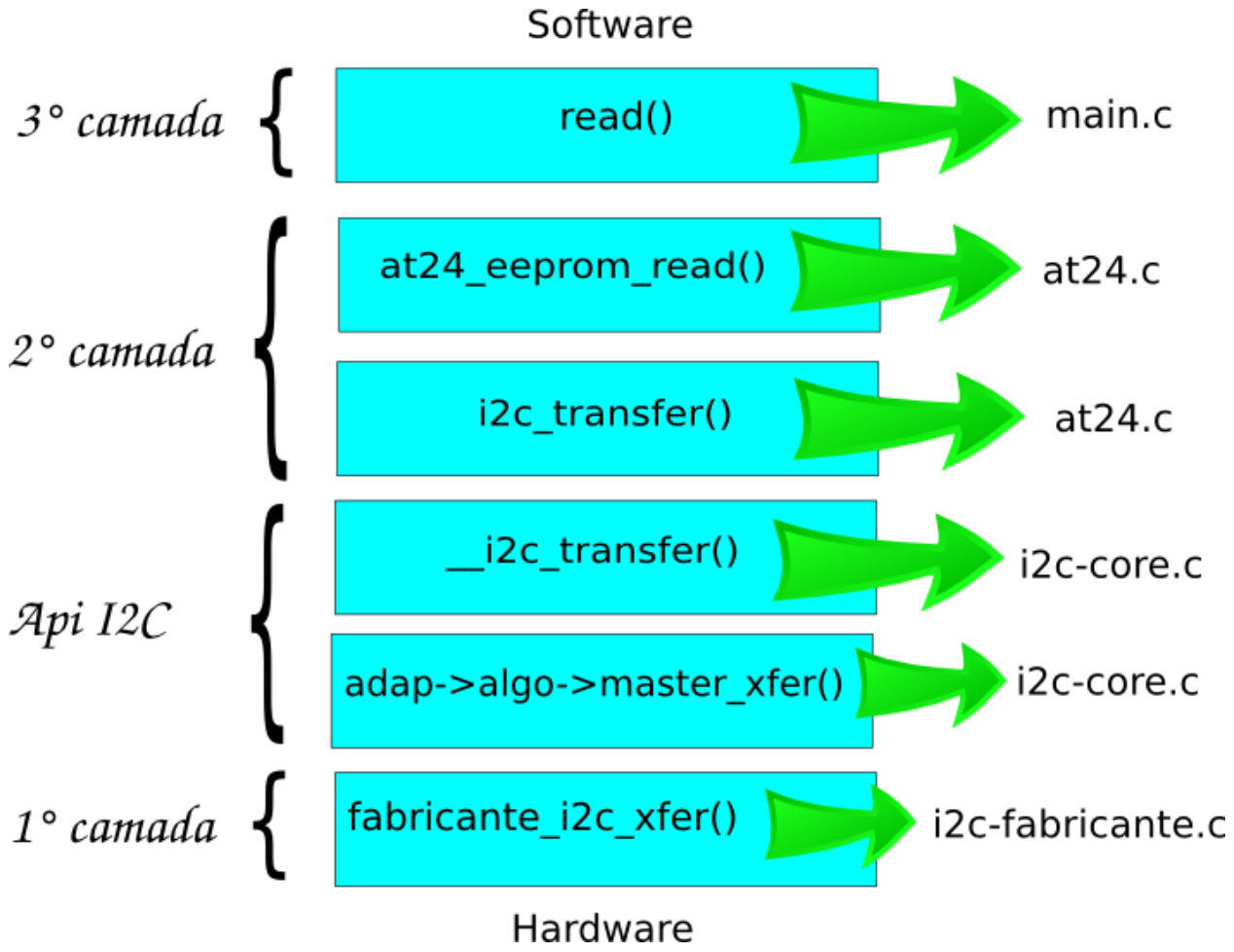
Ele é bastante usado em sistemas embarcados, principalmente nas arquiteturas ARM, PowerPC e MIPS. Então é muito importante entendê-lo. Se você pretende, por exemplo, fazer um BSP (Board Support Package) de uma placa nova, terá que fazer vários drivers de plataforma. É um trabalho extremamente complexo e árduo. Geralmente é necessária uma equipe para isso.

A camada de driver de plataforma

Para termos uma ideia melhor, será apresentado um exemplo prático. No Linux nós podemos imaginar que há três camadas de software distintas. A camada mais baixa é a dos plataform drivers. A segunda é a de linux device drivers. A terceira e última é a camada de aplicativos. 'Camada mais baixa' significa que está mais próxima do hardware. Assim, do ponto de vista de camadas, se tomarmos como exemplo uma memória EEPROM AT24cxx da ATMEL, acessada através de barramento I2C, teríamos:

- 1° camada: Driver de plataforma I2C (geralmente feito pelo fabricante);
- 2º camada: Driver de dispositivo at24.c;
- 3º camada: O software em espaço de usuário para ler e escrever a memória.

Então, usando a abordagem do Devfs ou a abordagem do Sysfs (Unified Device Driver Model), quando você chama *read()* lá no seu software, acontece resumidamente o seguinte:



A função *master_xfer()* é responsável pela comunicação I2C propriamente dita e é implementada no driver de plataforma I2C. A implementação dessa função e todo o código fonte do driver em si é diferente para cada SoC. Se a placa que você comprou tem um SoC que o fabricante não implementou o driver de plataforma I2C, você terá que fazê-lo. Um driver de plataforma qualquer é bem mais complicado de implementar do que um driver de dispositivo.

Existem poucas exceções onde um mesmo driver de plataforma serve para processadores diferentes. O driver i2c-mv64xxx.c, por exemplo, é um driver para controlador I2C de processadores da Marvell e que graças a um patch suporta também o controlador I2C do A10/A13 da Allwinner.

Comparação entre dois drivers de plataforma

Vamos ver exemplos de implementação para dois fabricantes diferentes.

Driver i2c-tegra.c (apenas algumas linhas):

```
static const struct i2c_algorithm tegra_i2c_algo = {
  .master_xfer    = tegra_i2c_xfer,
  .functionality  = tegra_i2c_func,
};

static int tegra_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[], int num)
{
        struct tegra_i2c_dev *i2c_dev = i2c_get_adapdata(adap);
        int i;
        int ret = 0;

      if (i2c_dev->is_suspended)
            return -EBUSY;

         ret = tegra_i2c_xfer_msg(i2c_dev, &msgs[i], end_type);
        return ret ?: i;
}
```

Driver i2c-omap.c (apenas algumas linhas):

```
static const struct i2c_algorithm omap_i2c_algo = {
   .master_xfer    = omap_i2c_xfer,
   .functionality  = omap_i2c_func,
};

static int omap_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[], int num)
{
      struct omap_i2c_dev *dev = i2c_get_adapdata(adap);

      int i;
      int r;

      if (dev->set_mpu_wkup_lat != NULL)
            dev->set_mpu_wkup_lat(dev->dev, dev->latency);

      for (i = 0; i < num; i++) {
       r = omap_i2c_xfer_msg(adap, &msgs[i], (i == (num - 1)));

              if (r != 0)
                      break;
        }
         return r;
}
```

Esses trechos são as implementações da função master_xfer para estes SoCs baseados em ARM. Percebeu o quanto eles são diferentes? E o que eles têm em comum? A primeira coisa que notamos em comum é a estrutura i2c_algorithm. Todo driver de plataforma I2C terá obrigatoriamente ela. E deverá ter uma função responsável por implementar a função de callback da API I2C, *master_xfer()*. A partir dai, cada fabricante implementa sua master_xfer de

acordo com seu hardware, conforme vimos acima. Leia o código completo destes drivers no kernel Linux e você verá o quanto são distintos.

Driver I2C Tegra

Driver I2C Omap

Mas existem elementos comuns a todos os drivers de plataforma. Eles terão sempre as seguintes estruturas:

```
struct platform_device {
        const char      *name;
        u32             id;
        struct device   dev;
        u32             num_resources;
        struct resource *resource;
};
```

representando o Platform device. E a outra estrutura:

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*suspend_late)(struct platform_device *, pm_message_t state);
        int (*resume_early)(struct platform_device *);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
};
```

representando o Platform driver. Nesse aspecto, ele se assemelha ao driver de dispositivo, porque ambos apresentam uma estrutura principal com funções de callback *probe()* e *remove()*.

Na primeira estrutura (platform_device) serão definidos aspectos ligados ao hardware propriamente dito, como endereços de registradores mapeados em memória e linhas de interrupção. E na segunda (platform_driver), deverão ser definidas ao menos as funções *probe()* e *remove()*, através das quais o kernel insere e remove o driver de plataforma do sistema. As funções *shutdown()*, *suspend()* e *resume()* estão relacionadas a gerenciamento de energia.

O Platform driver é registrado no sistema através da função *platform_driver_register()* e o Platform device através da função

platform_device_register(). A função **platform_driver_register()** pode ser encontrada em diversos drivers do kernel Linux, como pode ser visto aqui. Já a função **platform_device_register()** poderá ser encontrada em arquivos de *board*, especialmente a estrutura *platform_device*, como será visto num exemplo dado abaixo.

Suporte a Device Tree

Outro ponto importante para salientar é o uso do Device Tree. Como dito no artigo de introdução, é mandatório que todo driver de plataforma que é desenvolvido atualmente já inclua suporte a Device Tree. Ele passou a ser usado na versão 3.1 do kernel, no caso da plataforma ARM. Em relação ao driver de plataforma i2c-tegra.c, o trecho de código que oferece suporte a Device Tree é mostrado abaixo:

```
if (pdev->dev.of_node) {
    const struct of_device_id *match;

    match = of_match_device(tegra_i2c_of_match, &pdev->dev);
    i2c_dev->hw = match->data;
    i2c_dev->is_dvc = of_device_is_compatible(pdev->dev.of_node, "nvidia,tegra20-i2c-dvc");
} else if (pdev->id == 3) {
    i2c_dev->is_dvc = 1;
}
```

A função *of_match_device()* permite obter a entrada correspondente na estrutura tegra_i2c_of_match, mostrada logo abaixo. É útil para obter o campo especifico data, usado para alterar o comportamento do driver dependendo da variante do dispositivo detectado. Abaixo é mostrado a tabela de compatibilidade para os controladores I2C que esse driver suporta:

```
/* Match table for of_platform binding */

static const struct of_device_id tegra_i2c_of_match[] = {
{ .compatible = "nvidia,tegra114-i2c", .data = &tegra114_i2c_hw, },
{ .compatible = "nvidia,tegra30-i2c", .data = &tegra30_i2c_hw, },
{ .compatible = "nvidia,tegra20-i2c", .data = &tegra20_i2c_hw, },
{ .compatible = "nvidia,tegra20-i2c-dvc", .data = &tegra20_i2c_hw, },
{},
};
```

Os valores atribuídos a compatible são strings usadas para ligar um platform device ao platform driver. Abaixo é mostrado o código do Device Tree propriamente dito para o controlador I2C-1 do Tegra20:

```
i2c@7000c000 {
    compatible = "nvidia,tegra20-i2c";
    reg = ;
    interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&tegra_car TEGRA20_CLK_I2C1>,
            <&tegra_car TEGRA20_CLK_PLL_P_OUT3>;
    clock-names = "div-clk", "fast-clk";
    resets = <&tegra_car 12>;
    reset-names = "i2c";
    dmas = <&apbdma 21>, <&apbdma 21>;
    dma-names = "rx", "tx";
    status = "disabled";
};
```

O que é um driver de dispositivo

De acordo com a definição da documentação do kernel, driver de dispositivo é uma estrutura estaticamente alocada. Em termos práticos, é a camada de software usada para comunicação com um dispositivo externo ao processador ou SoC e que está associado a algum barramento como, por exemplo, barramento I2C, barramento SPI ou barramento USB. Para o exemplo dado acima da memória EEPROM, o barramento é I2C. Se em vez da memória AT24Cxxx, fosse a memória AT25DF641, o barramento seria SPI. Então todo dispositivo externo se comunica com o processador através de um barramento.

O driver de dispositivo, diferente do driver de plataforma, teoricamente funciona em qualquer processador, mesmo que sejam de arquiteturas diferentes. Ou seja, seu código é independente de hardware, enquanto que o driver de plataforma depende do hardware. O tópico do artigo anterior Escrevendo Device Drivers Portáveis se refere especificamente a drivers de dispositivo. O driver de dispositivo é uma camada concebida para ser portável. Pode-se até rodar o mesmo driver de dispositivo em um Linux Desktop e em uma placa com Linux Embarcado.

Existem essencialmente três tipos de dispositivos: dispositivos de rede, dispositivos de bloco e dispositivos de caractere. Uma memória I2C EEPROM como a AT24Cxx se encaixa bem na categoria de dispositivo de caractere. Desta forma, nós teríamos sob o diretório /dev sua representação através de um nó,

como AT24C, através do qual o usuário pode ler e escrever na memória. Porém, essa abordagem é tipica do kernel 2.4.x, mas que também pode ser usada nos kernels 2.6.x e 3.x. Outra possibilidade é a abordagem que pode ser usada para o desenvolvimento de drivers a partir do kernel 2.6, o Unified Device Driver Model. Esse modelo disponibiliza uma interface para comunicação com o espaço de usuário através de entradas no diretório /sys e é o modelo usado no driver at24.c.

No entanto, para o desenvolvedor de software, não muda nada. Pode ser usada a mesma API de acesso a arquivos *open()*, *read()*, *write()* e *close()*.

Como eles interagem

No tópico acima já foi dado um exemplo do que acontece quando você chamada *read()*. Desta forma, a partir de uma chamada de sistema no software o kernel passa por varias funções até chegar na camada de drivers de dispositivo. E dai, ele continua mais adiante chamando outras funções, até chegar na camada de driver de plataforma. Para, finalmente, chegar no dispositivo.

Funciona como, conforme citada acima, três camadas de software: aplicação, driver de dispositivo e driver de plataforma. Se algumas dessas duas últimas camadas não estiver presente, você não poderá acessar o dispositivo de barramento a partir de um software.

É importante salientar que alguns periféricos dispensam o uso da camada de driver de dispositivo, como é o caso de conversores A/D e D/A, caso seja utilizado o framework IIO para acessá-los. O framework já disponibiliza diretamente no diretório /sys, entradas para o espaço de usuário. Outro exemplo é o controlador de GPIO. Isso se aplica geralmente a periféricos internos ao SoC que não estão associados a um barramento de comunicação, como I2C, SPI ou USB.

Entradas para acesso a dispositivo no diretório /dev

Para que o usuário possa acessar um dispositivo é necessário um mecanismo para criar nós de dispositivos em uma pasta acessível a ele.

Muitos drivers de dispositivo no kernel 2.4 são implementados como drivers de caractere. Eles usam a cdev, uma abstração do kernel para drivers de caractere. Eles também podem ser usados nas versões do kernel 2.6.x e 3.x. A estrutura

principal usado nesses drivers é file_operations. Nela se define as operações para abrir e fechar arquivo, leitura e escrita, entre outras.

Major e Minor number

Tudo no Linux, assim como no Unix, é representado através de arquivos, incluindo dispositivos de hardware. Todos os dispositivos são como arquivos regulares. Eles podem ser abertos, fechados, lidos e escritos usando as mesmas chamadas de sistema que são usadas para manipular arquivos. O kernel Linux usa o par de números major e minor para representar um dispositivo de caractere. Eles se encontram na pasta /dev do sistema de arquivos.

Todos os dispositivos controlados pelo mesmo driver tem um major number em comum. Os minor numbers são usadas para distinguir entre diferentes dispositivos e seus controladores. O major number também identifica o tipo de dispositivo no sistema(se é uma memória I2C ou um terminal serial, por exemplo). Você pode alocar esses números de forma estática ou dinâmica.

Na alocação estática você deve escolher o major number que não esteja sendo utilizado. O registro estático deve ser feito através da função *register_chrdev_region()*.

Você pode conferir a lista de major number utilizados através do arquivo /proc/devices

Character devices:

```
1 mem
4 /dev/vc/0
4 tty
10 misc
13 input
29 fb
253 watchdog
254 rtc
```

Na alocação dinâmica o kernel irá atribuir o major number através da função *alloc_chrdev_region()* . Porém essa técnica não garante que sempre será utilizado o mesmo major number para determinado driver.

Como dito anteriormente o user-space se comunica com os device drivers através de arquivos de dispositivos residentes no /dev. Ao listar esse diretório temos como exemplo:

```
crw-rw-rw-    1 root  root    1,   3 Apr 11  2002 null
crw-------    1 root  root      10,   1 Apr 11  2002 psaux
crw-------    1 root  root    4,   1 Oct 28 03:04 tty1
crw-rw-rw-    1 root  tty         4,  64 Apr 11  2002 ttys0
crw-rw----    1 root  uucp    4,  65 Apr 11  2002 ttyS1
crw--w----    1 vcsa  tty         7,   1 Apr 11  2002 vcs1
crw--w----    1 vcsa  tty         7, 129 Apr 11  2002 vcsa1
crw-rw-rw-    1 root  root    1,   5 Apr 11  2002 zero
```

Veja que nas colunas 5 e 6 temos as informações do major e minor numbers. A criação desses arquivos podem ser realizadas basicamente de três maneiras:

1. Em tempo de construção do sistema de arquivos.

2. Utilizando o comando mknod.

3. Através de um sistema de gerenciamento de dispositivos, como por exemplo o udev.

Arquivo Board

Até o kernel 3.0 as plataformas ARM utilizavam extensamente o arquivo de board para realizar o instanciamento dos platform devices, dentre outras coisas.

Vejamos o exemplo da Beaglebone no arquivo arch/arm/mach-omap2/board-omap3beagle.c para instanciar os leds presentes na placa.

Declaração das estruturas:

```
387 static struct gpio_led_platform_data gpio_led_info = {
388           .leds          = gpio_leds,
389           .num_leds      = ARRAY_SIZE(gpio_leds),
390 };
391
392 static struct platform_device leds_gpio = {
393           .name   = "leds-gpio",
394           .id     = -1,
395           .dev    = {
396                  .platform_data  = &gpio_led_info,
397           },
398 };

428 static struct platform_device *omap3_beagle_devices[] __initdata = {
429           &leds_gpio,
430           &keys_gpio,
431           &madc_hwmon,
432 };
```

E finalmente o instanciamento realizado na função omap3_beagle_init:

```
509           platform_add_devices(omap3_beagle_devices,
510                          ARRAY_SIZE(omap3_beagle_devices));
```

Entradas para acesso a dispositivo no diretório /sys

Também é possível disponibilizar no espaço de usuário entradas no diretório /sys usando a interface de arquivo SYSFS. A vantagem do SYSFS é que ele organiza os dispositivos em classes, barramentos, tipos de dispositivos, etc.

Os sysfs exporta detalhes internos de implementação do kernel e depende de suas estruturas internas do kernel. É consenso pelos desenvolvedores do kernel Linux que ele não fornece uma API interna estável. Portanto, há aspectos da interface sysfs que podem não ser estáveis em todas as versões do kernel.

Atualmente há três lugares para classificação de dispositivos: /sys/block, /sys/class e /sys/bus. É planejado que estas pastas não conterão quaisquer diretórios de dispositivo em si, mas apenas listas de symlinks apontando para a árvore unificada /sys/devices. Todos os três lugares tem regras completamente diferentes de como acessar informações de dispositivo.

Pode-se criar entradas no diretorio /sys usando a função int *sysfs_create_bin_file()* e remover com a função *sysfs_remove_bin_file()*. As

funções de escrita e leitura podem ser definidas através da estrutura bin_attribute:

```
struct bin_attribute {
    struct attribute attr;
    size_t size;
    void *private;

    ssize_t (*read)(struct file *, struct kobject *, struct bin_attribute *,
        char *, loff_t, size_t);
    ssize_t (*write)(struct file *, struct kobject *, struct bin_attribute *,
        char *, loff_t, size_t);
    int (*mmap)(struct file *, struct kobject *, struct bin_attribute *attr,
        struct vm_area_struct *vma);
};
```

Exemplo de atribuição para leitura:
https://lxr.free-electrons.com/source/drivers/misc/eeprom/at24.c#L572

Exemplo de atribuição para escrita:
https://lxr.free-electrons.com/source/drivers/misc/eeprom/at24.c#L586

Conclusão

Neste artigo foi mostrado a diferença entre drivers de plataforma e drivers de dispositivo, quais as características de cada um e como eles interagem. Além disso, foram abordadas duas formas de acesso a drivers de dispositivo pelo usuário: entradas no /dev e entradas no /sys.

No próximo artigo nós iremos implementar um driver hello world para mostrar a estrutura básica de um driver de dispositivo.

Artigo escrito em conjunto com Diego Sueiro.

Referências

https://linuxtutorial.info/modules.php?name=MContent&pageid=94

https://www.makelinux.net/ldd3/chp-3-sect-2

https://www.kernel.org/doc/Documentation/i2c/instantiating-devices

https://www.kernel.org/doc/Documentation/sysfs-rules.txt

https://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf

Autor: Vinicius Maciel

Cursando Tecnologia em Telemática no IFCE. Trabalho com programação de Sistemas Embarcados desde 2009. Tenho experiência em desenvolvimento de software para Linux embarcado em C/C++. Criação ou modificação de devices drivers para o sistema operacional Linux. Uso de ferramentas open source para desenvolvimento e debug incluindo gcc, gdb, eclipse.

Publicado originalmente em: https://embarcados.com.br/linux-device-drivers/

Exemplo de driver para Linux Embarcado

Neste artigo daremos um pequeno e simples exemplo de como codificar um device driver para Linux Embarcado. Na verdade esse exemplo funciona no Linux Desktop também, a única diferença será o compilador usado e arquivo para compilação Makefile. Então vamos lá!

Preparando o ambiente para desenvolvimento

Para que possamos executar o driver numa determinada placa cujo processador é diferente daquele onde o código será desenvolvido, nós precisamos usar um toolchain cruzado ou cross-toolchain. Mas se você for executar esse exemplo no PC, você pode usar o compilador gcc nativo. E se você já tem um toolchain instalado em seu sistema, pode pular esta parte do artigo e ir direto para a compilação do kernel.

Das plataformas utilizadas em Linux Embarcado, a plataforma ARM é a mais usada e difundida atualmente. Assim as instruções a seguir visam apenas placas com processadores ARM. Mas você poderia rodar esse exemplo em qualquer placa ARM ou mesmo placas com processadores diferentes como MIPS e PowerPC, fazendo as devidas adaptações. Para processadores MIPS, por exemplo, você deve usar algo como gcc-mips-linux-gnueabi.

Será utilizado para o exemplo o cross-toolchain da Linaro gcc-linaro-arm-linux-gnueabihf-4.7, por ser estável e por rodar em praticamente qualquer distribuição de sua escolha. E se o Linux que roda em sua placa não

suporta ponto flutuante por hardware, use a versão gcc-linaro-arm-linux-gnueabi-4.7, ou seja, sem o hf no final.

1 – Digite o seguinte comando em seu Linux Desktop para baixá-lo:

```
wget
ttps://releases.linaro.org/13.04/components/toolchain/binaries/gcc-linaro-arm-linux-gnueabihf-
.7-2013.04-20130415_linux.tar.xz
```

2 – Descompacte o compilador:

```
$ tar -xJf gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux.tar.xz
```

3 – Crie a pasta toolchains/gnueabi sob /opt:

```
# mkdir -p
/opt/toolchains/gnueabi
```

4 – Mova a pasta do compilador para a pasta /opt/toolchains/gnueabi:

```
# mv gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux /opt/toolchains/gnueabi/
```

5 – Adicione o diretório do compilador ao PATH de seu usuário:

```
$ mcedit ~/.bashrc (use seu editor de texto preferido)
```

E adicione à última linha:

```
export
PATH="/opt/toolchains/gnueabi/gcc-linaro-arm-linux-gnueabihf-4.7-2013.04-20130415_linux/bin/:
$PATH"
```

6 – Teste o funcionamento básico (antes de fazer o teste, feche e abra o terminal novamente):

```
arm-linux-gnueabihf-gcc -v
```

Você deverá ver na última linha: gcc version 4.7.3 20130328 (prerelease) (crosstool-NG linaro-1.13.1-4.7-2013.04-20130415 – Linaro GCC 2013.04)

Para seguirmos com o resto do tutorial, é necessário que o leitor tenha instalado em sua máquina o git, além de ferramentas de desenvolvimento básicas. Instale-as através do comando:

```
# apt-get install git git-email build-essential lzop u-boot-tools
```

O git-email será útil se você pretende enviar patches para algum repositório que utiliza o git, como o kernel Linux.

Gerando uma imagem de kernel para processadores da Allwinner

O kernel Linux usado nas placas com processadores da Allwinner como A13-Olinuxino e Cubieboard é um kernel derivado do trabalho de uma comunidade na internet conhecida como comunidade sunxi. O seu mantenedor é um desenvolvedor chamado Alejandro Mery. Existe também uma versão especial para o u-boot, o u-boot-sunxi. O nome sunxi é pelo fato de o fabricante do processador, Allwinner, ser chinês. Por isso ele foi batizado como linux-sunxi.

Existe também um trabalho que está em andamento para incluir o suporte aos processadores da Allwinner no kernel oficial. Mas ainda muitos periféricos estão sem driver, como o audio e o vídeo. Dessa forma, nós iremos usar o kernel da comunidade sunxi. Para baixá-lo, digite:

```
git clone https://github.com/linux-sunxi/linux-sunxi.git
```

Isso pode demorar um pouco! Depois de concluído, configure e compile o kernel com os seguintes comandos:

```
$ cd linux-sunxi
$ make ARCH=arm sun4i_defconfig
$ make ARCH=arm menuconfig
$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- uImage modules
$ make ARCH=arm INSTALL_MOD_PATH=output modules_install
```

A opção sun4i_defconfig é usada para processadores da Allwinner modelo A10. Mude para a opção adequada de acordo com seu processador. Se tudo deu certo, dentro da pasta do código fonte do kernel, no caminho arch/arm/boot, deverá ter o arquivo de imagem do kernel uImage. E em output/ os módulos do kernel. Agora transfira o kernel e módulos para a placa.

Gerando uma imagem de kernel para processadores da Texas

O kernel Linux mantido pela Texas Instruments está entre um dos mais estáveis entre processadores ARM. Desde de muito tempo, eles têm funcionários dedicados à manutenção e suporte ao kernel Linux. As instruções abordadas aqui se aplicam à placa BeagleBone Black rodando a distribuição Debian. Se o leitor for usar o driver na distribuição Ångström Linux, que já vem gravada na memória da Beaglebone Black na revisão B, deve usar um compilador sem ponto flutuante, ou seja, sem o hf no final. Na BeagleBone Black revisão C, a distribuição gravada na memória flash é o Debian.

Para baixar o kernel para a placa BeagleBone Black execute os seguintes comandos:

```
$ git clone https://github.com/beagleboard/linux.git
$ cd linux
$ git checkout 3.14
```

Agora compile o kernel com os seguintes comandos:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bb.org_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j4 zImage dtbs
LOADADDR=0x82000000
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j4 modules
$ make ARCH=arm INSTALL_MOD_PATH=output modules_install
```

A opção bb.org_defconfig é a configuração de kernel para a placa BeagleBone. E dtbs se refere ao arquivo de Device Tree, explicado no artigo Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo. zImage e modules referem-se ao kernel e módulos do kernel respectivamente. Você deve ter percebido alguns comandos em comum no processo de compilação dos dois processadores. Os mesmos comandos se aplicam para a compilação do kernel para outros processadores ARM, com ou sem suporte a Device Tree.

Se tudo deu certo, deverá existir um arquivo zImage em arch/arm/boot, assim como no caso da Allwinner. Além disso, haverá o arquivo de Device Tree, am335x-boneblack.dtb, em arch/arm/boot/dts e os módulos do kernel em output/. Agora transfira o kernel, o arquivo de Device Tree e os módulos para a placa.

