UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

CÉSAR A. S. PASTORINI

Porte do Sistema Operacional Fuchsia para um Single Board Computer

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Marcelo de Oliveira Johann

Porto Alegre
2021

AGRADECIMENTOS

Agradeço primariamente aos meus Pais pelo bootstrap disso que sou. Desde o início do (meu) tempo, o Seu Luiz e a Dona Eloisa continuaram firmes no zelo e amparo.

Agradeço também à Pequena Hirwin, minha namorada e minha companheira absoluta.

Não posso deixar de agradecer aos meus amigos, por tantas boas boas experiências e pela parceria.

Faço um agradecimento aqui para todos os professores do Instituto de Informática cujo trabalho foi fundamental não apenas pelo conhecimento puro fornecido, mas principalmente pelo ensino de formas de pensar e resolver problemas.

E não posso deixar de agradecer a meu orientador Prof. Marcelo Johann que me aturou por todo esse tempo e sempre esteve disposto a ajudar.

RESUMO

O processo de porte de um sistema operacional consiste em implementar o suporte para os diversos componentes do hardware de forma que possam ser usados através de uma interface simples e uniforme. Esse trabalho realiza o porte do sistema operacional Fuchsia para uma Single Board Computer baseada no SoC RK3328 da Rockchip. O boot do kernel é feito através do U-Boot e usa o firmware de PSCI do Trusted Firmware-A da ARM. São desenvolvidos drivers para os subsistemas do SoC e porta serial. O framework de drivers do Fuchsia e os mecanismos fundamentais pelos quais o software consegue interagir e controlar dispositivos de hardware são introduzidos de forma a contextualizar e explicar o funcionamento dos drivers desenvolvidos.

Palavras-chave: Porte. rock64. google fuchsia. zircon. single board computer. sistemas operacionais. microkernel.

Porting the Fuchsia operating system to a Single Board Computer

ABSTRACT

The process of porting an operating system consists of implementing support for the various hardware components so that they can be used through a simple and uniform interface. The Fuchsia operating system is ported to a Single Board Computer based on Rockchip's RK3328 SoC. The Zircon kernel is booted using the U-Boot bootloader and uses the PSCI firmware from ARM's Trusted Firmware-A. Drivers for the SoC subsystems and UART are then developed. The Fuchsia driver framework and the fundamental mechanisms that software uses to interact and control hardware devices are introduced in order to contextualize and explain the operation of the developed drivers.

Keywords: porting, bringup, rock64, google fuchsia, zircon, drivers, u-boot, single board computer, operating systems, microkernel.

LISTA DE ABREVIATURAS E SIGLAS

ABI	Application Binary Interface
API	Application Programming Interface
CRU	Control and Reset Unit
DDK	Driver Development Kit
DID	Device ID
DLAB	Divisor Latch Access Bit
DLH	Divisor Latch High
DLL	Divisor Latch Low
DSL	Domain Specific Language
EL	Exception Level
FIDL	Fuchsia Interface Definition Language
GIC	Generic Interrupt Controller
GPIO	General Purpose Input Output
GRF	General Registers File
IDL	Interface Definition Language
IO	Input-output
IPC	Inter-process Communication
LCR	Line Control Register
MMIO	Memory-Mapped IO
MMU	Memory Management Unit
PID	Product ID
PLL	Phase-locked Loop
POSIX	Portable Operating System Interface
PSCI	Power State Coordination Interface

RBR	Receive Buffer Register
SBC	Single Board Computer
SoC	System on Chip
TEE	Trusted Execution Environment
TF-A	Trusted Firmware-A
TFTP	Trivial File Transfer Protocol
THR	Transmitter Holding Register
UART	Universal Asynchronous Receiver-Transmitter
VID	Vendor ID
VMAR	Virtual Memory Address Region
VMO	Virtual Memory Object
ZBI	Zircon Boot Image

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas operacionais são essenciais para o uso prático de dispositivos de hardware. Devido à complexidade e diversidade do hardware, seria extremamente difícil escrever um software que conseguisse utilizar os recursos de hardware de uma forma eficiente e, ao mesmo tempo, implementasse as mesmas funcionalidades em um dispositivo ligeiramente diferente.

Esse problema de *portabilidade* é resolvido por sistemas operacionais através de diversas abstrações de software que acabam criando uma espécie de máquina virtual, onde um programa (o software executando) não interage diretamente com o hardware, mas o faz através do sistema operacional (SILBERSCHATZ; GALVIN; GAGNE, 2018). Apesar de não ser perfeito, esse esquema resolve a maior parte do problema de portabilidade.

O porte de um sistema operacional para um dispositivo de hardware específico consiste em suportar os diversos componentes desse dispositivo de forma que aplicações executando no sistema consigam utilizar as funcionalidades e recursos do hardware. As alterações necessárias no sistema operacional para tal tarefa dependem do nível de suporte já existente em relação ao hardware em questão e também de quais funcionalidades do hardware se quer utilizar.

1.1 Motivação

O porte de um sistema operacional é uma tarefa desafiadora porque, além de exigir um conhecimento de diversos princípios e mecanismos de hardware, é necessário um conhecimento da arquitetura de *drivers* do sistema operacional em questão. Os detalhes de funcionamento dos diversos componentes de hardware e software envolvidos fazem ser difícil ter uma visão global de como as abstrações de software interagem com o hardware.

Sistemas operacionais baseados no *kernel* Linux são o padrão de fato em sistemas embarcados e, consequentemente, tem um suporte abrangente de diversos componentes de hardware. Isso inclui as *Single Board Computers*, que são um bom modelo de sistemas embarcados utilizados em dispositivos como smartphones, tablets, smartwatches, smart TVs, etc. Devido à ubiquidade do *kernel* Linux, já existem trabalhos que descrevem o processo de porte dele para sistemas embarcados (BELLONI, 2015; CLEMENT, 2016).

O sistema operacional Fuchsia[1] representa uma boa oportunidade para um estudo de caso de um porte. Por ser um sistema relativamente novo, o nível existente de suporte à dispositivos de hardware é menor e isso facilita a descoberta e apreciação de quais componentes e subsistemas de hardware precisam ser suportados. Além disso, o Fuchsia é baseado em um *microkernel* e implementa uma arquitetura de software extremamente modular e é interessante ver como isso têm influência no processo de porte.

1.2 Objetivos

Esse trabalho tem três objetivos inter-relacionados. O primeiro é portar o Fuchsia para a Rock64 de forma que uma aplicação executando no sistema operacional consiga utilizar um dispositivo periférico de entrada-saída.

O segundo objetivo é descrever o sistema operacional Fuchsia e, principalmente, seu *framework* de *drivers* de forma que seja possível compreender o funcionamento dos *drivers* desenvolvidos e como eles se inter-relacionam.

O terceiro objetivo é introduzir os princípios e mecanismos fundamentais através dos quais o software consegue interagir e controlar os dispositivos de hardware. Sem isso, é difícil entender como as diversas abstrações e construções de programação envolvidos em um porte mapeiam para as mecanismos que, ultimamente interagem com o hardware.

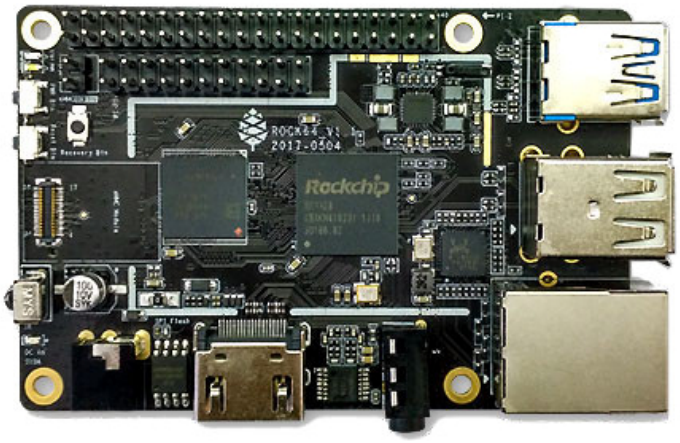
1.3 Rock64 e o SoC RK3328

O sistema embarcado para o qual o Fuchsia é portado neste trabalho é a Rock64 (Figura 1.1) que é uma *Single Board Computer* baseada no SoC RK3328 da Rockchip (PINE64, 2020b). Esse SoC integra a maioria dos componentes de hardware da placa, incluindo o processador Cortex-A53. A Figura 1.2 contém um diagrama simplificado do esquemático da Rock64 e ilustra alguns dos componentes integrados no RK3328. As principais fontes de documentação sobre o hardware são o datasheet e o manual de referência[2] do SoC (ROCKCHIP, 2017a; ROCKCHIP, 2017b).

[1] <https://fuchsia.dev>

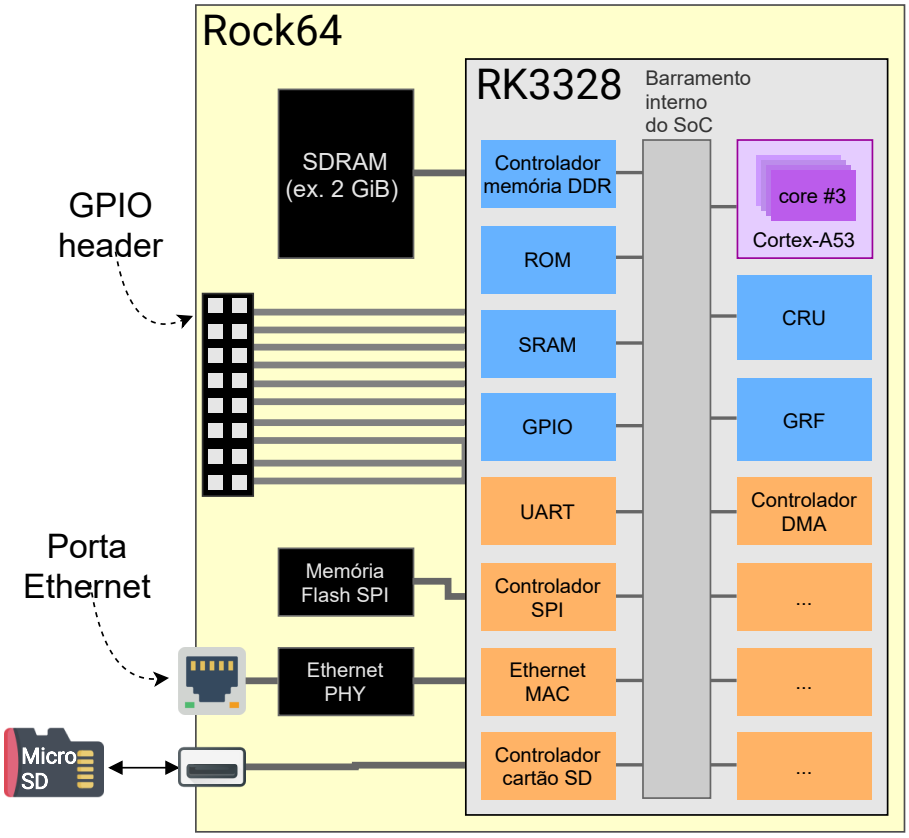
[2] Também conhecido como “Technical Reference Manual”, ou TRM

Figura 1.1: Rock64, a *Single Board Computer* utilizada neste trabalho.



Fonte: Adaptado de (PINE64, 2020b).

Figura 1.2: Componentes de hardware da Rock64.



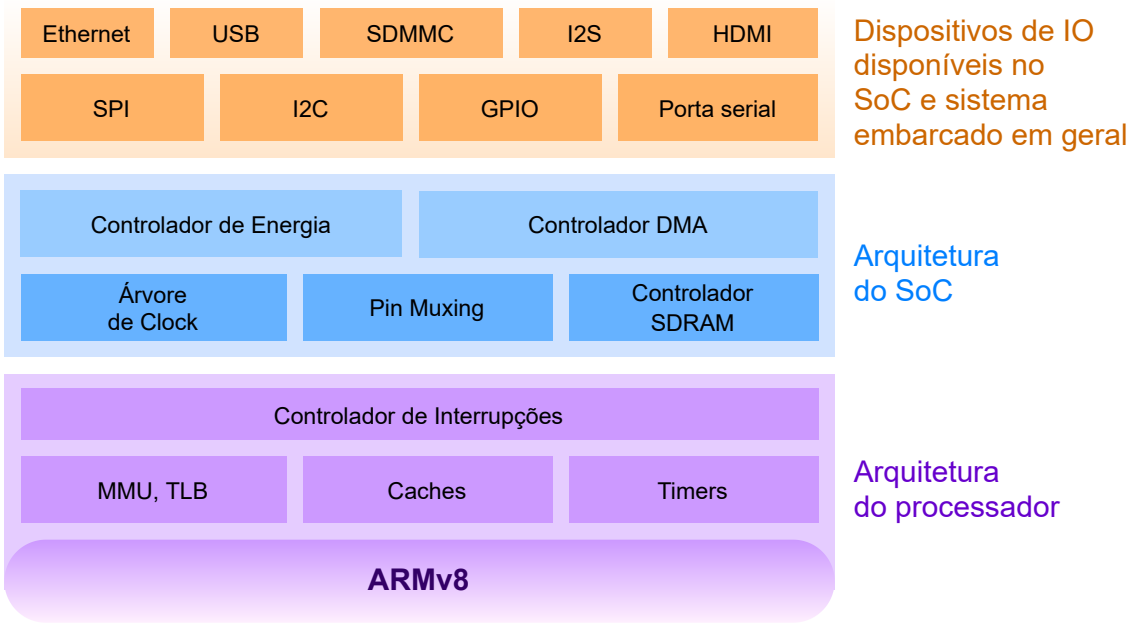
Fonte: O Autor.

2 CONCEITOS

Em termos gerais, a arquitetura de hardware de um computador de propósito geral pode ser resumida em quatro componentes: processador principal (ou CPU), memória RAM, controladores para periféricos de entrada-saída e um barramento principal (SILBERSCHATZ; GALVIN; GAGNE, 2018). Normalmente existem diversos outros componentes, muitos dos quais são processadores especializados para execução de tarefas como processamento de áudio e vídeo, aceleração gráfica (GPUs), processamento de sinais, etc.

Da perspectiva do software, os componentes e subsistemas de hardware a serem suportados podem ser visualizados em um diagrama como o da Figura 2.1. Esse diagrama ilustra de uma forma aproximada a dependência que dispositivos de hardware têm com serviços e mecanismos implementados por outros dispositivos.

Figura 2.1: Hierarquia aproximada de dispositivos de hardware a serem suportados.



Fonte: O Autor.

Na camada inferior está o processador principal (ou CPU) que é o componente de hardware mais importante pois ele é quem executa o software que controla e orquestra todos os outros componentes. O processador define a *Instruction Set Architecture* (ISA), que é a abstração fundamental representando a interface entre o hardware e o software (PATTERSON; HENNESSY, 2016). Além do conjunto de instruções, a ISA define diversas outras informações e mecanismos necessários para

utilizar os diversos recursos de um processador (PATTERSON; HENNESSY, 2016). Na prática, o suporte ao conjunto de instruções (e outros detalhes associados), é implementado no próprio *toolchain* de desenvolvimento de software em programas como compiladores, assemblers e linkers. Esse tipo de suporte está fora do escopo deste trabalho, já que todas todos os softwares utilizados suportam o conjunto de instruções da arquitetura ARMv8.

Além da ISA, a camada inferior na Figura 2.1, representa os componentes fundamentais de um processador moderno com arquitetura ARM. A *Memory Management Unit* (MMU), caches e timers estão muito acoplados à ISA porque possuem instruções e/ou registradores dedicados e têm influência direta na execução do software (ARM, 2019a). Já o Controlador de Interrupções (*Generic Interrupt Controller*, ou GIC na sigla em inglês) (ARM, 2021), por exemplo, tem acoplamento um pouco menor porque a maior parte da sua configuração pode ser feita pelo mesmo mecanismo (leia-se, *MMIO*) que os dispositivos das camadas superiores utilizam. Esses componentes precisam ser suportados explicitamente pelo software e não apenas pelo *toolchain*. Como esses componentes são fundamentais para um sistema operacional implementar suas abstrações e mecanismos básicos, eles costumam ser suportados no próprio *kernel* (SILBERSCHATZ; GALVIN; GAGNE, 2018).

A camada do meio na Figura 2.1 representa os mecanismos e dispositivos presentes no *System on Chip* (SoC). O SoC é o chip que integra o processador e a maior parte dos componentes de hardware em um sistema embarcado. Como será discutido na Seção 2.9, dentre outras coisas, o SoC contém mecanismos para controlar e distribuir diversos recursos de hardware necessários para a maior parte dos dispositivos integrados nele conseguirem funcionar e implementar suas funcionalidades.

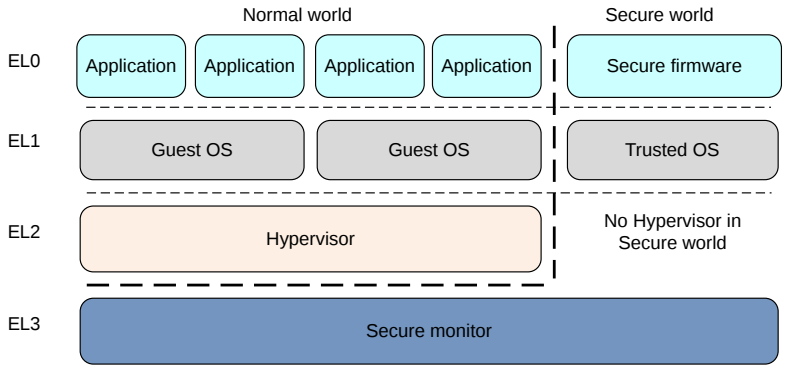
A camada superior na Figura 2.1 representa todos os outros dispositivos integrados no SoC e contém dispositivos periféricos de IO, controladores de barramentos, processadores especializados, etc. Esses dispositivos podem utilizar serviços de outros dispositivos, incluindo subsistemas do SoC.

2.1 Arquitetura ARMv8

A arquitetura ARM é composta de diversas versões sendo a mais recente — ARMv8-A — introduzida em 2011. Existem dois modos de execução. O primeiro, Aarch32 tem compatibilidade retroativa (na sua maior parte) com as versões ante-

riores da arquitetura. O segundo modo — Aarch64[1] — é novo e implementa, em especial, um novo conjunto de instruções com registradores de 64 bits, maior espaço de endereçamento e um novo modelo de exceções (ARM, 2019a). Em geral, quando se fala na arquitetura ARMv8, é subentendido que o modo de execução é o Aarch64. Inclusive, esse é o modo que um processador ARM começa a executar após um *reset*.

Figura 2.2: *Exception levels* na arquitetura ARMv8.



Fonte: Adaptado de (ARM, 2019b).

O modelo de exceções define quatro *exception levels* nos quais o processador pode operar (Figura 2.2). A diferença principal são os recursos do processador (e consequentemente do resto do hardware) que o software executando em cada nível consegue acessar em relação aos outros. As *exceptions* são o mecanismo através do qual o processador muda de *exception level*. O EL0 tem o objetivo de executar aplicações comuns de um sistema operacional e, por causa disso, tem o menor privilégio sem praticamente nenhum acesso à registradores e instruções relacionadas à configuração deste. Dessa forma, um software executando em EL0 não conseguiria alterar o funcionamento do processador de forma a afetar diretamente as garantias de segurança do resto do software executando. O nível EL1 é onde geralmente o *kernel* do sistema operacional executa, e esse nível tem acesso para controlar as configurações do nível inferior (EL0). Por exemplo, o *kernel* pode editar o mapa de memória virtual de um processo executando em EL0 ou ler/escrever na memória mapeada do processo. O nível EL1 também pode receber as interrupções de software geradas no nível EL0, que é o mecanismo pelo qual as chamadas de sistema são implementadas. Esse padrão se repete recursivamente até chegar no nível EL3.

A forma como os processadores ARMv8 lidam com interrupções de hardware é mapeada de forma elegante para *exceptions*: as interrupções de hardware são

[1]Também conhecido como ARM64 na terminologia do *kernel* Linux.

apenas mais um tipo de *exception* e têm uma entrada correspondente na tabela de *exception handlers*. Interrupções geradas por software (ex. via execução de uma instrução específica, instrução incorreta, ou acesso indevido) também têm uma entrada correspondente. Até o *reset* do processador é visto pelo software como a execução de um *exception handler* específico.