Driver Hello World

O leitor pode estar se perguntando que relação tem baixar ou compilar o kernel com a programação de um device driver. Para compilar o código do driver, é necessário que se tenha o código fonte do kernel. E se você for compilar um driver embutido no kernel, deverá necessariamente recompilar o código fonte do kernel! Além disso, não é possível carregar um driver num kernel que não foi gerado por você.

O driver exemplo será compilado como módulo (loadable module), ou seja, separado do kernel Linux. Mas não apenas isso, o código estará fora da árvore do kernel. Por esse motivo, o módulo precisa ser carregado por meio do comando insmod. Você não poderá usar o modprobe!

Salve o código abaixo como hello.c num diretório qualquer. Pode ser fora da pasta do código fonte do kernel.

```
/* hello.c */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
  pr_alert("Hello World!\n");
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Good bye cruel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Exemplo simples");
```

Vamos agora ao arquivo Makefile. Salve o conteúdo abaixo com o nome Makefile (com M maiúsculo mesmo) na mesma pasta do código fonte do driver.

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /caminho/para/kernel

all:
$(MAKE) -C $(KDIR) M=$$PWD

clean:
$(MAKE) -C $(KDIR) M=$(PWD) clean
endif
```

Altere o valor de KDIR de acordo com o caminho onde o código-fonte do kernel se encontra, o qual foi baixado e compilado anteriormente. Para compilar o driver simplesmente digite:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

O compilador referenciado nesse comando, arm-linux-gnueabihf-gcc, é o mesmo que baixamos nas instruções para preparação do ambiente. Se você usa outro, altere para o nome correto. Essa é a forma como os drivers de código fechado, feitos por certos fabricantes, como Nvidia, são feitos. É bastante prático quando é necessário fazer testes e alterações no código do driver, pelo fato de que não é necessário recompilar o kernel.

O leitor verificará que o driver foi compilado corretamente se o arquivo hello.ko for criado na pasta.

Explicação do código

1. __init

O kernel toma isso como sinal de que a função é usada apenas durante a fase de inicialização e libera recursos de memória utilizados, após a inicialização do kernel, quando o kernel imprime: *Freeing unused kernel memory: 236k freed*

Definido em: include/linux/init.h

2. __exit

Alguns drivers podem ser configurados como módulos (como o nosso exemplo acima). Nesses casos eles usam o exit. Entretanto, se eles são compilados embutidos no kernel, eles não necessitam do exit. Assim como o __init, indica que a função será colocada numa região do binário para ser descartada depois.

3. Cabeçalhos específicos do kernel Linux: linux/xxx.h

Não existe acesso à biblioteca C padrão na API do kernel Linux. Ele tem sua própria API. Para imprimir mensagens, por exemplo, usa-se printk() em vez de printf().

4. Uma função de inicialização – hello_init()

Chamada quando o módulo é carregado, retornando o código 0 em caso de sucesso e um valor negativo em caso de falha. Declarada pela macro module_init().

5. Uma função de cleanup – hello_exit()

Chamada quando o módulo é descarregado e declarada pela macro module_exit().

6. module_init()/module_exit()

A macro module_init() indica que função será chamada quando o módulo for carregado no kernel. O nome da função não importa, embora que <nome_do_módulo>_init() é uma convenção. Já a macro module_exit(), indica a função que será chamada quando o módulo é removido do sistema via rmmod.

7. Declarações de metadados

Declarações de metadados são feitas através das seguintes macros: MODULE_LICENSE(), MODULE_DESCRIPTION() and MODULE_AUTHOR(). Eles indicam a licença, a descrição do que módulo faz e o autor, respectivamente.

Algumas observações importantes:

1. A partir do módulo do kernel, apenas um limitado número de funções do kernel pode ser chamado;
2. Funções e variáveis devem ser explicitamente exportadas pelo kernel para serem visivéis para outros módulos do kernel;
3. Duas macros são utilizadas no kernel para exportar funções e variáveis:

- EXPORT_SYMBOL(nome_do_simbolo), exporta uma função ou variável para todos os módulos;
- EXPORT_SYMBOL_GPL (nome_do_simbolo), exporta uma função ou variável só para módulos GPL.

Um detalhe interessante é que em drivers modernos as macros module_init() e module_exit() não são mais usadas, embora ainda funcionem. Em vez delas, usa-se module_i2c_driver() para drivers de I2C, module_spi_driver para drivers de SPI, e assim por diante. Mas como nosso exemplo é apenas para ilustrar a adição e remoção do driver no sistema, não faria sentido usar module_i2c_driver(), por exemplo.

Transferindo o driver para a placa

Provavelmente, a forma mais fácil de transferir o driver ou um arquivo qualquer para a placa de desenvolvimento é através de ssh. Exemplo:

```
$ scp hello.ko usuario@192.168.1.5:/home/usuario
```

Inserindo e removendo o driver

Para carregar o módulo no Linux da placa, mude para pasta onde foi transferido o arquivo e digite como usuário root:

```
# insmod hello.ko
```

Para remover o módulo do sistema digite:

```
rmmod hello
```

Os mesmos comandos se aplicam ao Linux Desktop para testar o driver. Ao inserir o driver no sistema, o leitor deverá ver a mensagem “Hello World!” e quando removê-lo via rmmod, “Good bye cruel world!”. Caso não veja, simplesmente digite o comando:

```
dmesg | tail
```

O printk é utilizado na programação do kernel para imprimir mensagens para os logs do kernel. E pr_alert é definido como:

```
#define pr_alert(fmt, ...) \

    printk(KERN_ALERT pr_fmt(fmt), ##__VA_ARGS__)
```

A sintaxe de printk é: *printk (“log level” “message”,);*

Os níveis de log indicam a importância da mensagem que está sendo impressa. No kernel são definidos 8 níveis de log no arquivo printk.h:

```
#define KERN_EMERG "" /* system is unusable*/
#define KERN_ALERT "" /* action must be taken immediately*/
#define KERN_CRIT "" /* critical conditions*/
#define KERN_ERR "" /* error conditions*/
#define KERN_WARNING "" /* warning conditions*/
#define KERN_NOTICE "" /* normal but significant condition*/
#define KERN_INFO "" /* informational*/
#define KERN_DEBUG "" /* debug-level messages*/
```

Como normalmente o nível de log no sistema é configurado para 4, o leitor deverá ver as mensagens do driver sem precisar alterar o nível do log do seu sistema. No próximo artigo veremos um exemplo de driver I2C.

Para saber mais

– Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo

– Device Drivers para Linux Embarcado – Introdução

– Como se tornar um especialista em Linux embarcado

– Seminário Linux Embarcado 2011

– Anatomia de um Sistema Linux embarcado

– Embedded Linux Build Systems

– Controlador Industrial com Linux embarcado

– Como o Linux é Construído

– https://elinux.org/Building_BBB_Kernel#Downloading_and_building_the_Linux_Kernel

– https://www.ti.com/lsds/ti/tools-software/linux.page

– https://free-electrons.com/doc/training/linux-kernel/

– https://tuxthink.blogspot.com.br/2012/07/printk-and-console-log-level.html

Referências

https://www.kernel.org/

https://www.ti.com/

https://beagleboard.org/black

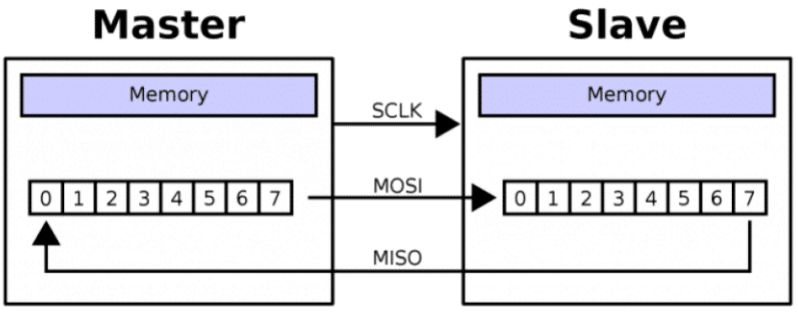
https://embarcados.com.br/linux-device-drivers-diferencas-entre-drivers-de-plataforma-e-de-dispositivo/

Autor: Vinicius Maciel

Cursando Tecnologia em Telemática no IFCE. Trabalho com programação de Sistemas Embarcados desde 2009. Tenho experiência em desenvolvimento de software para Linux embarcado em C/C++. Criação ou modificação de devices drivers para o sistema operacional Linux. Uso de ferramentas open source para desenvolvimento e debug incluindo gcc, gdb, eclipse.

Publicado originalmente em:
https://embarcados.com.br/exemplo-de-driver-para-linux-embarcado/

Comunicação SPI em Linux



O protocolo SPI

Dando continuidade aos artigos sobre acesso a dispositivos típicos de sistemas embarcados no Linux, vamos abordar neste artigo Comunicação SPI em Linux. Por enquanto apenas visando softwares em espaço de usuário.

O protocolo SPI (Serial Peripheral Interface) foi definido pela Motorola. É um protocolo serial síncrono semelhante ao protocolo I2C, mas que utiliza três fios para comunicação e pode alcançar velocidades superiores ao I2C. A interface física é composta pelos sinais MOSI, MISO e SCLK que ligam a todos os dispositivos na forma de barramento. Pode existir também um quarto fio para seleção do dispositivo com o qual a comunicação será feita.

Assim como o I2C, o SPI apresenta o conceito de mestre e escravo da comunicação. O mestre inicia a transferência de dados e controla o sinal de clock para estabelecer o sincronismo. A transferência de dados processa-se em full-duplex sobre os sinais MOSI (Master Output Slave Input) e MISO (Master Input Slave Output). A taxa de transferência é definida pelo processador, no papel de mestre, através do sinal SCLK. A seleção do periférico é feita através do sinal SSn. Nessa configuração, pode haver um mestre e vários escravos no barramento SPI, como mostrado na Figura 1.

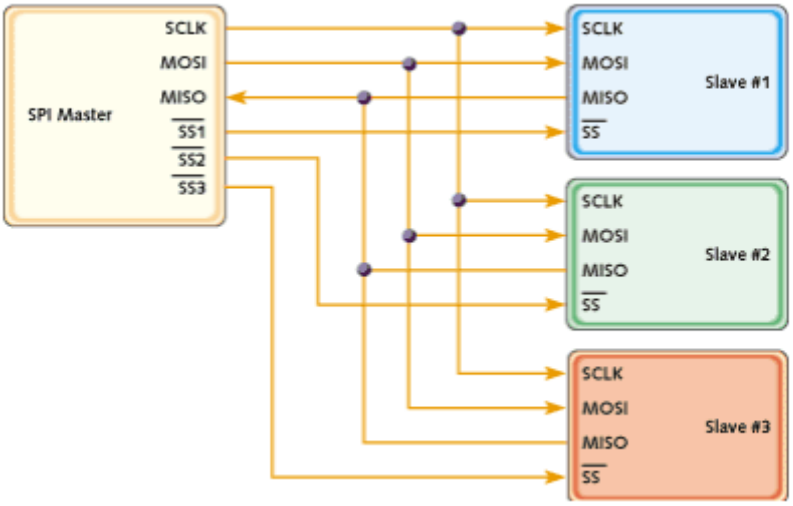


Figura 1 – Ligação entre um mestre e escravos no barramento SPI.

O fluxo de dados é controlado pelo mestre através do sinal de clock SCLK. Só há transferência enquanto o mestre pulsar o sinal SCLK. Em repouso o sinal SCLK encontra-se estável com o valor lógico definido por CPOL.

Habilitando o driver SPI em Linux

Assim como para o I2C, o Linux disponibiliza um driver genérico para SPI. Ele cria entradas no diretório /dev para que o usuário possa acessar o dispositivo SPI como um arquivo através da funções open(), close(), read() e write(). Por ser um driver genérico, ele pode não atender a todos os requisitos do seu projeto. Se você pretende usar o chipset ENC28J60 para construir uma interface ethernet, por exemplo, você precisará de um driver específico. Para esse chipset já existe um driver no Linux. Mas, para um outro chipset que não tenha suporte no Linux, você deverá desenvolvê-lo.

Saiba que, para se comunicar com um dispositivo SPI em Linux, por meio da sua infraestrutura, o processador da placa que você está usando precisa ter o driver do controlador SPI presente no kernel. No artigo Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo explico em detalhes o funcionamento geral dos drivers. Consulte essa referência!

Agora vamos habilitar o driver SPI no kernel. Selecione Device Driver:

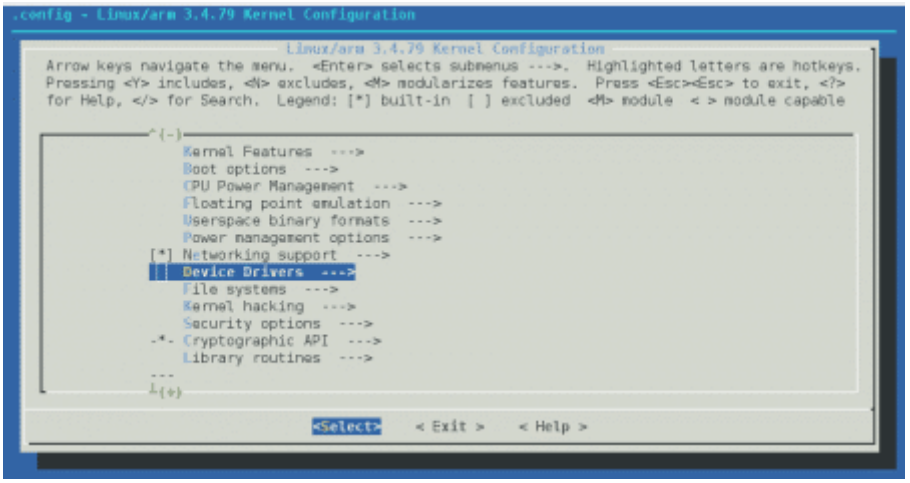


Figura 2 – Habilitando o driver de SPI (menu Device Drivers).

Habilite o suporte a SPI apertando 'y', depois aperte ENTER:

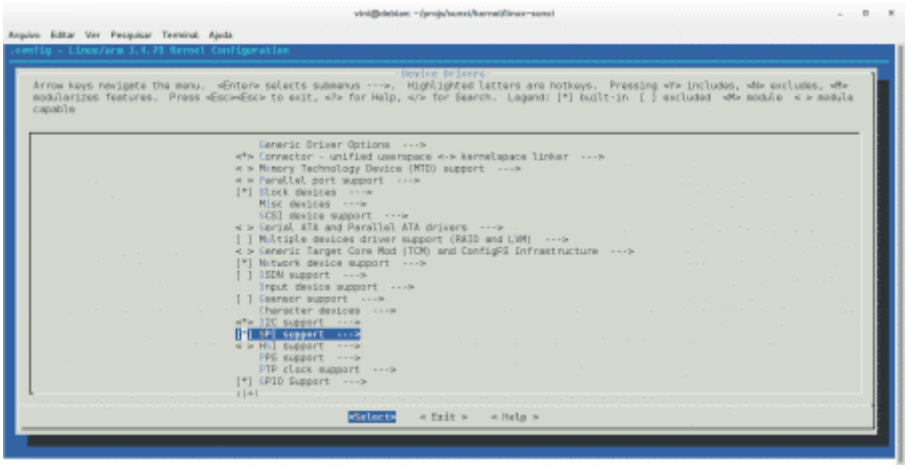


Figura 3 – Habilitando o driver de SPI (item SPI support).

Habilite o suporte ao driver SPI em Linux como módulo apertando ‘m’ ou ‘y’ para deixar o driver embutido no kernel:

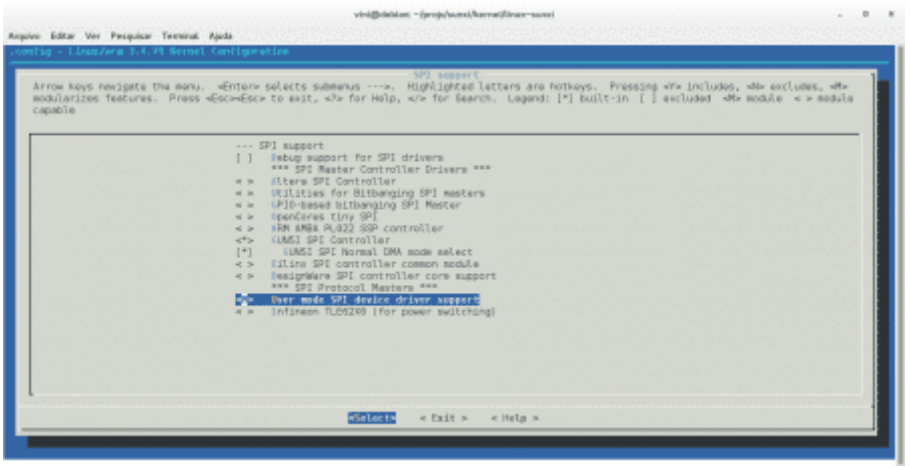


Figura 4 – Habilitando o driver de SPI (módulo ou built-in).

Agora basta compilar o kernel!

Comunicação SPI em Linux

Agora veremos um exemplo de software para escrever e ler um dispositivo qualquer que utiliza o protocolo SPI. O dispositivo mais simples para fazer um teste é uma memória Flash. Mas pode ser usado qualquer outro dispositivo, como um módulo RFID.

Exemplo de software para SPI:

```c
/*
* Exemplo de software para comunicacao SPI
*
* Autor: Vinicius Maciel
*
* comando para compilacao: gcc spi_teste.c -o spi_teste
*
* comando para cross compilacao: arm-linux-gnueabihf-gcc spi_teste.c -o spi_teste
*/
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include "spidev.h"
#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
/* Funcao que imprime mensagem de erro e aborta o programa
*
*/
static void pabort(const char *s)
{
    perror(s);
    abort();
}
// Nome do dispositivo de driver SPI no diretorio /dev
static const char *device = "/dev/spidev0.0"; // (Mude de acordo com sua placa)
// Modo de operacao do controlador SPI do SoC
static uint32_t mode;
// Quantidade de bits da palavra de transferencia SPI
static uint8_t bits = 8;
// Velocidade de comunicao
static uint32_t speed = 1000000;
// Delay entre bytes transferidos, caso necessario
static uint16_t delay;
/*
* Funcao que realiza uma transferencia full duplex,
* ou seja, que escreve e ler ao mesmo tempo
*
* Parametros ----
* fd: inteiro retornado pela funcao open()
* tx: para de escrita para mensagem SPI
* rx: buffer de leitura de mensagem SPI
*/
static void transfer(int fd, uint8_t *tx, uint8_t *rx)
{
    int ret;

    // Estrutura que contem as informacoes para a transmissao da mensagem
    struct spi_ioc_transfer tr = {
        .tx_buf = (unsigned long)tx,
        .rx_buf = (unsigned long)rx,
```

```
        .len = ARRAY_SIZE(tx),
        .delay_usecs = delay,
        .speed_hz = speed,
        .bits_per_word = bits,
    };
    // Funcao que faz a transferencia full duplex
    ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
    if (ret < 1)
        pabort("Nao foi possivel enviar mensagem SPI");
    // Imprimi a mensagem recebida, caso haja alguma
    for (ret = 0; ret < ARRAY_SIZE(rx); ret++) {
        printf("%x ", rx[ret]);
    }
    puts("");
}
int main(int argc, char *argv[])
{
    int ret = 0, i;
    int fd;
    // Buffers para leitura e escrita da mensagem SPI
    uint8_t tx_buffer[32], rx_buffer[32];

    // Abri o dispositivo de driver SPI e retorna um inteiro
    fd = open(device, O_RDWR);
    if (fd < 0)
    pabort("Erro ao abrir o dispositivo");
    // Escolha outros modos de operacao que o SPI da sua placa suporta
    mode = SPI_CPHA | SPI_CPOL | SPI_MODE_0;

    // Configura o modo de operacao
    ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
    if (ret == -1)
        pabort("Erro ao setar o modo do SPI");
    ret = ioctl(fd, SPI_IOC_RD_MODE32, &mode);
    if (ret == -1)
        pabort("Erro ao setar o modo do SPI");
    // Configura o tamanho da palavra de transferencia SPI para escrita
    ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
    if (ret == -1)
        pabort("Erro ao setar os bits por palavra");
    // Configura o tamanho da palavra de transferencia SPI para leitura
    ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
    if (ret == -1)
        pabort("Erro ao ler os bits por palavra");
    // Configura a maxima velocidade de transferencia para escrita
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret == -1)
        pabort("Erro ao setar a velocidade maxima em HZ");
    // Configura a maxima velocidade de transferencia para leitura
    ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
    if (ret == -1)
        pabort("Erro ao ler a velocidade maxima em HZ");
    // Imprimi informacoes de configuracao do SPI
    printf("Modo SPI: 0x%x\n", mode);
```

```
    printf("bits por palavra: %d\n", bits);
    printf("Maxima velocidade: %d Hz (%d KHz)\n", speed, speed/1000);

    // Preenche o buffer de transferencia com numeros de 0 a 9
    for (i = 0; i < 9; i++)
    tx_buffer[i] = i;
    // Faz a transferencia full duplex
    transfer(fd, tx_buffer, rx_buffer);
    // Fecha o dispositivo de driver SPI
    close(fd);
    return ret;
}
```

Para que o código seja compilado, é necessário incluir o seguinte código de cabeçalho:

```
/*
* include/linux/spi/spidev.h
*
* Copyright (C) 2006 SWAPP
*    Andrea Paterniani <a.paterniani@swapp-eng.it>
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
  */
#ifndef SPIDEV_H
#define SPIDEV_H
#include <linux/types.h>
/* User space versions of kernel symbols for SPI clocking modes,
* matching <linux/spi/spi.h>
*/
#define SPI_CPHA        0x01
#define SPI_CPOL        0x02
#define SPI_MODE_0        (0|0)
#define SPI_MODE_1        (0|SPI_CPHA)
#define SPI_MODE_2        (SPI_CPOL|0)
#define SPI_MODE_3        (SPI_CPOL|SPI_CPHA)
#define SPI_CS_HIGH        0x04
#define SPI_LSB_FIRST        0x08
#define SPI_3WIRE        0x10
#define SPI_LOOP        0x20
#define SPI_NO_CS        0x40
#define SPI_READY        0x80
#define SPI_TX_DUAL        0x100
#define SPI_TX_QUAD        0x200
#define SPI_RX_DUAL        0x400
#define SPI_RX_QUAD        0x800
/*---------------------------------------------------------------------------*/
/* IOCTL commands */
#define SPI_IOC_MAGIC            'k'
/**
* struct spi_ioc_transfer - describes a single SPI transfer
* @tx_buf: Holds pointer to userspace buffer with transmit data, or null.
*    If no data is provided, zeroes are shifted out.
* @rx_buf: Holds pointer to userspace buffer for receive data, or null.
* @len: Length of tx and rx buffers, in bytes.
* @speed_hz: Temporary override of the device's bitrate.
* @bits_per_word: Temporary override of the device's wordsize.
* @delay_usecs: If nonzero, how long to delay after the last bit transfer
```

```
*    before optionally deselecting the device before the next transfer.
* @cs_change: True to deselect device before starting the next transfer.
*
* This structure is mapped directly to the kernel spi_transfer structure;
* the fields have the same meanings, except of course that the pointers
* are in a different address space (and may be of different sizes in some
* cases, such as 32-bit i386 userspace over a 64-bit x86_64 kernel).
* Zero-initialize the structure, including currently unused fields, to
* accommodate potential future updates.
*
* SPI_IOC_MESSAGE gives userspace the equivalent of kernel spi_sync().
* Pass it an array of related transfers, they'll execute together.
* Each transfer may be half duplex (either direction) or full duplex.
*
*    struct spi_ioc_transfer mesg[4];
*    ...
*    status = ioctl(fd, SPI_IOC_MESSAGE(4), mesg);
*
* So for example one transfer might send a nine bit command (right aligned
* in a 16-bit word), the next could read a block of 8-bit data before
* terminating that command by temporarily deselecting the chip; the next
* could send a different nine bit command (re-selecting the chip), and the
* last transfer might write some register values.
*/
struct spi_ioc_transfer {
    __u64        tx_buf;
    __u64        rx_buf;
    __u32        len;
    __u32        speed_hz;
    __u16        delay_usecs;
    __u8        bits_per_word;
    __u8        cs_change;
    __u8        tx_nbits;
    __u8        rx_nbits;
    __u16        pad;
    /* If the contents of 'struct spi_ioc_transfer' ever change
        * incompatibly, then the ioctl number (currently 0) must change;
        * ioctls with constant size fields get a bit more in the way of
        * error checking than ones (like this) where that field varies.
        *
        * NOTE: struct layout is the same in 64bit and 32bit userspace.
        */
};
/* not all platforms use <asm-generic/ioctl.h> or _IOC_TYPECHECK() ... */
#define SPI_MSGSIZE(N) \
    ((((N)*(sizeof (struct spi_ioc_transfer))) < (1 << _IOC_SIZEBITS)) \
        ? ((N)*(sizeof (struct spi_ioc_transfer))) : 0)
#define SPI_IOC_MESSAGE(N) _IOW(SPI_IOC_MAGIC, 0, char[SPI_MSGSIZE(N)])
/* Read / Write of SPI mode (SPI_MODE_0..SPI_MODE_3) (limited to 8 bits) */
#define SPI_IOC_RD_MODE            _IOR(SPI_IOC_MAGIC, 1, __u8)
#define SPI_IOC_WR_MODE            _IOW(SPI_IOC_MAGIC, 1, __u8)
/* Read / Write SPI bit justification */
#define SPI_IOC_RD_LSB_FIRST        _IOR(SPI_IOC_MAGIC, 2, __u8)
#define SPI_IOC_WR_LSB_FIRST        _IOW(SPI_IOC_MAGIC, 2, __u8)
```

```
/* Read / Write SPI device word length (1..N) */
#define SPI_IOC_RD_BITS_PER_WORD    _IOR(SPI_IOC_MAGIC, 3, __u8)
#define SPI_IOC_WR_BITS_PER_WORD    _IOW(SPI_IOC_MAGIC, 3, __u8)
/* Read / Write SPI device default max speed hz */
#define SPI_IOC_RD_MAX_SPEED_HZ        _IOR(SPI_IOC_MAGIC, 4, __u32)
#define SPI_IOC_WR_MAX_SPEED_HZ        _IOW(SPI_IOC_MAGIC, 4, __u32)
/* Read / Write of the SPI mode field */
#define SPI_IOC_RD_MODE32        _IOR(SPI_IOC_MAGIC, 5, __u32)
#define SPI_IOC_WR_MODE32        _IOW(SPI_IOC_MAGIC, 5, __u32)
#endif /* SPIDEV_H */
```

Saiba mais sobre SPI

– Comunicação SPI

– Linux Kernel Documentation

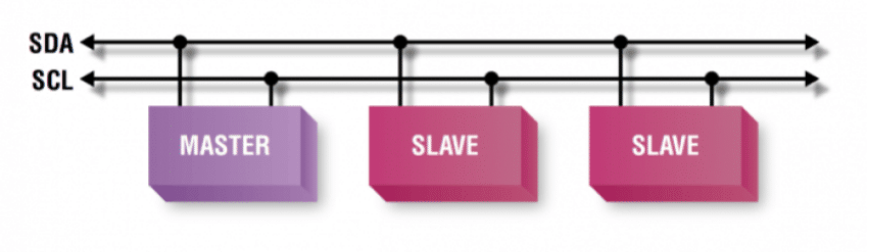
– Gerando PWM com a Raspberry PI

Autor:Vinicius Maciel

Cursando Tecnologia em Telemática no IFCE. Trabalho com programação de Sistemas Embarcados desde 2009. Tenho experiência em desenvolvimento de software para Linux embarcado em C/C++. Criação ou modificação de devices drivers para o sistema operacional Linux. Uso de ferramentas open source para desenvolvimento e debug incluindo gcc, gdb, eclipse.

Publicado orinalmente em: https://embarcados.com.br/comunicacao-spi-em-linux/

Exemplo de Device Driver I2C para Linux Embarcado



Depois de um longo tempo, iremos agora retomar a série sobre Device Drivers com um exemplo de device driver I2C para Linux Embarcado. Caso não tenha lido os artigos anteriores, sugiro que os leia para obter um conhecimento básico.

Em especial, leia o artigo que explica as diferenças entre device drivers e drivers de plataforma. Outro pré-requisito muito importante é o uso de ponteiros. Eles são muito usados no kernel Linux e nós veremos ponteiros por todo o código do driver.

Aqui será apresentado um exemplo de device driver I2C baseado em um driver já existente, mas reduzido ao máximo para torná-lo mais simples. O driver em que foi baseado o exemplo é o driver at24.c, que é usado para ler memórias I2C da família at24c como, por exemplo, a at24c512. Ele usa a abordagem de sysfs e cria entradas no diretório /sys para representar a memória como um arquivo e, assim, permitir que o usuário facilmente leia e escreva no dispositivo. O driver pode ser testado em qualquer placa com Linux embarcado que tenha os pinos I2C disponíveis.

Começaremos mostrando o código do device driver, seguido de seu Makefile, que é usado para compilá-lo. E será apresentado um exemplo de software para testar o device driver. Iniciemos baixando o código do device driver com o seguinte comando:

```
$ git clone https://github.com/vinifr/drivers.git
```

E entre na pasta do driver I2C dentro do kernel Linux com o comando:

```
$ cd drivers/i2c_sys/
```

O código do driver pode ser acessado aqui, mas é exibido abaixo para referência.

```
/*
* at24.c - handle most I2C EEPROMs
*
* Copyright (C) 2005-2007 David Brownell
* Copyright (C) 2008 Wolfram Sang, Pengutronix
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or
* (at your option) any later version.
*/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/mutex.h>
#include <linux/sysfs.h>
#include <linux/mod_devicetable.h>
#include <linux/log2.h>
#include <linux/bitops.h>
#include <linux/jiffies.h>
#include <linux/of.h>
#include <linux/i2c.h>
#include <linux/platform_data/at24.h>


struct at24_data {
    struct at24_platform_data chip;
    int use_smbus;

    /*
        * Lock protects against activities from other Linux tasks,
        * but not from changes by other I2C masters.
        */
    struct mutex lock;
    struct bin_attribute bin;

    u8 *writebuf;
    unsigned write_max;
    unsigned num_addresses;

    /*
        * Some chips tie up multiple I2C addresses; dummy devices reserve
        * them for us, and we'll use them with SMBus calls.
        */
    struct i2c_client *client[];
};

/*
* This parameter is to help this driver avoid blocking other drivers out
* of I2C for potentially troublesome amounts of time. With a 100 kHz I2C
* clock, one 256 byte read takes about 1/43 second which is excessive;
* but the 1/170 second it takes at 400 kHz may be quite reasonable; and
```

```
* at 1 MHz (Fm+) a 1/430 second delay could easily be invisible.
*
* This value is forced to be a power of two so that writes align on pages.
*/
static unsigned io_limit = 128;
module_param(io_limit, uint, 0);
MODULE_PARM_DESC(io_limit, "Maximum bytes per I/O (default 128)");

/*
* Specs often allow 5 msec for a page write, sometimes 20 msec;
* it's important to recover from write timeouts.
*/
static unsigned write_timeout = 25;
module_param(write_timeout, uint, 0);
MODULE_PARM_DESC(write_timeout, "Time (in ms) to try writes (default 25)");

#define AT24_SIZE_BYTELEN 5
#define AT24_SIZE_FLAGS 8

#define AT24_BITMASK(x) (BIT(x) - 1)

/* create non-zero magic value for given eeprom parameters */
#define AT24_DEVICE_MAGIC(_len, _flags)         \
    ((1 << AT24_SIZE_FLAGS | (_flags))          \
        << AT24_SIZE_BYTELEN | ilog2(_len))

static const struct i2c_device_id at24_ids[] = {
    /* needs 8 addresses as A0-A2 are ignored */
    { "24c00", AT24_DEVICE_MAGIC(128 / 8, AT24_FLAG_TAKE8ADDR) },
    /* old variants can't be handled with this generic entry! */
    { "24c01", AT24_DEVICE_MAGIC(1024 / 8, 0) },
    { "24c02", AT24_DEVICE_MAGIC(2048 / 8, 0) },
    /* spd is a 24c02 in memory DIMMs */
    { "spd", AT24_DEVICE_MAGIC(2048 / 8,
        AT24_FLAG_READONLY | AT24_FLAG_IRUGO) },
    { "24c04", AT24_DEVICE_MAGIC(4096 / 8, 0) },
    /* 24rf08 quirk is handled at i2c-core */
    { "24c08", AT24_DEVICE_MAGIC(8192 / 8, 0) },
    { "24c16", AT24_DEVICE_MAGIC(16384 / 8, 0) },
    { "24c32", AT24_DEVICE_MAGIC(32768 / 8, AT24_FLAG_ADDR16) },
    { "24c64", AT24_DEVICE_MAGIC(65536 / 8, AT24_FLAG_ADDR16) },
    { "24c128", AT24_DEVICE_MAGIC(131072 / 8, AT24_FLAG_ADDR16) },
    { "24c256", AT24_DEVICE_MAGIC(262144 / 8, AT24_FLAG_ADDR16) },
    { "24c512", AT24_DEVICE_MAGIC(524288 / 8, AT24_FLAG_ADDR16) },
    { "24c1024", AT24_DEVICE_MAGIC(1048576 / 8, AT24_FLAG_ADDR16) },
    { "at24", 0 },
    { /* END OF LIST */ }
};
MODULE_DEVICE_TABLE(i2c, at24_ids);

/*-------------------------------------------------------------------------*/

/*
* This routine supports chips which consume multiple I2C addresses. It
```

```
* computes the addressing information to be used for a given r/w request.
* Assumes that sanity checks for offset happened at sysfs-layer.
*/
static struct i2c_client *at24_translate_offset(struct at24_data *at24,
        unsigned *offset)
{
    unsigned i;

    if (at24->chip.flags & AT24_FLAG_ADDR16) {
        i = *offset >> 16;
        *offset &= 0xffff;
    } else {
        i = *offset >> 8;
        *offset &= 0xff;
    }

    return at24->client[i];
}

static ssize_t at24_eeprom_read(struct at24_data *at24, char *buf,
        unsigned offset, size_t count)
{
    struct i2c_msg msg[2];
    u8 msgbuf[2];
    struct i2c_client *client;
    unsigned long timeout, read_time;
    int status, i;

    memset(msg, 0, sizeof(msg));

    client = at24_translate_offset(at24, &offset);

    if (count > io_limit)
        count = io_limit;

    i = 0;
    if (at24->chip.flags & AT24_FLAG_ADDR16)
        msgbuf[i++] = offset >> 8;
    msgbuf[i++] = offset;

    msg[0].addr = client->addr;
    msg[0].buf = msgbuf;
    msg[0].len = i;

    msg[1].addr = client->addr;
    msg[1].flags = I2C_M_RD;
    msg[1].buf = buf;
    msg[1].len = count;

    /*
        * Reads fail if the previous write didn't complete yet. We may
        * loop a few times until this one succeeds, waiting at least
        * long enough for one entire page write to work.
        */
```

```
    timeout = jiffies + msecs_to_jiffies(write_timeout);
    do {
        read_time = jiffies;

        status = i2c_transfer(client->adapter, msg, 2);
        if (status == 2)
            status = count;
        dev_dbg(&client->dev, "read %zu@%d --> %d (%ld)\n",
                count, offset, status, jiffies);

        if (status == count)
            return count;

        /* REVISIT: at HZ=100, this is sloooow */
        msleep(1);
    } while (time_before(read_time, timeout));

    return -ETIMEDOUT;
}

static ssize_t at24_read(struct at24_data *at24,
        char *buf, loff_t off, size_t count)
{
    ssize_t retval = 0;

    /*
     * Read data from chip, protecting against concurrent updates
     * from this host, but not from other I2C masters.
     */
    mutex_lock(&at24->lock);

    while (count) {
        ssize_t    status;

        status = at24_eeprom_read(at24, buf, off, count);
        if (status <= 0) {
            if (retval == 0)
                retval = status;
            break;
        }
        buf += status;
        off += status;
        count -= status;
        retval += status;
    }

    mutex_unlock(&at24->lock);

    return retval;
}

static ssize_t at24_bin_read(struct file *filp, struct kobject *kobj,
        struct bin_attribute *attr,
        char *buf, loff_t off, size_t count)
```

```
{
    struct at24_data *at24;

    at24 = dev_get_drvdata(container_of(kobj, struct device, kobj));
    return at24_read(at24, buf, off, count);
}


/*
* Note that if the hardware write-protect pin is pulled high, the whole
* chip is normally write protected. But there are plenty of product
* variants here, including OTP fuses and partial chip protect.
*
* We only use page mode writes; the alternative is sloooow. This routine
* writes at most one page.
*/
static ssize_t at24_eeprom_write(struct at24_data *at24, const char *buf,
        unsigned offset, size_t count)
{
    struct i2c_client *client;
    struct i2c_msg msg;
    ssize_t status;
    unsigned long timeout, write_time;
    unsigned next_page;
    int i;

    /* Get corresponding I2C address and adjust offset */
    client = at24_translate_offset(at24, &offset);

    /* write_max is at most a page */
    if (count > at24->write_max)
        count = at24->write_max;

    /* Never roll over backwards, to the start of this page */
    next_page = roundup(offset + 1, at24->chip.page_size);
    if (offset + count > next_page)
        count = next_page - offset;

    i = 0;
    msg.addr = client->addr;
    msg.flags = 0;

    /* msg.buf is u8 and casts will mask the values */
    msg.buf = at24->writebuf;
    if (at24->chip.flags & AT24_FLAG_ADDR16)
        msg.buf[i++] = offset >> 8;

    msg.buf[i++] = offset;
    memcpy(&msg.buf[i], buf, count);
    msg.len = i + count;

    /*
       * Writes fail if the previous one didn't complete yet. We may
       * loop a few times until this one succeeds, waiting at least
```

```
        * long enough for one entire page write to work.
        */
    timeout = jiffies + msecs_to_jiffies(write_timeout);
    do {
        write_time = jiffies;

        status = i2c_transfer(client->adapter, &msg, 1);
        if (status == 1)
            status = count;

        dev_dbg(&client->dev, "write %zu@%d --> %zd (%ld)\n",
                count, offset, status, jiffies);

        if (status == count)
            return count;

        /* REVISIT: at HZ=100, this is sloooow */
        msleep(1);
    } while (time_before(write_time, timeout));

    return -ETIMEDOUT;
}

static ssize_t at24_write(struct at24_data *at24, const char *buf, loff_t off,
              size_t count)
{
    ssize_t retval = 0;

    /*
        * Write data to chip, protecting against concurrent updates
        * from this host, but not from other I2C masters.
        */
    mutex_lock(&at24->lock);

    while (count) {
        ssize_t    status;

        status = at24_eeprom_write(at24, buf, off, count);
        if (status <= 0) {
            if (retval == 0)
                retval = status;
            break;
        }
        buf += status;
        off += status;
        count -= status;
        retval += status;
    }

    mutex_unlock(&at24->lock);

    return retval;
}
```

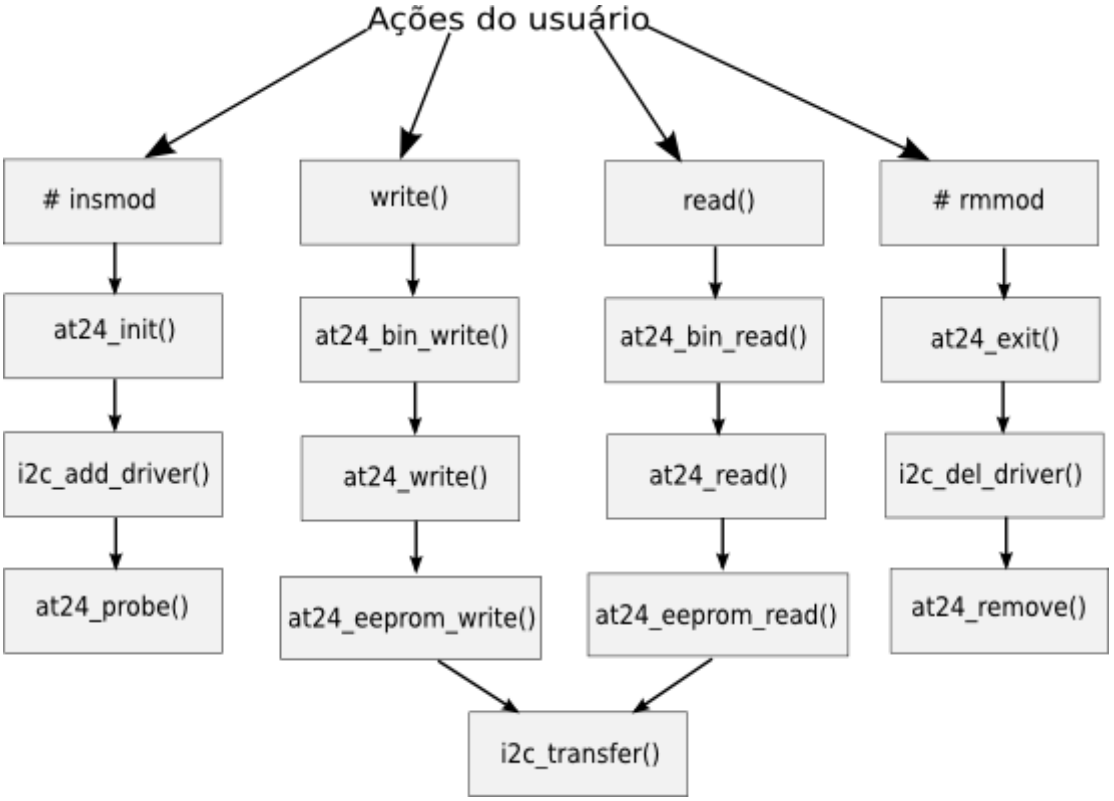
```
static ssize_t at24_bin_write(struct file *filp, struct kobject *kobj,
        struct bin_attribute *attr,
        char *buf, loff_t off, size_t count)
{
    struct at24_data *at24;

    if (unlikely(off >= attr->size))
        return -EFBIG;

    at24 = dev_get_drvdata(container_of(kobj, struct device, kobj));
    return at24_write(at24, buf, off, count);
}

#ifdef CONFIG_OF
static void at24_get_ofdata(struct i2c_client *client,
        struct at24_platform_data *chip)
{
    const __be32 *val;
    struct device_node *node = client->dev.of_node;

    if (node) {
        if (of_get_property(node, "read-only", NULL))
            chip->flags |= AT24_FLAG_READONLY;
        val = of_get_property(node, "pagesize", NULL);
        if (val)
            chip->page_size = be32_to_cpup(val);
    }
}
#else
static void at24_get_ofdata(struct i2c_client *client,
        struct at24_platform_data *chip)
{ }
#endif /* CONFIG_OF */

static int at24_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    struct at24_platform_data chip;
    //bool writable;
    int use_smbus = 0;
    struct at24_data *at24;
    int err;
    unsigned i, num_addresses;
    kernel_ulong_t magic;
    unsigned write_max;

    if (client->dev.platform_data) {
        chip = *(struct at24_platform_data *)client->dev.platform_data;
    } else {
        if (!id->driver_data)
            return -ENODEV;

        magic = id->driver_data;
        chip.byte_len = BIT(magic & AT24_BITMASK(AT24_SIZE_BYTELEN));
        magic >>= AT24_SIZE_BYTELEN;
```

```
    chip.flags = magic & AT24_BITMASK(AT24_SIZE_FLAGS);
    /*
    * This is slow, but we can't know all eeproms, so we better
    * play safe. Specifying custom eeprom-types via platform_data
    * is recommended anyhow.
    */
    chip.page_size = 1;

    /* update chipdata if OF is present */
    at24_get_ofdata(client, &chip);

    chip.setup = NULL;
    chip.context = NULL;
}


/* Use I2C operations unless we're stuck with SMBus extensions. */
if (i2c_check_functionality(client->adapter,
    I2C_FUNC_SMBUS_BYTE_DATA)) {
    use_smbus = I2C_SMBUS_BYTE_DATA;
} else {
    return -EPFNOSUPPORT;
}


if (chip.flags & AT24_FLAG_TAKE8ADDR)
    num_addresses = 8;
else
    num_addresses =    DIV_ROUND_UP(chip.byte_len,
        (chip.flags & AT24_FLAG_ADDR16) ? 65536 : 256);

at24 = devm_kzalloc(&client->dev, sizeof(struct at24_data) +
    num_addresses * sizeof(struct i2c_client *), GFP_KERNEL);
if (!at24)
    return -ENOMEM;

mutex_init(&at24->lock);
at24->use_smbus = use_smbus;
at24->chip = chip;
at24->num_addresses = num_addresses;

/*
    * Export the EEPROM bytes through sysfs, since that's convenient.
    * By default, only root should see the data (maybe passwords etc)
    */
sysfs_bin_attr_init(&at24->bin);
at24->bin.attr.name = "eeprom";
at24->bin.attr.mode = chip.flags & AT24_FLAG_IRUGO ? S_IRUGO : S_IRUSR;
at24->bin.read = at24_bin_read;
at24->bin.size = chip.byte_len;
at24->bin.write = at24_bin_write;
at24->bin.attr.mode |= S_IWUSR;

write_max = chip.page_size;
```

```
    if (write_max > io_limit)
        write_max = io_limit;
    at24->write_max = write_max;
    /* buffer (data + address at the beginning) */
    at24->writebuf = devm_kzalloc(&client->dev,
        write_max + 2, GFP_KERNEL);
    if (!at24->writebuf)
        return -ENOMEM;

    at24->client[0] = client;

    err = sysfs_create_bin_file(&client->dev.kobj, &at24->bin);
    if (err)
        goto err_clients;

    i2c_set_clientdata(client, at24);

    dev_info(&client->dev, "%zu byte %s EEPROM, %u bytes/write\n",
        at24->bin.size, client->name, at24->write_max);
    if (use_smbus == I2C_SMBUS_WORD_DATA ||
        use_smbus == I2C_SMBUS_BYTE_DATA) {
        dev_notice(&client->dev, "Falling back to %s reads, "
              "performance will suffer\n", use_smbus ==
              I2C_SMBUS_WORD_DATA ? "word" : "byte");
    }

    return 0;

err_clients:
    for (i = 1; i < num_addresses; i++)
        if (at24->client[i])
            i2c_unregister_device(at24->client[i]);

    return err;
}

static int at24_remove(struct i2c_client *client)
{
    struct at24_data *at24;
    int i;

    at24 = i2c_get_clientdata(client);
    sysfs_remove_bin_file(&client->dev.kobj, &at24->bin);

    for (i = 1; i < at24->num_addresses; i++)
        i2c_unregister_device(at24->client[i]);

    return 0;
}

/*-------------------------------------------------------------------------*/

static struct i2c_driver at24_driver = {
    .driver = {
```

```
        .name = "at24c",
        .owner = THIS_MODULE,
    },
    .probe = at24_probe,
    .remove = at24_remove,
    .id_table = at24_ids,
};

static int __init at24_init(void)
{
    if (!io_limit) {
        pr_err("at24: io_limit must not be 0!\n");
        return -EINVAL;
    }

    io_limit = rounddown_pow_of_two(io_limit);
    return i2c_add_driver(&at24_driver);
}
module_init(at24_init);

static void __exit at24_exit(void)
{
    i2c_del_driver(&at24_driver);
}
module_exit(at24_exit);

MODULE_DESCRIPTION("Driver for I2C EEPROMs");
MODULE_AUTHOR("David Brownell and Wolfram Sang");
MODULE_LICENSE("GPL");
```

Visão geral do driver



Visão geral do driver I2C para Linux Embarcado.

Explicação do código do driver

Geralmente os códigos de device drivers para Linux Embarcado seguem uma lógica que começa de baixo para cima. Assim, normalmente encontramos as funções de inicialização e de probe()/remove() nas últimas linhas do código. Então passemos agora para a explicação das principais linhas do código do driver.

Nas linhas 513, 514 e 515 temos as macros de metadados já explicadas no artigo Exemplo de driver para Linux Embarcado. E também nas linhas 505 e 511 temos as funções module_init()/module_exit(), também já explicadas no referido artigo.

1. i2c_add_driver(): linha 503

Essa função é responsável por registrar um dispositivo I2C no subsistema I2C do kernel Linux, que irá se comunicar com um dispositivo I2C slave. Ela recebe como parâmetro um ponteiro para estrutura struct i2c_driver. Essa estrutura guarda informações sobre o driver no membro driver e ponteiros para as funções

de callback probe() e remove() nos respectivos membros da estrutura. O membro id_table guarda a lista de dispositivos I2C suportados por esse driver.

Essa é a forma de dizer ao sistema que existe um driver chamado at24, que é capaz de se comunicar com os dispositivos listados em id_table e que tem como funções de callback at24_probe() e at24_remove().

2. i2c_del_driver(): linha 509

Essa função remove um dispositivo I2C no subsistema I2C do kernel. Assim como i2c_add_driver(), ela recebe como parâmetro um ponteiro para struct i2c_driver.

3. at24_probe(): linha 359

Todo device driver precisa fornecer ao menos a definição das funções de probe() e remove(). A função probe() é algo obscuro e difícil de depurar. Ela é chamada quando registramos o driver I2C usando i2c_add_driver(). Aqui está a sequência de chamadas para o caso do barramento I2C:

- i2c_add_driver()
- i2c_register_driver()
- driver_register()
- bus_add_driver()
- driver_attach()
- __driver_attach()
- driver_probe_device()
- really_probe() – aqui parece um beco sem saída, mas seguimos adiante
- i2c_device_probe() – chamada por dev->bus->probe dentro de really_probe
- at24_probe() – chamada por driver->probe() dentro de i2c_device_probe

Observe que a função i2c_device_probe() do barramento I2C, que está dentro de i2c-core.c, é definida na estrutura i2c_bus_type:

```
784 struct bus_type i2c_bus_type = {
785             .name           = "i2c",
786             .match           = i2c_device_match,
787             .probe           = i2c_device_probe,
788             .remove         = i2c_device_remove,
789             .shutdown       = i2c_device_shutdown,
790 };
791 EXPORT_SYMBOL_GPL(i2c_bus_type);
```

Quando a instrução driver->probe() é executada, é a função i2c_device_probe() que será chamada, uma vez que estamos usando o barramento I2C e esta função foi definida como callback de *.probe*, como mostra o código acima.

O ponteiro *struct i2c_client *client*, o primeiro parâmetro da função at24_probe(), provém da função i2c_new_device(), que é chamada em arquivos de *board*. Dentro da função at24_probe(), temos uma série de outras funções importantes.

3.1. at24_get_ofdata(): linhas 339, 354 e 388

Essa função recebe como parâmetro um ponteiro para *struct i2c_client* e outro ponteiro para *struct at24_platform_data*. Ela só executa algo na definição da linha 339, e serve para atualizar os parâmetros relacionados a modos de acesso (leitura/escrita) e o tamanho da página da memória. A sua existência, na linha 339, está condicionada ao teste de pré-processador *#ifdef CONFIG_OF*, o qual verifica se o kernel está usando Device Tree. Essas propriedades são lidas no Device Tree usando a função of_get_property(). As propriedades do nó são recuperadas por meio do ponteiro *client->dev.of_node*, atribuido na linha 343.

Obtenha mais informação sobre pré-processador e Device Tree.

3.2. i2c_check_functionality(): linha 396

Essa função verifica se o controlador I2C suporta determinada funcionalidade. No teste da linha 396 estamos verificando se há suporte para escrita e leitura de bytes por parte do controlador I2C. Então isso vai depender da placa que está sendo usada. Em geral, a maioria dos controladores I2C suportam leitura e escrita de bytes. Também é possivel verificar se há suporte para leitura e escrita de words ou operações em blocos. Veja a lista completa aqui.

3.3. devm_kzalloc(): linha 410

Essa função é usada para alocar memória. Porém, diferente da antiga kzalloc(), a memória é liberada automaticamente quando o driver é removido. Ela retorna

um ponteiro para a área de memória alocada em caso de sucesso ou NULL em caso de falha.

3.4. sysfs_bin_attr_init(): linha 424

Inicializa uma estrutura *bin_attribute* alocada dinamicamente. Ela é necessária apenas quando o atributo *lockdep* está presente no kernel. Por sua vez, o *lockdep* irá imprimir uma descrição da situação e um traço da *stack* para o log do kernel quando ele encontra uma sequência de locking que pode potencialmente causar um deadlock. Nas linhas 425 a 430 temos a inicialização de alguns membros da estrutura *bin_attribute*. Entre eles temos *bin.read* e *bin.write*, que definem as funções para leitura e escrita, respectivamente, do arquivo binário criado sob o diretório */sys*.

3.5. sysfs_create_bin_file(&client->dev.kobj, &at24->bin): linha 444

Cria um arquivo binário para o objeto passado como parâmetro sob o diretório */sys*. Recebe um ponteiro para *struct kobject* e um ponteiro constante para *struct bin_attribute*. O nome do arquivo é indicado em *bin.attr.name*. Essa função é mais indicada quando um arquivo precisa transferir grandes quantidades de dados, como uma memória.

3.6. i2c_set_clientdata(client, at24): linha 448

Essa função é usada para guardar um ponteiro *void *driver_data* que armazena informações do driver. No nosso caso, estamos guardando as informações da estrutura *at24* dentro da estrutura de client. Mais tarde, se for necessário, podemos recuperar esse ponteiro usando *i2c_get_clientdata()*. Também é possível recuperar esse ponteiro usando a função *container_of()* ou uma variante dela.

4. at24_remove: linha 469

Essa função é chamada quando o driver é removido do subsistema I2C do kernel, processo que é iniciado por i2c_del_driver(). Ela passa por várias funções, assim como probe(), e é chamada por *dev->bus->remove(dev)*, dentro de __device_release_driver(). As funções dentro dela executam a liberação de recursos.

4.1. i2c_get_clientdata(): linha 474

Como dito anteriormente, recupera o ponteiro salvo por i2c_set_clientdata(). Ela recebe como parâmetro um ponteiro para *struct i2c_client*, e retorna um ponteiro para *void*.

4.2. sysfs_remove_bin_file(): 475

Remove o arquivo binário do objeto passado como parâmetro e também apaga a entrada no diretório */sys*, criado por sysfs_create_bin_file().

4.3. i2c_unregister_device(): linha 478

Remove um dispositivo do subsistema I2C do kernel, que é o oposto ao que é feito pela função i2c_new_device().

5. at24_bin_write(): linha 325

Essa função é chamada pelo kernel quando uma operação de escrita é realizada sobre o arquivo binário criado por sysfs_create_bin_file(). Pode ser um comando *echo* ou a função de chamada de sistema write().

5.1. dev_get_drvdata(), container_of(): linha 334

Essa função retorna o ponteiro armazenado previamente por i2c_set_clientdata() em *driver_data*, executando a seguinte linha: *return dev->driver_data*. Ela precisa de um ponteiro para *struct device*. Como não temos nenhum ponteiro para *struct device* dentro de at24_bin_write(), precisamos de algum artifício para recuperar esse ponteiro. Isso é feito justamente por container_of(), que aproveita o ponteiro *struct kobject *kobj* vindo de *at24_bin_write* e recupera o ponteiro para *struct device*. Agora que temos um ponteiro válido na variável *at24*, chamamos at24_write(at24, buf, off, count).

5.2. at24_write(): linhas 335 e 294

Escreve os dados passados em *buf* na memória I2C, até atingir o número de bytes igual a *count*. Por enquanto, não iremos falar de mutex_lock() e mutex_unlock().

5.2.1. at24_eeprom_write(): linhas 308 e 233

Escreve o número máximo de bytes indicado por *write_max* ou *count* se for menor. Primeiro é recuperado o endereço do cliente por meio de at24_translate_offset(). Depois o valor de *count* é limitado por *at24->write_max*. A variável *msg* contém os parâmetros que serão passados para i2c_transfer(). O

membro *msg.buf* recebe a memória alocada em at24_probe() por *at24->writebuf*. Na posição zero, *msg.buf[0]*, é guardado o endereço de memória a ser escrito. E nas posições posteriores são guardados os dados a serem escritos, por meio de memcpy(). E *msg.len* guarda o número de bytes a serem escritos, incluindo o endereço mais os dados (i + count).

5.2.2. msecs_to_jiffies(): linha 273

Converte o valor passado em milissegundos para jiffies. A variável global *jiffies* mantém o número de ticks que ocorreram desde que o sistema foi inicializado. *Tick*, por sua vez, refere-se a uma interrupção do timer do sistema que, por padrão em processadores ARM, ocorre a cada 1 ms. Considerando que a variável *write_timeout* é por padrão 25, quando fazemos:

```
timeout = jiffies + msecs_to_jiffies(write_timeout);
```

Significa um tempo de 25 ms a partir do momento em que a função está sendo chamada.

5.2.3. i2c_transfer(): linha 277

Essa função é quem de fato faz a transferência de bytes no barramento I2C, tanto escrita como leitura. A operação de escrita ou leitura é indicada no membro *msg.flags*, sendo que zero indica escrita. Veja na linha 257 (*msg.flags = 0*). Ela precisa de um ponteiro para *struct i2c_adapter* o qual indica o driver do controlador I2C, um ponteiro para *struct i2c_msg* que indica a mensagem para transferir e um inteiro para indicar o número de mensagens para transferir.

5.2.4. time_before(t1, t2): linha 289

Testa se o tempo atual *t1* alcançou um tempo posterior *t2*. Se o tempo posterior não foi atingido, retorna verdadeiro. O primeiro parâmetro é o jiffie atual e o segundo parâmetro, o jiffie posterior.

6. at24_bin_read(): linha 214

Essa função é chamada pelo kernel quando uma operação de leitura é realizada sobre o arquivo binário criado por sysfs_create_bin_file(). Pode ser um comando *cat* ou a função de chamada de sistema read(). As funções chamadas em at24_bin_read() são as mesmas chamadas em at24_bin_write() com pequenas diferenças. Vejamos então apenas as funções em que há diferenças.

6.1. at24_eeprom_read(): linhas 197 e 128

Ler o número de bytes máximo indicado por *io_limit* ou *count* se menor. Nessa função temos um *array struct i2c_msg msg[2]*, que irá guardar duas mensagens. Na primeira mensagem, indicada por *msg[0]*, temos a operação de escrita, pois *msg[0].flag* foi omitido e é inicializado com zero. E na segunda mensagem, que é indicada por *msg[1]*, temos a operação de leitura. Veja na linha 154 (*msg[1].flags = I2C_M_RD*). As duas mensagens são transferidas quando indicamos 2 no último parametro de i2c_transfer(). Veja na linha 167 (*i2c_transfer(client->adapter, msg, 2)*). Isso é necessário pois a operação de leitura de uma memória requer que primeiro se escreva o endereço que irá ser lido, para então começarmos a ler. As outras funções são as mesmas de at24_eeprom_write().

7. module_param(): linhas 60 e 68

Essa macro é usada para que o usuário possa passar parâmetros para o módulo quando este é carregado no sistema. Para mais informações veja aqui. Associado a essa macro, temos MODULE_PARM_DESC(), que fornece uma descrição do parâmetro. Os parametros são armazenados em variavéis globais.

Código do Makefile do device driver

O código do Makefile é praticamente o mesmo do que foi usado neste artigo. Porém esse driver foi testado na placa Raspberry Pi 2, e por este motivo, aponta para o código fonte do kernel da Broadcom. Mas como dito antes, pode ser testado em qualquer placa.

Para compilar o driver digite:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

Agora estamos prontos para testar driver!

Exemplo de uso do driver I2C

A fim de que o driver funcione, é necessário adicionar informações para que o driver "saiba" que memória será usada. Isso pode ser feito de duas formas: através da estrutura *i2c_board_info* e a função i2c_register_board_info() para registrar as informações necessárias; ou através de um simples código de Device Tree.

Para a primeira abordagem, veja este exemplo.

Vejamos como proceder para o caso do Device Tree. Se você está usando a Raspberry 2, abra o arquivo bcm2709-rpi-2-b.dts, e dentro de bloco da *i2c1*, coloque:

```
at24@51 {
        compatible = "at24,24c512";
        pagesize = <512>;
        reg = <0x50>;
    };
```

Assim, o código completo do bloco i2c1 deve ficar assim:

```
&i2c1 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins>;
    clock-frequency = <100000>;

    at24@51 {
        compatible = "at24,24c512";
        pagesize = <512>;
        reg = <0x50>;
    };
};
```

Este código é para a antiga memória at24c512 da Atmel. Mude conforme a memória que você for testar. Você pode usar esse códido para praticamente qualquer placa com suporte a Device Tree, como a Beaglebone Black e outras. Agora recompile o Device Tree usando:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs
```

Depois transfira o arquivo de Device Tree compilado para a placa. No caso da Raspberry 2, será bcm2709-rpi-2-b.dtb, que deve ser colocado na pasta */boot* do SD-Card.

Agora finalmente podemos testar a leitura e escrita da mémoria I2C. Digite os seguintes comandos, como *root*:

```
# cd /sys/devices/platform/soc/3f804000.i2c/i2c-1/1-0050
# echo "Teste de memoria I2C na Raspberry" > eeprom
# hexdump -C -n 100 eeprom
```

O resultado deveria ser algo como:

```
00000000  54 65 73 74 65 20 64 65  20 6d 65 6d 6f 72 69 61  |Teste de memoria|
00000010  20 49 32 43 20 6e 61 20  52 61 73 70 62 65 72 72  | I2C na Raspberr|
00000020  79 0a ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |y...............|
00000030  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

Exemplo de software

Agora vejamos um exemplo de software para testar a leitura e escrita da memória através do driver I2C.

Código:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>


#define DEV_I2C        "/sys/devices/platform/soc/3f804000.i2c/i2c-1/1-0050/eeprom"

#define LEN        9


int main()
{
    int fd, i;
    char dado_wr[10];
    char dado_rd[10];

    fd = open (DEV_I2C, O_RDWR);    // abri o driver para operacoes de w/r
    if (fd == -1)
    {
        printf("Erro ao abrir eeprom0\n");
        return fd;
        /* error */
    }
    else
        printf("Driver aberto!\n");

    for (i=0; i < LEN; i++)
        dado_wr[i] = '1' + i;        // preenche memoria com valores de 1 a LEN
    write(fd, dado_wr, LEN);        // escreve os dados na memoria
    usleep(1000*10);
    pread(fd, dado_rd, LEN, 0);     // ler o dado escrito anteriormente
    for (i=0; i < LEN; i++)
        printf("%c", dado_rd[i]);    // imprime dados lidos da memoria
    putchar('\n');

    close(fd);                // fecha o driver
    printf("Driver fechado!\n");
```

```
    return 0;
}
```

Salve o código fonte como *i2c_teste.c*. Para fazer a compilação na própria Raspberry, digite:

```
$ gcc i2c_teste.c -o i2c_teste
```

Para compilar no PC, fazendo uma compilação cruzada, digite:

```
$ arm-linux-gnueabihf-gcc i2c_teste.c -o i2c_teste
```

Observe que as funções de chamada de sistema *open()*, *read()*, *write()* e *close()*, mostradas na figura acima (ações do usuário), são usadas no software de exemplo. Mas elas também são usadas nos comandos *echo* e *hexdump*, basta verificar no código fonte de cada um.

E assim concluímos mais um artigo sobre device drivers para linux embarcado. Para o próximo artigo, veremos um driver SPI.

Referências

- [1] Linux Device Drivers – Diferenças entre Drivers de Plataforma e de Dispositivo
- [2] Driver at24.c
- [3] Exemplo de driver para Linux Embarcado
- [4] Pré-processador C – Parte 1
- [5] Introdução ao uso de Device Tree e Device Tree Overlay em Sistemas Linux Embarcado
- [6] The Linux Kernel Module Programming Guide
- [7] Raspberry Pi I2C 256K EEPROM Tutorial

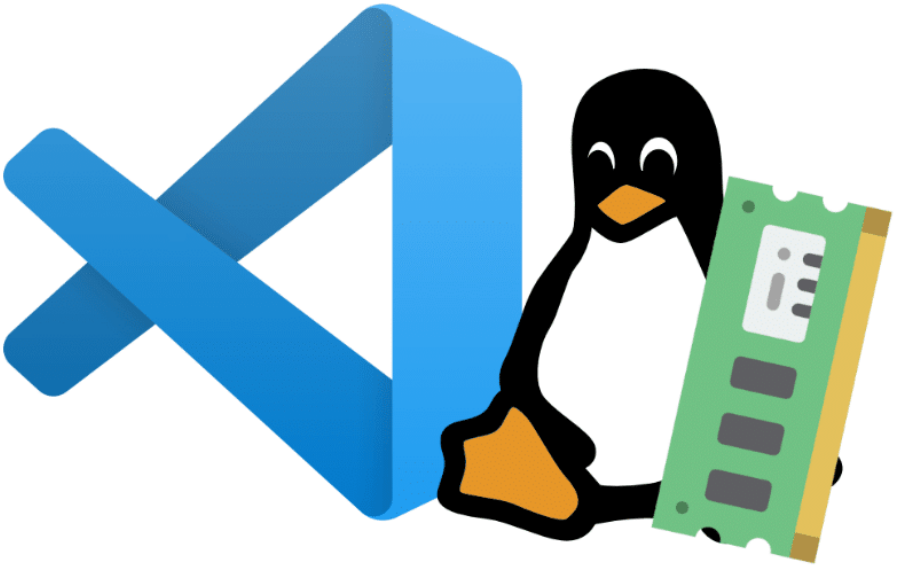
Fonte da imagem destacada: https://openelectronicsproject.blogspot.com.br/2015/08/i2c-protocol.html

Autor: Vinicius Maciel

Cursando Tecnologia em Telemática no IFCE. Trabalho com programação de Sistemas Embarcados desde 2009. Tenho experiência em desenvolvimento de software para Linux embarcado em C/C++. Criação ou modificação de devices drivers para o sistema operacional Linux. Uso de ferramentas open source para desenvolvimento e debug incluindo gcc, gdb, eclipse.

Publicado originalmente em:
https://embarcados.com.br/device-driver-i2c-para-linux-embarcado/

Extensão do Visual Studio Code para Kernel Linux Embarcado



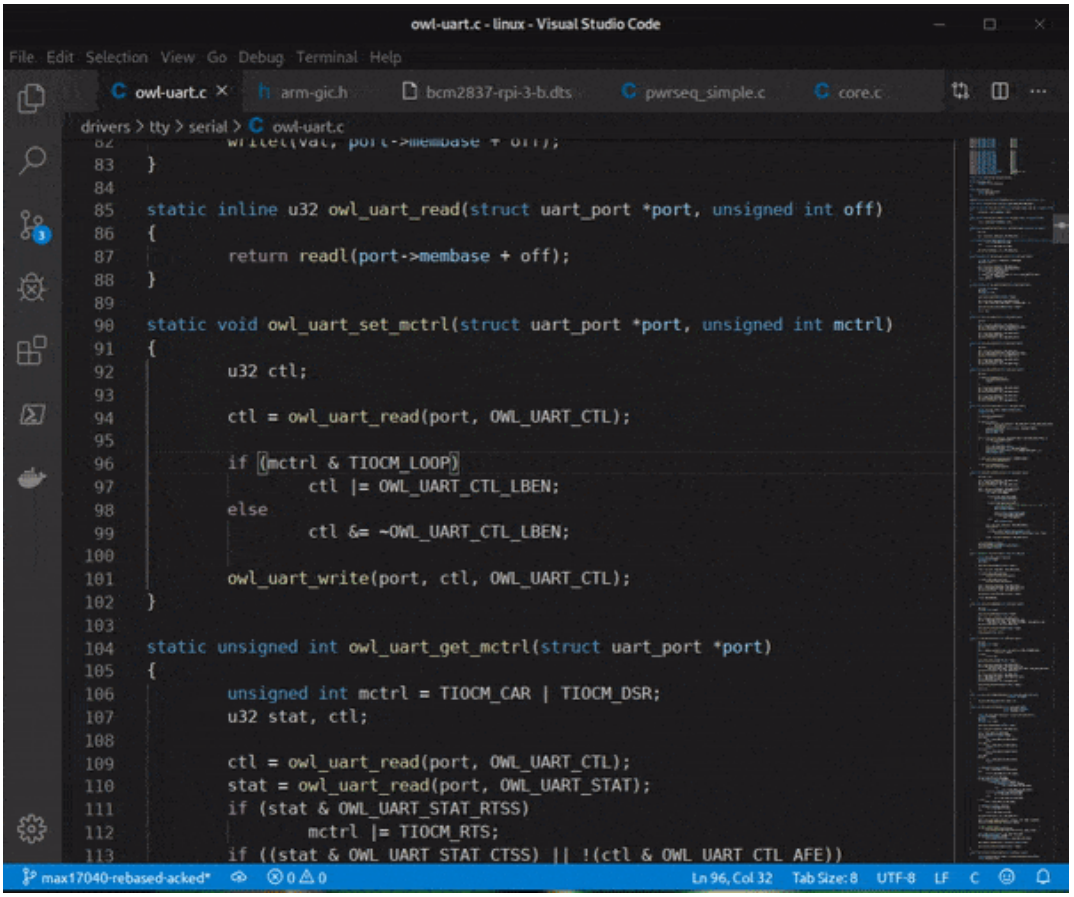
Se você está começando a trabalhar com, ou a estudar, desenvolver para sistemas embarcados microprocessados, talvez, você esbarre na tarefa de ter que desenvolver um módulo, ou editar algum device tree, para o Kernel Linux.

Então que ferramenta de edição de código ou IDE utilizar para ganhar agilidade nessas tarefas? O editor de código mais popular entre os desenvolvedores do Kernel Linux é o VIM, e seus muitos plugins. Não é minha intenção aqui entrar no mérito de qual a melhor IDE, editor de texto, nem blasfemar contra a religião das pessoas. É, o VIM tem até igreja para seus seguidores 😅. Brincadeiras a parte, cada um tem a sua preferência, e se adapta melhor a uma certa ferramenta.

Eu pessoalmente me adapto melhor ao VS Code. Usei o Atom por bastante tempo, mas o VS Code me ganhou, por ser similar ao Atom e pelas extensões, temas, APIs para criar extensões, tarefas e etc. Porém, como nem tudo são flores, para projetos com base de código grandes, caso do Kernel Linux, a extensão padrão da Microsoft para C/C++ é bem pesada, demora para indexar, devora sua memória RAM e deixa tudo travado 😢. Além de sentir falta de alguns plugins para agilizar tarefas em arquivos de device-tree. A solução foi criar uma extensão, pensando no desenvolvimento de Kernel Linux embarcado para o VS Code.

Extensão – Embedded Linux Kernel Dev

A extensão pode ser baixada no Visual Studio Marketplace. Ou no menu "View > Extensions" do próprio VS Code:



Estou compartilhando a extensão com licença MIT, open source. Fiquem livres para contribuir ou usar como quiserem.

Dependências

A extensão, por hora, funciona apenas em sistemas Linux e usa alguns pacotes para seu funcionamento correto. Antes de usar é necessário instalar as seguintes dependências em seu sistema:

- bash
- universal-ctags

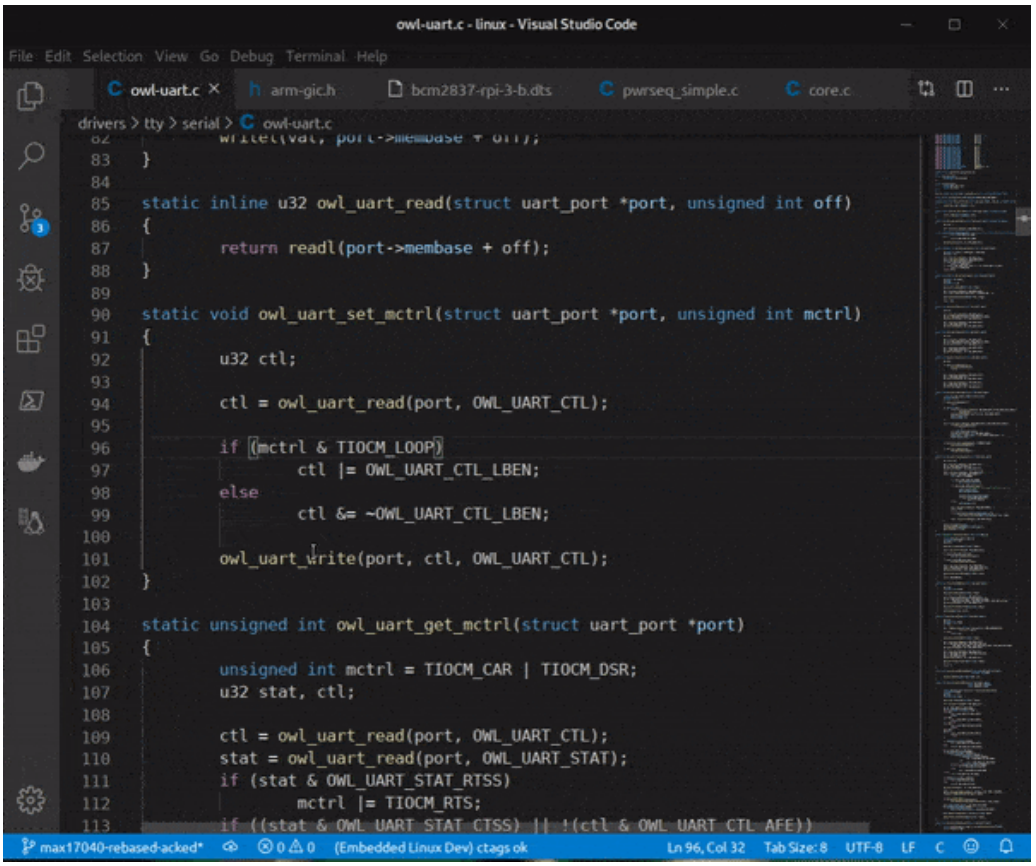
Um detalhe importante é que seja instalado o universal-ctags e não o exuberant-ctags. Pois o exuberant não indexa arquivos de device tree.

Para uma experiência completa de desenvolvimento para o kernel Linux, durante a instalação da extensão, as seguintes extensões serão requisitadas a serem instaladas em conjunto:

- DeviceTree (implementa syntax highlighting para arquivos de device-tree .dts e .dtsi)
- kconfig (implementa syntax highlighting para arquivos Kconfig)

Funcionalidades

Todas as funcionalidades da extensão podem ser executadas, por comandos de click, pela barra de atividades:

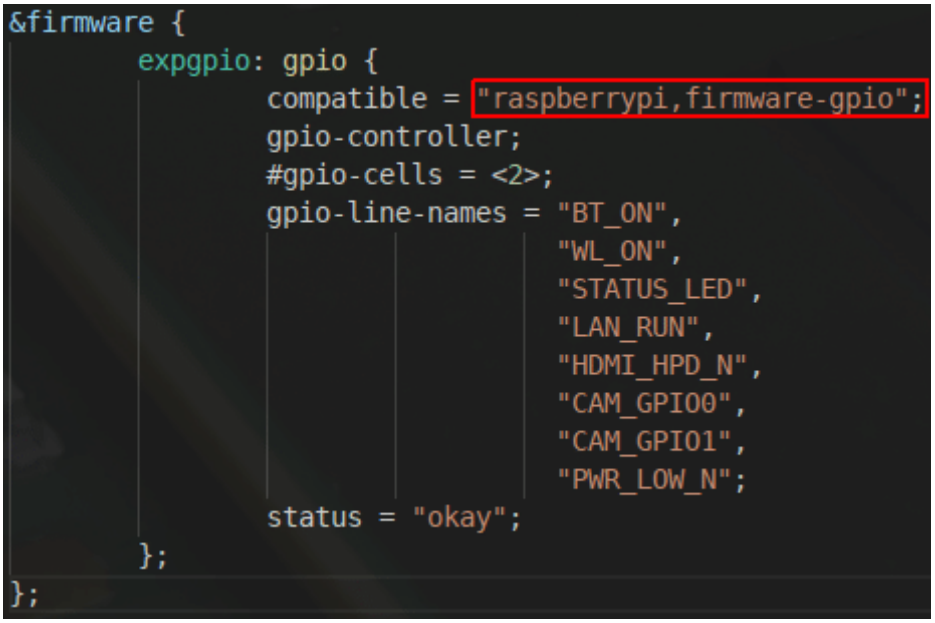


Nos próximos tópicos, irei descrever cada uma das funcionalidades da extensão, bem como descrever o problema que resolve.

Device Driver From Compatible

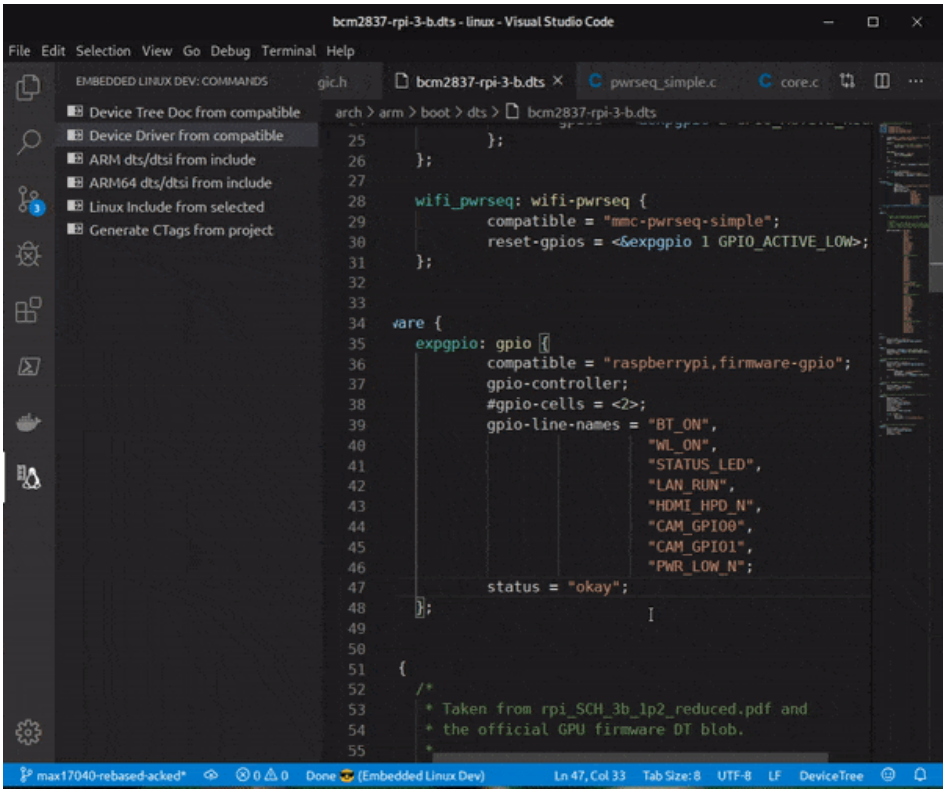
Um das tarefas repetitivas, no meu dia dia, era a procura pelo match em arquivos ".c", código de driver, de uma certa string "compatible". Vamos usar o

arquivo bcm2837-rpi-3-b.dts como exemplo. Esse é o device tree source que descreve o hardware do Raspberry Pi 3B para o kernel Linux. Note o trecho retirado do arquivo:

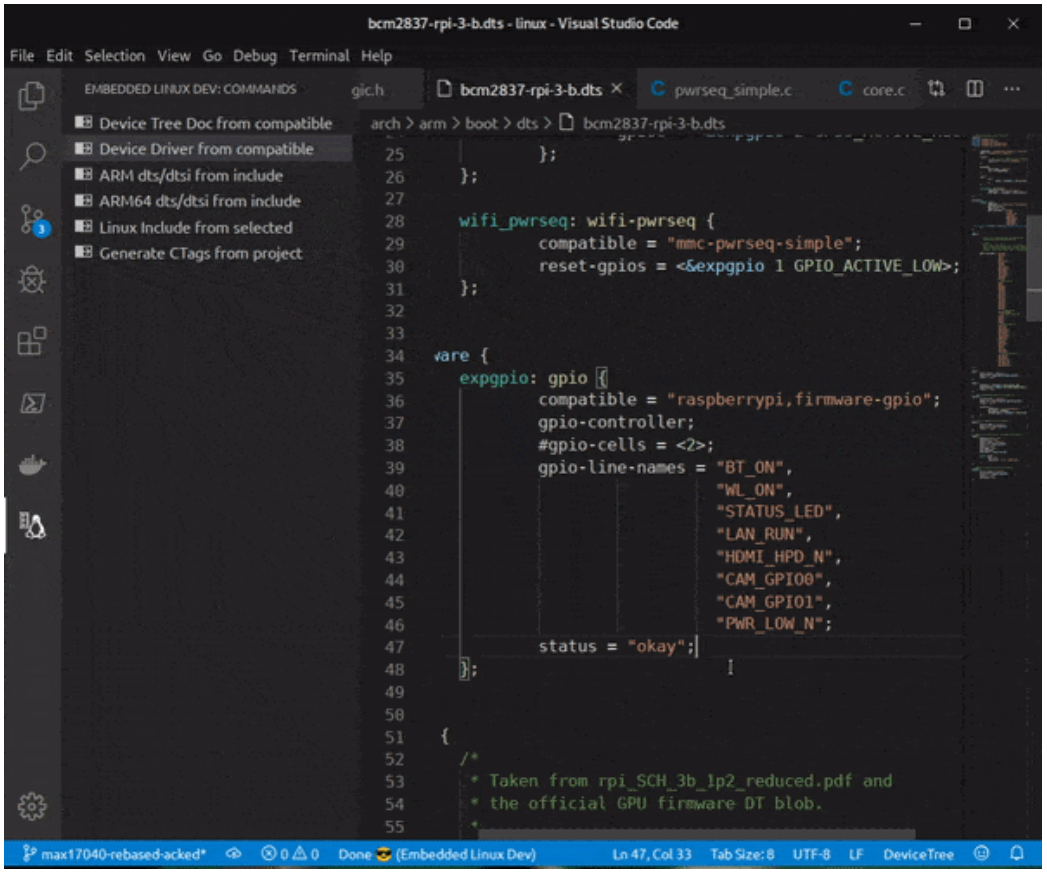


O "compatible", em destaque na imagem, é a string que faz o "bind" entre a descrição do hardware, nó do device tree, e o driver do kernel que irá usar essa descrição. Para saber qual é o arquivo ".c" do driver que será utilizado, eu fazia uma busca, um grep, entre os arquivos ".c", pela string do compatible. Pegava as respostas listadas, pelo grep, e abria o arquivo no editor.