Há ainda outro modo de execução ortogonal aos níveis EL0-EL3, que divide o estados do processador em *Seguro* e *Não seguro* (Figura 2.2). A ideia é prover ainda outro nível de controle de acesso à recursos do próprio processador e, consequentemente, do resto do hardware, de forma a suportar um padrão próprio de software da ARM. Isso será discutido na Seção 2.7.

2.2 Memory-mapped IO

A comunicação do processador com dispositivos externos se dá através da leitura e escrita de registradores presentes nestes dispositivos (SILBERSCHATZ; GALVIN; GAGNE, 2018). Esses registradores são diferentes dos presentes no arquitetura do processador, os quais são acessados diretamente (e explicitamente) através das instruções de máquina (i.e. *assembly*).

Normalmente, um dispositivo exporta um conjunto de registradores com tamanho fixo (ex. 32 bits) onde cada registrador tem um ou mais campos de bit; esses campos são definidos para cada registrador e podem ser lidos, escritos ou ambos; e isso pode variar entre os diferentes registradores exportados pelo dispositivo. Além de ser algo específico a um dado dispositivo, essas definições de campos de bit e a semântica associada com cada um podem variar dinamicamente de acordo com o estado em que o dispositivo se encontra. A informação de quais registradores e o formato e significado de cada campo de bit são fornecidas no *datasheet* ou manual do dispositivo em questão.

Existem duas maneiras para um processador se comunicar diretamente com esses registradores. A primeira é através do uso de instruções *assembly* dedicadas. Essas instruções de leitura e escrita levam à utilização de um espaço de endereços dedicado para fazer IO que é independente do espaço de memória que as outras instruções acessam. Esse método é conhecido como *Port-mapped IO* (PMIO) pois os endereços nesse espaço dedicado à IO são conhecidos como portas.

A outra maneira, onde o processador utiliza as mesmas instruções de acesso à memória RAM, é conhecido como *Memory-mapped IO* (MMIO). Nesse caso, o conjunto de registradores de um dispositivo externo é visto pelo processador como uma simples região da memória. Com isso, todos os dispositivos externos acessíveis pelo processador vivem no mesmo espaço de endereçamento e, tecnicamente, a memória RAM é apenas mais um dispositivo externo no espaço de endereços visto pelo processador.

Um processador pode usar qualquer um destes métodos. Por exemplo, os processadores da arquitetura x86 utilizam ambos (SILBERSCHATZ; GALVIN; GAGNE, 2018). A arquitetura ARM utiliza exclusivamente o método de MMIO e, por isso, o resto desse texto considera apenas este método. Apesar disso, muitos dos conceitos são equivalentes para o caso de PMIO.

É importante notar que a associação entre um endereço de memória e um dispositivo externo é realizada fora do processador. Por exemplo, um processador ARM que executa uma instrução para escrever um byte no endereço `0xFF201000` não tem como saber se esse endereço aponta para uma posição na memória RAM ou para um registrador de algum outro dispositivo[2].

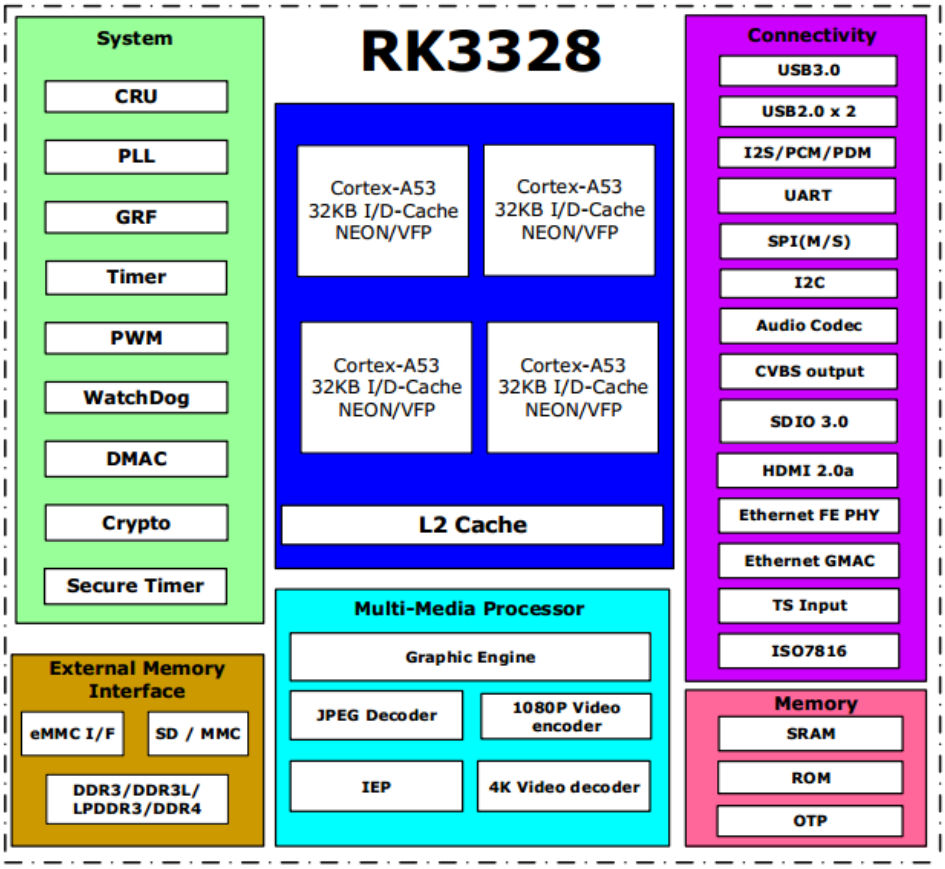
2.3 Mapa de memória

O conjunto de registradores de um dado dispositivo externo discutido anteriormente costumam ser agrupados em uma região contínua do espaço de endereçamento do processador. A associação entre regiões do espaço de memória e dispositivos externos correspondentes leva à noção de um mapa de memória. Esse mapa é fundamental para a utilização do hardware porque ele fornece a localização dos dispositivos no espaço de endereçamento do processador.

Como mencionado na Seção 2.2, esse mapeamento é implementado fora do processador. No caso de processadores ARM, que são usualmente integrados em um SoC, esse mapa é definido pelo SoC. A Figura 2.3 contém a lista completa dos dispositivos integrados no RK3328 e a Figura 2.4, disponível no manual do SoC (ROCKCHIP, 2017b), contém o mapa de memória do RK3328. Todos os dispositivos

[2] O sistema de memória virtual da arquitetura ARMv8 possibilita diferenciar isso em relação ao comportamento dos caches, mas é algo que precisa ser configurado a priori em *runtime* e, ainda assim, para o processador em si não há diferença.

Figura 2.3: Componentes do SoC RK3328 da Rockchip.



Fonte: Disponível em (ROCKCHIP, 2017a).

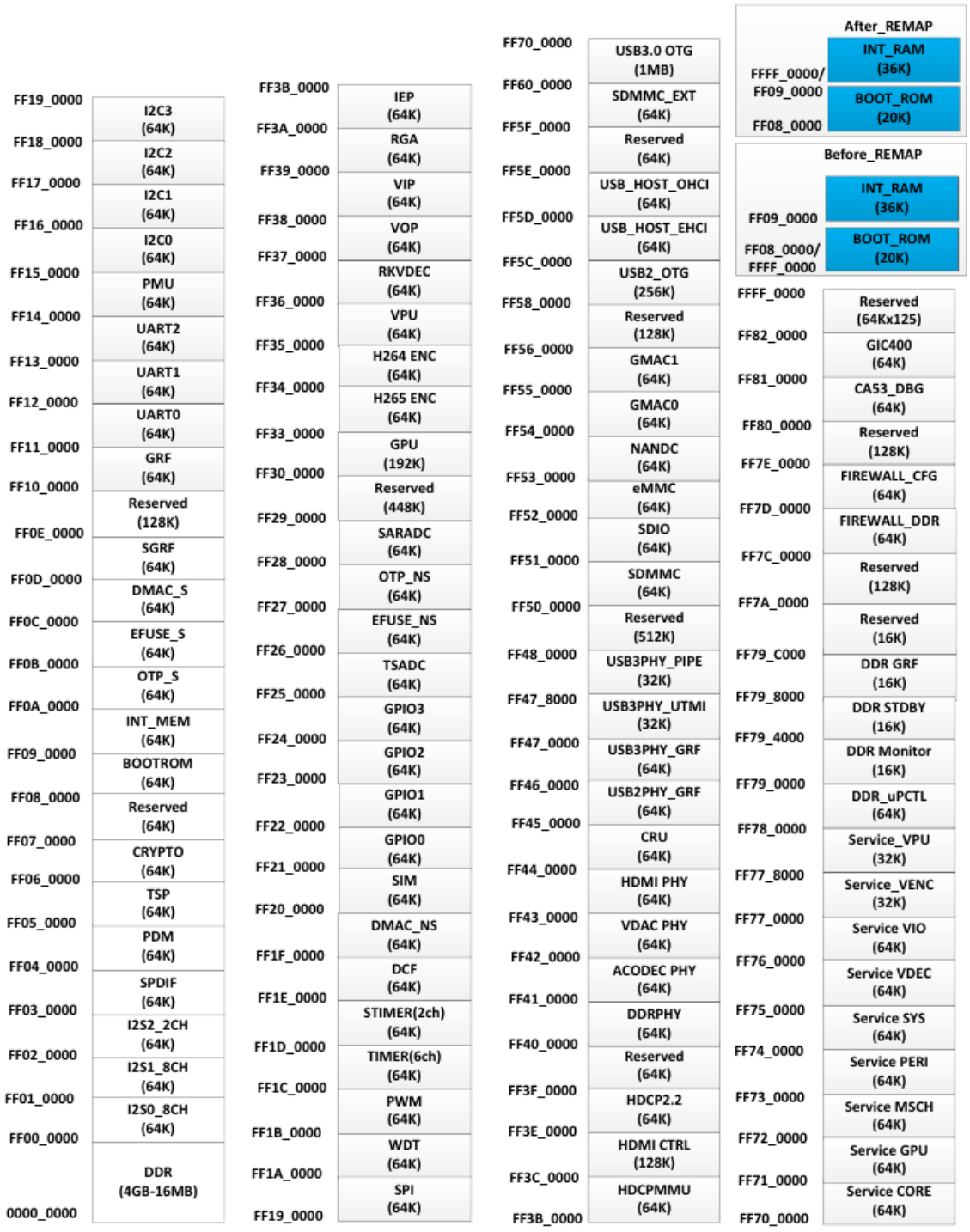
do SoC têm um região de MMIO associada no mapa de memória. Alguns dispositivos até têm mais de uma região associada.

É importante observar que um mesmo dispositivo de hardware pode ter mais de uma região no espaço de endereçamento associada à ele, com os registradores em cada região não necessariamente sendo iguais.

2.4 *Drivers*

A utilização de MMIO pelo processador para interagir com um dispositivo externo leva à noção de *driver*, que é um software capaz de interagir com um dispositivo de hardware específico e prover uma interface de software para tal controle. Isso é, um *driver* tem o conhecimento especifico da forma (ou protocolo) necessário para fazer com que um dado dispositivo execute as funções desejadas.

Figura 2.4: Mapa de memória do SoC RK3328.



Fonte: Fig. 1-1 na página 13 de (ROCKCHIP, 2017b).

Por um lado, os *drivers* são independentes do ambiente de software onde executam. Por exemplo, um *driver* pode ser um software executando como parte do *kernel* de um sistema operacional ou na forma de um processo de aplicação comum. Um bom exemplo disso, documentado em (MARX, 2019), mostra que um *driver* implementado para o *kernel* Linux é bem similar (em termos de arquitetura) a um *driver* feito para o Fuchsia. Mesmo que as abstrações de software utilizadas sejam diferentes, no final, a lógica (ou protocolo) utilizado para interagir com um dado dispositivo externo é a mesma e o *driver* vai executar as mesmas instruções de leitura e escrita nos registradores do dispositivo.

Porém, por outro lado, um *driver* geralmente não é trivialmente portável justamente por causa desse ambiente de execução e das abstrações de software utilizadas. Na prática, um *driver* é desenvolvido para um ambiente de execução específico da mesma forma que um programa de aplicação é desenvolvido para executar em um dado sistema operacional.

É interessante observar que *drivers* podem ser implementados por qualquer software que tenha alguma forma de acesso (normalmente direto como MMIO) à dispositivos de hardware no ambiente em que está executando. Um bom exemplo disso são os programas *bare metal*, que são softwares executando em um ambiente fora do sistema operacional. Os firmwares e *bootloaders* discutidos nas seções a seguir são exemplos de programas *bare metal* que, em sua maior parte, precisam interagir com dispositivos de hardware para conseguirem realizar suas funções. De uma forma estruturada ou não, esses softwares acabam implementando *drivers* que variam em complexidade e funcionalidade. O *bootloader* U-Boot discutido nas Seções 2.7.1 e 4.2 é um exemplo de software que, devido às suas diversas funcionalidades, possui uma infraestrutura de código relativamente completa e flexível (um *framework*) para o desenvolvimento de *drivers*.

2.5 Interrupções de hardware

As interrupções de hardware são um outro aspecto fundamental na interação com dispositivos de hardware pois elas possibilitam aos dispositivos externos notificar o processador (SILBERSCHATZ; GALVIN; GAGNE, 2018). Isto é, a comunicação não só é possível na direção processador para os dispositivos, mas também

o contrário. Apesar de ser apenas uma notificação, as interrupções possibilitam ao processador fazer outras tarefas e não depender exclusivamente de *polling* para verificar o estado dos dispositivos externos.

Assim, elas fazem também fazem parte da interface necessária para controlar (ou usar) um dispositivo de forma eficiente. As interrupções são associadas um-para-um com números, que são as *interrupt IDs*. Esse mapeamento é feito pelo SoC e, no caso do RK3328, está disponível no manual (ROCKCHIP, 2017b). Vale observar que um dispositivo de hardware pode ter zero ou mais interrupções de hardware e não necessariamente o uso delas é necessário para controlar (ou usar) determinadas funcionalidades do dispositivo.

Como mencionado antes, as interrupções de hardware na arquitetura ARMv8 são mapeadas para um tipo específico de exception no processador. Porém, diferentemente do mapa de memória, é necessário uma certa configuração para que elas gerem exceptions no processador.

O GIC é o gerenciador de interrupções programável padrão na arquitetura ARMv8 (ARM, 2021) e, como mencionado anteriormente, sua interface de configuração e uso têm uma região correspondente no mapa de memória do SoC. O GIC precisa de uma certa configuração antes de poder ser utilizado. Então, de certa maneira, o GIC precisa de um *driver*. Os detalhes de configuração do GIC estão fora do escopo desse trabalho.

No mapa de memória da Figura 2.4 o GIC-400 está na região que inicia no endereço físico `0xFF810000`[3].

2.6 Memória virtual

A memória virtual é um mecanismo que tem uso pervasivo em todo tipo de software. Além dos componentes de sistemas operacional, isso inclui a maioria dos softwares discutidos nas próximas seções. Memória virtual aqui denota o mecanismo do processador em si, e não necessariamente o uso desse mecanismo para implementar funcionalidades como aumento da quantidade de memória disponível sistema através de *paging* para disco (SILBERSCHATZ; GALVIN; GAGNE, 2018; PATTERSON; HENNESSY, 2016).

[3]O GIC400 é o nome de um componente específico desenvolvido pela ARM que implementa o padrão GICv2 que têm compatibilidade com os padrões GICv3 e GICv3 utilizados pelo processador Cortex-A53 (CORTEX-A53. . . , 2013).

Há diversas formas de configurar o sistema de memória virtual, cujo escopo está fora desse trabalho. É possível não utilizar esse mecanismo, mas a grande maioria dos softwares utilizam porque, dentre vários motivos, os caches do processador têm seu funcionamento atrelado à esse mecanismo. No caso de aplicações comuns, esse mecanismo e os mapeamentos correspondentes são largamente controlados pelo *kernel* do sistema operacional e seu uso é essencial na execução de processos (SILBERSCHATZ; GALVIN; GAGNE, 2018).

Nesse caso, independente do mapeamento específico utilizado, o uso desse mecanismo faz com que os endereços de memória utilizados não necessariamente correspondam à endereços físicos como os endereços do mapa de memória discutido na Seção 2.3. Isso pode ter uma grande influência no uso de mecanismos como MMIO discutidos na Seção 2.2.

2.7 Ambiente de execução ARMv8 típico

O propósito da existência do nível EL3 e da divisão Seguro/Não-seguro é possibilitar a implementação de um *Trusted Execution Environment* (TEE). A ideia é ter uma garantia provida pela arquitetura do processador de forma a isolar em tempo de execução um sistema operacional normal e alguns serviços especiais tidos como seguros. Nesse caso, o software executando no nível EL3 (o *Secure Monitor* na Figura 2.2) faria a ponte entre o software do mundo *Seguro* e *Não seguro*.

A ARM desenvolve o *Trusted Firmware-A* (TF-A) que é um conjunto de softwares com os componentes fundamentais para implementar um TEE[4]. Esse projeto implementa uma série de serviços de software padronizados pela ARM, incluindo a interface (em termos de código *assembly*) para fazer a chamada desses serviços via o software executando em EL3[5].

Um desses serviços é o *Power State Coordination Interface* (PSCI) que, pelo menos até o momento, é necessário para conseguir executar o *kernel* do Fuchsia. O PSCI implementa serviços de gerenciamento dos *cores* e o desligamento (e reset) do processador[6]. Ele é utilizado pelo Fuchsia para fazer o *boot* todos os *cores* do processador.

[4]O código e links para documentação estão disponíveis em <https://github.com/ARM-software/arm-trusted-firmware>

[5]Essa interface (ou ABI) é conhecida como SMCCC, de *SMC Calling Convention*, onde SMC (*Secure Monitor Call*) é uma instrução *assembly* para fazer chamadas ao software executando em EL3.

[6]Mais informações em <https://developer.arm.com/documentation/den0022/latest/>.

O ambiente de execução é carregado e inicializado durante o processo de *boot*, o qual é discutido em mais detalhes na seção a seguir.

2.7.1 Sequência de *boot* em sistemas ARM

Existe um certo padrão na sequência de *boot* de sistemas baseados em processadores ARM. Embora os conceitos não sejam tão diferentes em comparação com outras arquiteturas de processadores, a terminologia associada e, em especial, os softwares utilizados, requerem um certa discussão porque eles determinam o ambiente de execução em que o resto do software executa.

O início do processo de *boot* começa com um sinal de reset físico que faz com que uma exceção do tipo *reset* seja gerada em todos os *cores* do processador. Dentro do modelo de exceções da arquitetura ARMv8, o código do *exception handler* dessa exceção geralmente está no endereço `0xFFFF0000`[7]. Esse endereço aponta para uma memória ROM contida no SoC onde o processador está integrado. Essa memória costuma ser chamada de *Mask ROM*, *Boot ROM* ou até *Trusted ROM*. O software contido nessa memória ROM é conhecido como *firmware* por não poder ser alterado facilmente. Na terminologia ARM, esse *firmware* é chamado de *AP_BL1* (ou simplesmente BL1) — onde BL significa *Boot Loader*. O local onde a Boot ROM está no mapa de memória do RK3328 pode ser visto no canto superior direito da Figura 2.4.

Esse primeiro estágio executa no nível EL3 Seguro e, logo, tem acesso completo ao hardware. A principal função desse estágio é carregar e executar o próximo estágio, BL2, a partir de algum dispositivo de armazenamento. Como nesse momento a memória DRAM (*Dynamic* RAM, i.e, a memória RAM comum) ainda não está inicializada, o BL1 utiliza uma pequena memória SRAM presente no SoC[8].