Agora com a extensão, dentro de um arquivo de device-tree, ".dts" ou ".dtsi", posso clicar com o mouse sobre uma string de "compatible" e selecionar o comando "Device Driver from compatible". O VS Code irá realizar o match, busca, automaticamente e vai abrir o arquivo de código, ".c", do driver que implementa o "compatible":



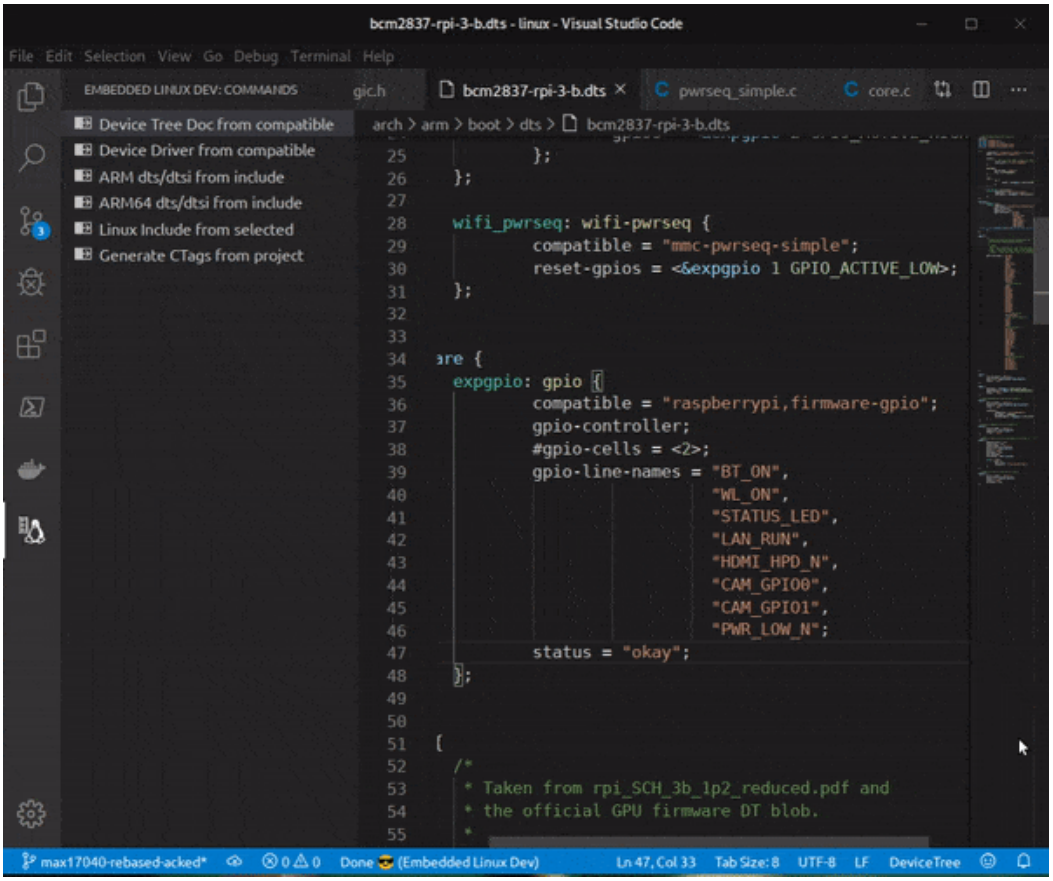
Essa funcionalidade também pode ser selecionada no menu de contexto, com click do botão direito:



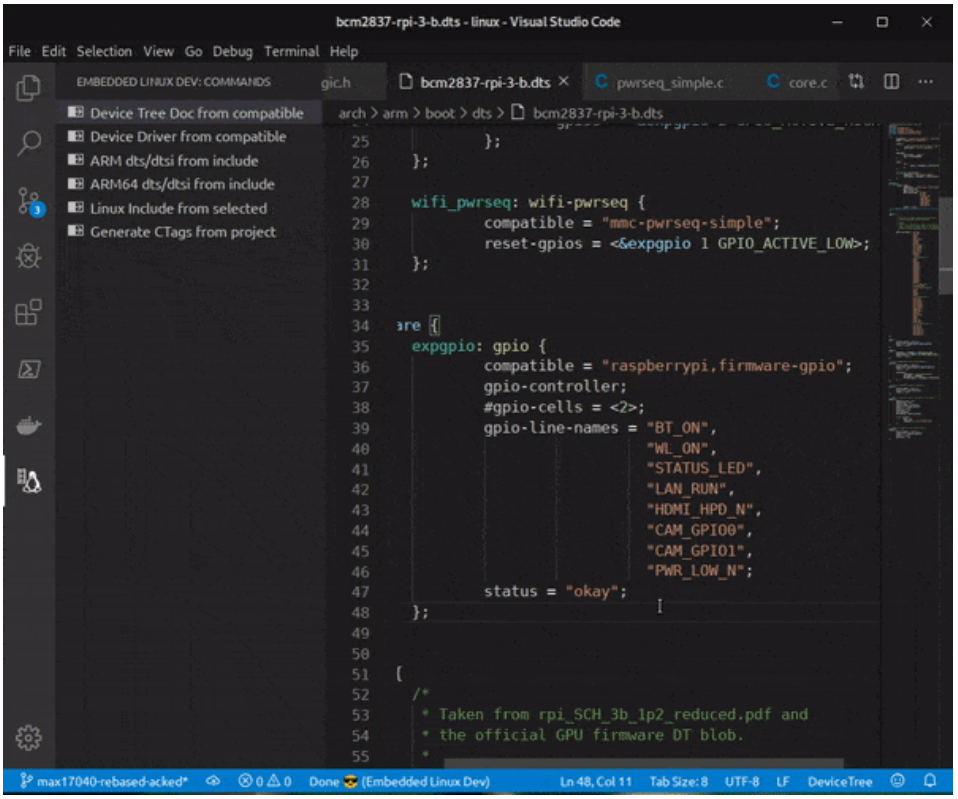
Device Tree Doc From Compatible

Aqui o mesmo problema de procura por string de “compatible” acontecia, só que para arquivos de documentação. Todos os device tree bindings tem uma documentação em “Documentation/devicetree/bindings”.

Agora com a extensão dentro de um arquivo de device-tree, “.dts” , “.dtsi”, ou código “.c”, posso clicar com o mouse sobre uma string de “compatible” e selecionar o comando “Device Tree Doc From Compatible”. O VS Code irá abrir o arquivo de documentação respectivo ao “compatible”:



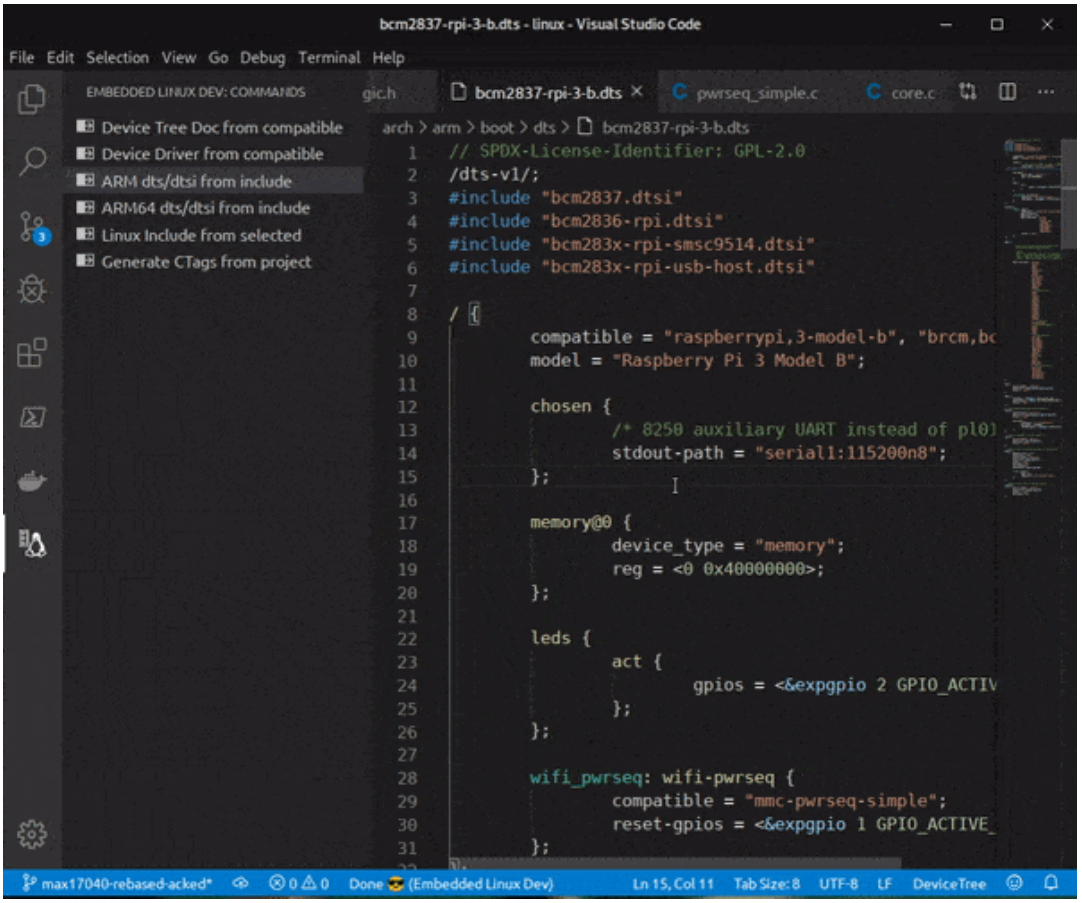
Essa funcionalidade também pode ser selecionada no menu de contexto, com click do botão direito:



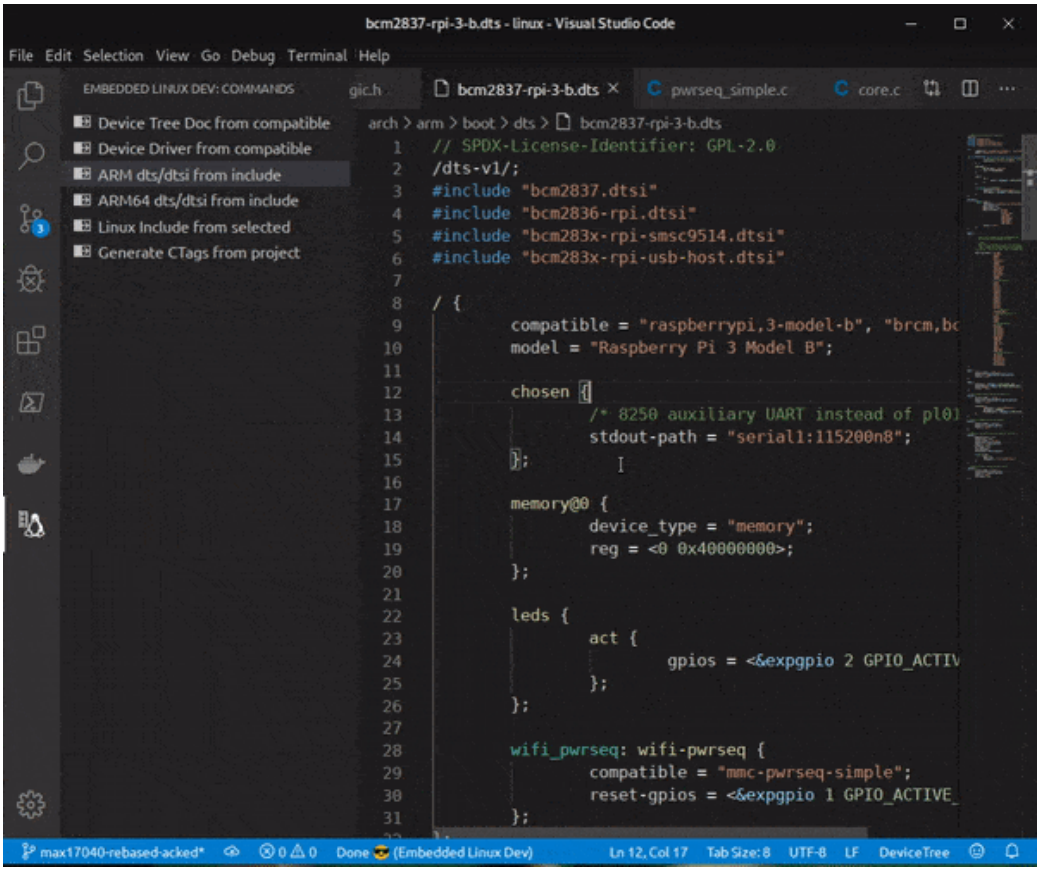
ARM/ARM64 dts/dtsi From Include

Geralmente arquivos ".dts" tem seus includes, ".dtsi". A repetição que eu tinha aqui era de ter que ir até o terminal, digitar o caminho e nome do arquivo ".dtsi" para abrir o include.

Agora com a extensão dentro de um arquivo de device-tree, ".dts" ou ".dtsi", apenas clico com o mouse sobre o string, de um device-tree include, e selecione o comando "ARM dts/dtsi From Include". O VS Code irá abrir o arquivo correspondente automaticamente:



Essa funcionalidade também pode ser selecionada no menu de contexto, com click do botão direito:

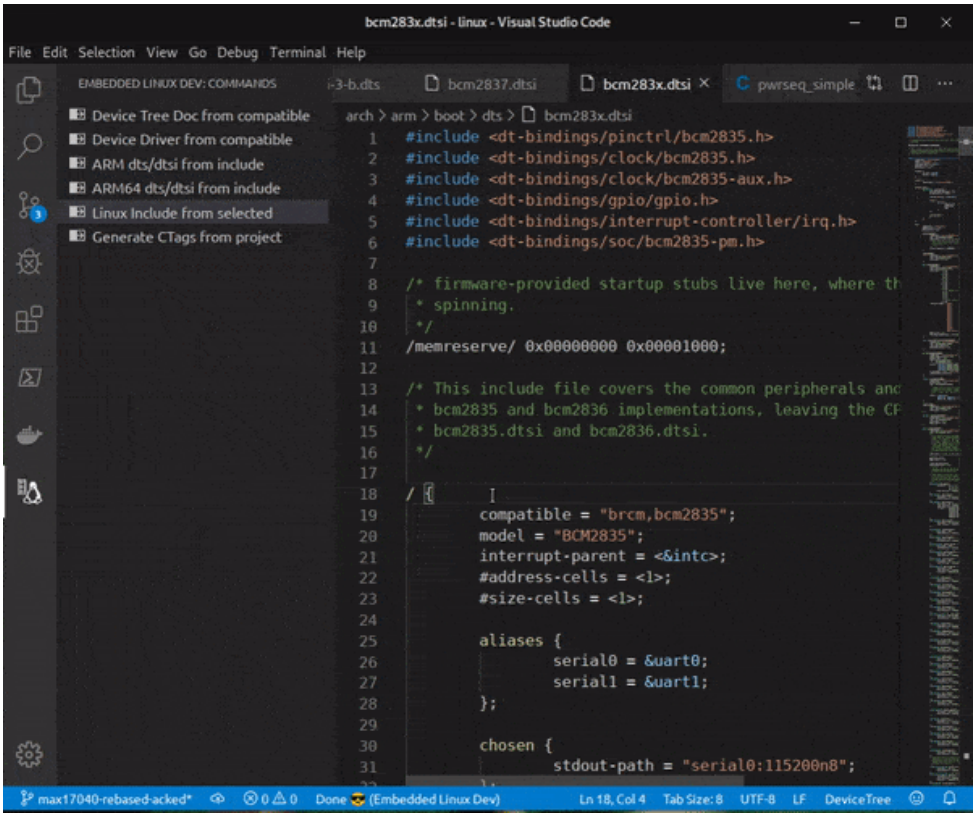


Há duas opções deste comando, uma para ARM e outra para ARM64, pois os devices-tree para cada uma dessas ficam em caminhos diferentes.

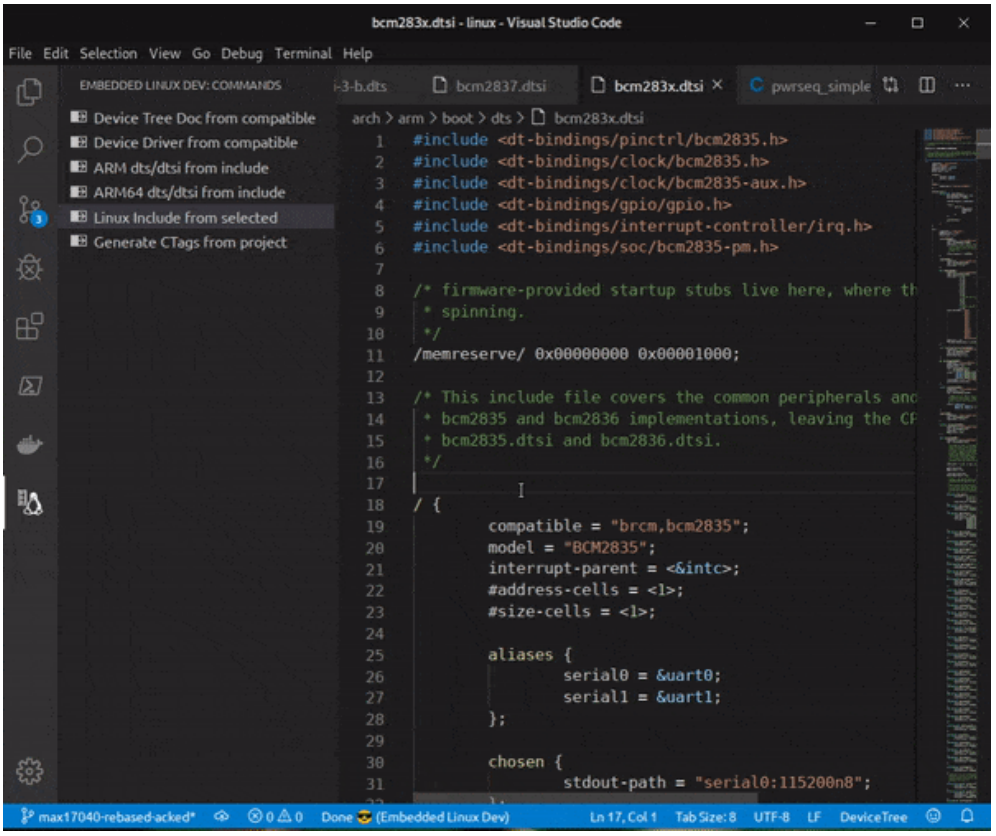
Linux Include From Selected

Essa é uma funcionalidade da qual sentia falta, que funcionava na extensão padrão do C/C++, para abrir includes ".h".

Dentro de arquivos ".c", ".dts" ou ".dtsi", clique com o mouse sobre um string de um include e selecione o comando "Linux Include From Selected". O VS Code irá abrir o include correspondente:



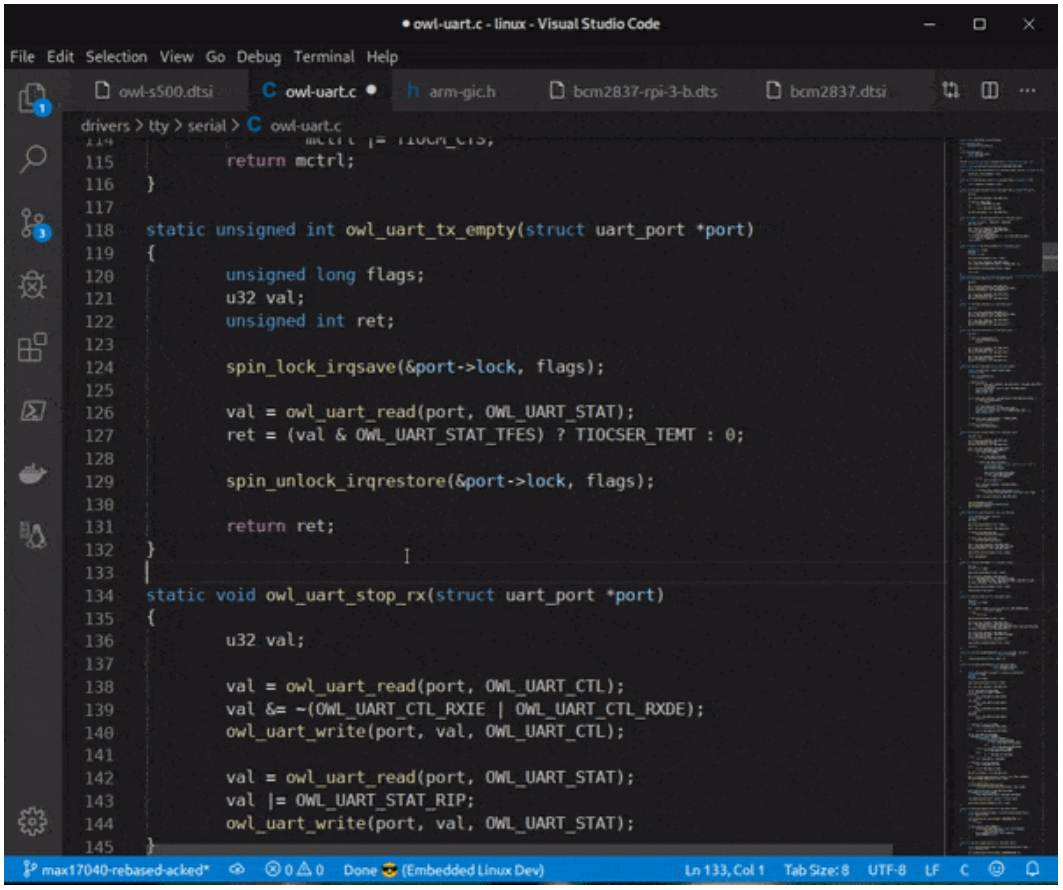
Essa funcionalidade também pode ser selecionada no menu de contexto, com click do botão direito:



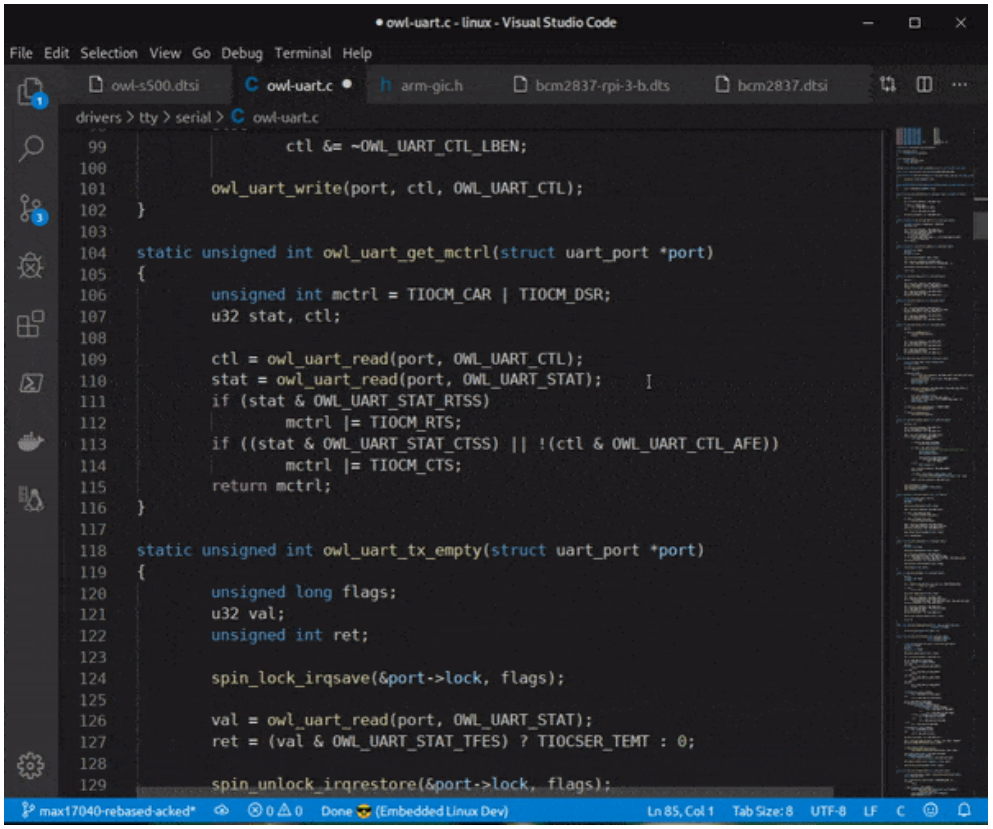
Generate CTags

Por último, mas, nem de longe, a menos importante. Essa funcionalidade gera um arquivo ".vscode-ctags" na raiz da pasta que foi aberta. Esse arquivo é o índice de tags gerado pelo universal-ctags. Ele é necessário para gerar as ajudas de navegação no código do projeto:

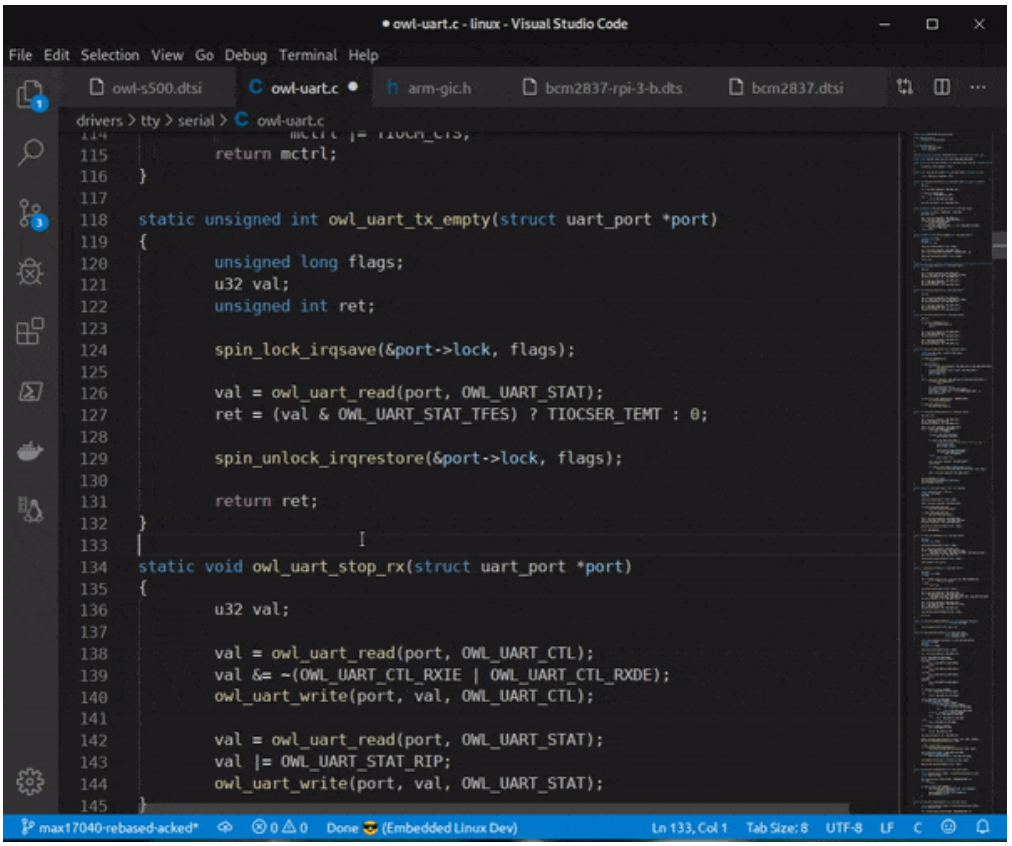
- Pular para a definição:



- Dicas para completar código:



- Mouse hover tags:



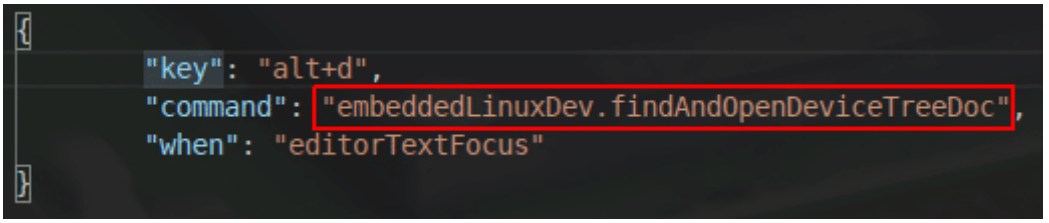
O Ctags resolve o problema das dicas de código e navegação da extensão C/C++ do VS Code. As funcionalidades são mais básicas, não possui um code linter por exemplo, mas em contrapartida é bem mais leve e fluído.