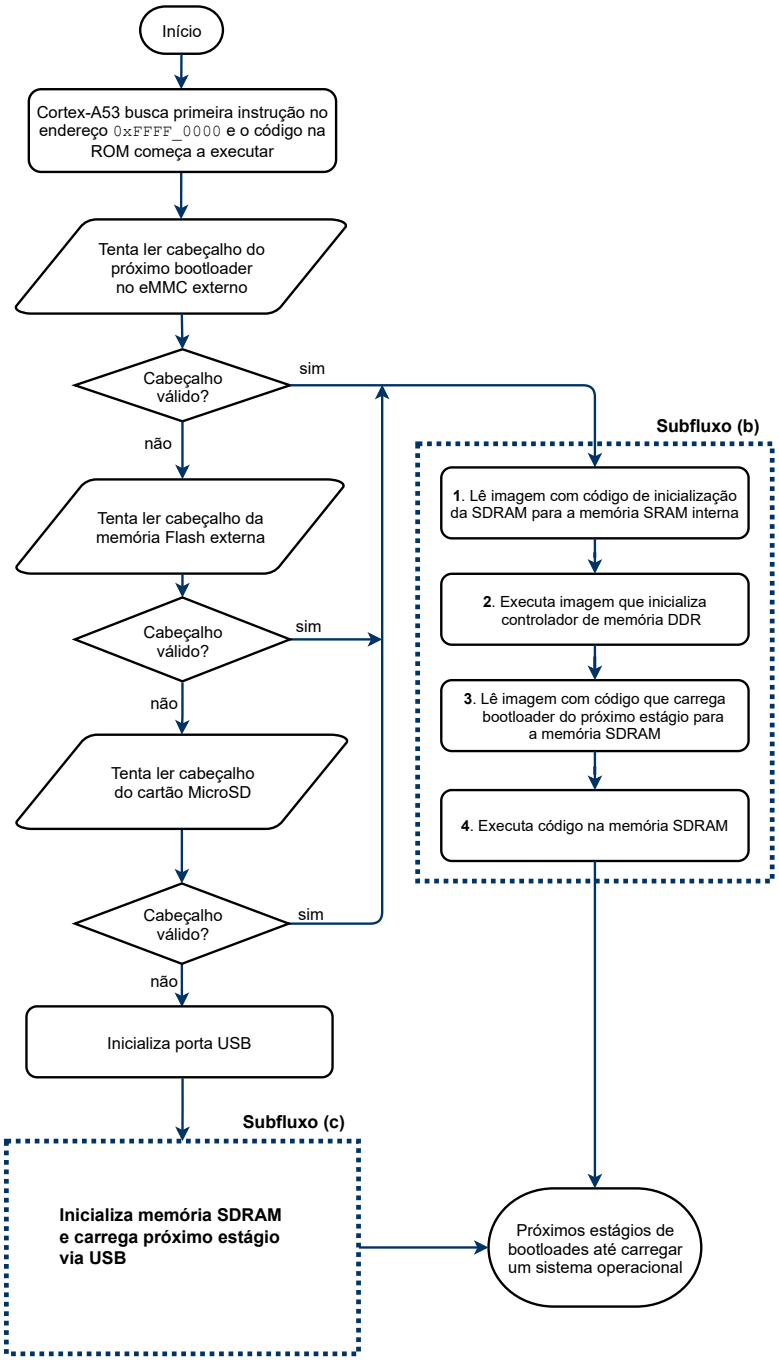
O dispositivo de armazenamento a partir do qual a imagem do BL2 é carregada pelo BL1 depende do fabricante do SoC, geralmente sendo utilizados cartões SD/EMMC ou memórias flash acessíveis por SPI. O diagrama da Figura 2.5 mostra o fluxo de execução que o BL1 do RK3328 segue para carregar a imagem do próximo estágio de *bootloader*.

O segundo estágio, BL2, é responsável por inicializar a memória DRAM e

[7]Esse endereço é definido pelo SoC

[8]A memória SRAM do RK3328 está no endereço `0xFF090000` (Figura 2.4).

Figura 2.5: Fluxo de execução do firmware na BootROM do RK3328.



Fonte: Adaptado da Figura na pág. 14 de (ROCKCHIP, 2017b).

carregar o próximo estágio do *bootloader* nesta memória. É o BL2 que carrega e executa o firmware que dá origem ao ambiente de execução discutido na Seção 2.7.

O terceiro estágio, BL3, costuma ser dividido em 3 sub estágios: BL31, BL32 e BL33. O primeiro corresponde ao firmware do ambiente de execução que implementa serviços como o PSCI acessíveis através da interface SMCCC. O BL32 é responsável por iniciar o *Trusted OS*, que seria um pequeno sistema operacional que

implementaria o TEE (também discutido na Seção 2.7)[9]. Ao contrário do estágio anterior, esses estágios BL31 e BL32 são residentes em memória porque eles contém software que continuam executando mesmo após o processo de *boot*.

O estágio BL33 é o estágio final e representa a versão completa (ou "rica") do *bootloader*. Um exemplo comum é U-Boot, que é um *bootloader* muito utilizado em sistemas embarcados, sendo suportado pela maioria dos SoCs. O U-Boot tem diversos módulos que podem ser usados para gerar os diversos estágios de *bootloaders* discutidos anteriormente.

Os SoCs da Rockchip costumam ter um bom suporte no U-Boot. Com exceção da BootROM, todos os componentes da sequência de *boot* podem ser obtidos a partir do U-Boot que é um software de código fonte aberto. Também é possível obter o firmware de runtime (o BL31) a partir do Trusted Firmware da ARM, que suporta o RK3328. Assim, toda a pilha de softwares necessários para bootar um sistema operacional na Rock64 pode ser gerada a partir de projetos de código fonte aberto. De fato, essa é a solução utilizada nesse trabalho.

A Figura 2.6 mostra a sequência de softwares executados durante o *boot* do RK3328. A fase onde o firmware da BootROM busca o BL2 nos dispositivos de armazenamento externo está esquematizado no diagrama da Figura 2.5.

2.8 Estrutura de dados *Device Tree*

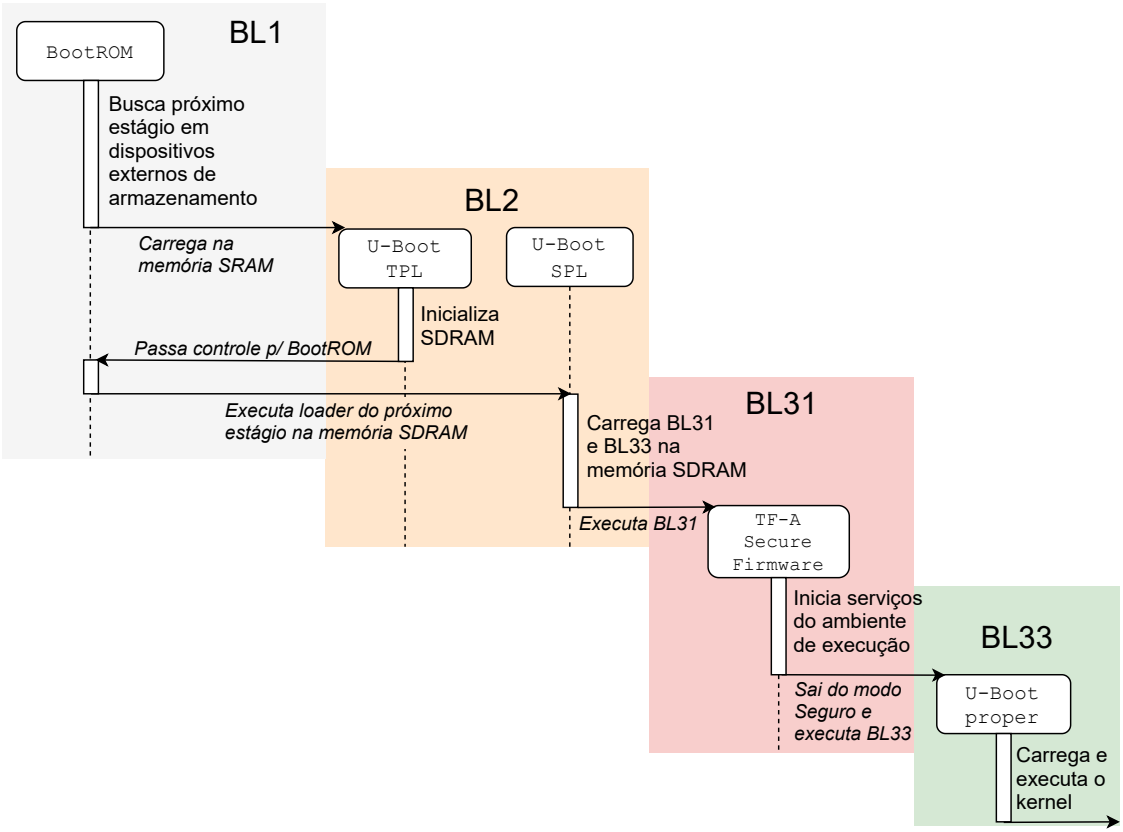
A arquitetura ARM não tem um mecanismo para a descoberta automática de hardware. Não existe uma padronização para o mapa de memória (Seção 2.3) como, por exemplo, localizações fixas para determinados dispositivos. Por causa disso, o software não consegue descobrir essas informações em tempo de execução.

A *Device Tree*, também conhecida como *Flattened Device Tree* (FDT) ou *Device Tree file*, é uma estrutura de dados utilizada para descrever os componentes de hardware disponíveis em uma plataforma e tem o objetivo de resolver esse problema[10]. Além do mapa de memória e da descrição dos dispositivos associados, uma *device tree* pode conter diversas outras informações que costumam ser usadas, por exemplo, como parâmetros para o *kernel* de um sistema operacional.

[9] O Trusted OS é não é utilizado nesse trabalho, logo não há um estágio BL32.

[10] Mais informações disponíveis em https://en.wikipedia.org/wiki/Device_tree e <https://elinux.org/Device_Tree_Reference>

Figura 2.6: Sequência de softwares executados no *boot* do RK3328.



Fonte: O Autor.

Uma *device tree* é definida em arquivos de texto simples com as extensões `.dts`[11] e `.dtsi`[12]. Esses arquivos têm uma sintaxe própria que permite comentários e podem ser montados por composição a partir de diferentes arquivos. Um compilador associado, geralmente chamado de `dtc`[13], transforma esses arquivos para um formato binário com extensão `.dtb`[14]. É esse arquivo binário que é usado pelos softwares discutidos anteriormente.

Também existe suporte à esse formato em *bootloaders* que costumam implementar a funcionalidade de editar a imagem binária de uma *device tree* diretamente na memória. Isso dá a flexibilidade de inserir e substituir parâmetros em tempo de execução. As *device trees* também podem ser usadas pelos próprios *bootloaders* de estágio final de forma a não precisarem conter parâmetros *hardcoded* para os *drivers* que eles utilizam.

[11]Device tree source
[12]Device tree include
[13]Device tree compiler
[14]Device tree blob

2.9 Subsistemas do SoC

Existem vários componentes específicos à um SoC — ou à uma família de SoCs — como controladores relacionados à memória DRAM, pontes entre diferentes barramentos internos e gerenciamento de energia. Devido à integração de diversos componentes em um SoC, e à necessidade de recursos para cada dispositivo de hardware integrado, os SoCs também possuem subsistemas próprios para fazer a configuração e distribuição destes recursos. As subseções a seguir discutem os dois principais subsistemas utilizados nesse trabalho. Esses subsistemas estão ilustrados na camada do SoC no diagrama da Figura 2.1.

2.9.1 Pin-muxing

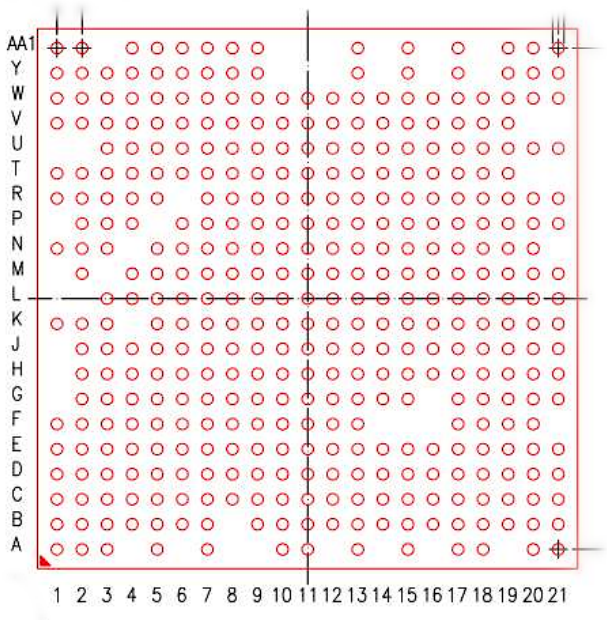
A Figura 2.7 mostra os pinos físicos no *chip* do RK3328. Esses pinos são o único meio pelo qual os componentes integrados no SoC conseguem interagir com o ambiente externo ao SoC. A principal forma de identificar esses pinos é através das suas coordenadas ilustradas na Figura. Por exemplo, o pino mais à esquerda na camada inferior do chip é chamado “A1”.

No caso do RK3328, são 391 pinos. Mesmo parecendo ser um número relativamente grande, essa quantidade não é suficiente para todos os pinos de todos os dispositivos integrados no SoC. Por causa disso, um mesmo pino frequentemente precisa ser compartilhado entre diversos dispositivos o que o que leva à necessidade de um mecanismo de multiplexação.

O termo ‘pino’, nesse contexto, denota dois tipos diferentes de pinos: um deles se refere aos pinos do SoC que estão ilustrados na Figura 2.7. Esses são os pinos utilizados de forma multiplexada entre os diferentes dispositivos integrados no SoC. O outro tipo de pino se refere aos pinos internos ao SoC, que pertencem a determinados dispositivos integrados no SoC. Os dispositivos integrados que precisam interagir com o mundo externo geralmente possuem vários desses pinos, cada um representando um sinal de IO distinto. Para diminuir a confusão, o resto desse texto se refere à estes últimos pinos como ‘sinais’. Os sinais aos quais um pino é conectado também costumam ser chamados de ‘funções’. Assim, em geral, um pino do SoC costuma ter diversas ‘funções’.

O controle desse mecanismo de multiplexação é implementado e exposto na

Figura 2.7: Pinos físicos do SoC RK3328.



Fonte: Adapatado da Fig. 2-3 em (ROCKCHIP, 2017a).

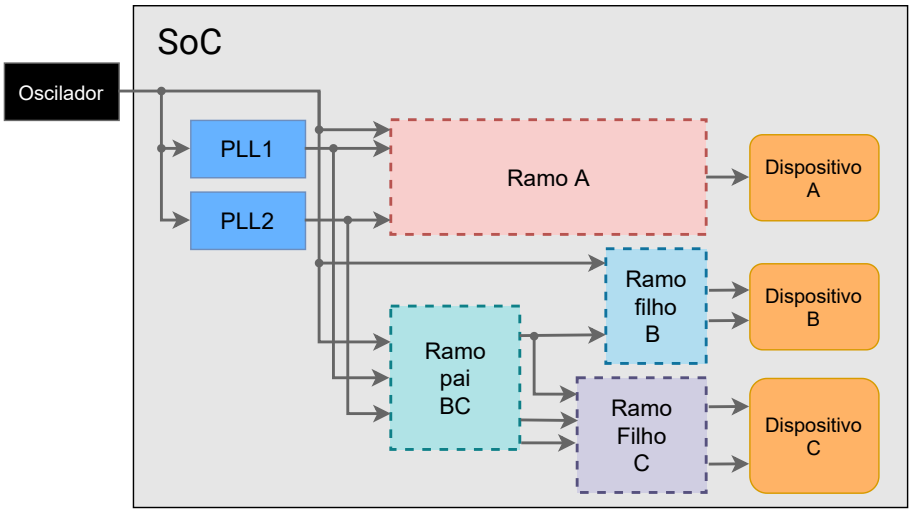
forma de um conjunto de registradores no mapa de memória do SoC. Desse jeito, do ponto de vista do software, o controle desse mecanismo é feito da mesma forma que a interação com outros dispositivos do SoC é realizada: i.e., através de MMIO.

Esse mesmo esquema é utilizado para o controle de outros aspectos dos SoCs. O controle é feito através de registradores espalhados em (geralmente) diversas regiões do mapa de memória. No caso do RK3328, os registradores do controle de multiplexação de pinos estão contidos na região do mapa de memória (Fig. 2.4) denominada GRF (de *General Register Files*), a qual inicia no endereço `0xFF100000`. Todos os registradores dessa região, e também da região CRU mencionada na próxima Seção, têm um tamanho de 32 bits.

2.9.2 Árvore de *clock*

Outra necessidade dos componentes integrados no SoC são os sinais de *clock*. Para funcionar, cada dispositivo necessita de um (ou mais) sinais de *clock* com uma frequência apropriada. Muitas vezes a velocidade de operação desses dispositivos pode ser modulada através da frequência desse sinal de *clock* (dentro de um determinado intervalo). Dependendo do dispositivo e da natureza da função deste, a possibilidade de mudar a velocidade de operação é um requerimento obrigatório

Figura 2.8: Árvore de *clock* simples de exemplo.



Fonte: O Autor.

para ele operar seguindo um padrão associado (p. ex., autonegociação de velocidade poderia fazer parte da especificação do protocolo que o dispositivo implementa).

Em razão disso, um SoC contém diversos circuitos para sintetizar os sinais de *clock* com frequência adequada para cada dispositivo integrado. Esses circuitos costumam ter uma topologia similar à uma árvore devido ao fato da maioria dos sinais de *clock* serem derivados de um único sinal externo ao SoC e em uma sequência de estágios onde os próximos sinais são gerados a partir dos anteriores até, eventualmente, gerar o sinal de *clock* apropriado de cada dispositivo.

A Figura 2.8 contém uma árvore de *clock* simplificada para fins de exemplo. O primeiro nível da árvore é ocupado por *Phase Locked Loop* (PLL)s, que são circuitos cujo objetivo é gerar sinais de *clock* com frequência maior do que o *clock* de referência (o oscilador externo). Esses *clocks* de frequência alta (usualmente na ordem de giga hertz), além de serem necessários para determinados dispositivos, fazem com que os circuitos que derivam os próximos sinais de *clock* sejam relativamente simples (constituídos, basicamente, de divisores e multiplexadores).

Diversos pontos dessa árvore de *clock* podem ser configurados, incluindo os PLLs e especialmente os circuitos de "ramo" que são responsáveis pelo *clock* de dispositivos específicos. E, novamente, essas configurações são feitas através de MMIO em certas regiões do mapa de memória.

No caso do RK3328, a região denominada por CRU (de *Clock and Reset Unit*) tem os registradores que controlam esses parâmetros.

3 SISTEMA OPERACIONAL FUCHSIA

O sistema operacional Fuchsia vêm sendo desenvolvido pelo Google pelo menos desde 2016. Até o presente momento, são poucas as informações oficiais sobre seu propósito e, consequentemente, há muita especulação. O código fonte do Fuchsia é aberto, utilizando licenças permissivas (BSD 2.0, MIT e Apache 2.0) e o desenvolvimento pode ser acompanhado através dos repositórios *git* e sistema de *review* de código associado[1].

O Fuchsia foi desenvolvido praticamente do zero incluindo seu *kernel* Zircon. Inicialmente derivado do LK[2], um *kernel* pequeno e simples utilizado no Android como *bootloader* e base para o Trusty TEE[3], o Zircon se tornou um *kernel* completo a partir do qual o Fuchsia implementa o resto do sistema operacional.

O Zircon é baseado em uma arquitetura de *microkernel* e uma de suas características mais notáveis é que as primitivas que ele implementa para o *userspace*, junto com as chamadas de sistema correspondentes, possibilitam a implementação de um modelo de segurança baseado em *capabilities*[4]. Esse modelo é utilizado extensivamente nas abstrações e interfaces de software (APIs) do Fuchsia.

Outra característica importante é a utilização da *Fuchsia Interface Definition Language* (FIDL), que é uma linguagem de definição de interfaces de software desenvolvida especificamente para o Fuchsia[5]. A utilização da FIDL, primeiramente, se deve ao fato do Zircon ser um *microkernel* — o que requer a utilização de *Interprocess Communication* (IPC) (SILBERSCHATZ; GALVIN; GAGNE, 2018). Isso significa que, basicamente, todos os serviços utilizados por um processo são disponibilizados (e implementados) por outros processos. Os protocolos utilizados na comunicação entre processos são importantes porque eles definem a interface (tanto API quanto ABI) disponível no sistema. Sendo utilizada extensivamente na definição de protocolos IPC, a FIDL é responsável pela maior parte das APIs (e ABIs) do Fuchsia[6]. Até mesmo as chamadas de sistema do Zircon têm uma parte considerável de seu código fonte definidos automaticamente a partir de protocolos FIDL[7].

---

[1] O repositório git principal está disponível em <https://fuchsia.googlesource.com/fuchsia/> e sistema de review de código em <https://fuchsia-review.googlesource.com/>

[2] <https://github.com/littlekernel/lk/wiki/Introduction>

[3] Trusted Execution Environments (TEE) são discutidos na Seção 2.7.

[4] <https://en.wikipedia.org/wiki/Capability-based_security>

[5] https://fuchsia.dev/fuchsia-src/concepts/fidl/overview

[6] https://fuchsia.dev/fuchsia-src/concepts/system/abi/system

[7] https://fuchsia.dev/fuchsia-src/concepts/kernel/life_of_a_syscall

Essas características, por si só, poderiam ser utilizadas de diversas formas para implementar as mais variadas arquiteturas de software em *userspace*. Seria largamente possível[8], por exemplo, ignorar a possibilidade de um modelo de *capabilities* e implementar um ambiente POSIX tradicional. Isto é, ter uma biblioteca em linguagem C definindo a API (e ABI) correspondentes, um sistema de arquivos global, *file descriptors* com as funções canônicas (`open()`, `read()`, `write()`, etc.), um *shell* Unix com *pipelines*, etc[9].

A arquitetura do *userspace* no Fuchsia, porém, é consideravelmente diferente tendo um conceito próprio de componentes como unidade básica de software executável. Esse *framework* de componentes é discutido na Seção 3.6.

3.1 Sistema de *build*

O sistema de *build* do Fuchsia é implementado com as ferramentas GN e Ninja desenvolvidas pelo Google[10]. Essas ferramentas possibilitam a definição de diversas abstrações que são usadas para implementar uma espécie de *framework*. Cada tipo de software no repositório do Fuchsia utiliza um tipo de funcionalidade desse *framework* para especificar seus arquivos de código fonte, dependências, parâmetros para o compilador e outros informações relevantes.

Esse sistema também suporta as diversas linguagens utilizadas no desenvolvimento do Fuchsia, que na sua maior parte são C++ e Rust com alguns poucos remanescentes em C[11].

Porém, o usuário não executa diretamente as ferramentas com as quais o sistema de *build* é desenvolvido. Ao invés disso é utilizado um *frontend* implementado através de um script de *shell* chamado `fx`[12]. Esse script possuí vários subcomandos para tarefas diversas onde nem todas estão relacionadas ao sistema de *build*. Para este trabalho, os comandos relevantes são:

---

[8] Alguns detalhes, como *thread killing*, não são possíveis de implementar sem emulação pois não são suportados pelo *kernel*. Mais informações em https://fuchsia.dev/fuchsia-src/reference/syscalls/task_kill

[9] `Starnix` é uma proposta similar: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0082_starnix

[10] Mais informações sobre o GN e o Ninja podem ser encontradas aqui: https://fuchsia.dev/fuchsia-src/concepts/build_system/intro

[11] Linguagens de programação utilizadas no Fuchsia: https://fuchsia.dev/fuchsia-src/contribute/governance/policy/programming_languages

[12] Informações sobre o `fx` podem ser encontradas aqui https://fuchsia.dev/fuchsia-src/development/build/fx

1. `fx set <product>.<board>`
2. `fx build`

O comando `fx set` recebe os dois principais parâmetros nos quais o sistema de *build* do Fuchsia é baseado.

A `board` define o dispositivo de hardware onde os binários resultantes do *build* vão executar. Consequentemente, essa configuração acaba definindo a arquitetura do processador e também o conjunto de *drivers* básicos que serão incluídos. De forma semelhante à configuração anterior, as *boards* podem ser listadas com o comando `fx list-boards`[13].

O `product` especifica, basicamente, todo o resto. Isso inclui os softwares que vão executar em runtime e também os parâmetros e configurações desse software. Devido ao fato do Fuchsia usar um *microkernel*, o `product` acaba definindo quais APIs vão estar disponíveis em userland pois os softwares disponíveis para execução vão determinar os serviços disponíveis em runtime. Os produtos disponíveis no repositório do Fuchsia podem ser listados pelo comando `fx list-products` [14].

3.2 Processo de *boot*

Antes de começar a executar, um *kernel* precisa (ou supõe) que o hardware esteja em um estado minimamente consistente e conhecido. Assim como um programa comum, um *kernel* também utiliza informações de entrada que configuram diversas parâmetros utilizados em *runtime* e possibilitam o sistema operacional continuar seu processo de inicialização. Essas informações precisam ser passadas ao *kernel* de uma forma apropriada e, junto com outros requerimentos de execução, acabam especificando o protocolo de *boot* do *kernel*.

O principal imagem resultante do processo de *build* do Fuchsia é o arquivo no formato *Zircon Boot Image* (ZBI) que, dentre outras coisas, contém a imagem binária do *kernel*. Esse arquivo é um container para diversos arquivos e itens de dados[15]. Além da imagem do *kernel*, o arquivo mais importante contido no ZBI

[13]Os arquivos GN que definem as *boards* podem ser vistos em <https://cs.opensource.google/fuchsia/fuchsia/+/main:boards/>

[14]Os arquivos GN que definem cada produto podem ser vistos em <https://cs.opensource.google/fuchsia/fuchsia/+/main:products/>

[15]Arquivo de cabeçalho em linguagem C com a especificação do formato ZBI: <https://cs.opensource.google/fuchsia/fuchsia/+/main:zircon/system/public/zircon/boot/image.h>

é o BOOTFS que é a implementação de uma *initramfs* específica do Fuchsia[16]. A maior parte dos conteúdo do BOOTFS é determinado pelo parâmetro `product` do sistema de *build* visto anteriormente. Em geral, o arquivo ZBI encapsula todas as informações que o Zircon, e também o Fuchsia como um todo, necessitam para o *boot*.

O processo de *boot* do Fuchsia consiste em dois passos:

1. Extração da imagem binária do *kernel* contida no arquivo ZBI para um local apropriado na memória RAM.
2. Edição do arquivo ZBI em memória adicionando itens de dados especificando informações relevantes para o *boot*.

O último passo é opcional uma vez que o arquivo ZBI original pode conter toda a informação necessária sobre o hardware para execução do *kernel*. Entretanto, como partes do arquivo ZBI podem ficar residentes na memória durante a execução do sistema, alguns itens (como a imagem binária do *kernel*) são normalmente removidos para poupar espaço.

O protocolo de *boot* do Zircon na arquitetura ARM requer que a MMU, caches e interrupções estejam desabilitados. Todos os parâmetros de entrada que o *kernel* utiliza são encapsulados no arquivo ZBI em memória e o endereço deste é passado ao *kernel* através do registrador `X0` do processador.

O software que faz essas tarefas é o *bootloader*, mais precisamente, o último estágio de *bootloader* como o BL33 visto na Seção 2.7.1. Normalmente, o suporte ao formato das imagens e protocolo de *boot* específico de um *kernel* são suportados diretamente no *bootloader* — especialmente no caso de *kernels* mais comuns como o Linux. O fato do Zircon ser relativamente novo faz com que ele não possua suporte oficial nos *bootloaders* mais comuns. Até existe um suporte no U-Boot para placas utilizadas como plataforma de desenvolvimento pelo Google[17], mas os *patches* estão implementados com muitos trechos de códigos *hardcoded* para estas placas específicas. A utilização deles envolveria aplicar os *patches* no código fonte do U-Boot e a recompilação do mesmo[18]. Assim, este trabalho desenvolve um método próprio para o *boot* usando versão *upstream* (ou oficial) do U-Boot.

[16] https://en.wikipedia.org/wiki/Initial_ramdisk

[17] Até o momento, publicamente, são suportadas as placas vim2 e vim3 da Khadas e a imx8mevk da NXP. Mais informações em https://third-party-mirror.googlesource.com/u-boot/

[18] Ironicamente, a recompilação do U-Boot acabou sendo necessária para bootar o Zircon na Rock64 (ver Seção 4.2.1).

3.2.1 Boot-shim

O *boot-shim* é um pequeno programa *bare metal* disponível no código fonte do Fuchsia que tem o objetivo de implementar o suporte ao *boot* do Zircon. Até o presente momento, ele serve como um método alternativo à implementação direta desse suporte em um *bootloader* como o U-Boot.

Em termos da sequência de estágios no *boot* vistos na Seção 2.7.1, o boot-shim funciona como um estágio adicional de *bootloader*. O boot-shim é simples e, por si só, não interage com nenhum dispositivo de hardware. Ele apenas executa as tarefas descritas na Seção anterior. Através da definição de algumas macros e funções em linguagem C, é possível gerar uma versão do boot-shim que provê as informações sobre o hardware em questão. Isto é, juntando o código genérico do boot-shim com alguns parâmetros específicos do hardware alvo, o sistema de *build* gera uma imagem binária apropriada do boot-shim capaz de fazer o *boot* do *kernel* Zircon.

O boot-shim suporta a passagem de alguns parâmetros através de uma FDT (vista na Seção 2.8). Em especial, o endereço do ZBI em memória é passado através de um parâmetro da FDT.

3.3 *Drivers* do *kernel*

A escolha de onde implementar os *drivers* em um sistema operacional é uma decisão arquitetural. Tecnicamente, um *driver* poderia executar em qualquer ambiente de software, desde que esse ambiente forneça meios para o *driver* interagir com o dispositivo de hardware associado.

Como o Zircon é implementado seguindo o princípio da arquitetura de *microkernel*, a grande maioria dos *drivers* executam *fora* do *kernel* — i.e., em *userspace* como processos normais. Esses *drivers* são discutidos na Seção 3.8.

Porém, há certos *drivers* que executam dentro do *kernel* e que suportam dispositivos tidos como essenciais para o *kernel* implementar funcionalidades básicas. Desses dispositivos, os mais importantes na arquitetura ARM são o *Generic Interrupt Controller* (GIC) e o *timer*. Ambos estão na camada de mais baixo nível no diagrama da Figura 2.1 — isto é, são dispositivos associados à arquitetura do processador. Os parâmetros utilizados por esses *drivers* são passados ao *kernel* no processo de *boot* e são discutidos nas Seção 4.1.

A MMU e caches não possuem um *driver* na forma de um módulo (ou unidade coesa) de software porque estes dispositivos têm um alto acoplamento com o conjunto de instruções da arquitetura ARM. Isto é, são controlados através de registradores e têm instruções próprias para tal. Assim, o suporte é implementado nos diversos pontos do *kernel* onde são necessários na forma de chamadas de procedimentos comuns.

3.4 *Capabilities*

Os *handles* são construções do Zircon que possibilitam o software executando em *userspace* referenciar objetos do *kernel*. A grande maioria das chamadas de sistema utilizam um *handle* como escopo. Isto é, a função que elas implementam se aplica ao objeto do *kernel* para qual o *handle* faz referência.

Do ponto de vista do *userspace*, um *handle* é simplesmente um número inteiro de 32 bits. Uma propriedade essencial dos *handles* é não serem passíveis de falsificação, que é devida à associação entre um *handle* (o número inteiro) e um objeto ser uma informação associada a um processo e gerenciada pelo *kernel*. Assim, se um processo "advinhar" (ou criar) um *handle* válido (i.e., associado a um objeto existente do *kernel*), é devido ao fato desse *handle* já existir na tabela de *handles* do processo.

Além de estar associado à um objeto do *kernel*, um *handle* também é associado à um conjunto de *rights* os quais permitem que certas ações sejam feitas ao *handle* em questão, ou ao objeto apontado pelo *handle*[19]. Por exemplo, existe uma chamada de sistema para duplicar um *handle* (i.e., criar uma cópia dele) de forma que o novo *handle* também aponte para o mesmo objeto que o original. Essa ação só é possível se o *handle* original tiver o right `ZX_RIGHT_DUPLICATE`. Um *handle* criado a partir de outro não consegue ter mais rights do que o *handle* original. Outro right importante é o `ZX_RIGHT_TRANSFER` que permite a um *handle* ser movido para outro processo através de um canal de comunicação IPC apropriado.

Os *handles* e rights, junto com a forma como são tratados pelas chamadas de sistema do Zircon, possibilitam que um modelo de segurança baseado em *capabilities* seja implementado — onde os *handles* são *capabilities*. Nesse modelo, o que um processo tem (as *capabilities*) determinam o que ele pode fazer.

[19] https://fuchsia.dev/fuchsia-src/concepts/kernel/rights

O Fuchsia implementa alguns tipos específicos de *capabilities* a partir dessas primitivas[20]. Porém, no fundo, os componentes fundamentais dessas *capabilities* mais especializadas ainda são as primitivas de *handles*. Os namespaces discutidos na Seção 3.5 são as estruturas que organizam as *capabilities* de um processo.

3.4.1 Exemplo de utilização

Como mencionado no início deste capítulo, uma das consequências do Zircon ser um *microkernel* é que a vasta maioria dos serviços utilizados por um processo são implementados por outros processos. Assim, um processo precisa realizar IPC para utilizar esses serviços. É útil discutir um exemplo um pouco mais detalhado sobre *handles* e *capabilities* e como elas permitem (ou possibilitam) processos acessar serviços de outros processos.

Channels são o mecanismo mais comum para realização de IPC no Zircon (e no resto do Fuchsia)[21]. Um *channel* é criado pela chamada de sistema `zx_channel_create()`[22] a qual, caso execute com sucesso, retorna dois *handles* representando os *endpoints* (ou lados) do *channel* recém criado. Bytes escritos em um *endpoint* podem ser lidos no outro e vice-versa.

Após serem criados, esses *handles* estão no processo que executou a chamada de sistema mencionada. Isso é, os outros processos executando não têm como utilizar nenhum dos *endpoints* desse *channel* para enviar (ou receber) bytes. A partir desse ponto, para oferecer determinado serviço à outro processo, um desses *handles* precisa ser enviado (ou transportado) para o outro processo de alguma forma. Esse envio é feito através de *channels* e, assim, o processo já precisa ter um *handle* com um dos *endpoints* do *channel* em questão.

Assim, supondo que os *handles* estejam em processos diferentes, o *channel* criado poderia ser usado para o outro processo solicitar um determinado serviço do primeiro. Nesse caso — do ponto de vista do processo que solicita o serviço (o processo cliente) — a propriedade de *capability* do *handle* do *endpoint* dele se deve, primeiramente, ao simples fato dele possuir o *handle*. Adicionalmente, ambos os *handles* retornados pela chamada de sistema `zx_channel_create()` não têm o

[20] https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities
[21] https://fuchsia.dev/fuchsia-src/reference/kernel_objects/channel
[22] https://fuchsia.dev/fuchsia-src/reference/syscalls/channel_create

right `ZX_RIGHT_DUPLICATE` e, logo, não podem ser duplicados. Essa propriedade garante que a posse do *handle* é o única maneira de obter o serviço em questão, o que faz o *handle* ser uma *capability*.

3.5 Namespaces

Inicialmente, um processo criado pelo Zircon não tem nenhuma *capability*[23]. Não existe nenhuma autoridade ambiente disponível *a priori*. Após ser criado, um processo pode receber um conjunto de *capabilities*. Essas *capabilities* iniciais são o único meio pelo qual o processo consegue utilizar serviços do sistema. De fato, essas *capabilities* recebidas determinam exatamente quais serviços o processo vai conseguir utilizar.

Os *namespaces* são uma convenção no Fuchsia de organizar essas *capabilities* iniciais na forma de um sistema de arquivo simples[24]. Mais especificamente, um namespace é uma estrutura de dados do tipo tabela que mapeia nomes (*strings*) para objetos. Esses objetos geralmente são *handles* do Zircon, mas também podem ser um sub-*namespace* e isso leva a formação de uma hierarquia que pode ser vista como um mini sistema de arquivos na memória do processo.

Essas *capabilities* iniciais que o processo recebe não são entregues como um *array* de *handles* cru mas, em fato, estão embutidas em um protocolo que inclui uma serialização do namespace para o processo[25].

3.6 *Framework* de componentes

Ao longo de seu desenvolvimento, o Fuchsia criou e refinou uma forma elegante de estruturar o software executando em *userspace* em termos do modelo de segurança baseado em *capabilities* que as primitivas do *kernel* Zircon fazem possível.

O conceito mais importante desse esquema são os componentes devido ao fato deles encapsularem *capabilities*. Um componente no Fuchsia é, basicamente, uma estrutura contendo dois conjuntos de *capabilities*: consumidas e fornecidas. O outro aspecto importante é a existência de um método bem definido de roteamento

[23] https://fuchsia.dev/fuchsia-src/concepts/process/sandboxing
[24] https://fuchsia.dev/fuchsia-src/concepts/process/namespaces
[25] https://fuchsia.dev/fuchsia-src/concepts/booting/program_loading

de *capabilities* entre instâncias de componentes. É através desse mecanismo que os *handles* representando as *capabilities* solicitadas (ou declaradas) são entregues aos componentes[26].

O *framework* de componentes é implementado pelo `component_manager` que é um programa iniciado logo após a execução do primeiro processo em *userspace*. O `component_manager` implementa todas as abstrações básicas do *framework* como os tipos possíveis de *capabilities* e é responsável pelo roteamento das *capabilities* entre os componentes. Esse programa é um dos poucos softwares no Fuchsia que não são componentes[27]. Com exceção de algumas *capabilities* fornecidas pelo *framework* (i.e. pelo próprio `component_manager`), todas as *capabilities* são definidas, implementadas e fornecidas por componentes.

Componentes são normalmente definidos através de arquivos de texto simples com extensão `.cml` chamados de *component manifests source*, ou simplesmente *component manifests*. Esses arquivos são utilizados em uma forma compilada que é gerada através de um compilador apropriado[28].

Para fins de exemplo e também para ilustrar alguns dos outros conceitos desse *framework*, a Listagem 1 mostra a declaração do componente denominado `console`[29], o qual implementa (i.e. cria) uma *capability* do tipo protocolo.

A primeira seção (linhas 2-6) especifica informações sobre a execução do componente e incluí o *runner* que deve ser utilizado para tal. Nem todo componente contém um software que deve ser executado — alguns podem ser definidos apenas para encapsular um conjunto de *capabilities* a serem roteadas para outros componentes. Mas os componentes que contém um programa executável, declaram isso através de um *component runner*. O componente da listagem contém um programa que deve ser executado pelo *ELF runner*[30]. Um component runner é um tipo de *capability* criada através da implementação de um protocolo FIDL correspondente e serve para separar a definição de um componente da forma como ele é executado[31].

---

[26] https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/life_of_a_protocol_open

[27] https://fuchsia.dev/fuchsia-src/concepts/components/v2/introduction

[28] Uma discussão sobre as diferentes formatos associados à declaração de componentes pode ser vista no seguinte RFC: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0093_component_manifest_design_principles

[29] Essa declaração está disponível em <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/bringup/bin/console/meta/console.cml>

[30] https://fuchsia.dev/fuchsia-src/concepts/components/v2/elf_runner

[31] https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/runners

Listagem 1: *Component manifest* do componente `console`.

```
1  {
2   program: {
3     runner: "elf",
4     binary: "bin/console",
5     // [...] parâmetros omitidos por brevidade
6   },
7   capabilities: [
8     { protocol: "fuchsia.hardware.pty.Device" },
9   ],
10  use: [
11    { protocol: "fuchsia.boot.RootResource" },
12    { protocol: "fuchsia.boot.WriteOnlyLog" },
13    { protocol: "fuchsia.logger.Log" },
14  ],
15  expose: [
16    {
17      protocol: "fuchsia.hardware.pty.Device",
18      from: "self",
19    },
20  ],
21 }
```

A seção `use` (linhas 10-14) declara as *capabilities* que o componente utiliza e são instaladas no namespace do processo executando o programa do componente. Nesse exemplo, todas as *capabilities* são protocolos FIDL.

A seção `expose` (linhas 15-20 na Listagem 1) disponibiliza *capabilities* que o componente tem para que sejam usadas pelo componente pai. A *capability* exposta é criada pelo próprio componente e foi declarada entre as linhas 7-9. Como mencionado antes, essa *capability* é do tipo protocolo, o que significa que ela implementa um protocolo FIDL. Nesse caso é o protocolo `fuchsia.hardware.pty.Device`, o qual implementa um dispositivo de terminal emulado comum em sistemas Unix[32].

3.7 *Debuglog* e chamadas de sistema de *debug*

O Zircon utiliza uma porta serial para escrever as mensagens de *log* emitidas durante o seu processo de *boot*. Essa mesma porta serial também é disponibilizada para *userspace* através do *debuglog* e de chamadas de sistema específicas habilitadas

[32]https://fuchsia.dev/reference/fidl/fuchsia.hardware.pty

via parâmetro na linha de comando do *kernel*[33]. Essas chamadas de sistema, quando habilitadas, podem ser usadas por qualquer processo.