Atalhos de Teclado

Outro funcionalidade interessante é que a extensão também exporta todos os comandos, para serem usados em atalhos de teclado. Utilize os seguintes nomes para assinar os comandos aos atalhos de sua preferência:

- embeddedLinuxDev.findAndOpenDeviceTreeDoc
  - Device Tree Doc From Compatible
- embeddedLinuxDev.findAndOpenDeviceTreeMatchDriver
  - Device Driver From Compatible
- embeddedLinuxDev.openArmDtsDtsi
  - ARM dts/dtsi From Include
- embeddedLinuxDev.openArm64DtsDtsi
  - ARM64 dts/dtsi From Include
- embeddedLinuxDev.findAndOpenDeviceTreeMatchDriver
  - Generate CTags

Exemplo, definindo o atalho “alt+d” para executar o comando “Device Tree Doc From Compatible”:



Confira também o vídeo aonde mostro na prática a utilização da extensão:

Conclusões

Essas foram as funcionalidades implementadas, que resolvem e agilizam meu workflow, para desenvolvimento de kernel Linux embarcado, utilizando o VS Code.

Eu considero o VS Code bem amigável e com uma estrutura bem intuitiva. A utilização de Linux para dispositivos e soluções embarcados é crescente, e eu vejo o VS Code como a ferramenta, para edição de código, de escolha para

desenvolvedores que estão migrando para o mundo do Linux embarcado. Digo minha experiência pessoal, ainda lá na época que migrei do Windows e Visual Studio, para o Linux e Atom. Achei o VIM e as outras opções meio “esquisitas” ao que estava acostumado.

A extensão está em uma versão Alfa. Há coisas que poderiam ser melhoradas ou incluídas. Fiquem a vontade para contribuir. O bacana do open source é isso. Você não fica preso a uma ferramenta, pode estender ela, reutilizar funcionalidades, escrever a sua própria 😎.

Espero que tenham gostado da extensão, se for útil pra você deixe eu saber. Comente aqui nos comentários. Abraços!

Autor:Matheus Castello

Cientista da Computação atuando desde 2013 nas áreas de sistemas embarcados, Android e Linux. Trabalhou diretamente com o Kernel Linux embarcado em modelos de smartphones e tablets Samsung comercializados na America Latina. Colaborou no upstream para o Kernel Linux, árvore git do Linus Torvalds, para o controlador de GPIO da família de processadores utilizados nos Raspberry Pi. Palestrante FISL 2018 e Linux Developer Conference Brazil 2018.

Publicado originalmente em:
https://embarcados.com.br/extensao-do-vs-code-para-kernel-linux-embarcado/

Systemd – Adicionando scripts na inicialização do Linux

Em diversos projetos e produtos que criamos, com Linux embarcado ou na utilização de servidores Linux para back-end, front-end, banco de dados e etc, se faz necessário ter um gerenciamento completo de certos scripts e programas durante todo o ciclo de vida dele (desde o boot até falhas repentinas). Um caso bem comum, por exemplo, são scripts Python para tratamento de dados recebidos via MQTT.

Seja qual for o motivo, muitos desses scripts devem ser inicializados junto do sistema operacional e permanecerem em execução 100% do tempo, pois se não estiverem em execução, alguma etapa do seu projeto irá parar, podendo gerar perdas incalculáveis. Um grande desafio também é manter esse script monitorado, a fim de reiniciá-lo automaticamente caso pare, executar ações antes e/ou depois de iniciá-los e etc.

O mais comum é adicionar a execução do nosso script/serviço dentro do arquivo "rc.local", que é inicializado junto do sistema, mas isso tem diversos problemas. Por exemplo: Se nosso script/serviço vier a parar (encerrado pelo usuário, travamento, falta de memória e etc), ele não será reiniciado automaticamente, podendo gerar muitos problemas.

Para solucionar, vamos aprender um pouco sobre o Systemd, um sistema que gerencia quase toda a inicialização do sistema, padrão em diversas distribuições Linux, que também consegue gerenciar todo ciclo de vida de um serviço (nosso script), sendo facilmente configurado com incontáveis parâmetros de configuração.

O que é Systemd?

Systemd é um sistema que gerencia a inicialização de boa parte do sistema operacional e de serviços, estando presente nas mais diversas distribuições, inclusive nos Raspberry Pi's, sendo que seu aprendizado é muito importante para quem pretende se aprofundar um pouco mais em Linux.

Um processo resumido de boot pode ser composto por:

1. Bootloader.
2. Kernel.
3. Systemd.

Unidades (Units): Unidades são arquivos de texto que contém instruções de inicialização de um recurso, como por exemplo: Serviço, timer, path, mount, automount, socket e etc.

Unidade alvo (Target): Uma unidade alvo é um agrupamento de outras unidades que levam o sistema a um ponto específico no processo de inicialização, indicando que certos serviços e etc (dependências) já foram executados.

Entendendo sobre a árvore de boot

O systemd é focado em alta paralelização e dependências, contendo diversos targets que indicam em qual ponto da inicialização já foi atingido (ou pode ser atingido) e quais as próximas unidades que devem ser executadas para se chegar no target desejado. Sendo comumente comparados aos "runlevels do System V".

Podemos utilizar esses pontos ao nosso favor para inicializar unidades o mais breve possível, já que estaremos cientes de que certas dependências já foram atendidas. A seguir uma pequena lista de alguns targets importantes do systemd. Uma breve ordem em que cada um ocorre pode ser melhor visto na figura 1 abaixo.

- sysinit.target: Ponto em que apenas os serviços mais básicos necessários para o restante da inicialização foram executados. (single-user, non-graphical).
- basic.target: Ponto em que todos file systems, devices, swaps, sockets, timers e etc já foram montados e inicializados. Há todo o necessário para executar daemons e scripts de propósito geral. (single-user, non-graphical).
- multi-user.target: Ponto em que o sistema já permite multi-usuários (non-graphical) no sistema.
- graphical.target: Ponto em que o sistema já inicializou a interface gráfica do sistema.
- default.target: Ponto padrão de inicialização do systemd, sendo apenas um apelido (alias, symlink) para o multi-user.target ou graphical.target. Ao fim de todo processo de inicialização, o sistema chega ao ponto "default.target" e está pronto para uso.
- network.target: Ponto em que o mínimo necessário para o funcionamento da parte de redes do computador já foi atingido. Atenção, isso não indica que as interfaces já obtiveram IP e etc. Veja com mais detalhes nas referências.
- rescue.target: Inicia o sistema com o terminal no modo "rescue", apenas com o file system e serviços básicos necessários para alguma manutenção.
- emergency.target: Inicia o sistema com o terminal no modo "emergency", sendo o modo mais simples possível, iniciando o menor número de serviços e normalmente deixando o file system apenas em "read-only". Mais utilizado para encontrar problemas ao inicializar o sistema operacional.
- reboot.target: Leva o sistema para o estado de desligamento e o reinicia.
- poweroff.target: Leva o sistema para o estado de desligado.

Como foi dito, é um sistema de dependências, logo, conseguimos fazer algumas coisas muito interessantes, como por exemplo: Executar comandos/scripts na falha de outro serviço ou até antes do reinicio/desligamento do sistema. Isso facilita bastante a gerência de servidores, onde a parada de um serviço (banco de dados, broker e etc) deve ser notificada e assim por diante.

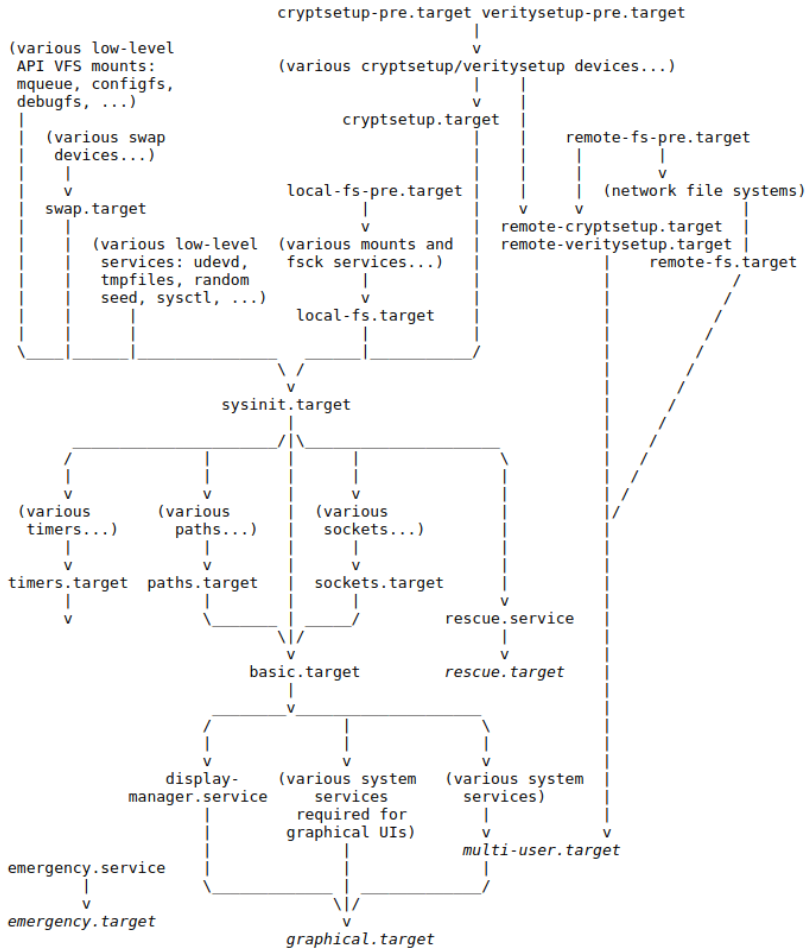


Figura 1: Processo de inicialização do systemd resumido em uma árvore.

Comandos relevantes para gerenciamento do systemd

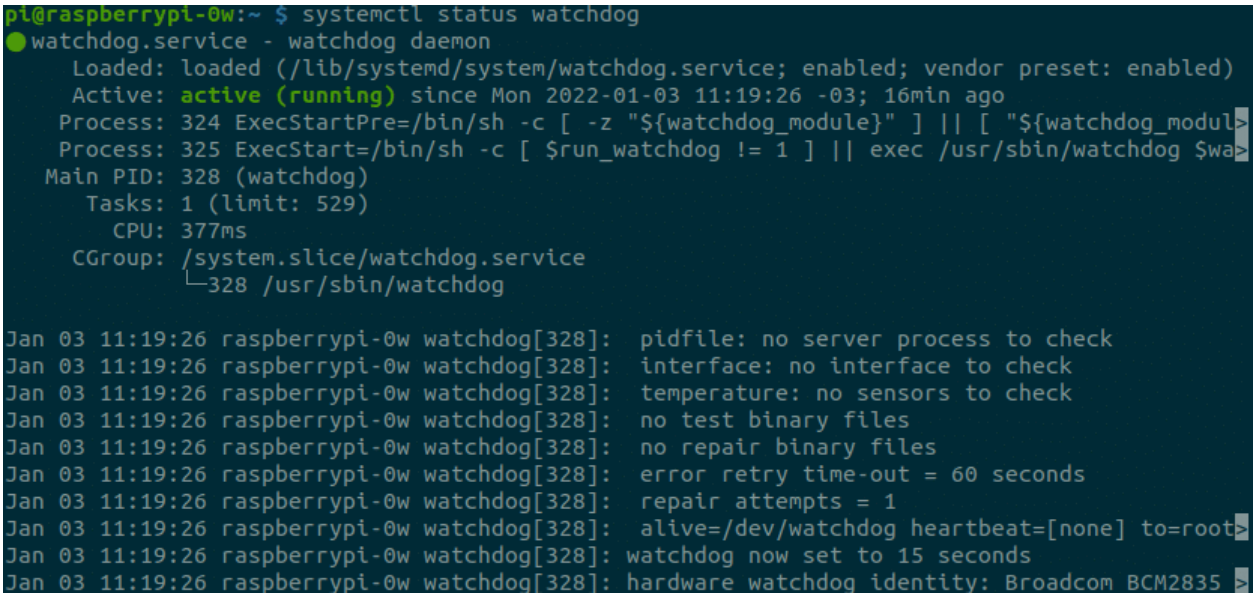


Figura 2: Status de um serviço de Watchdog no Raspberry Pi Zero W.

- systemctl status: Mostra detalhes sobre o serviço, como log, se está em execução e etc.
- systemctl start: Inicia o serviço.
- systemctl stop: Encerra o serviço.
- systemctl enable: Adiciona o serviço na inicialização do Linux.
- systemctl disable: Remove o serviço da inicialização do Linux.

Criando um novo serviço para nosso script

Agora que já temos uma noção básica do que é o systemd e seu funcionamento, vamos criar uma nova unidade de serviço para executar nosso script Python automaticamente durante a inicialização do Linux e também garantir que ele fique sempre em execução, sendo reiniciado automaticamente caso venha a ser encerrado.

O script em questão é um supervisório que criei para monitorar estatísticas de CPU, RAM, temperatura e etc de dispositivos Linux. Um simples script Python que coleta informações e salva em um banco de dados localmente na memória RAM (SQLite3) de tempo em tempo. Lembre-se que é apenas um exemplo demonstrativo.

Obtendo o script do github

```
git clone https://github.com/urbanze/supix.git
```

Criando o arquivo de serviço

Atenção, os diretórios utilizados se referem ao meu computador, que podem ser diferentes do seu. Altere se necessário, principalmente a pasta do script.

```
nano /etc/systemd/system/supix.service

[Unit]
Description=Supervisor to LOG your system
stats inside RAM.

[Service]
Type=simple
User=root
ExecStart=/usr/bin/python3 -u
/home/ze/supix/cpu.py
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Após a criação do arquivo de serviço, basta iniciá-lo com o comando “systemctl start supix”. Para ver se a execução do serviço foi bem sucedida, veja com comando de status “systemctl status supix”. Podemos notar que o serviço está ativo (running) e não retornou nenhum erro.

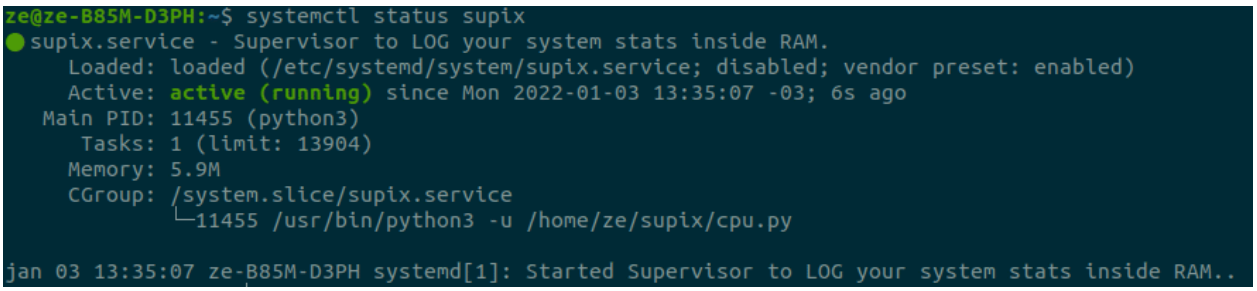


Figura 3: Novo serviço iniciado.

Agora, queremos que esse script (serviço) seja iniciado junto com o sistema caso o computador venha a ser reiniciado e etc. Execute “systemctl enable supix” e pronto! Nosso serviço será iniciado automaticamente quando o sistema iniciar.

No arquivo de serviço que criamos anteriormente, há algumas das inúmeras configurações disponíveis para uma unidade do systemd. Adicionamos, por exemplo, “Restart=always” e “RestartSec=10”. Isso indica ao Systemd que ele deve reiniciar o serviço após 10 segundos, caso venha a ser encerrado por falhas, OOM Killer ou mesmo o usuário. Para testar isso, encerre o processo

manualmente pelo comando “kill” ou até “htop”, você notará que após 10 segundos, o serviço estará ativo novamente. Isso é maravilhoso!

Algumas opções bem interessantes são sobre ordem e dependência de execução. Às vezes precisamos que um serviço só execute após algum outro serviço ou até que só possa ser executado enquanto outro serviço esteja ativo. Para isso, temos as opções “After=” e “Requires=”, respectivamente. Isso é útil, por exemplo, para iniciar um script que monitora dados MQTT do broker Mosquitto, porém, ele só deve ser iniciado após o broker. Logo, adicionamos “After=mosquitto.service”.

Outros exemplos de opções interessantes são para execução de algo antes (ExecStartPre=) ou depois da execução do serviço (ExecStartPost=) , ou até mesmo quando ocorrer uma falha e o serviço encerrar (ExecStopPost=) , como por exemplo, enviar um email! Podemos inclusive reiniciar o sistema inteiro se assim desejar. Se você tiver interesse e precisa de alguma lógica mais profunda para sua unidade, não se esqueça de consultar as referências e documentos do Systemd.

Ainda podemos especificar qual o tipo (type=) de serviço e como ele se comporta diante do sistema:

- simple: O serviço é iniciado como processo principal, não sendo necessário seu encerramento para o próximo serviço executar.
- oneshot: O serviço é iniciado e bloqueia o restante da inicialização até seu próprio encerramento.
- idle: O serviço é iniciado apenas quando todos os outros já foram inicializados.

Referências

https://www.freedesktop.org/software/systemd/man/systemd.unit.html

https://www.freedesktop.org/software/systemd/man/bootup.html

https://www.freedesktop.org/software/systemd/man/systemd.special.html

Autor: José Morais

Graduado em Engenharia de Computação, é um ávido entusiasta da computação aplicada em qualquer segmento do mercado. Trabalha como Engenheiro de Sistemas Embarcados e Diretor de sua própria empresa, TecnoEVO, que atua no ramo de IoT, entregando maneiras inteligentes para monitoramento, controle e automação de processos em equipamentos e ambientes. Também é um ativo contribuidor para

comunidade brasileira, contando com dezenas de bibliotecas, palestras e artigos publicados. Focando seus esforços para ambientes IoT, conta com vastos conhecimentos na área computacional, como por exemplo: Hardware, software, redes, segurança, RTOS, banco de dados, Linux, computação em nuvem e visão computacional.

Utilizando o udev para criar automações com porta USB no Linux

Já pensou o quão útil pode ser saber se um determinado periférico USB (mouse, teclado, câmera e afins) foi conectado ou desconectado de seu projeto Linux embarcado?

Vamos exemplificar pensando em um terminal de autoatendimento ou tótem, o qual possui periféricos como display touchscreen, máquina de cartões e etc, acredito ser interessante saber se houve alteração na conexão dos periféricos e quando isto ocorreu. Podendo criar scripts para salvar logs de conexões ou até mesmo enviar os dados para a internet.

Figura 1 – Totens e terminais de autoatendimentos – Fonte: www.seat.ind.br, 2021

Recentemente em um projeto pessoal, me deparei com a necessidade de saber quando um determinado dispositivo USB foi conectado ou desconectado de uma Raspberry pi. Portanto, gostaria de compartilhar nesse artigo a experiência que tive em atender a necessidade.

Por razões didáticas, simplificaremos a situação utilizando um dispositivo Linux embarcado, no caso uma Beaglebone Green e um pendrive para simular o periférico que desejamos monitorar a conexão e desconexão. O intuito dessa simplificação será apenas criar um arquivo txt informando a data do evento e o tipo de evento (conexão ou desconexão).

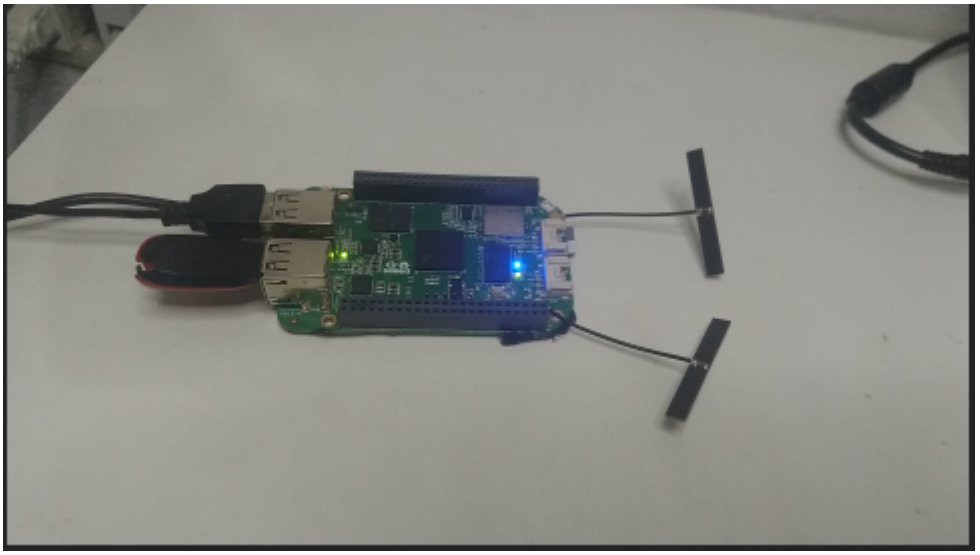


Figura 2 – Hardware utilizado – Fonte: Autoria Própria, 2022

Apenas evidenciando, o intuito desse artigo é apresentar que é possível aproveitar os eventos de conexão e desconexão dos periféricos USB para criar automações em aplicações Linux, apresentando um exemplo inicial de como criar as automações utilizando a ferramenta udev do Linux.

Requisitos

Este artigo é destinado a todas as pessoas que tenham contato ou interesse em trabalhar com Linux e criar automações. Entretanto, é recomendando principalmente para a execução da parte prática conhecimentos nos seguintes assuntos:

- Básico em Linux;
- Básico em manipulação do terminal do Linux;
- Básico em python.

Conhecendo o uDEV

O uDEV trata-se de um gerenciador de dispositivos para Linux, considerado como sucessor do devfsd e hotplug. Na prática, o udev é responsável por comunicar ao sistema operacional sobre os dispositivos físicos conectados ao hardware, baseado nas informações recebidas do kernel. Caso deseje obter mais informações acerca do udev, recomendo a leitura do seguinte artigo.

Assim como seus antecessores já citados, o udev também gerencia os dispositivos conectados, criando um subdiretório específico para cada dispositivo em /dev, por exemplo:

```
/dev/sda
```

Uma característica interessante do udev de mencionar é a possibilidade de criar scripts para os eventos recebidos via kernel, scripts os quais são chamados de *rules.* Na próxima secção abordamos em detalhes as *rules* do udev.

Regras uDEV

Como já mencionado, o udev é capaz de notificar o sistema operacional sobre as alterações nas conexões dos periféricos, além disso, ele permite o registro de manipuladores de eventos (em inglês, *Event Handlers*) personalizados, ou seja, é possível criar scripts para serem executados quando o udev detecta uma alteração específica no periférico. Exemplificando, é possível criar um script para ser executado sempre que um determinado dispositivo é desconectado.

Os scripts mencionados são chamados de regras (em inglês, *rules*). Existem dois níveis de regras, as regras de administrador salvas em /etc/udev/rules.d/ e as regras de usuários salvas em /usr/lib/udev/rules.d/. As regras de administrador precedem as regras de usuário na inicialização do udev.

Existe uma sintaxe própria para a criação de regras, baseadas em comandos e chaves, recomendo a leitura dessa especificação para maior detalhamento. Para este artigo, vamos nos ater aos seguintes comandos.

Comando	Descrição
+=	Adicionar valor ao final da lista de valores presentes na chave
==	Comparar igualdade entre valores
=	Definir valor para chave

Com relação às chaves, o quadro abaixo apresenta as chaves abordadas no exemplo prático.

Chave	Descrição
ACTION	Nome da ação do evento desejado (exemplos: “add”, “remove”)
SUBSYSTEM	Nome do subsistema onde deseja-se capturar o evento
ENV{*chave*}	Utilizado para acessar uma propriedade do dispositivo
RUN	Corresponde à ação que se deseja executar caso a regra seja ativada

Desenvolvimento prático

Certo, chegou o momento de “colocarmos a mão na massa”. Utilizei uma Beaglebone green para fazer o artigo, mas, na prática, esse procedimento é válido para qualquer dispositivo Linux que possua o udev. Foi utilizado um pen drive San Disk para fazermos os testes de conexão/desconexão.

Coletando informações do dispositivo USB

A primeira etapa é verificar como o udev identifica os eventos de conexão e desconexão do periférico de testes, para isso digite no terminal.

```
sudo udevadm monitor -p
```

Com a inserção do comando, você poderá verificar as informações dos eventos recebidos e os detalhes do dispositivo. Conecte o periférico USB e verifique a saída do terminal, deve ver algo parecido com a imagem abaixo.

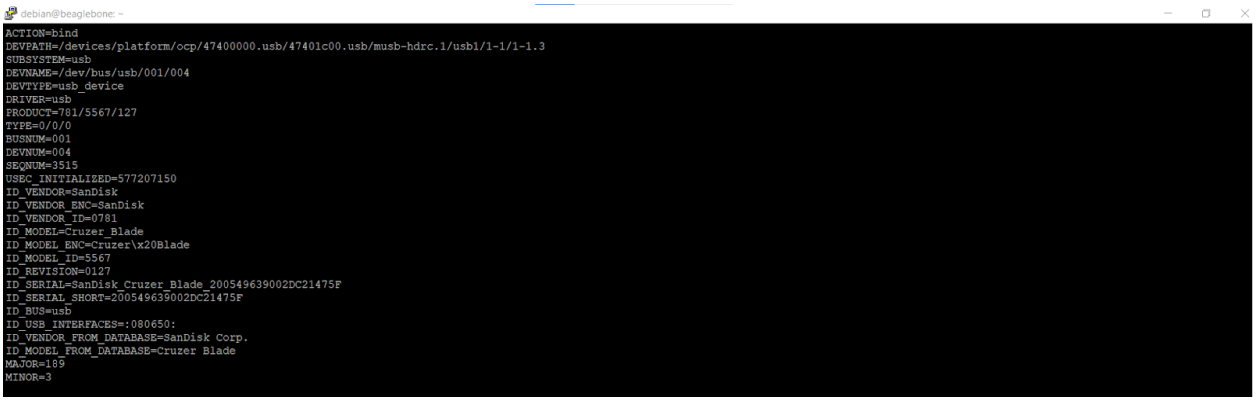


Figura 3 – Leitura do udevadm monitor – Fonte: Autoria própria, 2022

Da imagem acima utilizaremos as informações de ID_VENDOR_ID, ID_MODEL_ID, DEVTYPE e SUBSYSTEM, de preferência salve as informações mencionadas. Em seguida, desconecte o periférico USB. A imagem abaixo exemplifica como devem ser as mensagens recebidas na saída no terminal.

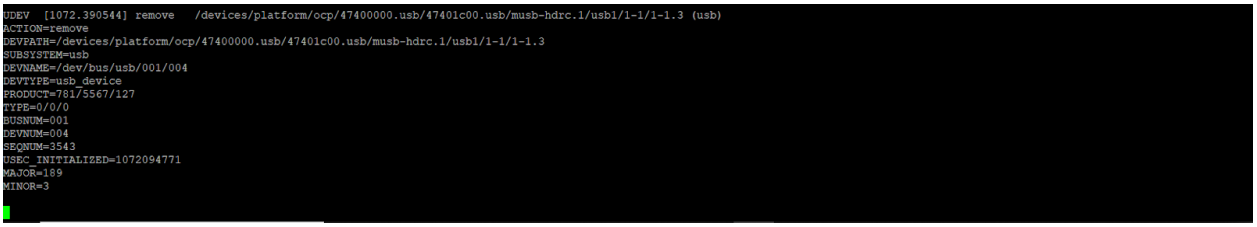


Figura 4 – Leitura do udevadm monitor após desconexão – Fonte: Autoria própria, 2022

Utilizaremos as informações de SUBSYSTEM, DEVTYPE e PRODUCT. Com todas as informações coletadas, é possível criar a *rule* no udev.

Criando os scripts

Como mencionado no início desse artigo, deseja-se criar um arquivo de log simples indicando a conexão e desconexão do pen drive. Com o intuito puramente didático, criaremos dois scripts python, um para salvar o evento de conexão e o outro para salvar o evento de desconexão.

O código do script de conexão (arquivo connected.py) é apresentado abaixo.