O *debuglog* é um objeto do Zircon disponibilizado para softwares que implementam funcionalidades básicas — como os programas iniciais do *boot*, o `component_manager` e o subsistema de *drivers* — de forma que eles consigam emitir mensagens de *log* que possam ser visualizadas através da porta serial. O *debuglog* também funciona como um *backend* para a infraestrutura de *logging*, o que possibilita às mensagens de *log* do resto do Fuchsia serem visualizadas na porta serial.

3.8 *Framework* de *drivers*

De forma similar ao *framework* de componentes discutido na Seção anterior, o Fuchsia também possui um *framework* de *drivers*. Com exceção dos *drivers* mencionados na Seção 3.3, todos os *drivers* no Fuchsia executam em processos normais de *userspace*.

O *framework* de *drivers* é implementado pelo `driver_manager` que é um componente normal do Fuchsia. Isto é, ele é definido através de um *component manifest* o qual declara as *capabilities* utilizadas e fornecidas, etc. Por causa disso, um *driver* no Fuchsia é um conceito definido por este componente e não é uma entidade que existe devido a algum modelo (ou abstração) específica definida pelo *kernel* (à exceção do Zircon ter uma arquitetura de *microkernel*, o que normalmente significa que os *drivers* executam fora do *kernel*).

Apesar do `driver_manager` ser o componente Fuchsia responsável pelo *framework* de *drivers*, ainda há um outro programa que é essencial nesse *framework*: o `driver_host`. Um *driver* no Fuchsia é compilado para um arquivo ELF que sozinho não é executável como a imagem binária de um programa normal. O *driver* é estruturado de tal forma a utilizar um *runtime* que implemente as abstrações que fazem a ponte entre o *driver* e os detalhes de operações associadas como gerenciamento do *driver* e interação com os componentes Fuchsia normais. O `driver_host` é um programa normal (i.e. um binário que não é um componente Fuchsia) que implementa esse runtime.

A forma como o sistema de *drivers* abstrai os dispositivos de hardware e seu suporte é através de três abstrações principais: *device*, *protocolo* e *driver*.

[33] <https://fuchsia.dev/fuchsia-src/gen/boot-options>

De forma similar aos protocolos FIDL, um protocolo é uma interface de software (i.e. API). A maior parte dos protocolos do *framework* de *drivers* são definidos na linguagem Banjo (Seção 3.8.3). Da mesma forma que os protocolos FIDL são usados através de uma arquitetura de cliente-servidor no Fuchsia, os protocolos do *framework* de *drivers* também são usados por servidores e clientes. A diferença é que aqui, ao invés de componentes Fuchsia, os *drivers* são os programas cliente e servidor. Assim, os *drivers* são as entidades que implementam protocolos e também usam protocolos. Um mesmo *driver* pode implementar diversos protocolos e usar diversos protocolos.

Um *device* é uma instanciação de um protocolo e representa a ligação entre uma implementação particular desse protocolo e o meio pelo qual clientes podem usar (ou acessar) o protocolo implementado. Assim, um *device* é similar a uma *capability* no sentido que ele encapsula o acesso à implementação de um protocolo.

É interessante notar que o *framework* de *drivers* está passando por uma migração. Historicamente, o *framework* de *drivers* foi desenvolvido antes (e de forma independente) da versão atual do *framework* de componentes. Neste momento, os *drivers* não são componentes (no sentido do *framework* de componentes). Isto é, eles não necessariamente executam em um container com *capabilities* encapsuladas, especificadas por um *manifest* e que podem ser roteadas para outros pontos da topologia de componentes. Consequentemente, há diversas outras funcionalidades associadas a componentes — como *runners*, *packages*, *resolvers*, facilidade de atualização e *efemeridade*[34] — que não podem ser utilizadas. Entretanto, a arquitetura do *framework* de *drivers* é, no final, muito similar. O que muda é a forma como a implementação e acesso a protocolos é realizado e distribuído. Em suma, nesse momento, *drivers* e componentes existem em ambientes diferentes onde a interação entre eles não é uniforme.

3.8.1 *Driver Development Kit*

A API do *framework* de *drivers* é conhecida como *Driver Development Kit* (DDK) e é definida em linguagem C[35]. Também há uma biblioteca feita em cima dessa API, chamada *DDK Templates Library* (DDKTL), que encapsula as funções

[34]https://fuchsia.dev/fuchsia-src/concepts/principles/updatable
[35]A DDK é definida em <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/lib/ddk/>

e estruturas de dados da versão em C na forma de classes C++ com templates[36]. Essas classes são apenas *wrappers* e não alteram a ABI da DDK.

A DDK define diversas funções básicas relacionadas à publicação de *devices* e à execução de *drivers*. As principais entidades dessa API são *devices*, protocolos e *drivers*. Uma boa parte da API corresponde a um protocolo básico definido em C (e há *wrapper* correspondente em C++) conhecido como *device protocol*. Ele define a interface básica que um *device* deve implementar e tem operações como `open()`, `read()` e `write()` similares a uma API POSIX. Esse protocolo também inclui operações relacionadas ao controle de energia como `suspend()` e `resume()` além de diversas outras operações internas para suporte aos mecanismos do *framework* de *drivers*. Em termos dessa API, um *driver* é simplesmente um programa que implementa um conjunto específico de funções.

Os `driver_host`s contém a implementação de diversas funções da API e também têm uma implementação padrão para as funções não essenciais que um *driver* poderia não querer sobrescrever. Além disso, eles fazem o mapeamento entre as funções do *device protocol* e a implementação fornecida pelo *driver*. É através desse mecanismo que os `driver_host`s conectam os *drivers* ao resto do Fuchsia. Por essa razão, os `driver_host`s são o *runtime* dos *drivers*.

A DDK também incluí diversos outros protocolos que são declarados na linguagem Banjo (e FIDL, ver Seção 3.8.3). Além de protocolos para dispositivos de hardware específicos — como controladores *Ethernet*, cartões SD/EMMC, portas seriais, barramentos e interfaces (I2C, SPI, USB, etc.) — são definidos protocolos que não correspondem diretamente a dispositivos reais; isto é, representam a interface de "alto nível" (ou abstrata) que os dispositivos podem oferecer.

Por exemplo, os detalhes da forma como os dados em um cartão micro SD são acessados é diferente se comparado à um pendrive — o pendrive está conectado através de um barramento USB, o que significa que entre o processador e os dados do pendrive existe um dispositivo de hardware com toda uma complexidade própria (i.e., o barramento USB). Ainda assim, um pendrive é um dispositivo de armazenamento que a partir de um certo nível de abstração opera de forma muito similar ao um cartão micro SD. Nesse caso, o protocolo `fuchsia.hardware.block.Block`[37] define essa interface comum.

[36] https://fuchsia.dev/fuchsia-src/concepts/drivers/driver_development/using-ddktl

[37] <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.block/block.fidl;l=255>

Além de protocolos relacionados à dispositivos de hardware, a DDK também define parte da sua API através de alguns protocolos especiais. Esses protocolos são implementados por um *driver* específico que, por causa disso, é tão essencial ao *framework* de *drivers* quanto o `driver_manager` e os `driver_host`s. Os outros *drivers* normais que tem acesso à esses protocolos (i.e., são clientes) de certa forma também são diferenciados e isso gera uma certa classificação de *drivers*. Esse assunto será discutido na Seção 3.8.5.

3.8.2 *Driver binding*

Um *driver* é carregado em resposta à publicação de um *device*. Um *device* contém um conjunto de atributos que são utilizados pelo `driver_manager` para tentar localizar um *driver* apropriado. O atributo mais são os `proto_id`s que são identificadores do protocolos que o *device* representa (além do *device protocol* presente em todo *device*). Esses identificadores são macros C definidas pela DDK e têm o formato `ZX_PROTOCOL_`*`NAME`*[38]. Esses identificadores correspondem implicitamente à protocolos Banjo. Por exemplo, o identificador `ZX_PROTOCOL_BLOCK` corresponde ao protocolo `fuchsia.hardware.block.Block` mencionado na seção anterior.

Um *driver* que publica um *device* especificando um protocolo deve fornecer uma implementação desse protocolo, pois esse *driver* vai ser o programa no lado "servidor" da interface que esse protocolo especifica.

Além do `proto_id`, um *device* normalmente contém outros atributos como *Vendor ID* (VID), *Product ID* (PID) e *Device ID* (DID). Esses atributos são usados para *devices* que correspondem de forma mais próxima a dispositivos de hardware reais e possibilitam que um *driver* específico para o dispositivo seja utilizado (ver Seção 3.8.6).

Para que o `driver_manager` consiga localizar o *driver* apropriado para o *device* publicado, os *drivers* precisam declarar de alguma forma os atributos do *device* que eles desejam utilizar. Assim, o `driver_manager` vai poder comparar os atributos do *device* publicado com os atributos especificados por um dado *driver* e decidir se há uma correspondência. Se houver, o *driver* é carregado em um `driver_host` e associado ao *device* de tal forma que o *driver* vira um cliente dos protocolos

[38] A lista de identificadores é declarada aqui: <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/lib/ddk/include/lib/ddk/protodefs.h>

especificados no *device*. Esse *device* é conhecido como *device* pai do *driver*. O processo de localizar, carregar e associar um *driver* a um *device* é conhecido como *binding*.

Um *driver* especifica os atributos dos *devices* que ele pode (ou "quer") fazer *binding* através de uma pequena *Domain Specific Language* (DSL) própria do *framework* de *drivers* do Fuchsia[39]. Essa linguagem é composta de uma série de regras que compararam os atributos declarados pelo *driver* com os atributos provenientes do *device* em questão. Essas regras são compiladas por um compilador associado e formam o *binding program* contido na imagem binária do *driver*. É através desse *binding program* que o `driver_manager` decide se houve uma correspondência. Os arquivos com o código fonte do *binding program* usam a extensão `.bind`.

3.8.3 Protocolos Banjo (e FIDL)

Os protocolos da DDK, com exceção do *device protocol*, eram originalmente definidos na linguagem Banjo, que é uma IDL derivada da FIDL. A principal diferença entre elas é que os protocolos Banjo tem o objetivo de realizar comunicação dentro de um mesmo processo sem utilizar passagem de mensagens. Por causa disso, e para obter uma ABI estável, o código correspondente gerado pelo compilador Banjo são basicamente ponteiros de funções em linguagem C com um nível de indireção. O compilador também suporta C++ para o qual são geradas classes *wrapper* (que encapsulam) os ponteiros da versão em C de tal forma que a ABI ainda seja mantida. Em termos de estrutura de código fonte, as classes C++ geradas pelo compilador são muito semelhantes ao código com a definição dos protocolos Banjo.

Os protocolos da DDK não são exatamente a mesma coisa que as *capabilities* do tipo protocolo do *framework* de componentes do Fuchsia (Seção 3.6). Essa diferença é devida à forma como a versão atual do *framework* de *drivers* implementa IPC em comparação ao *framework* de componentes e é uma das razões da migração mencionada anteriormente.

Como parte dessa migração, o Banjo está sendo substituído por protocolos definidos na linguagem FIDL[40], a qual está sendo alterada para suportar os protocolos Banjo através de novos valores para alguns atributos da linguagem[41]. Isto

[39] https://fuchsia.dev/fuchsia-src/concepts/drivers/device_driver_model/driver-binding
[40] https://fuchsia.dev/fuchsia-src/contribute/roadmap/2021/stable_driver_runtime
[41] Review do *patch* para suporte na linguagem FIDL: <https://fuchsia-review.googlesource.com/

significa que os protocolos definidos na linguagem Banjo foram transformados em protocolos FIDL com atributos especiais, e o compilador associado (`banjoc`) foi incluído no compilador FIDL de forma que os códigos C/C++ gerados ainda sejam os mesmos.

3.8.4 Recursos para interface de hardware

Os mecanismos que os *drivers* do Fuchsia usam para interfacear com os dispositivos de hardware são implementados através de algumas primitivas do *kernel* Zircon desenvolvidas especificamente para esse fim. Essas primitivas são objetos do *kernel* e chamadas de sistema correspondentes que lidam com mecanismos de acesso ao hardware como interrupções e MMIO.

A principal forma com que os *drivers* desenvolvidos neste trabalho interagem com o hardware é através de MMIO. Os objetos do *kernel* que a DDK usa para possibilitar o uso de MMIO são VMOs[42] e VMARs[43] que são as primitivas normais do Zircon para lidar com memória. A partir delas, a DDK usa uma chamada de sistema especial — `zx_vmo_create_physical()`[44] — que permite alocar uma região de memória no espaço de endereçamento físico ao invés do espaço de endereçamento virtual onde os processos executam. O acesso à essa chamada de sistema é controlado através de um tipo específico de *handle*, o que possibilita que MMIO possa ser usado como um recurso através de uma *capability*. As interrupções de hardware também são suportadas de uma forma similar e tem recursos correspondentes que funcionam como *capabilities*.

A DDK define algumas camadas de abstração em cima dessas primitivas na forma de classes C++ e protocolos Banjo específicos. Esses últimos são utilizados de uma maneira similar ao modelo de *capabilities* do *framework* de componentes, no sentido de que eles possibilitam que *capabilities* sejam transferidas entre um *driver* do *framework* e alguns tipos de *drivers*.

---

c/fuchsia/+/503781>

[42] https://fuchsia.dev/fuchsia-src/reference/kernel_objects/vm_object

[43] https://fuchsia.dev/fuchsia-src/reference/kernel_objects/vm_address_region

[44] https://fuchsia.dev/fuchsia-src/reference/syscalls/vmo_create_physical

3.8.5 *Drivers* de plataforma e o `platform-bus`

O *driver* `platform-bus` é um *driver* especial do *framework* e não corresponde a nenhum dispositivo real. Ele é o *driver* por meio do qual o `driver_manager` realiza o *boot* do sistema de *drivers*. Isto é, o `platform-bus` não faz *binding* com nenhum *device* (pois não há nenhum) e é instanciado em um `driver_host` de maneira artificial (i.e., *hardcoded*).

Uma vez carregado, o `platform-bus` inicia a árvore de dispositivos através da publicação de um *device* com protocolo `ZX_PROTOCOL_PBUS` (que corresponde ao protocolo Banjo `fuchsia.hardware.platform.bus.PBus`[45]). Isso significa que o *driver* `platform-bus` é o servidor desse protocolo. O `ZX_PROTOCOL_PBUS`, ou simplesmente PBUS, é um protocolo especial do *framework* usado para dois objetivos inter-relacionados:

1. Criar os 'recursos de interface de hardware' (Seção 3.8.4).
2. Publicar os *device* com um protocolo associado que permite o acesso aos recursos criados no item anterior.

São os *device* criados no item 2 que acabam iniciando a árvore de *device* e *drivers* que, no final, representam (e implementam) o suporte aos dispositivos de hardware no Fuchsia.

O `platform-bus` recebe do `driver_manager` os *handles* especiais do Zircon que permitem à ele usar as chamadas de sistema mencionadas na Seção 3.8.4. Porém, o `platform-bus` nunca passa cópias desses *handles* para outros *drivers*. Ele apenas passa os recursos de interface de hardware criados a partir desses *handles*.

O cliente do protocolo PBUS — isto é, o *driver* que faz *binding* ao *device* contendo esse protocolo — é denominado *board driver* e tem a função de publicar os *device* correspondentes aos dispositivos de hardware suportados na plataforma. A plataforma denota o conjunto de dispositivos de hardware disponíveis e, neste trabalho correspondem aos dispositivos integrados no SoC RK3328 da Rock64 (camadas superior e do meio na Figura 2.1). O *board driver* tem embutido em si diversos tipos de informações que descrevem a plataforma como regiões de MMIO, interrupções de hardware, sinais de *clock*, pinos físicos utilizados do SoC, identificação dos dispositivos de hardware (VID, PID, DID), etc. Esse é o mesmo tipo de informação

[45] <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.platform.bus/platform-bus.fidl>

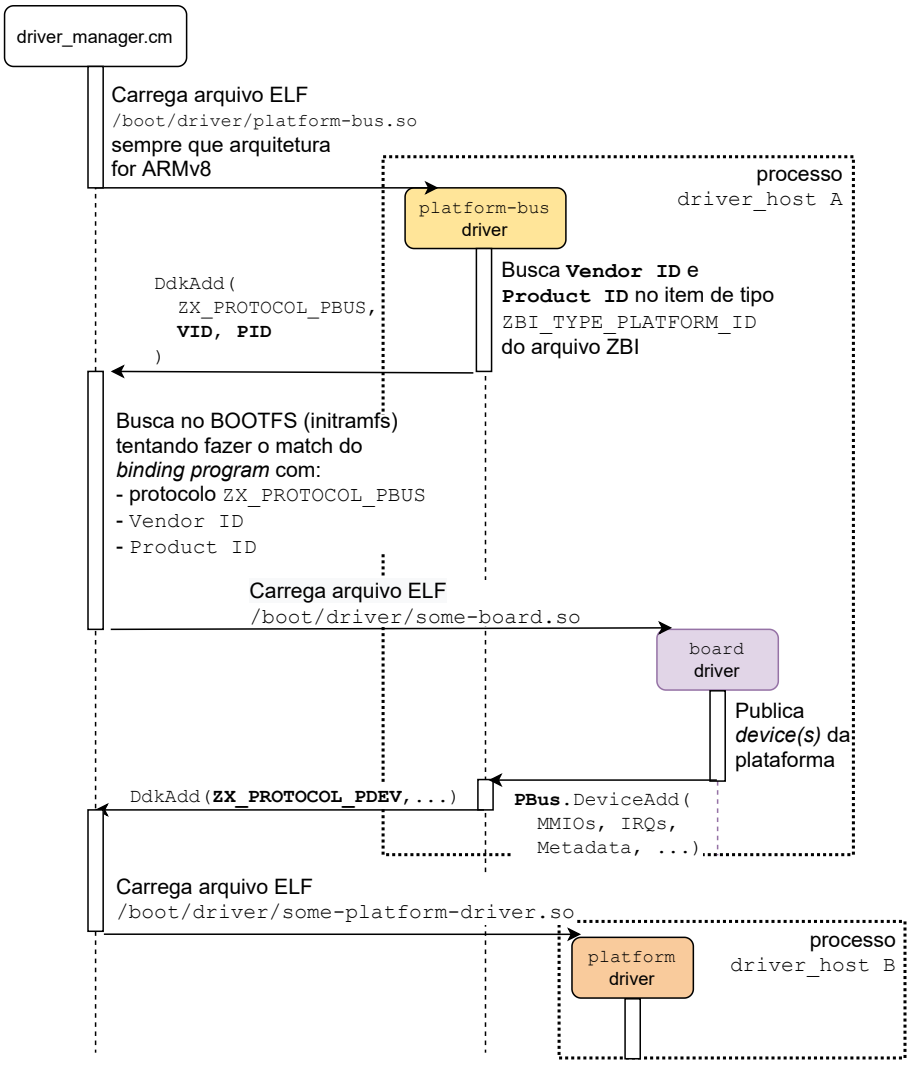
que as *Device Trees*, discutidas na Seção 2.8, codificam. Até o momento, porém, o *framework* de *drivers* não tem suporte à essa estrutura de dados e o *board driver* possui elas de forma *hardcoded*.

O *board driver* usa as operações do protocolo PBUS apenas para definir (ou declarar) os *devices* e seus recursos de interface de hardware associados. É o *driver* `platform-bus` quem de fato publica os *devices* contendo os recursos criados via API C/C++ da DDK. Esses *device* são associados ao protocolo `ZX_PROTOCOL_PDEV`[46], que também é implementado pelo `platform-bus`. O `ZX_PROTOCOL_PDEV`, ou simplesmente PDEV contém as operações que permitem aos *drivers* que fizerem *binding* à esses *device* receberem os recursos de interface de hardware que o `platform-bus` criou a partir das especificações passadas pelo *board driver*. Esses últimos *drivers* são denominados *drivers* de plataforma e são os únicos tipos de *drivers* que têm acesso direto aos recursos de interface de hardware. Os outros *drivers* acessam o hardware indiretamente através deles.

O diagrama de sequência na Figura 3.1 ilustra os eventos importantes desde o *binding* do `platform-bus` até o *binding* de um *driver* de plataforma. O `platform-bus` publica um *device* por meio da API C++ da DDK (o método `DdkAdd()`). Além de estar associado ao protocolo PBUS, esse *device* contém os atributos *Vendor ID* e *Product ID* que o `platform-bus` recupera de um item de dados do arquivo ZBI. O *binding program* do *board driver* procura fazer uma correspondência positiva contra esses atributos. Assim, ele consegue declarar o produto de hardware específico suportado de forma a não ser carregado para algum outro produto qualquer. O `driver_manager` procura o *driver* e, caso encontre, instância-o no mesmo processo `driver_host` do *driver* `platform-bus`. O *device* pai do *board driver* é o *device* publicado pelo `platform-bus`, logo, o *board driver* consegue usar o protocolo PBUS como cliente. A partir daí, o *board driver* utiliza as operações (também chamados de métodos) do PBUS para solicitar ao `platform-bus` a publicação dos diversos *devices* e seus recursos de hardware (e outras informações) associadas. O `platform-bus`, no lado servidor do protocolo PBUS, cria os recursos de interface de hardware especificados e — junto com outras informações passadas pelo *board driver* — os coloca em um *device* publicado através da API C/C++ da DDK. Esse *device* está associado ao protocolo PDEV, também servido pelo `platform-bus`, que então inicia novamente

[46] Esse identificador corresponde ao protocolo Banjo `fuchsia.hardware.platform.device.PDev` definido em <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.platform.device/platform-device.fidl>

Figura 3.1: Sequência de eventos até o *binding* de um *driver* de plataforma.



Fonte: O Autor.

o processo de *binding*. Caso o `driver_manager` localize um *driver* adequado, este é carregado. Dessa vez, dependendo do método específico que o *board driver* usou, esse *driver* é instanciado em um processo `driver_host` próprio.

Existe ainda um outro tipo de *driver* conhecido como 'protocol implementation driver' que implementa protocolos úteis relacionados à configuração de recursos da plataforma. Elestambém são *drivers* de plataforma pois o *device* ao qual fazem *binding* implementa o protocolo PDEV. A diferença é que esse *device* pai também está associado ao protocolo PBUS e, por causa disso, o *driver* é capaz de fazer um registro do protocolo útil que ele implementa. Isso possibilita ao *board driver* acesso como cliente o qual ele usa para fazer configurações específicas dos recursos controlados por esses *drivers*. Esse tipo de *driver* será discutido em mais detalhes nas Seções 5.2.2 e 5.3.2.

3.8.6 *Drivers* de implementação e *drivers* genéricos

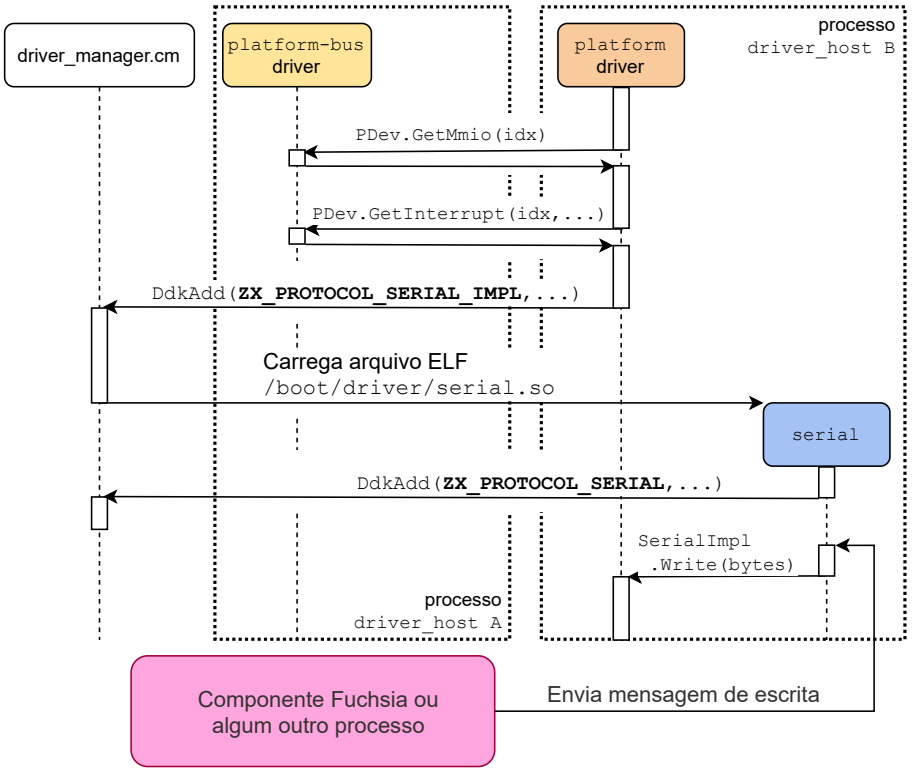
Os *drivers* do Fuchsia que implementam suporte aos dispositivos de hardware reais costumam dividir tal implementação em duas partes: uma específica ao dispositivo em questão e outra genérica. Ambas são implementadas por *drivers*, mas não pelo mesmo *driver*. O *driver* genérico implementa funcionalidades que seriam comuns a qualquer *driver* controlando o mesmo tipo de dispositivo.

Por exemplo, o Capítulo 6 implementa um *driver* para uma porta serial. A funcionalidade de serializar diversas operações concorrentes de leitura e escrita em buffers apropriados é feita por um *driver* serial genérico já existente. O *driver* de implementação apenas faz operações de leitura e escrita, sem lidar esse tipo de funcionalidade.

Além do `driver_manager` e dos `driver_hosts`, os *drivers* genéricos são os últimos programas que normalmente fazem a interface entre o mundo dos *drivers* (que se comunicam através de protocolos Banjo) e o resto do Fuchsia, que se comunica através de protocolos FIDL.

O diagrama de sequência da Figura 3.2 ilustra os eventos importantes desde o *binding* do *driver* de implementação até os eventos relacionados à uma operação realizada por um programa externo ao sistema de *drivers*. O *driver* de implementação é, normalmente, um *driver* de plataforma e, assim, consegue recuperar os recursos de interface de hardware necessários através do protocolo PDEV. Após fazer isso,

Figura 3.2: Sequência de eventos até o *binding* de um *driver* genérico.



Fonte: O Autor.

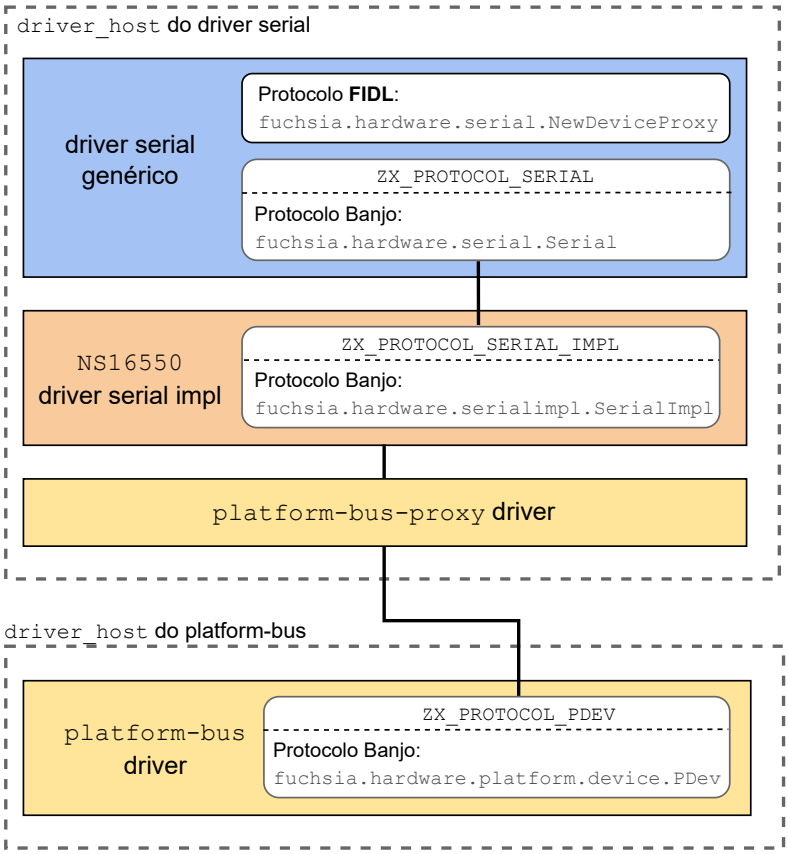
ele publica um *device* associado ao protocolo que ele implementa — nesse caso `ZX_PROTOCOL_SERIAL_IMPL`[47]. Dessa forma, os *drivers* que fizerem *binding* nesse *device* vão conseguir usar esse protocolo como clientes. O *driver* que faz *binding* nesse "protocolo de implementação" é o *driver* genérico. O *binding program* dele não especifica nenhum atributo do tipo *Vendor ID*, *Product ID*, etc., pois esses detalhes específicos ao dispositivo de hardware são suportados pelo *driver* de implementação. Após inicializar, o *driver* genérico publica um *device* que implementa o protocolo dele, que neste caso é o `ZX_PROTOCOL_SERIAL`[48]. Através desse *device*, outros *drivers* podem usar a porta serial. O *driver* genérico também implementa o protocolo FIDL `fuchsia.hardware.serial.NewDeviceProxy`[49] que é mecanismo pelo qual o resto do Fuchsia consegue acessar a porta serial.

[47]Esse identificador corresponde ao protocolo Banjo `fuchsia.hardware.serialimpl.SerialImpl` definido em <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.serialimpl/serial-impl.fidl>

[48]Esse identificador corresponde ao protocolo Banjo `fuchsia.hardware.serial.NewDeviceProxy` definido em <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.serial/serial.fidl>

[49]Nesse momento, esse protocolo FIDL é diferente do protocolo "Banjo" correspondente. A versão FIDL implementada pelo *driver* genérico é definida em <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/fidl/fuchsia.hardware.serial/serial.fidl>

Figura 3.3: *drivers* de implementação e genérico e seus `driver_host`s.



Fonte: O Autor.

A Figura 3.3 ilustra a relação entre esses *drivers* e os protocolos que usam e implementam. Ela também ilustra os processos `driver_host` onde é possível ver o uso de um driver interno do *framework* que é instanciado no `driver_host` do *driver* de plataforma para que esse consiga utilizar o protocolo PDEV. O *driver* `platform-bus-proxy` é necessário devido ao fato dos protocolos Banjo poderem ser usados apenas para comunicação no mesmo processo.

3.8.7 *Composite Devices*

O *framework* de *drivers* disponibiliza uma abstração para organizar os *drivers* em uma estrutura que modela de forma mais próxima a dependência de recursos que um dado dispositivo requer, onde esses recursos são fornecidos por outros dispositivos também controlados por *drivers* do *framework*.

Isso é feito por meio de um *composite device* implementado pelo *framework* o qual está associado ao protocolo `ZX_PROTOCOL_COMPOSITE`. A função desse *device*

é agregar outros *devices*, os quais são chamados de fragmentos. Os *composite devices* são normalmente criados através do método `CompositeDeviceAdd(...)` no protocolo PBUS e isso quer dizer que o *board driver* é o *driver* que indiretamente cria esses *device*. Para fazer isso, o *board driver* precisa orquestrar e primeiro criar os sub-*device* que os *composite devices* vão utilizar.

O *driver* `platform-bus` também implementa o protocolo `ZX_PROTOCOL_PDEV` no *composite device* solicitado pelo *board driver*. Assim, o *driver* que faz *binding* no *composite device* também consegue obter os recursos de interface de hardware especificados pelo *board driver*.

O *driver* que faz *binding* no *composite device* consegue obter os *devices* fragmento contidos nele através de uma operação apropriada que o *composite device* implementa. Por meio disso, um *driver* no Fuchsia consegue abstrair os recursos que seu dispositivo utiliza delegando seu controle para outros *drivers* através desses fragmentos.

3.8.8 Interface entre *drivers* e componentes

A maioria dos *device* publicados pelos *drivers* são organizados em uma árvore de dispositivos pelo `driver_manager`, o qual monta um *namespace* na forma de diretório com a estrutura dessa árvore. Esse *namespace* é conhecido por *Device Filesystem* ou *DevFS*. O `driver_manager`, por meio de seu *component manifest*, exporta esse diretório que, a partir daí, pode ser usado por outros componentes Fuchsia através do mecanismo de *capability routing*.

A maior parte dos itens nesse diretório são *handles* que apontam para os *device* publicados pelos *drivers*. O `driver_manager` junto com os processos `driver_host` implementam os protocolos de *file IO*[50] do Fuchsia nesses *handles* e mapeiam mensagens como `Open()`, `Read()` e `Write()` para as operações correspondentes do *device protocol* associado aos *device*. Outras mensagens são encaminhados para a operação `message()` do *device protocol* que é usado pelos *drivers* para implementarem outros protocolos FIDL relacionados à sua operação.

[50] <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/fidl/fuchsia.io/io.fidl>

4 PORTE I: *BOOT* DO *KERNEL*

O processo de porte de um sistema operacional, em geral, consiste em fazer com que este suporte os itens nas camadas da Figura 2.1. Esse suporte precisa começar da camada de mais "baixo nível", que representa a arquitetura do processador. Como visto no início deste trabalho, esse suporte é tão fundamental que, em geral, acaba sendo implementado no *kernel*.

Pelo fato de o *kernel* do Fuchsia já suportar a arquitetura ARMv8[1] — e também pela placa utilizada neste trabalho ser baseada em um SoC com processador Cortex-A53, o qual implementa a arquitetura ARMv8 — o suporte básico da arquitetura já deve, teoricamente, funcionar.

O objetivo deste capítulo é descrever o processo pelo qual uma imagem apropriada do *kernel* Zircon é gerada de forma a ser executada na Rock64. Esse processo consiste em duas etapas:

1. Gerar uma imagem adequada do Zircon a partir do sistema de *build* do Fuchsia.
2. Carregar e executar essa imagem na memória RAM da Rock64.

Na maior parte, as tarefas em cada etapa são independentes entre si. Porém, como será visto nas próximas seções, há certos parâmetros utilizados na primeira etapa que precisam ser usados na segunda etapa.

4.1 Criação da `board` no sistema de *build*

Como visto na Seção 3.1, a interface do sistema de *build* do Fuchsia se resume, basicamente, à execução dos seguintes comandos:

- `scripts/fx set <product>.<board>`
- `scripts/fx build`

O *product* é uma configuração que especifica o conjunto de softwares incluídos na imagem final gerada pelo *build*. Nesse momento, seria ideal ter a configuração com o menor número de softwares possíveis já que objetivo é conseguir executar o Fuchsia. Nesse caso, o valor mais adequado — de fato, definido exatamente para

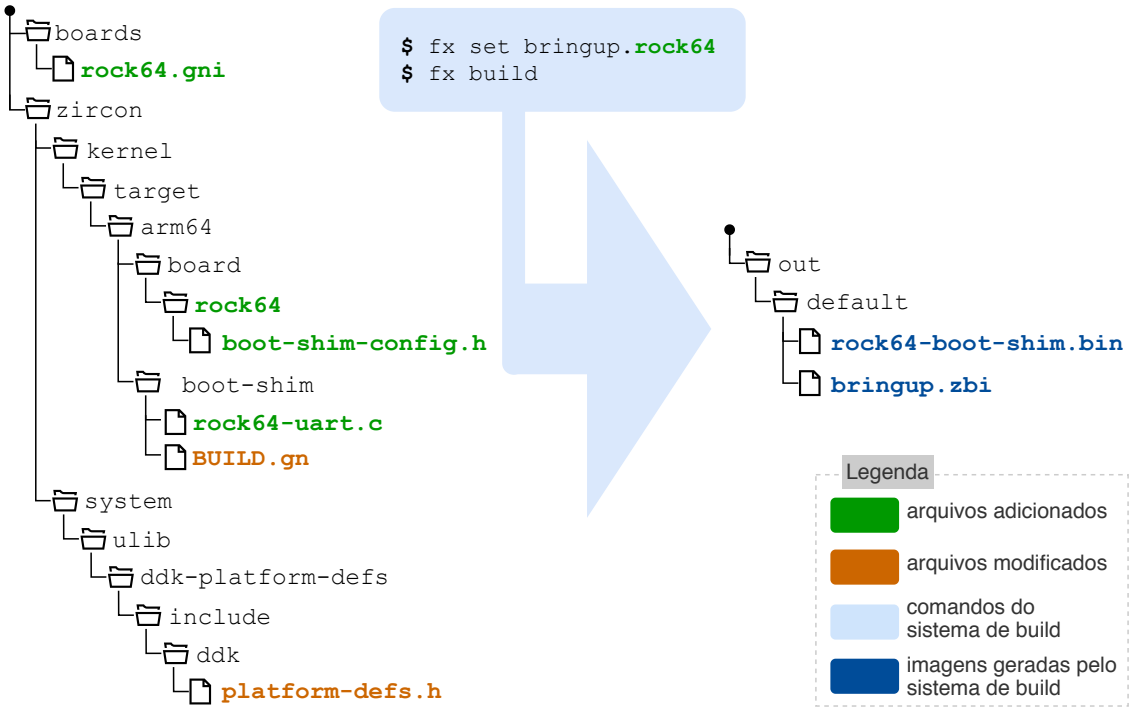
[1] As arquiteturas suportadas estão disponíveis em https://fuchsia.dev/fuchsia-src/concepts/architecture/architecture_support

esse fim[2] — é o `bringup`, pois ele incluí o mínimo necessário para realizar o porte.

A *board*, ou *placa*, define a plataforma de hardware final onde o produto vai executar. Embora tecnicamente fosse possível utilizar alguma placa previamente definida — como a '`qemu-arm64`' — seria improvável que o Fuchsia (ou mesmo o *kernel*) conseguisse executar, pois os detalhes do hardware da Rock64 são diferentes da máquina emulada pelo QEMU. O mais adequado é definir uma placa própria para a Rock64.

O diagrama na Figura 4.1 ilustra os arquivos que foram criados (ou alterados) para adicionar a placa `rock64` no sistema de *build*. O *commit git* desta adição está no Apêndice A.2. O resto dessa seção discute as informações mais relevantes e pontos interessantes dessa adição.

Figura 4.1: Arquivos relacionados à adição da Rock64 ao sistema de *build*.



Fonte: O Autor.

No repositório com o código fonte do Fuchsia, as placas suportadas estão no diretório `boards`[3]. Cada placa é definida por um arquivo GN com a extensão `gni`. O arquivo criado neste trabalho para representar a Rock64 é o `rock64.gni`. As principais linhas desse arquivo são as seguintes:

[2]https://fuchsia.dev/fuchsia-src/concepts/build_system/bringup
[3]Pode ser visualizado em <https://cs.opensource.google/fuchsia/fuchsia/+/main:boards/>

```
import("//boards/arm64.gni")
board_name = "rock64"
board_package_labels += [ "//zircon/kernel/target/arm64/boot-shim:rock64"
    ↪ ]
```

A linha com a variável `board_package_labels` define a lista de dependências. Nesse momento, a única dependência necessária é o boot-shim o qual, como visto na Seção 3.2.1, é o programa que faz a ponte entre um *bootloader* propriamente dito (como o U-Boot) e o *kernel* do Fuchsia. Além do boot-shim, existem outras dependências implícitas que são incluídas devido ao produto '`bringup`' e também pelo arquivo `arm64.gni`. A variável `board_name` define o nome da placa que será usado no comando `fx set`.

O *kernel* Zircon não inclui nenhum parâmetro relacionado à Rock64 (ou o RK3328) durante o *build*. Esses parâmetros são passadas a ele via itens do arquivo ZBI que são adicionados pelo boot-shim em tempo de execução durante o processo de *boot*. A maior parte desses parâmetros são definidos através de uma função C definida fora do código principal do boot-shim, a qual tem o seguinte protótipo:

```
void append_board_boot_item(zbi_header_t* bootdata);
```

A struct `zbi_header_t` representa o arquivo ZBI em memória. A implementação dessa função, e a declaração dos parâmetros que ela inclui no ZBI, estão no arquivo `boot-shim-config.h` ilustrado na Figura 4.1.