```
from datetime import datetime
import os

'''
    Funcao principal
'''
def main():
    #pega a data atual
    now = datetime.now()
    caminho = "/home/debian/usb-automation/"

    #salvar o log
    with open(caminho + "log.txt", "a") as file:
        file.write(f'{now.strftime("%d/%m/%Y %H:%M:%S")} -
Dispositivo Conectado\r\n')


#executa a funcao principal
main()
```

O código do script de desconexão (arquivo disconnected.py) é apresentado abaixo.

```
from datetime import datetime
import os

'''
    Funcao principal
'''
def main():
    #pega a data atual
    now = datetime.now()
    caminho = "/home/debian/usb-automation/"

    #salvar o log
    with open(caminho + "log.txt", "a") as file:
        file.write(f'{now.strftime("%d/%m/%Y %H:%M:%S")} -
Dispositivo Desconectado\r\n')


#executa a funcao principal
main()
```

Caso você não tenha conhecimento em python, recomendo a leitura desse artigo que indica alguns livros para o estudo da tecnologia.

Dando sequência ao procedimento, coloque ambos os scripts em uma pasta, no meu caso coloquei os arquivos em /home/debian/usb-automation. Caso você queira alterar o caminho onde os scripts serão salvos, altere a variável caminho presente nos scripts, com o caminho atual de onde estão localizados os scripts.

Criando as regras

Conhecendo as informações do dispositivo, podemos criar a regra udev para detectar a conexão e desconexão do pen drive. Usando o terminal, criaremos o arquivo de regra chamado "pen-drive.rules" por meio do editor de texto nano. Para isso, digite o seguinte comando.

```
sudo nano /etc/udev/rules.d/pen-drive.rules
```

Após inserção do comando, o nano abrirá um arquivo em branco. Agora basta digitar a seguinte regra.

```
ACTION=="add", SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ENV{ID_VENDOR_ID}=="0781",
ENV{ID_MODEL_ID}=="5567", RUN+="/usr/bin/python3 /home/debian/usb-automation/connected.py"
ACTION=="remove", SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device",
ENV{PRODUCT}=="781/5567/127", RUN+="/usr/bin/python3
/home/debian/usb-automation/disconnected.py"
```

Analisando a regra acima e destacando os principais pontos, começando pela primeira linha de comando, temos a chave ACTION com o valor "add" o que significa que deseja-se criar uma regra para o evento de conexão, temos a chave ENV{ID_MODEL_ID} com o valor "5567" o qual indica o device id do dispositivo que queremos monitorar.

Prosseguindo a análise, a chave ENV{ID_VENDOR_ID} está com o valor "0781" indicando o *vendor id* do pen drive, por fim, a chave RUN recebe o valor "/usr/bin/python3 /home/debian/usb-automation/connected.py" indicando que o script python de conexão deve ser executado quando o udev detectar a conexão do pen drive.

Ainda fazendo análise, a segunda linha da regra é similar a primeira linha, porém com algumas ressalvas, sendo elas:

- Chave ACTION recebe valor "remove", pois é desejado verificar a desconexão.
- Chave RUN recebe o valor que indica a execução do script python para desconexão.

Após a criação do arquivo de regra, devemos atualizar a tabela de regras e forçar o recarregamento das regras. Digite o seguinte comando no terminal.

```
sudo udevadm control --reload-rules ; sudo udevadm trigger
```

Após inserção dos comandos, a regra deve ser recarregada e já estará em execução.

Teste Prático

Certo, chegou o momento de validar a regra salva no udev e os scripts python. Para testar, conecte o periférico usb desejado e verifique no arquivo log.txt se o script registou o evento de conexão, faça o mesmo para o evento de desconexão.

Conclusão

O udev é um gerenciador de dispositivos para Linux, o qual proporciona a abstração dos periféricos conectados, facilitando o uso dos periféricos nos ambientes Linux. Outro ponto importante de ser mencionado é a capacidade de criar scripts, chamados de *rules,* que são executados quando o udev detecta uma alteração específica no periférico.

A capacidade de criar *rules* amplia as possibilidades para os projetos Linux, pois é possível agregar mais dados sobre os periféricos conectados em sua aplicação, podendo assim, agregar maior valor para seu projeto.

Os resultados demonstrados nesse artigo comprovam a eficácia e a facilidade de uso do udev em aplicações.

Referências

BUENO, Cleiton. Linux – O poderoso udev. 2015. Disponível em: https://cleitonbueno.com/linux-o-poderoso-udev/. Acesso em: 05 fev. 2022.

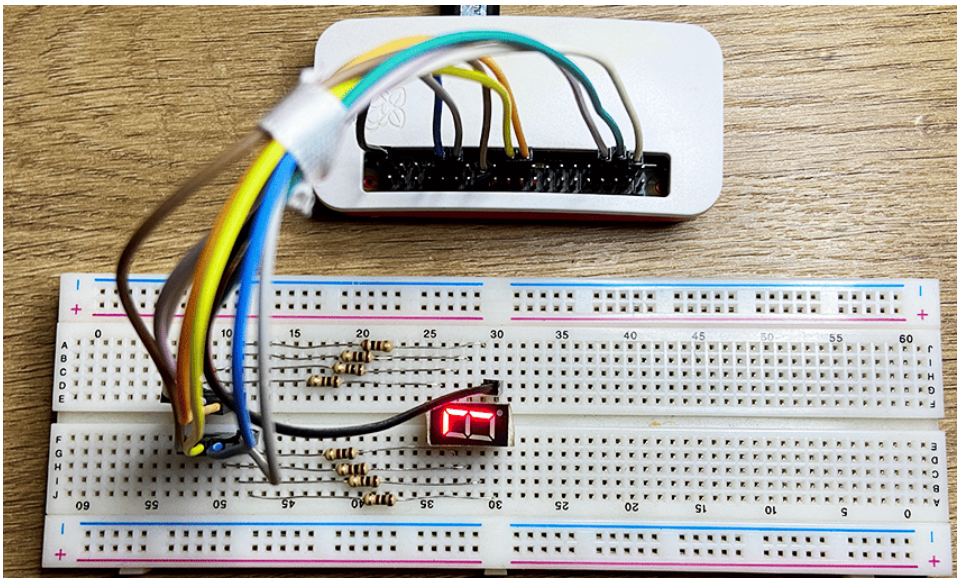
UDEV. Disponível em: https://wiki.archlinux.org/title/udev#About_udev_rules. Acesso em: 05 fev. 2022.

Autor: Yago Caetano

Graduado em Tecnologia em Automação Industrial pela FATEC SBC, e graduando em Engenharia da Computação pela FTT. Sou uma pessoa entusiasmada com tecnologias, sejam sistemas embarcados, aplicativos móveis e aplicativos da web

Publicado originalmente em:
https://embarcados.com.br/utilizando-o-udev-para-criar-automacoes-com-porta-usb-no-linux/

Drivers Linux: O passo a passo do driver para um display de 7 segmentos



Este artigo mostra o passo a passo do desenvolvimento de um driver baseado em *device-tree* para um display de 7 segmentos de cátodo comum. Neste caso estamos utilizando o computador de placa única Raspberry Pi Zero W (com a distro Raspbian 10 LTS) e o display HS-3161, mas é possível replicar para outros computadores de placa única e outros displays trocando a pinagem e fazendo as devidas mudanças no arquivo de overlay do *device-tree*.

O repositório com o código completo pode ser acessado aqui.

Módulos de driver

Uma característica muito interessante das distribuições Linux é a possibilidade de adicionar ou remover algumas funcionalidades do sistema operacional enquanto ele já está operando. Os compilados de códigos que podem ser carregados a qualquer momento ao *kernel* para adicionar essas funcionalidades são chamados de módulos e podem ser classificados em diversos grupos de acordo com a sua aplicabilidade no sistema. Neste artigo estaremos tratando especificamente de um driver carregado ao *kernel* via módulo, ou seja, um código que permite a comunicação do *kernel* com um *hardware* (neste caso o display HS-3161) e que pode ser carregado ao sistema dinamicamente.

Esquemático e pinagem

Para facilitar o entendimento do projeto, a tabela a seguir mostra quais GPIOs da Raspberry Pi foram conectadas a cada pino do display:

Segmento	**Pino HS-3161**	**GPIO Raspberry**
a	10	PA2
b	9	PA3
c	8	PA4
d	5	PA5
e	4	PA6
f	2	PA7
g	3	PA9
dp	7	PA10

Tabela 1 – Pinagem utilizada no projeto

O componente HS-3161 é um display de LEDs de 7 segmentos com configuração de cátodo comum, ou seja, os cátodos de todos os LEDs são conectados e devem ser aterrados. Para acender cada um dos segmentos precisamos fornecer pelo menos a mínima corrente de operação deste LED.

Como os segmentos devem ser acionados separadamente, cada um precisou ser conectado a uma *GPIO* diferente da Raspberry Pi, a qual fornece uma tensão de 3.3V nesses pinos quando a saída lógica é 1. O modelo de display usado foi HS-3161AS, que possui segmentos de LED vermelho, assim, considerando que a queda em cada LED seria de aproximadamente 2V a utilização de um resistor de 100Ω resultaria em uma corrente de 13mA, que é o suficiente para um LED operar (normalmente a corrente de operação fica entre 6mA e 20mA):

$$i = \frac{(3{,}3-2)}{100} = 13mA$$

A seguir pode-se observar o esquemático do circuito montado:

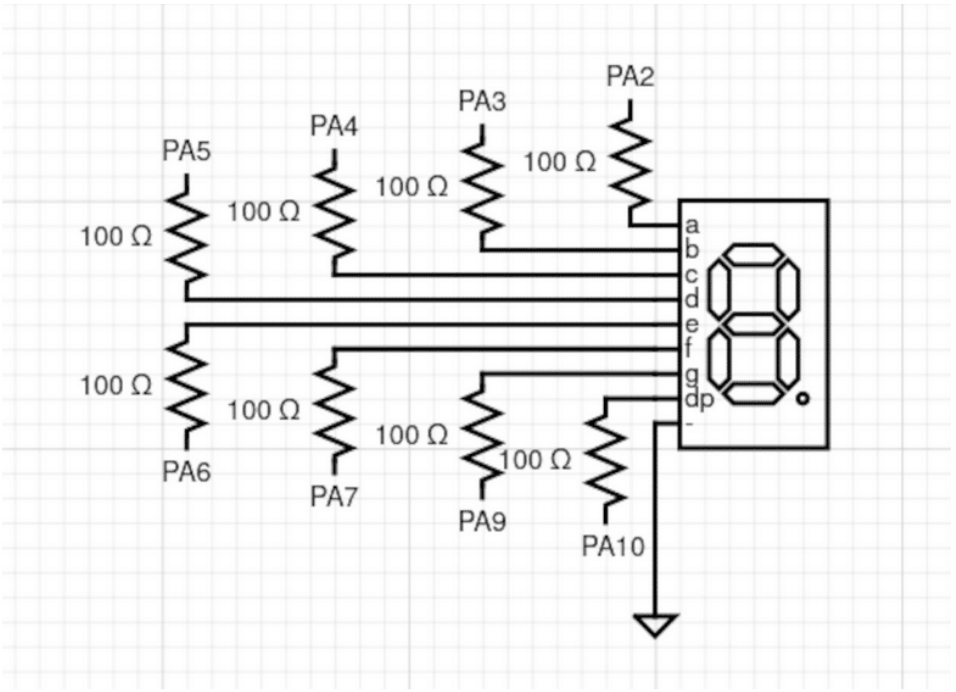


Figura 1 – Esquemático do circuito do HS-3161

Como o driver funciona?

Depois de carregar o módulo será possível encontrar os arquivos do driver na pasta */sys/class/7segment*. Para acender e desligar o LED do ponto deve-se escrever, respectivamente, 1 e 0 no arquivo *enableDP*. O valor escrito no arquivo *value* será *o número* exibido no display, caso seja escrito um valor superior a 9 será exibido o caractere E de erro. A leitura dos arquivos irá retornar o último valor escrito.

Passo a Passo

Headers do kernel

Para escrever um módulo de driver para uma distribuição Linux são necessários alguns *headers* em C que contém funções, *structs* e variáveis indispensáveis para o desenvolvimento do *kernel*. Para instalar esses *headers* via terminal para qualquer distribuição baseada em Debian em qualquer versão de kernel basta usar o seguinte comando:

```
sudo apt-get install linux-headers-$(uname -r)
```

No caso específico da Raspberry Pi também é possível fazer a instalação através do comando:

```
sudo apt-get install raspberrypi-kernel-headers
```

Makefile

Para gerar o compilado do módulo do driver a partir do código C é necessário um arquivo Makefile. O ideal é criar um diretório, neste caso foi criado um chamado LinuxDeviceDrivers e lá um arquivo chamado Makefile com o seguinte conteúdo:

```
obj-m += 7segment.o
all : modulo

modulo:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Por padrão este Makefile irá compilar o arquivo C do módulo, que foi chamado de 7segment.c, e gerar o arquivo 7segment.ko necessário para carregar o módulo no kernel.

Funções do Módulo

No arquivo 7segment.c devem estar definidas duas funções referentes ao módulo, uma que será chamada quando o módulo é carregado no *kernel* e outra que é chamada quando ele é removido do *kernel*. A seguir pode-se observar um exemplo de declaração dessas funções:

```
#include <linux/module.h>
#include <linux/kernel.h>

/***************Funções de
Módulo****************/

MODULE_LICENSE("GPL");

static int segmentsDisplay_init(void)
{
    printk(KERN_ALERT "MÓDULO
INICIALIZADO!");
    return 0;
}

static void segmentsDisplay_exit(void)
{
    printk(KERN_ALERT "MÓDULO REMOVIDO!");
}

module_init(segmentsDisplay_init);
module_exit(segmentsDisplay_exit);
```

Para essa implementação foram necessários dois *headers*. No *header* *module.h* consta a definição das macros *MODULE_LICENSE(), module_init() e module_exit()*, através delas atribui-se, respectivamente, a licença do módulo e as funções de inicialização e remoção do módulo. A função de inicialização por padrão retorna um *static int* e, neste caso, quando chamada, emite a mensagem "MÓDULO INICIALIZADO" no *log* do *kernel* (acessado via terminal pelo comando *dmesg*). Já a função de remoção do módulo não retorna nada por definição e aqui emite a mensagem "MÓDULO REMOVIDO" no *log* do *kernel.*

Para carregar o módulo no *kernel* deve-se compilar o arquivo C através do Makefile, para isso basta utilizar o comando *make* e depois que o arquivo .ko for gerado deve-se proceder como indicado a seguir:

```
sudo insmod 7segment.ko
```

Já para remoção do módulo pode ser realizada no terminal por:

```
sudo rmmod 7segment
```

Diretório no */sys/class*

Para acessar os arquivos do drive do display é necessário criar uma classe e consequentemente um diretório no caminho */sys/class.* As informações da classe

são registradas em uma *struct* chamada *class* definida no *header device.h*, a seguir consta como esta struct foi declarada globalmente no código:

```
static struct class *device_class = NULL;
```

Depois é preciso alocar um espaço da memória para esta classe, aqui utilizamos a função *kzalloc* definida no *header slab.h*. Abaixo está indicado o trecho de código que foi incrementado na função de inicialização do módulo para que a classe fosse devidamente criada e nomeada como 7segment:

```
/*Criando a classe no diretório /sys/class*/
        device_class = (struct class *)kzalloc(sizeof(struct  class),GFP_ATOMIC);
        if(!device_class){
           printk("Erro na alocação da classe!");
        }
        device_class->name = "7segment";
        device_class->owner = THIS_MODULE;
        ret = __class_register(device_class,&key);
```

O procedimento correto quando o módulo é removido é limpar a memória alocada para essa classe e destruir essa classe no */sys/class*, portanto a função de remoção do módulo também deve ser modificada, com o acréscimo das seguintes linhas de código:

```
/*Destruindo a classe no /sys/class*/
class_unregister(device_class);
class_destroy(device_class);
```

Arquivos do driver

Após a criação da classe deve-se criar os arquivos que permitem a integração do usuário com display e que estarão localizados dentro do diretório */sys/class/7segment.* Para cada arquivo desta classe é preciso atribuir uma *struct* chamada *class_attribute*, que encontra-se no *header device.h,* e também duas funções: uma que é chamada quando o arquivo é aberto e outra quando ele é fechado. Como explicitado acima, existem dois arquivos para este driver: um para acender/desligar o led do ponto e outro para o usuário definir qual número será mostrado no display. O trecho seguinte exemplifica a declaração e definição das funções para um deles, o arquivo *value*:

```
/********Declaração das funções dos arquivos********/
static ssize_t show_value( struct class *class, struct class_attribute *attr, char *buf );
static ssize_t store_value( struct class *class, struct class_attribute *attr, const char *buf,
size_t count );

/***************Variaveis Globais*****************/
volatile int value_display;

/********Funções de escrita e leitura arquivo value*********/
static ssize_t show_value( struct class *class, struct class_attribute *attr, char *buf ){

      printk("Valor do display - LEITURA!");
      return sprintf(buf, "%d", value_display);
}

static ssize_t store_value( struct class *class, struct class_attribute *attr, const char *buf,
size_t count ){
       printk("Valor do display - ESCRITA!");
       sscanf(buf,"%d",&value_display);
       return count;
}
```

Quando o arquivo *value* for lido, por exemplo, a função *show_value* irá exibir o último número escrito no arquivo, que é sempre registrado na variável *value_display,* e exibirá a frase "Valor do display – ESCRITA!" no *log* do *kernel.* Mas para isso acontecer deve-se relacionar um atributo de cada arquivos com as funções de arquivo e com classe criada no */sys/class*. A seguir temos a exemplificação da definição de um atributo de arquivo:

```
struct class_attribute *attr_value = NULL;
```

E, então, como deve-se registrar esse atributo, criando o arquivo, na função de inicialização do módulo:

```
 /*Criando o arquivo /sys/class/7segment/value*/
   attr_value = (struct class_attribute *)kzalloc(sizeof(struct class_attribute ),GFP_ATOMIC);
   attr_value->show = show_value;
   attr_value->store = store_value;
   attr_value->attr.name = "value";
   attr_value->attr.mode = 0777 ;
   ret = class_create_file(device_class, attr_value);
   return 0;
}
```

Um ponto muito interessante é que uma das variáveis que atributos de arquivos recebem é a *mode*, ela define qual a permissão de leitura e escrita destes arquivos. A permissão 0777 permite que qualquer usuário escreva ou leia o arquivo, na maioria das aplicações ela é indesejada, é recomendado que o

desenvolvedor atente-se para qual permissão é mais adequada para o seu projeto.

Para destruir um arquivo na remoção do módulo deve-se liberar a memória alocada para seu atributo na função de remoção do módulo:

```
/*Destruindo o arquivo do /sys/class*/
kfree(attr_value);
```

Device-Tree

Nos sistemas operacionais baseados em Linux temos uma estrutura de dados chamada *device-tree* que é responsável por informar a descrição dos periféricos e componentes do *hardware* ao *kernel*. As portas de entrada e saída de dados (GPIOs) são mapeadas nessa estrutura, por isso podemos acessá-las em um módulo de driver fazendo uma *overlay* no *device-tree*.

Uma overlay no *device-tree* consiste em um arquivo de dados que altera a atual composição da *device-tree* do *hardware* em questão, ele é compilado e acoplado dinamicamente ao *kernel*. A estrutura da overlay para o Raspbian, que reserva os pinos para este projeto, foi salva em um arquivo chamado overlay.dts, seu conteúdo pode ser observado abaixo:

```
/dts-v1/;
        /plugin/;
        /{
        compatible = "brcm,bcm2835";
        fragment@0 {
        target-path = "/";
        __overlay__{
        my_device{
                compatible = "emc-logic,7segment";
                status = "okay";
                a-gpio = <&gpio 2 0>;
                b-gpio = <&gpio 3 0>;
                c-gpio = <&gpio 4 0>;
                d-gpio = <&gpio 5 0>;
                e-gpio = <&gpio 6 0>;
                f-gpio = <&gpio 7 0>;
                g-gpio = <&gpio 9 0>;
                dp-gpio = <&gpio 10 0>;                                 };
        };
        };
        };
```

O formato da *overlay* pode mudar para cada distro de Linux, mas parte da formatação do arquivo sempre se mantém a mesma. Em geral, é importante

atentar-se às variáveis *compatible,* a primeira refere-se a compatibilidade do processador (o *bcm2835* é o SoC utilizado na Raspberry Pi Zero W) e a segunda, a compatibilidade do driver, indica o nome do driver e qual a empresa ou desenvolvedor responsável pela manutenção dele. Este segundo *compatible* será indispensável nos próximos passos para que o módulo reconheça qual *overlay* contém as alterações necessárias para que o driver funcione corretamente.

A *overlay*, então, precisa ser compilada e depois carregada no *device-tree.* Esse procedimento também varia com cada distro, o Makefile com a adição do comando para a compilação da *overlay* no Raspbian segue abaixo:

```
obj-m += 7segment.o
      all : modulo dt

      dt: overlay.dts
        dtc -@ -I dts -O dtb -o overlay.dtbo
overlay.dts

      modulo:
        make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) modules

      clean:
        make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) clean
        rm -rf overlay.dtbo
```

Para carregar o *overlay* no Raspbian basta executar o comando no terminal:

```
sudo dtoverlay overlay.dtbo
```

Driver

Por fim, temos que realizar a integração da *overlay* de *device-tree* com o módulo do driver para que seja possível alterar o estado das *GPIOs* da Raspberry Pi de acordo com o que é escrito nos arquivos localizados no */sys/class/7segment.* Para isso precisamos definir duas *structs*, a primeira, *of_device_id,* está declarada no *header* *of_device.h* e leva como parâmetro o *compatible* do driver explicado no tópico acima, ela é usada para identificar todos os dispositivos na *device-tree* que utilizam esse driver. A segunda, *platform_driver,* declarada no *header* *platform_device.h*, é uma *struct* para a definição de um driver genérico de um dispositivo, ela recebe como parâmetros a *struct of_device_id* (caso utilize uma *overlay* com determinado *compatible*) e as funções de inicialização e remoção desse driver:

```
/***************Structs Globais*****************/
    static struct of_device_id driver_ids[] = {
       {.compatible = "emc-logic,7segment"},
       {/* end node */}
    };
   static struct platform_driver display_driver = {
       .probe = gpio_init_probe,
       .remove = gpio_exit_remove,
       .driver = { .name = "display_driver",
       .owner = THIS_MODULE,
       .of_match_table = driver_ids, }
   };
```

É preciso, então, definir e declarar as funções de inicialização (conhecida como *probe*) e de remoção do driver:

```
/***************Variaveis Globais*****************/
    struct gpio_desc *a, *b, *c, *d;
    struct gpio_desc *e, *f, *g, *dp;

    /******Declaração funções do Driver********/
    static int gpio_init_probe(struct platform_device *pdev);
    static int gpio_exit_remove(struct platform_device *pdev);

    /********Funções do Driver*********/
    static int gpio_init_probe(struct platform_device *pdev){
        printk("DRIVER INICIALIZADO!");
        a = devm_gpiod_get(&pdev->dev, "a", GPIOD_OUT_LOW);
        b = devm_gpiod_get(&pdev->dev, "b", GPIOD_OUT_LOW);
        c = devm_gpiod_get(&pdev->dev, "c", GPIOD_OUT_LOW);
        d = devm_gpiod_get(&pdev->dev, "d", GPIOD_OUT_LOW);
        e = devm_gpiod_get(&pdev->dev, "e", GPIOD_OUT_LOW);
        f = devm_gpiod_get(&pdev->dev, "f", GPIOD_OUT_LOW);
        g = devm_gpiod_get(&pdev->dev, "g", GPIOD_OUT_LOW);
        dp = devm_gpiod_get(&pdev->dev, "dp", GPIOD_OUT_LOW);
        return 0;
    }
   static int gpio_exit_remove(struct platform_device *pdev){
        printk("DRIVER REMOVIDO!");
        return 0;
    }
```

Note que na função de inicialização, explicitada acima, é utilizada a função *devm_gpiod_get* para inicializar todas as *GPIOs,* referentes a cada um dos segmentos definidos na *overlay,* em nível lógico baixo e atribuí-las às *structs gpio_desc,* de modo que seja possível alterar seus estados posteriormente. Para mais detalhes dessa função e *struct* é interessante verificar o *header* *gpio/consumer.h*. Com a declaração e definição das funções de inicialização e

remoção do driver é preciso registrá-lo no módulo, para isso os seguintes trecho foram, respectivamente, acrescentados nas funções de inicialização e remoção do módulo:

```
/*Inicializando o  driver*/  if(platform_driver_register(&display_driver)){
        printk("ERRO! Não foi possível carregar o driver! \n");          return -1;

}


/*Remoção do driver*/
platform_driver_unregister(&display_driver);
```

As últimas alterações para garantir o funcionamento pleno deste módulo de driver devem ser feitas nas funções de escrita e leitura dos arquivos *enableDP* e *value.* Para escrever qualquer nível lógico nas *GPIOs* dos segmentos do display utilizamos a função *gpiod_set_value,* o exato segmento que terá o nível lógico alterado é reconhecido pela *struct gpio_desc* que lhe foi atribuída na função de inicialização do driver. Assim, no caso do ponto, o nível lógico do segmento dp deve receber o último valor escrito pelo usuário no arquivo *enableDP.* Já o valor escrito pelo usuário no arquivo *value* (armazenado na variável *value_display)* deve modificar o nível lógico dos segmentos de a,b,c,d,e,f e g para que o número correto seja exibido no display. O trecho de código a seguir mostra, por exemplo, como exibir o número 3 no display:

```
gpiod_set_value(a, 1);
gpiod_set_value(b, 1);
gpiod_set_value(c, 0);
gpiod_set_value(d, 1);
gpiod_set_value(e, 1);
gpiod_set_value(f, 0);
gpiod_set_value(g, 1);
```

Conclusão

Existem diversas maneiras de se fazer um *driver* de um dispositivo em uma distro Linux, este artigo, apesar de demonstrar o passo a passo de um *driver* específico, esclarece alguns pontos que podem ser muito úteis em diversas aplicações como, por exemplo, criar uma classe no */sys/class*, acessar *GPIOs* via uma *overlay* no *device-tree* e registrar atributos e funções de arquivos. Uma pequena demonstração do resultado pode ser visualizado no video abaixo:

Referências

Linux Device Drivers, 3rd Edition – Jonathan Corbet, Alessandro Rubini e Greg Kroah-Hartman – O'Reilly Media.

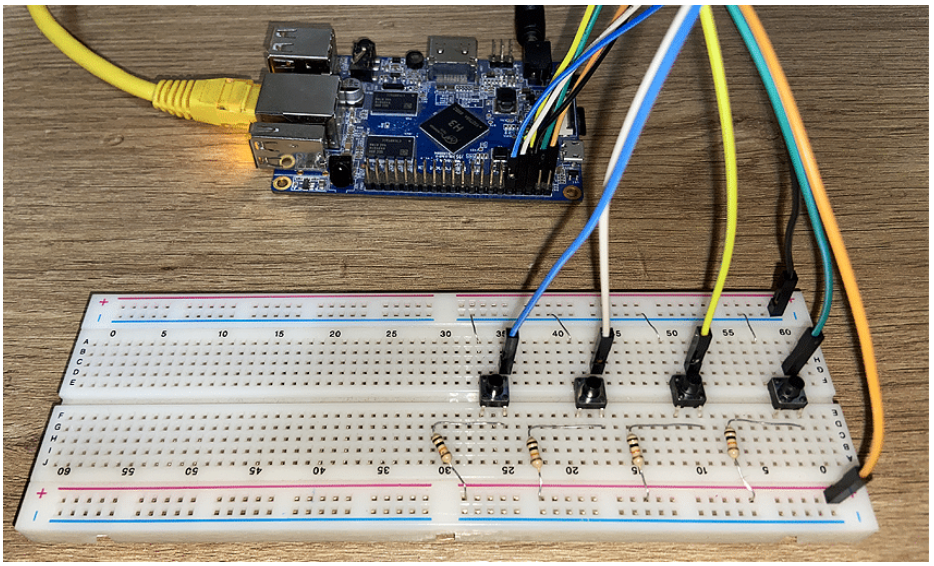
https://github.com/Johannes4Linux/Linux_Driver_Tutorial/tree/main/21_dt_gpio

Autora:Brenda Jacomelli https://www.emc-logic.com/

Engenheira eletricista pela USP de São Carlos e co-proprietária da empresa Emc-Logic, na qual atua com consultoria, treinamento e desenvolvimento de software para sistemas embarcados microcontrolados e microprocessados.