Os parâmetros definidos no `boot-shim-config.h` são:

- Número de *clusters* no processador e quantidade de *cores* em cada *cluster*. No caso do RK3328 há apenas um processador Cortex-A53 com todos os 4 *cores* em um único *cluster*.
- Mapa de memória básico, onde o espaço de endereços físicos do processador é particionado entre regiões que endereçam memória RAM e regiões que endereçam espaços de MMIO dos dispositivos integrados ao SoC. No caso do RK3328, o diagrama da Figura 2.4 mostra que a memória SDRAM está alocada entre os endereços `0x00000000` e `0xFF000000`[4] (não inclusive). E as regiões de MMIO estão definidas entre `0xFF000000` e `0xFFFF0000` (não inclusive).
- Endereço físico do MMIO da UART2, que é a porta serial do RK3328 utilizada

[4] A versão da Rock64 utilizada nesse trabalho é a `ROCK64_V2 201700713` a qual possuí 4GiB de RAM.

como console de *debug* e conectada aos pinos físicos de GPIO da Rock64. O *kernel* necessitada disso para implementar o *debuglog* e as syscalls de *debug*[5].

- Endereço físico de MMIO do GIC-400[6].
- *Flag* para que o *kernel* utilize a instrução `smc` para fazer chamadas ao serviço de PSCI via Secure Monitor executando em EL3.
- Número das interrupções de hardware do *timer* padrão da arquitetura ARMv8.
- *Vendor ID* (`VID`) e *Product ID* (`PID`) da plataforma. A função desses números é fazer com que o *driver* correto (nesse caso, da Rock64) seja carregado quando o subsistema de *drivers* do Fuchsia for iniciado (Seção 3.8.5).

O arquivo `rock64-uart.c`, apesar de não ser essencial, é útil porque define um *driver* simples que escreve *strings* para a porta serial. Esse *driver* têm o endereço em memória da porta serial padrão da Rock64 (a UART2 do RK3328 no endereço `0xFF130000`) *hardcoded* em seu código. Com isso, é possível ver que o código do boot-shim realmente está executando e também fazer *debugging* do código através de `printf()`s.

A imagem final do boot-shim gerada pelo sistema de *build* está no arquivo `rock64-boot-shim.bin`. O arquivo ZBI contendo o *kernel* é o `bringup.zbi`. É importante observar que esse arquivo ZBI é editado em memória e a versão passada ao *kernel* é ligeiramente diferente.

O *commit git* com as mudanças representadas na Figura 4.1 está disponível no Apêndice A.2. Com esse *commit*, o comando `fx list-boards` deve retornar a `rock64` na lista de placas suportadas.

4.2 Carregar e executar o *kernel*

A execução do *kernel* de um sistema operacional é, basicamente, a execução de um programa *bare metal*. No caso do Zircon, como visto na Seção 3.2, há uma certa complexidade nesse processo devido à fatores como:

1. a imagem binária do *kernel* estar empacotada no arquivo ZBI.
2. todos os parâmetros de configuração do *kernel* serem passados via arquivo ZBI

[5] *Debuglog* e as syscalls de *debug* são discutidos na Seção 3.7.

[6] Esses parâmetros podem ser encontrados na Tabela 3.1 no manual do GIC-400 disponível em <https://developer.arm.com/documentation/ddi0471/b>.

em memória.

3. forma específica que o arquivo ZBI é passado ao *kernel*.

O suporte a essas particularidades do *kernel* Zircon é implementado pelo boot-shim (discutido na Seção 3.2.1) o qual, por si só, também é um programa *bare metal*.

Para carregar e executar o *kernel*, este trabalho utiliza o *bootloader* U-Boot que é comum em sistemas embarcados e possui diversas funcionalidades que facilitam implementar o protocolo de *boot* de um *kernel* não suportado. Como visto no final da Seção 2.7.1, o U-Boot e o Trusted Firmware-A da ARM são capazes de fornecer todos os módulos de software necessários para implementar um fluxo de *boot* nos SoCs da Rockchip como o RK3328 utilizado na Rock64.

Aqui, U-Boot se refere ao último estágio na sequência de *bootloaders*. Isto é, o estágio que corresponde ao *bootloader* propriamente dito, onde todas as funcionalidades estão carregadas e disponíveis para uso.

O U-Boot pode ser utilizado de forma interativa através de um *shell* de linha de comando similar aos *shells* do Unix como o *bash*. As funcionalidades do U-Boot são implementadas (e utilizadas) através da execução de comandos nesse *shell*. Também são suportados comandos de controle de fluxo e variáveis que, junto com os outros comandos, acabam definindo uma linguagem de programação simples que pode ser usada para criar *scripts*.

As funcionalidades do U-Boot utilizadas neste trabalho são:

- Transferência de arquivos através de link *Ethernet* via protocolo IP.
- Manipulação em memória de *Flattened Device Trees* (FDTs).
- Inspeção e alteração direta de conteúdo em memória.
- Transferência de controle (i.e. execução) de programas *bare metal*.
- Suporte à automação de tarefas por meio de scripts.

O link *Ethernet* é utilizado para aumentar a eficiência do ciclo de desenvolvimento, compilação e teste, pois a alternativa (como carregar o *kernel* a partir de um cartão micro SD) diminuiria a velocidade desse fluxo de trabalho.

A sequência de comandos necessários para executar o Zircon na Rock64 foram implementados na forma de um script do U-Boot (Listagem 2)[7]. O script começa carregando os arquivos relevantes através do protocolo TFTP via link *Ethernet* (li-

[7] O script em si está definido entre as linhas 2-17

Listagem 2: Script do U-Boot para bootar o *kernel* do Fuchsia na Rock64.

```
1  setenv boot_zircon '
2  setenv ipaddr 10.0.0.9;
3  setenv serverip 10.0.0.1;
4
5  tftpboot ${fdt_addr_r} dummy-device-tree.dtb;
6  tftpboot ${kernel_addr_r} rock64-boot-shim.bin;
7  tftpboot ${ramdisk_addr_r} bringup.zbi;
8
9  fdt addr ${fdt_addr_r};
10 fdt resize 2024;
11 fdt chosen ${ramdisk_addr_r} 0;
12 fdt set /chosen bootargs "virtcon.disable kernel.bypass-debuglog";
13 fdt set /memory reg <0 0x200000 0 0xfcd00000>;
14
15 dcache flush;
16
17 go ${kernel_addr_r}';
18 run boot_zircon
```

nhas 5-7), que são enviados por um servidor *Trivial File Transfer Protocol* (TFTP) executando no computador de desenvolvimento. Esses arquivos são colocados diretamente na memória RAM nos endereços correspondentes às variáveis utilizadas[8]. Os endereços para os quais elas expandem estão ilustrados na Figura 4.2.

Há algumas restrições sobre esses endereços e que influenciam os parâmetros utilizados na configuração do boot-shim vistos na seção anterior. Em particular, o endereço da FDT (o arquivo `dummy-device-tree.dtb`) é *hardcoded* no boot-shim e será discutido logo adiante nesta seção. O outro requerimento é que endereço do ZBI seja alinhado à 4 KiB[9] e exista espaço suficiente após o arquivo ZBI para o boot-shim colocar a imagem binária do *kernel* contida no próprio ZBI[10]. Seguindo esses requerimentos, os arquivos poderiam ser carregados em quaisquer outros endereços.

O arquivo `dummy-device-tree.dtb` é uma FDT vazia que é utilizada em tempo de execução pelo script do U-Boot para passar três parâmetros ao boot-shim:

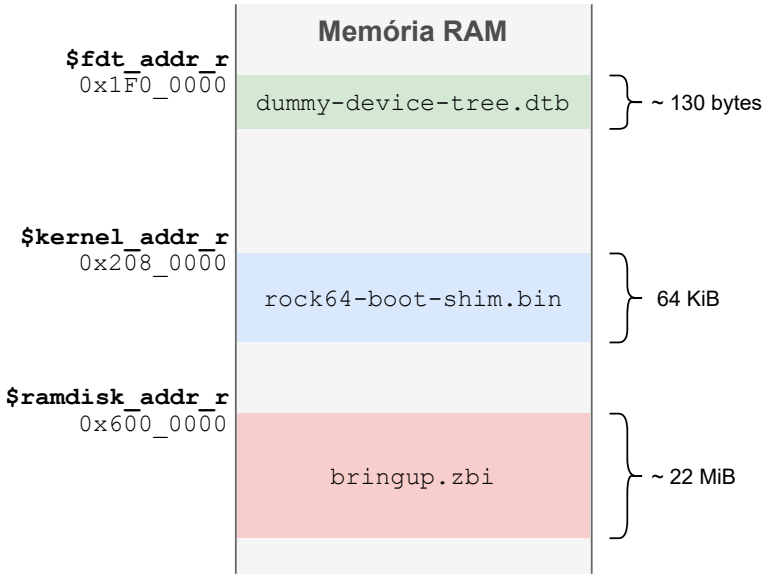
- Endereço em memória do arquivo ZBI, que é definido no parâmetro '`linux,initrd-start`' do nodo '`chosen`' na FDT (linha 11).

[8] Cada *build* do U-Boot, como o utilizado aqui neste trabalho, define um conjunto de variáveis com valores parametrizados de acordo com o hardware alvo.

[9] 4 KiB é o tamanho de página do sistema de memória virtual utilizado pelo Zircon.

[10] O tamanho típico da imagem binária do *kernel* é 1.6 MiB.

Figura 4.2: Locais da memória RAM dos arquivos relevantes para a execução do Zircon.



Fonte: O Autor.

- Linha de comando do *kernel*[11] (linha 12).
- Região do mapa de memória onde está a memória RAM (linha 13). Isso acaba definindo o tamanho da memória RAM utilizada pelo Fuchsia.

Essas informações são inseridas na FDT pelo script do U-Boot no momento em que é executado e, assim, permite a especificação dos valores em tempo de execução. Os outros parâmetros que o *kernel* necessita estão contidos no própria variante do boot-shim criada na Seção anterior (definidos no arquivo `boot-shim-config.h`). Esses parâmetros foram definidos lá por serem informações que não se espera que variem entre diferentes versões da Rock64.

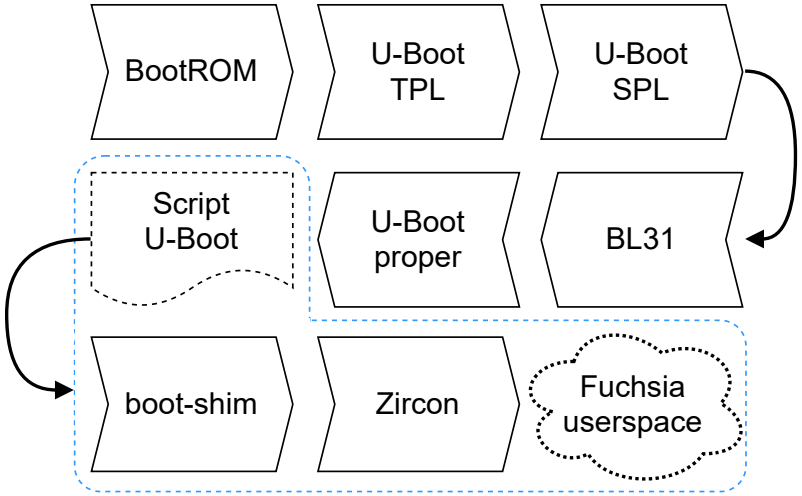
O comando `dcache flush` (linha 15 do script) é discutido na subseção logo a seguir.

A linha 17 concluí a declaração do script do U-Boot com o comando `go`, que faz um *jump* para o endereço de memória especificado (i.e, executa o boot-shim).

A Figura 4.3, ilustra a sequência completa de softwares executando na Rock64 desde o *boot* até o *userspace* do Fuchsia. Os itens após o U-Boot Proper destacados correspondem aos softwares específicos a este trabalho.

[11] Mais informações sobre os parâmetros possíveis da linha de comando do *kernel* disponível em https://fuchsia.dev/fuchsia-src/gen/boot-options e https://fuchsia.dev/fuchsia-src/reference/kernel/kernel_cmdline.

Figura 4.3: Sequência de softwares executados do *boot* ao *userspace* no Fuchsia.



Fonte: O Autor.

4.2.1 Problemas encontrados

É interessante discutir dois problemas que aconteceram durante o desenvolvimento descrito neste capítulo.

O primeiro, apesar de ser simples de resolver, necessitou de um *hack* onde mais uma informação precisou ser definida explicitamente (i.e. *hardcoded*) no boot-shim.

A única entrada do boot-shim é a FDT que ele usa para descobrir o endereço do arquivo ZBI. E a forma como a FDT é passada ao boot-shim é através do registrador `x0` do processador. Porém, o comando `go` do U-Boot utilizado para executar um programa *bare metal* (linha 17 na Listagem 2), não provê um jeito de escrever nos registradores do processador. A solução foi criar outra macro C, opcional no boot-shim que, quando definida, contém o endereço da FDT em memória. O *commit git* que implementa essa solução está disponível no Apêndice A.1. Isso foi implementado em uma mudança anterior ao *commit* que adiciona uma placa para a Rock64 no sistema de *build* (representado na Figura 4.1).

O segundo problema era que o *kernel* executava apenas até um determinado ponto logo no início da sua sequência de inicialização (ou *boot*). A partir desse ponto, uma *exception* era gerada pelo processador e isso causava um *kernel panic* no Zircon.

O processo para bootar o Zircon descrito neste capítulo foi desenvolvido e testado, inicialmente, com o U-Boot executando em uma máquina emulada no QEMU.

E esse teste *funcionou*. A máquina utilizada era a mesma já suportada pelo Fuchsia, que é a máquina por trás da *board* '`qemu-arm64`'. Essa máquina é diferente da Rock64, o que faz com que os parâmetros do arquivo `boot-shim-config.h` e os valores das variáveis no script do U-Boot também sejam diferentes. Porém, os parâmetros no teste do QEMU foram ajustados para os valores corretos da máquina emulada em questão. E mesmo que não fossem, seria esperado que o *boot* do *kernel* fosse mais longe do que o ponto inicial onde o problema estava acontecendo.

Um dos "sintomas" que aconteciam em algumas tentativas era que o cabeçalho do arquivo ZBI em memória estava corrompido e isso, às vezes, acontecia durante a execução do boot-shim sem ele nem ter chegado ao ponto de mexer no ZBI em memória. Isso parecia acontecer aleatoriamente, sem nenhuma mudança de parâmetros que pudesse causar alguma mudança.

No final, o problema acontecia devido a uma suposição equivocada sobre os comandos do U-Boot fazerem um *flush* dos *caches* do processador à medida que fossem executadados. Os arquivos recebidos pelo link *Ethernet* através dos comandos `tftpboot`, de alguma forma, ficavam "incoerentes" entre os caches do processador e a memória RAM. A solução foi simples: bastou adicionar o comando `dcache flush` logo antes de passar o controle para o boot-shim. Porém, como a versão binária do U-Boot disponibilizada pela comunidade da Rock64 não incluía esse comando, foi necessário compilar uma nova versão do firmware que incluísse esse comando. Isso também requer o BL31 do Trusted Firmware-A da ARM (discutido na Seção 2.7), pois o Zircon utiliza o serviço PSCI para bootar os outros *cores* do processador. O Apêndice B mostra como fazer isso e como gravar as imagens no cartão de memória SD.

A execução do boot-shim (e consequentemente, do Zircon) pode ser considerada a execução de um *self-modifying code*, pois o boot-shim inicialmente não está na memória RAM e é carregado lá através do link *Ethernet*. Em um processador ARM, o procedimento para executar um código desse tipo é fazer a invalidação do cache de instruções e fazer um *flush* do cache de dados de forma que o cache de instruções seja carregado com as novas instruções do programa[12]. O U-boot tem o comando `icache`, além do `dcache`, mas a chamada deste último no script de *boot* não fez diferença; isto é, o *boot* continuou funcionando como já estava usando apenas o comando `dcache`. Usando apenas o `dcache` e modificando o código dele para

[12]Mais informações em https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/caches-and-self-modifying-code

fazer um *flush* apenas no cache de dados L1 resultou no mesmo problema. Porém, modificando para fazer um *flush* apenas no cache L2 fazia o *boot* voltar a funcionar. O uso de instruções *assembly* de 'barreiras de memória' (`isb`, `dmb` e `dsb`) não fizeram diferença nos resultados das modificações realizadas.

4.3 Resultado

Resolvidos os problemas da seção anterior, o *kernel* conseguiu executar a sua sequência de inicialização completamente, incluindo execução do primeiro processo o qual é responsável por iniciar o *bootstrap* do *userspace* Fuchsia[13].

Como o produto '`bringup`' incluí o `component_manager` e diversos componentes básicos (incluindo o `driver_manager`), ao final desse processo de inicialização já existem diversos componentes executando, o que incluí também alguns *drivers* de teste.

Um desses componentes, o `console-launcher`[14], tem a função de iniciar um *shell* na porta serial de *debug* do *kernel*. Mesmo sendo acessível por qualquer processo, a porta serial do *kernel* é acessada através de uma *capability* do tipo protocolo criada e disponibilizada pelo componente `console`[15]. O `console-launcher` cria um processo para o *shell* e passa todo o seu namespace de componente para esse processo.

O *shell* utilizado é `dash`[16] que foi modificado pelos desenvolvedores do Fuchsia de tal forma a ver o namespace associado a seu processo como se fosse o sistema de arquivos Unix, onde está executando[17]. Assim, através desse console é possível executar alguns comandos e a experiência é similar a utilização de um *shell* Unix comum.

[13]https://fuchsia.dev/fuchsia-src/concepts/booting/userboot

[14]<https://cs.opensource.google/fuchsia/fuchsia/+/main:src/bringup/bin/console-launcher/meta/console-launcher.cml>

[15]Esse componente foi usado como exemplo na Seção 3.6

[16]Dash é normalmente o *shell* `/bin/sh` em sistemas Linux.

[17]Essa "visão" é possível devido, principalmente, à utilização da biblioteca `fdio` que cria uma camada de abstração em cima de namespaces. Mais informações em https://fuchsia.dev/fuchsia-src/concepts/system/life_of_an_open

5 PORTE II: *DRIVERS* DO SOC

Este capítulo trata do processo de desenvolvimento de *drivers* para os subsistemas do SoC discutidos na Seção 2.9. Dependendo do caso, esses *drivers* não são necessários para utilizar determinadas funcionalidades de outros dispositivos.

Por exemplo, a porta serial UART2 do RK3328 é utilizada desde os primeiros estágios de *bootloaders*. O próprio script do U-Boot visto na Seção 4.2 é enviado à Rock64 através dessa porta. Assim, no momento em que o Fuchsia executa, a porta serial já está configurada com determinados parâmetros. Isso significa que o *driver* do próximo capítulo poderia ser desenvolvido e funcionar sem a necessidade dos *drivers* desenvolvidos aqui. Porém, a diferença que os *drivers* desenvolvidos aqui fazem é tornar possível que outros *drivers* implementem funcionalidades específicas em seus dispositivos de hardware. O *driver* da porta serial, usado no exemplo anterior, não conseguiria suportar *baudrates* a partir de um certo valor se não fosse pelo serviço oferecido pelo *driver* implementado na Seção 5.3.2.

5.1 *Board driver*

Como visto na Seção 3.8.5, a principal função do *board driver* é criar indiretamente os *devices* a partir dos quais os *drivers* são carregados. Sem ele, os *drivers* desenvolvidos nas próximas seções e no Capítulo 6 não poderiam ser carregados. O *board driver* desenvolvido aqui também é usado mais tarde para aplicar algumas configurações através dos protocolos implementados pelos *drivers* desenvolvidos nas Seções 5.2.2 e 5.3.2.

O *board driver* espera fazer *binding* em um *device* associado ao protocolo `ZX_PROTOCOL_PBUS`. Esse *device* é publicado pelo *driver* `platform-bus` com os *Vendor ID* e *Product ID* provenientes do arquivo ZBI. Esses atributos foram especificados na Seção 4.1.