Publicado originalmente em:
https://embarcados.com.br/driver-linux-para-display-de-7-segmentos/

Drivers Linux: Módulo de driver para múltiplos dispositivos.



Este artigo tem como objetivo exemplificar a criação de um módulo de *driver* que pode ser utilizado para mais de um dispositivo através da adição de *overlays* na *device-tree.* O código do módulo demonstrado no decorrer do artigo refere-se a um driver para uma Orange Pi (armbian 22.05) que monitora as saídas de múltiplos circuitos com *push buttons* e pode ser acessado na íntegra aqui.Para mais detalhes do que são *drivers* de dispositivos carregados no *kernel* via módulos é recomendada a leitura deste artigo.

Esquemático circuitos

Abaixo está representado o exemplo do circuito com *push button* para o qual o módulo foi feito, neste caso se optou pelo resistor na configuração *pull up* e a saída Vout foi conectada a uma GPIO da Orange Pi. Portanto, quando o *push button* está em repouso, na saída Vout mede-se a tensão 3,3V (nível lógico alto ou 1) e quando o botão é pressionado mede-se uma tensão próxima de 0V (nível lógico baixo ou 0).

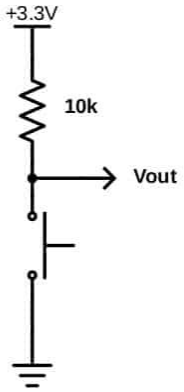


Figura 1 – Esquemático circuito com *push button*

Como o driver funciona?

Este driver irá funcionar para um número indeterminado, porém limitado, de circuitos com *push buttons*. Toda vez que desejar monitorar mais um desses circuitos é necessário executar o *shell script overlayGen* e passar o número da GPIO da Orange Pi como parâmetro (a pinagem completa desse computador pode ser acessado aqui), o sistema, então, será reiniciado e a *overlay* criada será inserida no *device tree* do armbian.

Com todas as *overlays* adicionadas e com o módulo já carregado no *kernel* será possível acessar um diretório no caminho */sys/class* para cada um dos circuitos previamente adicionados, o formato do nome desses diretórios é *push_button_x* , sendo x um número que identifica sequencialmente cada um dos circuitos. Portanto, caso você esteja monitorando três circuitos será possível acessar os seguintes diretórios: *push_button_0, push_button_1 e push_button_2.* Nesses diretórios será possível encontrar o arquivo *pressNum* que irá conter a quantidade de vezes que o botão em questão foi pressionado.

Passo a Passo

Headers do kernel

Para escrever um módulo de driver para uma distribuição Linux são necessários alguns headers em C que contém funções, *structs* e variáveis indispensáveis para o desenvolvimento do *kernel*. Para instalar esses *headers* via terminal para qualquer distribuição baseada em Debian (como o armbian) e qualquer versão de *kernel* basta usar o seguinte comando:

```
sudo apt-get install linux-headers-$(uname -r)
```

Makefile

Para gerar o compilado do módulo do driver a partir do código C é necessário um arquivo Makefile. O ideal é criar um diretório, neste caso foi criado um chamado multi-devices-driver e lá um arquivo chamado Makefile com o seguinte conteúdo:

```
obj-m += pshBtns.o
all : modulo

modulo:
    make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build
M=$(PWD) clean
```

Por padrão este Makefile irá compilar o arquivo C do módulo, que foi chamado de pshBtns.c e gerar o arquivo pshBtns.ko necessário para carregar o módulo no *kernel*.

Funções do driver

Usualmente quando opta-se por carregar um *driver* no *kernel* via módulo cria-se no código principal, neste caso chamado *pshBtns.c*, duas funções para o módulo, uma de inicialização e uma de remoção, e as chama através das macros *module_init* e *module_exit* definidas no *header* *module.h*. Essa abordagem que foi utilizada neste artigo aqui não é tão trivial quando se deseja utilizar um mesmo módulo de *driver* para um número imprevisível de dispositivos, portanto para isso utilizamos o método de inicialização e remoção de *drivers* implementado no *header* *platform_device.h*.

O trecho de código abaixo exemplifica melhor essa abordagem, primeiro definimos duas *structs*, a primeira, *of_device_id*, está declarada no *header* *of_device.h* e leva como parâmetro o *compatible* do driver, que é uma *label* que identifica todos os dispositivos na *device-tree* que utilizam esse driver. A segunda, *platform_driver* é uma *struct* para a definição de um driver genérico de um dispositivo, ela recebe como parâmeteros a *struct of_device_id* e as funções de inicialização e remoção desse driver.A macro module_platform_driver garante

que quando o módulo for carregado/removido no *kernel* as funções de inicialização e remoção, respectivamente pbtn_init_probe e pbtn_exit_remove, sejam chamadas sempre que identificado algum dispositivo no *device-tree* com o compatible definido. Então, por exemplo, se no *device-tree* existirem dois dispositivos com o *compatible*"emc-logic, pshBtns" será impressa duas vezes a mensagem Inicialização! no log do *kernel* (acessado via comando *dmesg*) quando o módulo for carregado. O análogo ocorre quando o módulo for removido.

```
#include <linux/module.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/of_device.h>

/* declaração de funções */

static int pbtn_init_probe(struct platform_device *pdev);
static int pbtn_exit_remove(struct platform_device *pdev);

/* Strcuts */

static struct of_device_id pshBtns_ids[] = {
    {.compatible = "emc-logic,pshBtns"},
    {/* end node */}
};

static struct platform_driver pshBtns_driver = {
    .probe = pbtn_init_probe,
    .remove = pbtn_exit_remove,
    .driver = {
        .name = "pshBtns_driver",
        .owner = THIS_MODULE,
        .of_match_table = pshBtns_ids,
    }
};

/* Funções de inicialização e remoção do driver*/

static int pbtn_init_probe(struct platform_device *pdev){
    printk(KERN_ALERT"Inicialização!");
    return 0;
}

static int pbtn_exit_remove(struct platform_device *pdev){
    printk("Remoção!\n");
    return 0;
}


/* Módulo*/
```

```
MODULE_LICENSE("GPL");
module_platform_driver(pshBtns_driver);
```

Para carregar o módulo no *kernel* deve-se compilar o arquivo C através do Makefile, para isso basta utilizar o comando *make* e depois que o arquivo .ko for gerado deve-se proceder como indicado a seguir:

```
sudo insmod pshBtns.ko
```

Já para remoção do módulo pode ser realizada no terminal por:

```
sudo rmmod pshBtns
```

Overlays no Device-tree

Nos sistemas operacionais baseados em Linux temos uma estrutura de dados chamada *device-tree* que é responsável por informar a descrição dos periféricos e componentes do *hardware* ao *kernel.* As portas de entrada e saída de dados (*GPIOs)* são mapeadas nessa estrutura, por isso podemos acessá-las em um módulo de driver fazendo uma *overlay* no *device-tree.*

Uma *overlay* no *device-tree* consiste em um arquivo de dados que altera a atual composição da *device-tree* do *hardware* em questão, ele é compilado e acoplado assim como um módulo do *kernel.* Aqui queremos uma *overlay* para o armbian com a seguinte estrutura:

```
/dts-v1/;
/plugin/;
/{
    compatible = "allwinner,sun8i-h3";
    fragment@0 {
       target-path = "/";
       __overlay__{
               my_device_0{
               dev_num = <0>;
               compatible = "emc-logic,pshBtns";
               status = "okay";
               pBtn_label = "rpi-gpio-12";
               pBtn_gpio = <12>;
               };
       };
    };
};
```

Note, que além do *compatible* indicado nessa *overlay* existem mais três variáveis importantes que devem ser identificadas pelo módulo para garantir o funcionamento do *driver*: *dev_num,* que indica qual o número do dispositivo, *pBtn_label* e *pBtn_gpio* que armazenam as informações da GPIO que está sendo utilizada pelo dispositivo. Supondo que esse arquivo de *overlay* receba o nome de *overlay.dts*,para carregá-la no armbian deve-se utilizar o seguinte comando e depois reiniciar o sistema:

```
armbian-add-overlay overlay.dts
```

Como o intuito aqui é utilizar esse módulo para múltiplos dispositivo o processo de criação de *overlays* foi automatizado*,* o código a seguir refere-se ao *shell script* chamado *overlayGen*que cria *overlays* já no padrão do armbian e desse módulo, as carrega no *kernel* e reinicia o sistema para que as modificações sejam feitas de forma correta.

```
#!/bin/bash

declare -i i=0

while [[ $i -lt 6 ]]
do
    if [[ -f "overlay-$i.dts" ]]
    then
        echo "File overlay-$i.dts exists"
    else
        echo "File  overlay-$i.dts does not exist"

        echo "/dts-v1/;" > overlay-$i.dts
        echo "/plugin/;" >> overlay-$i.dts
        echo "/{" >> overlay-$i.dts
        echo "    compatible = \"allwinner,sun8i-h3\";" >> overlay-$i.dts
        echo "    fragment@0 {" >> overlay-$i.dts
        echo "        target-path = \"/\";" >> overlay-$i.dts
        echo "        __overlay__{" >> overlay-$i.dts
        echo "                my_device_$i{" >> overlay-$i.dts
        echo "                dev_num = <$i>;" >> overlay-$i.dts
        echo "                compatible = \"emc-logic,pshBtns\";" >> overlay-$i.dts
        echo "                status = \"okay\";" >> overlay-$i.dts
        echo "                pBtn_label = \"rpi-gpio-$1\";" >> overlay-$i.dts
        echo "                pBtn_gpio = <$1>;" >> overlay-$i.dts
        echo "                };" >> overlay-$i.dts
        echo "        };" >> overlay-$i.dts
        echo "    };" >> overlay-$i.dts
        echo "};" >> overlay-$i.dts

        armbian-add-overlay overlay-$i.dts
        sudo reboot -h now

        break
    fi
    ((i++))
done
```

O script é bem simples, os nomes dos arquivos das overlays geradas por ele são padronizados como overlay-x.dts, sendo x o número do dispositivo que não pode ser repetido e que é sempre inicializado no 0. No caso exemplificado neste artigo limitamos a seis o número máximo de circuitos de *push button* que o driver pode monitorar, mas esse limite pode variar dependendo da disponibilidade de GPIOs do computador de placa única utilizado. Então, quando o script é executado ele verifica se já existem todos os arquivos overlay-x.dts, com x de 0 a 5, e caso não ele os cria sequencialmente levando em conta o número da GPIO passado por parâmetro. Para executar esse script com a GPIO 12 da Orange Pi, por exemplo, deve-se executar no terminal:

```
./overlayGenerator.sh 12
```

Integração módulo com device-tree

A integração do módulo com os dados dos dispositivos armazenados na *device-tree* é feita na função de inicialização do *driver* e é necessária para que seja possível identificar qual *push button* foi pressionado via interrupções das GPIOs. O primeiro passo é declarar algumas variáveis:

```
/* Defines */

#define MAX_DEV_NUM 6

/* Variábeis globais */

int device_num = 0;
int times_onProbe = 0;
int times_onRemove = 0;

struct device_info{
    int pBtn_gpio;
    int dev_num;
    const char *pBtn_label;
    char buffer[15];
    uint numOf_presses;
    int irq_num;
};
```

A constante MAX_DEV_NUM vai determinar qual o limite de circuitos com push button que esse *driver* atende, apesar desse número poder ser adaptado para atender melhor às necessidades do hardware é IMPORTANTÍSSIMO que ele seja o mesmo utilizado como limite no *script* *overlayGen* para que não haja problemas com a alocação de memória. A variável *device_num* armazena o número do dispositivo que estamos tratando no momento, a *times_onProbe* e a *times_onRemove* contam respectivamente quantas vezes as funções de inicialização e remoção foram chamadas. A *struct device_info* armazena todos os dados importantes de cada dispositivo: seu número de identificação, os dados da GPIO e IRQ, o número de vezes que ele foi pressionado e o nome do seu diretório (buffer).

Assim, é possível alocar memória (usando o *kmalloc* do *header* *slab.h)* para o número máximo de dispositivos que podem ser usados e armazenar as informações de todos os dispositivos que estão sendo monitorados na função inicialização através das funções device_property_read_u32 e device_property_read_string (mais detalhes no *header* *property.h)*:

```
struct device_info *pBtn_info = NULL;

static int pbtn_init_probe(struct platform_device *pdev){
    int ret;
    struct device *dev = &pdev->dev;

    printk(KERN_ALERT"Inicialização!");

    /* Caso seja a primeira vez na função: alocar memória */
    if(times_onProbe == 0){
       /* alocação device_info */
      pBtn_info = (struct device_info *)kmalloc(MAX_DEV_NUM*sizeof(struct device_info),
GFP_ATOMIC);
      if(!pBtn_info){
            return -ENOMEM;
               printk("device_info erro de alocação");
       }
      printk("Primeira vez na inicialização!");
    }else{
    printk("Não é a primeira vez na inicialização!");
       }


    /* Qual o nº do dispositivo sendo inicializado? */
    ret = device_property_read_u32(dev,"dev_num",&device_num);
    (pBtn_info+device_num)->dev_num = device_num;
    /* Armazenando os dados na struct */
    ret = device_property_read_string(dev,"pBtn_label",&(pBtn_info+device_num)->pBtn_label);
    ret = device_property_read_u32(dev,"pBtn_gpio",&(pBtn_info+device_num)->pBtn_gpio);
    sprintf((pBtn_info+device_num)->buffer, "%s_%d", "push_button", device_num);
    printk("Device number is: %d", device_num);
    printk("GPIO pin is: %d", (pBtn_info+device_num)->pBtn_gpio);

    times_onProbe = times_onProbe + 1;

    return 0;
}
```

Na remoção do módulo a memória deve ser liberada:

```
static int pbtn_exit_remove(struct platform_device *pdev){

        /* Caso seja a primeira vez na função: desalocar memória */
    if(times_onRemove == 0){
        printk("Primeira vez na função de remoção!");
        kfree(pBtn_info);
    }

    times_onRemove = times_onRemove + 1;

    printk("Remoção!\n");

    return 0;

}
```

Classes e arquivos

Como citado anteriormente, cada dispositivo adicionado para ser monitorado pelo *driver* tem um diretório no caminho */sys/class* e um arquivo chamado *pressNum* que contém o número de vezes que o *push button* em questão foi pressionado. Por padrão, o nome deste diretório é *push_button_(número do dispositivo)*, então, por exemplo, se o dispositivo que está sendo inicializado tem na *overlay* a variável *dev_num* indicando 0, seu diretório será */sys/class/push_button_0*.

Para um dispositivo ter um diretório no */sys/class* ele precisa de uma classe, as informações de uma classe são sempre registradas em uma *struct* chamada *class* definida no *header* *device.h*. Cada um dos dispositivos possui um classe diferente, portanto sempre que a função de inicialização for chamada deve-se registrar uma nova classe com o nome armazenado na variável buffer da struct pBTn_info, na primeira vez que a função de inicialização for chamada, no entanto, deve-se por segurança alocar o espaço na memória para as classes do número máximo de dispositivos que podem ser monitorados:

```
struct device_info *pBtn_info = NULL;
static struct class *device_class = NULL;

static int pbtn_init_probe(struct platform_device *pdev){

int ret;
struct device *dev = &pdev->dev;
static struct lock_class_key __key;

    printk(KERN_ALERT"Inicialização!");

    /* Caso seja a primeira vez na função: alocar memória */
    if(times_onProbe == 0){

       /* class allocation */
    device_class = (struct class *)kmalloc(MAX_DEV_NUM*sizeof(struct class),GFP_ATOMIC);
    if(!device_class){
            printk("classe erro de alocação");
            return -ENOMEM;
      }
       /* alocação device_info */
      pBtn_info = (struct device_info *)kmalloc(MAX_DEV_NUM*sizeof(struct device_info),
GFP_ATOMIC);
      if(!pBtn_info){
            printk("device_info erro de alocação");
            return -ENOMEM;
      }
      printk("Primeira vez na inicialização!");
    }else{
    printk("Não é a primeira vez na inicialização!");
       }


    /* Qual o nº do dispositivo sendo inicializado? */
    ret = device_property_read_u32(dev,"dev_num",&device_num);
    (pBtn_info+device_num)->dev_num = device_num;
  /* Armazenando os dados na struct */
    ret = device_property_read_string(dev,"pBtn_label",&(pBtn_info+device_num)->pBtn_label);
    ret = device_property_read_u32(dev,"pBtn_gpio",&(pBtn_info+device_num)->pBtn_gpio);
    sprintf((pBtn_info+device_num)->buffer, "%s_%d", "push_button", device_num);
    printk("Device number is: %d", device_num);
    printk("GPIO pin is: %d", (pBtn_info+device_num)->pBtn_gpio);
    /* Registro da classe de cada um dos dispositivos */
  (device_class+device_num)->name = (pBtn_info+device_num)->buffer;
  (device_class+device_num)->owner = THIS_MODULE;
  ret = __class_register((device_class+device_num),&__key);

    times_onProbe = times_onProbe + 1;

    return 0;
}
```

Como em cada um dos diretórios encontra-se o arquivo *pressNum*, também é necessário alocar memória para eles e registrá-los. Assim, para cada um dos

arquivos é preciso atribuir uma *struct* chamada *class_attribute*, que também encontra-se no *header device.h,* e duas funções: uma que é chamada quando o arquivo é aberto e outra quando ele é fechado.

Neste caso, iremos usar as mesmas funções de arquivos para todos os dispositivos, portanto, toda vez que a função de abertura de arquivo for chamada precisamos identificar de qual dispositivo o usuário quer ler o número de vezes que o *push button* foi pressionado. Para isso lemos o nome da classe que o arquivo pertence, seu décimo terceiro caractere refere-se ao número do dispositivo, que aqui não ultrapassa uma dezena, transformamos isso para um inteiro e fazemos a conversão da tabela ASCII (o caractere 0 corresponde ao inteiro 48, por isso sempre retira-se 48 do valor obtido):

```
static ssize_t show_pressNum( struct class *class, struct class_attribute *attr, char *buf );
static ssize_t store_pressNum( struct class *class, struct class_attribute *attr, const char
*buf, size_t count );

/* Funções de arquivos: ler e escrever */

static ssize_t show_pressNum( struct class *class, struct class_attribute *attr, char *buf ){
    uint value = 0;
    int num = 0;

    num = (int)(*class).name[12];

    num = num - 48;

    printk("Numero do push button: %d, Arquivo lido!", num);
    value = (pBtn_info+num)->numOf_presses;
    return sprintf(buf, "%d", value);
}
```

A partir disso, basta alocar memória para os arquivos e registrá-los na função de inicialização, de maneira análoga ao que fizemos para as classes:

```
struct device_info *pBtn_info = NULL;
static struct class *device_class = NULL;
struct class_attribute *class_attr = NULL;

static int pbtn_init_probe(struct platform_device *pdev){

int ret;
struct device *dev = &pdev->dev;
static struct lock_class_key __key;

    printk(KERN_ALERT"Inicialização!");

    /* Caso seja a primeira vez na função: alocar memória */
    if(times_onProbe == 0){

       /* alocação classe */
    device_class = (struct class *)kmalloc(MAX_DEV_NUM*sizeof(struct class),GFP_ATOMIC);
    if(!device_class){
            printk("classe erro de alocação");
            return -ENOMEM;
      }
       /* alocação atributo de arquivo */
      class_attr = (struct class_attribute *)kmalloc(MAX_DEV_NUM*sizeof(struct class_attribute),
GFP_ATOMIC);
      if(!class_attr){
               printk("atributo de arquivo erro de alocação");
            return -ENOMEM;
      }
       /* alocação device_info */
      pBtn_info = (struct device_info *)kmalloc(MAX_DEV_NUM*sizeof(struct device_info),
GFP_ATOMIC);
      if(!pBtn_info){
               printk("device_info erro de alocação");
            return -ENOMEM;
       }
      printk("Primeira vez na inicialização!");
    }else{
    printk("Não é a primeira vez na inicialização!");
       }


    /* Qual o nº do dispositivo sendo inicializado? */
    ret = device_property_read_u32(dev,"dev_num",&device_num);
    (pBtn_info+device_num)->dev_num = device_num;
  /* Armazenando os dados na struct */
    ret = device_property_read_string(dev,"pBtn_label",&(pBtn_info+device_num)->pBtn_label);
    ret = device_property_read_u32(dev,"pBtn_gpio",&(pBtn_info+device_num)->pBtn_gpio);
    sprintf((pBtn_info+device_num)->buffer, "%s_%d", "push_button", device_num);
    printk("Device number is: %d", device_num);
    printk("GPIO pin is: %d", (pBtn_info+device_num)->pBtn_gpio);

    /* Registro da classe de cada um dos dispositivos */
  (device_class+device_num)->name = (pBtn_info+device_num)->buffer;
  (device_class+device_num)->owner = THIS_MODULE;
```

```
  ret = __class_register((device_class+device_num),&__key);

    /* Registro atributo de arquivos */
    (*(class_attr + device_num)).show = show_pressNum;
    (*(class_attr + device_num)).store = store_pressNum;
    (*(class_attr + device_num)).attr.name = "pressNum";
    (*(class_attr + device_num)).attr.mode = 0777 ;
    ret = class_create_file((device_class+device_num), &(*(class_attr+device_num)));

    times_onProbe = times_onProbe + 1;

    return 0;
}
```

Lembrando que toda memória alocada na inicialização deve ser liberada na remoção:

```
static int pbtn_exit_remove(struct platform_device *pdev){

  if(times_onRemove == 0){
        printk("Primeira vez na função de remoção!\n");
       class_unregister(device_class);
       class_destroy(device_class);
       kfree(class_attr);
        kfree(pBtn_info);
    }else{
       printk("Não é a primeira vez na função de remoção!");
        class_unregister((device_class+times_onRemove));
        class_destroy((device_class+times_onRemove));
    }

    times_onRemove = times_onRemove + 1;

    printk("Remoção!\n");

    return 0;

}
```

GPIO e Interrupção

Por fim, como o intuito é registrar quantas vezes o *push button* de cada dispositivo foi pressionado, precisamos alocar um pino de entrada digital para os circuitos e também habilitar a interrupção para cada um deles. Já vimos como as *overlays* são inseridas na *device tree* passando o pino GPIO por parâmetro e, na inicialização de cada um dos dispositivos, já vimos como armazenar o número desse pino na *struct* pBtn_info. Agora, utilizaremos funções definidas nos *headers* *consumer.h*, *gpio.h* e *interrupt.h* para registrar as GPIOs e as

interrupções dos dispositivos na função de inicialização, para isso deve-se acrescentar na função os seguintes trechos de código:

```
 /* Alocando a GPIO para o dispositivo*/
  if(gpio_request((pBtn_info+device_num)->pBtn_gpio, (pBtn_info+device_num)->pBtn_label)){
        printk("Erro!\n");
        return -1;
    }
  /* Definindo a GPIO como entrada  */
  if(gpio_direction_input((pBtn_info+device_num)->pBtn_gpio)) {
        printk("Erro!\n");
        gpio_free((pBtn_info+device_num)->pBtn_gpio);
        return -1;
    }
    /* Salvando o número da IRQ correspondente a GPIO */
  (pBtn_info+device_num)->irq_num = gpio_to_irq((pBtn_info+device_num)->pBtn_gpio);
  printk("IRQ num:%d", (pBtn_info+device_num)->irq_num);

  /* Habilitando a interrupção - trigger de 1 para 0 */
    if(request_irq((pBtn_info+device_num)->irq_num, (irq_handler_t) gpio_irq_handler,
IRQF_TRIGGER_FALLING, "my_gpio_irq", NULL) != 0){
        printk("Interrupt error!\n: %d\n", (pBtn_info+device_num)->irq_num);
        gpio_free((pBtn_info+device_num)->pBtn_gpio);
        return -1;
    }

    /* Habilitando debounce de 100ms */
    if(gpio_set_debounce((pBtn_info+device_num)->pBtn_gpio,100)){
        printk("Debounce!\n");
}
```

Logo, define-se a função de interrupção que quando chamada, neste contexto, deve identificar em qual pino houve o *trigger* e adicionar 1 ao número de quantas vezes o *push button* em questão foi pressionado:

```
static irq_handler_t gpio_irq_handler(unsigned int irq, void *dev_id, struct pt_regs *regs);

/* Callback da interrupção */

static irq_handler_t gpio_irq_handler(unsigned int irq, void *dev_id, struct pt_regs *regs) {
    int irq_count;

  printk(KERN_ALERT "Interrupção!\n");

    /*Identificando o pino*/
  for(irq_count=0; irq_count<=times_onProbe;irq_count++){
       if((pBtn_info+irq_count)->irq_num == irq){
        break;
    }
  }

    printk("O push button %d foi pressionado", (pBtn_info+irq_count)->dev_num);

    /* Evitando estourar a memória do inteiro */
    if((pBtn_info+irq_count)->numOf_presses <= 1000000){
       (pBtn_info+irq_count)->numOf_presses = (pBtn_info+irq_count)->numOf_presses + 1;
    }
  return (irq_handler_t) IRQ_HANDLED;
}
```

Na função de remoção do módulo as GPIOs devem ser liberadas e a interrupção desabilitada:

```
static int pbtn_exit_remove(struct platform_device *pdev){

    if(times_onRemove == 0){
       printk("Primeira vez na função de remoção!\n");
    class_unregister(device_class);
    class_destroy(device_class);
    kfree(class_attr);
      kfree(pBtn_info);
       free_irq(pBtn_info->irq_num,NULL);
    gpio_free(pBtn_info->pBtn_gpio);
    }else{
      printk("Não é a primeira vez na função de remoção!");
      class_unregister((device_class+times_onRemove));
      class_destroy((device_class+times_onRemove));
      free_irq((pBtn_info+times_onRemove)->irq_num,NULL);
      gpio_free((pBtn_info+times_onRemove)->pBtn_gpio);

    }

    times_onRemove = times_onRemove + 1;

  printk("Remoção!\n");

  return 0;

}
```

Conclusão

No decorrer deste artigo foi demonstrada uma maneira de fazer um módulo de *driver* que possa ser utilizado para mais de um dispositivo semelhante, aqui fizemos esse módulo especificamente para uma Orange Pi (rodando armbian) e para monitorar a saída digital de um circuito simples com *push button*, mas os conceitos podem ser replicados para qualquer plataforma que rode qualquer distribuição Linux e para qualquer dispositivo: monitorando via GPIO, I2C, SPI, UART e etc. O interessante do circuito com *push button* é que ele simula eletricamente o comportamento de alguns sensores digitais como, por exemplo, os sensores de presença. Uma pequena demonstração do resultado deste driver sendo utilizado para monitorar quatro dispositivos pode ser visualizado no vídeo abaixo:

Referências

Linux Device Drivers, 3rd Edition – Jonathan Corbet, Alessandro

Rubini e Greg Kroah-Hartman – O’Reilly Media.

Autora: Brenda Jacomelli https://www.emc-logic.com/

Engenheira eletricista pela USP de São Carlos e co-proprietária da empresa Emc-Logic, na qual atua com consultoria, treinamento e desenvolvimento de software para sistemas embarcados microcontrolados e microprocessados.

Publicado originalmente em:
https://embarcados.com.br/drivers-linux-modulo-de-driver-para-multiplos-dispositivos/