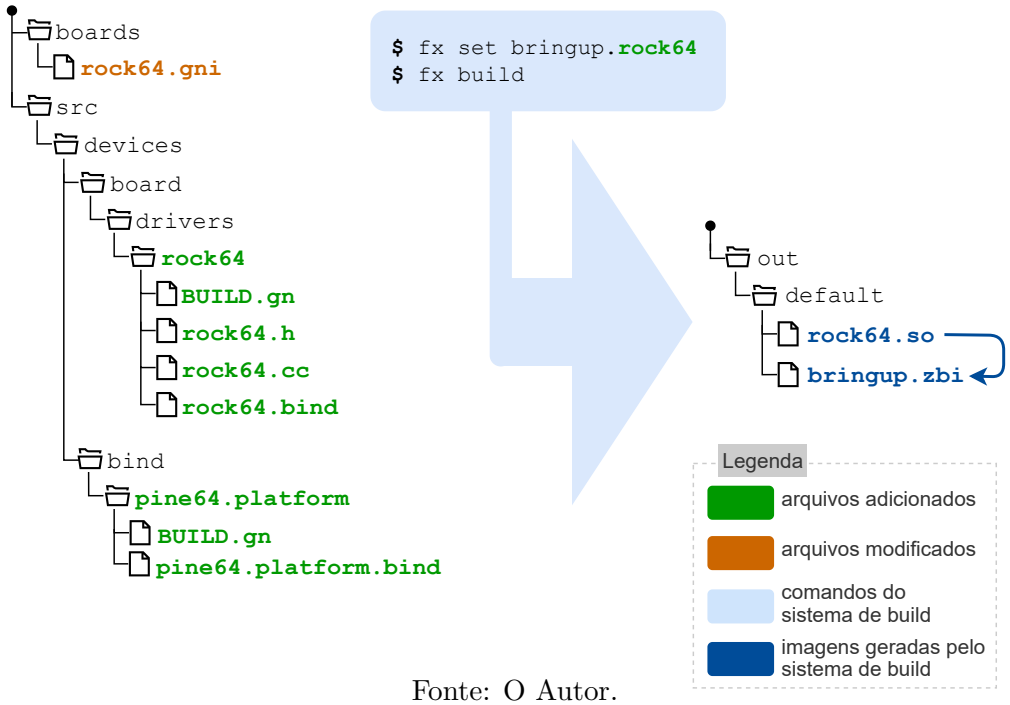
Listagem 3: *Binding program* do *driver* `rock64`.

```
1 fuchsia.BIND_PROTOCOL == fuchsia.platform.BIND_PROTOCOL.BUS;
2 fuchsia.BIND_PLATFORM_DEV_VID ==
  ↪ pine64.platform.BIND_PLATFORM_DEV_VID.PINE64;
3 fuchsia.BIND_PLATFORM_DEV_PID ==
  ↪ pine64.platform.BIND_PLATFORM_DEV_PID.ROCK64;
```

O *binding program* do *board driver* utilizando esses atributos está na Listagem 3. Os atributos `PINE64` e `ROCK64` foram declarados na biblioteca `pine64.platform` definida no arquivo `pine64.platform.bind`. O atributo `fuchsia.platform.BIND_PROTOCOL.BUS` é a forma como o identificador `ZX_PROTOCOL_PBUS` é utilizado na DSL do *binding program*.

A Figura 5.1 ilustra os arquivos relacionados à criação do *board driver*. O *commit git* que faz isso está disponível no Apêndice A.3. O arquivo `rock64.so` é a imagem binária do *driver* e é inserido automaticamente pelo sistema de *build* na *initramfs* embutido na imagem ZBI. Dessa forma, o *driver* consegue ser localizado pelo `driver_manager` em tempo de execução o que possibilita o seu *binding*.

Figura 5.1: Arquivos relevantes na criação do *driver* `rock64`.



Fonte: O Autor.

A classe C++ que define o *board driver* no arquivo `rock64.h` está reproduzida na Listagem 4. Os protocolos Banjo que um *driver* implementa (i.e., que ele é um servidor) são declarados seguindo esse padrão de herança de classes. A classe `ddk::Device<T>`[1] representa o *device protocol* e, no caso desse *driver*, é o único protocolo que ele implementa. Esse protocolo é obrigatório, e a classe `ddk::Device<T>` possui uma implementação padrão para quase todas as operações desse protocolo. O método `DdkRelease()` é a função de *cleanup* para quando o *driver* for descarregado e não é tratada neste trabalho.

[1]Essa classe é documentada em <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/lib/ddktl/include/ddktl/device.h;l=26>

Listagem 4: Declaração da classe com o código do *driver* `rock64`.

```
1  class Rock64 : public ddk::Device<Rock64> {
2   public:
3    Rock64(zx_device_t* parent, const ddk::PBusProtocolClient& pbus)
4        : ddk::Device<Rock64>(parent), pbus_(pbus) {}
5
6    static zx_status_t Bind(void* ctx, zx_device_t* parent);
7
8    void DdkRelease() { delete this; }
9
10  private:
11   zx_status_t Start();
12
13   ddk::PBusProtocolClient pbus_;
14 };
```

O método *static* `Bind(void* ctx, zx_device_t* parent)` representa a função `main()` do *driver* e é executada pelo `driver_host` para que o *driver* faça o *binding*. O *driver* obrigatoriamente sinaliza que concluiu o *binding* publicando o seu *device*, o que é feito através do método `DdkAdd("driver_name")` chamado na linha 40 do arquivo `rock64.cc` (Figura 5.1). A variável `parent` no método `Bind()` faz referência ao *device* pai ao qual o *driver* em questão está fazendo *binding*.

O trecho de código da Listagem 5, declarado entre as linhas 27-32 do `rock64.cc`, obtém uma referência ao protocolo `ZX_PROTOCOL_PBUS` que o *device* pai implementa.

Listagem 5: Trecho do driver `rock64` que cria a conexão com o *device* pai.

```
1  ddk::PBusProtocolClient pbus(parent);
2
3  if (!pbus.is_valid()) {
4    zxlogf(ERROR, "%s: Failed to get ZX_PROTOCOL_PBUS", __func__);
5    return ZX_ERR_NO_RESOURCES;
6  }
```

A classe `ddk::PBusProtocolClient` é gerada pelo compilador Banjo a partir do protocolo PBUS e é a forma pela qual esse protocolo é usado por um *driver* cliente. A instância dessa classe é guardada na variável de instância `pbus_` do *driver* (linha 26 do arquivo `rock64.h` na Figura 5.1) para uso posterior.

Nesse momento, o *board driver* não faz nada além de fazer o *binding* que, do ponto de vista do *driver*, significa se registrar na árvore de *drivers* mantida pelo `driver_manager`. Isso é um requisito que independe do *driver* implementar algum protocolo particular.

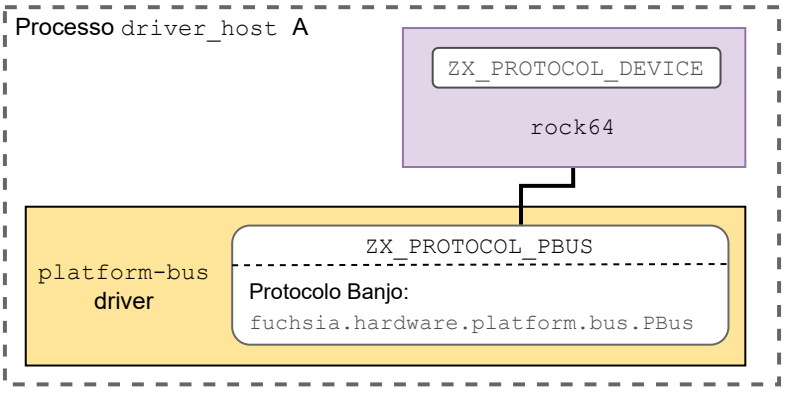
Durante as próximas Seções, quando novos *drivers* forem sendo definidos, esse *driver* vai ser modificado para publicar os *device* necessários para os próximos *drivers* fazerem *binding*.

5.1.1 Resultado

Quando o Fuchsia é executado na Rock64 utilizando o arquivo ZBI gerado pelo *build*, é possível observar que o *driver* conseguiu fazer o *binding* através do *debuglog* que, dentre diversas mensagens de *log*, mostra as mensagens do `driver_manager` indicando que o *driver* foi localizado e está sendo carregado. Além disso, esse *driver* envia uma mensagem de "Hello World!" para o *log*, o que também fica visível.

Adicionalmente, é possível executar o comando `dm dump` no *shell* acessível através da porta serial, o qual imprime um *dump* da árvore de dispositivos e *drivers* que mostra em quais processos de `driver_host`s os *drivers* estão carregados. A saída desse comando mostra que o *driver* `rock64.so` está carregado *no mesmo* `driver_host` que o *driver* `platform-bus`, o que é o comportamento esperado.

Figura 5.2: *Driver* `rock64` após fazer *binding*.



Fonte: O Autor.

A Figura 5.2 ilustra a instância do *driver* `rock64` conectado como cliente ao protocolo PBUS que seu *device* pai implementa. O protocolo `ZX_PROTOCOL_DEVICE` é o *device protocol* implementado pelo *driver* `rock64` (e por todos os *drivers* do Fuchsia). Esse *device* não é usado por nenhum outro *driver* e não será mais ilustrado nos próximos diagramas desse texto.

5.2 *Driver* de *pin-muxing*

O objetivo aqui é implementar o *driver* que realiza a função de multiplexação de pinos (Seção 2.9.1). Apesar de nem todos os pinos do RK3328 serem utilizados para IO, ou algum sinal analógico, — alguns são usados apenas para fins elétricos — os 391 pinos são uma quantia *grande* para implementar um *driver* neste trabalho. Por causa disso, e considerando que o próximo capítulo implementa um *driver* de porta serial, apenas os pinos utilizados nessa porta (que são dois) precisam ser suportados pelo *driver* feito aqui.

O RK3328 tem 3 *Universal Asynchronous Transmitter Receivers* (UARTs), que são os dispositivos controladores de portas seriais. A UART2 é a porta serial padrão utilizada pelos *bootloaders* e é a porta cuja região de MMIO é informada ao *kernel* pelo boot-shim. Por causa disso, ela é a porta serial onde as mensagens do *debuglog* são enviadas e o console do Fuchsia está conectado. Os pinos do SoC onde os principais sinais da UART0 estão conectados não estão acessíveis no *header General Purpose IO* (GPIO) da Rock64, e assim, na prática, não são acessíveis fisicamente.

A porta restante é a UART1, e é por isso que o *driver* desenvolvido nesta seção suporta essa porta e o *driver* do próximo capítulo usa essa mesma porta.

5.2.1 Descrição do hardware

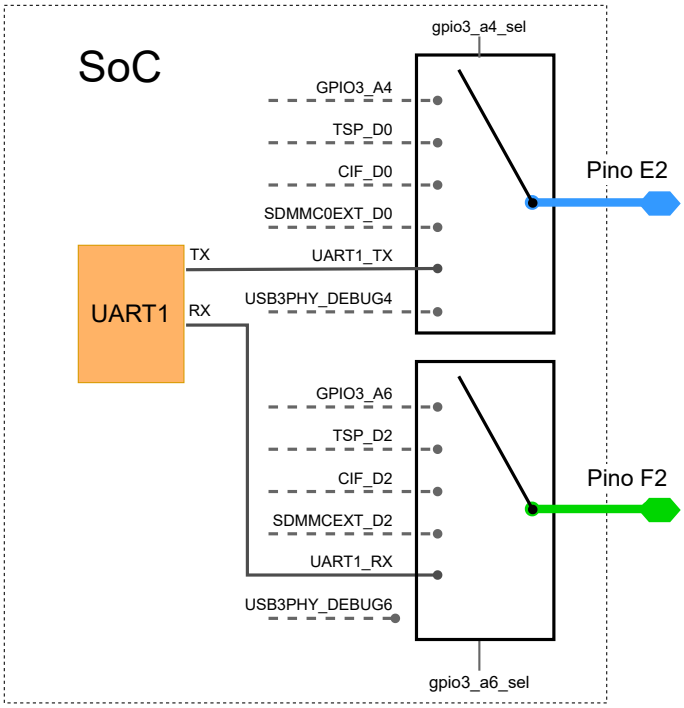
O desenvolvimento do *driver* de *pin-muxing* requer um conhecimento dos mecanismos relevantes e interfaces de software usadas para controlá-los. Esta subseção discute os sinais relevantes da UART1 no RK3328, já que o *driver* desenvolvido aqui suporta apenas os pinos dessa porta serial.

Os sinais mais importantes de uma UART são o TX e RX (*transmissor* e *receptor*, respectivamente). Há outros sinais, mas para uma operação básica, esses são suficientes. Com base nisso, as perguntas essenciais que precisam ser respondidas para que o *driver* desenvolvido aqui consiga realizar sua função são:

1. As saídas TX e RX da UART1 estão conectadas a quais pinos do RK3328?
2. Esses pinos correspondem a quais pinos do *header GPIO* da Rock64?
3. Onde, no mapa de memória do RK3328, estão os registradores que controlam as funções multiplexadas desses pinos?

O tipo de informação necessária para responder a primeira pergunta é, geralmente, codificada em tabelas que mapeiam os pinos do SoC aos sinais dos dispositivos integrados no SoC. O datasheet do RK3328 mostra que um dos sinais multiplexados no pino E2 é o UART1_TX[2]. Também costuma ser dito que a UART1_TX é uma das funções do pino E2. De forma semelhante, o sinal UART1_RX é uma das funções multiplexadas no pino F2. A partir das informações nessa tabela, é possível descobrir quais componentes têm seus sinais multiplexados nesses pinos. Isso está ilustrado no diagrama da Figura 5.3.

Figura 5.3: Sinais conectados aos pinos E2 e F2 no RK3328.



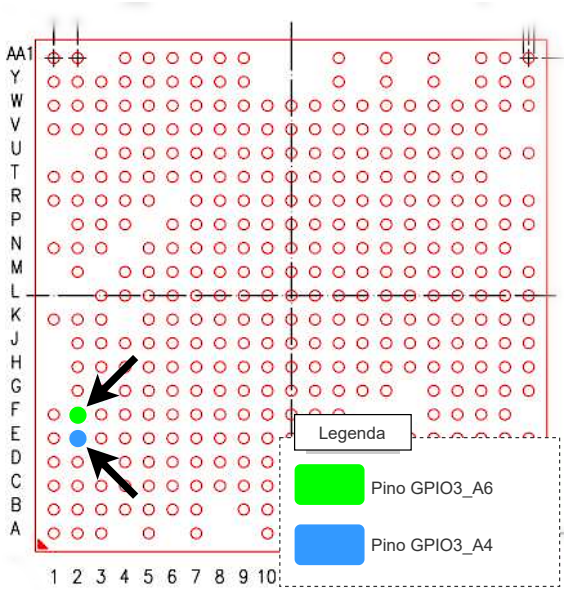
Fonte: O Autor.

Devido ao tipo de circuito que faz a multiplexação das várias funções associada ao pino físico ser semelhante a um circuito de GPIO, os pinos físicos do SoC também costumam ter como nome alternativo a função GPIO associada ao pino. Assim, no caso dos pinos E2 e F2, seus nomes mais comuns correspondem à GPIO3_A4 e GPIO3_A6 respectivamente. A localização desses pinos no chip do RK3328 estão ilustradas no diagrama da Figura 5.4.

A partir desses nomes alternativos, é possível descobrir quais pinos do *header* GPIO da Rock64 que estão conectados nos pinos correspondentes do SoC. Esses pinos estão indicados na Figura 5.5. Isso responde a segunda questão levantada anteriormente.

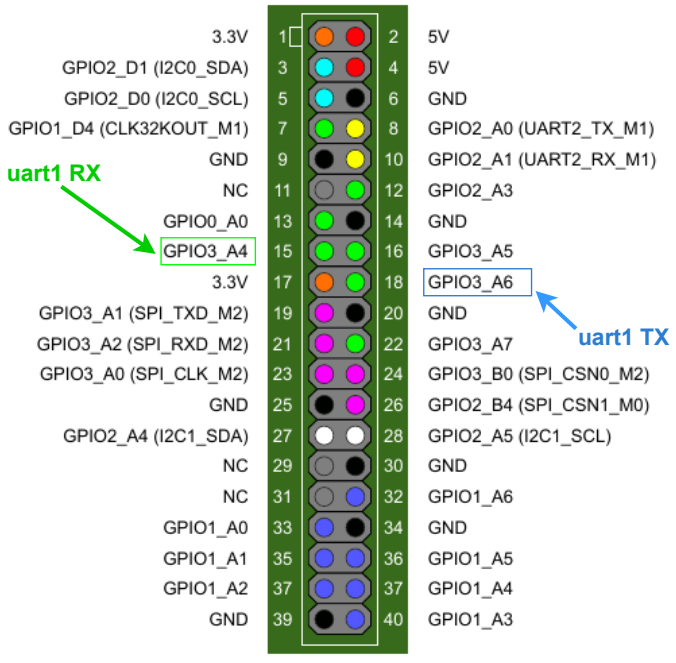
[2]Isso está disponível na Tabela 2-1 de (ROCKCHIP, 2017a).

Figura 5.4: Pinos do RK3328 nos quais os sinais RX e TX da UART1 estão conectados.



Fonte: Adaptado da Fig 2-3 em (ROCKCHIP, 2017a).

Figura 5.5: Localização dos pinos da UART1 no *header GPIO* da Rock64.



Fonte: Baseado em (PINE64, 2020a).

O manual do RK3328 contém as informações necessárias para responder à terceira questão. Uma busca pelas *strings* gpio3_a6 e gpio3_a4 levam às tabelas descrevendo os campos de bit dos registradores que controlam os seletores nos multiplexadores da Figura 5.3. De fato, os seletores nessa Figura foram nomeados

Figura 5.6: Campo `gpio3_a6_sel` no registrador `GRF_GPIO3AH_IOMUX`.

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_enable Bit0~15 write enable "When bit16=1, bit0 can be written by software. When bit16=0, bit 0 cannot be written by software; When bit 17=1, bit 1 can be written by software. When bit 17=0, bit 1 cannot be written by software; ..... When bit 31=1, bit 15 can be written by software. When bit 31=0, bit 15 cannot be written by software;
15:9	RO	0x0	reserved
5:3	RW	0x0	gpio3_a6_sel GPIO3A[6] iomux select 3'b000: gpio 3'b001: tsp_d2 3'b010: cif_data2 3'b011: sdmmc0ext_d2 3'b100: uart1_rx 3'b101: usb3phy_debug6 3'b110: reserved 3'b111: reserved

Fonte: Manual do RK3328, pág. 146 (ROCKCHIP, 2017b).

Figura 5.7: Campo `gpio3_a4_sel` no registrador `GRF_GPIO3AL_IOMUX`.

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_enable Bit0~15 write enable "When bit16=1, bit0 can be written by software. When bit16=0, bit 0 cannot be written by software; When bit 17=1, bit 1 can be written by software. When bit 17=0, bit 1 cannot be written by software; ..... When bit 31=1, bit 15 can be written by software. When bit 31=0, bit 15 cannot be written by software;
15	RO	0x0	reserved
14:12	RW	0x0	gpio3_a4_sel GPIO3A[4] iomux select 3'b000: gpio 3'b001: tsp_d0 3'b010: cif_data0 3'b011: sdmmc0ext_d0 3'b100: uart1_tx 3'b101: usb3phy_debug4 3'b110: reserved 3'b111: reserved

Fonte: Manual do RK3328, pág. 145 (ROCKCHIP, 2017b).

exatamente por causa disso. Como mencionado, ao final da Seção 2.9.1, esses registradores estão na região GRF do mapa de memória do RK3328. O registrador `GRF_GPIO3AH_IOMUX` contém o campo `gpio3_a6_sel` (bits 5-3) e está a um *offset* de `0x003c` bytes do início dessa região. Similarmente, o registrador `GRF_GPIO3AL_IOMUX` contém o campo `gpio3_a4_sel` (bits 14-12) e está a um *offset* de `0x0038` bytes no *General Register Files* (GRF). A Figuras 5.6 e 5.7 mostram a correspondência entre os campos mencionados e as respectivas funções selecionadas nos multiplexadores da Figura 5.3.

Um campo importante nesses registradores GRF, também presente na mai-

oria dos registradores que controlam outros subsistemas do RK3328, é o campo `write_enable`. Ele representa uma máscara de bits que controla quais campos (e valores) do registrador em questão devem ser escritos.

5.2.2 *Driver* `rk3328-gpio`

O diagrama da Figura 5.8 ilustra os arquivos relacionados à criação do driver `rk3328-gpio`. O *commit git* que cria o *driver* está no Apêndice A.4.

O protocolo da DDK que define a interface usada por *drivers* de *pin-muxing* é o `ZX_PROTOCOL_GPIO_IMPL` (Listagem 6), que corresponde ao protocolo Banjo `fuchsia.hardware.gpioimpl.GpioImpl`[3]. Assim, o objetivo do *driver* `rk3328-gpio` é implementar esse protocolo.

Listagem 6: Trecho da declaração do protocolo Banjo `ZX_PROTOCOL_GPIO_IMPL`.

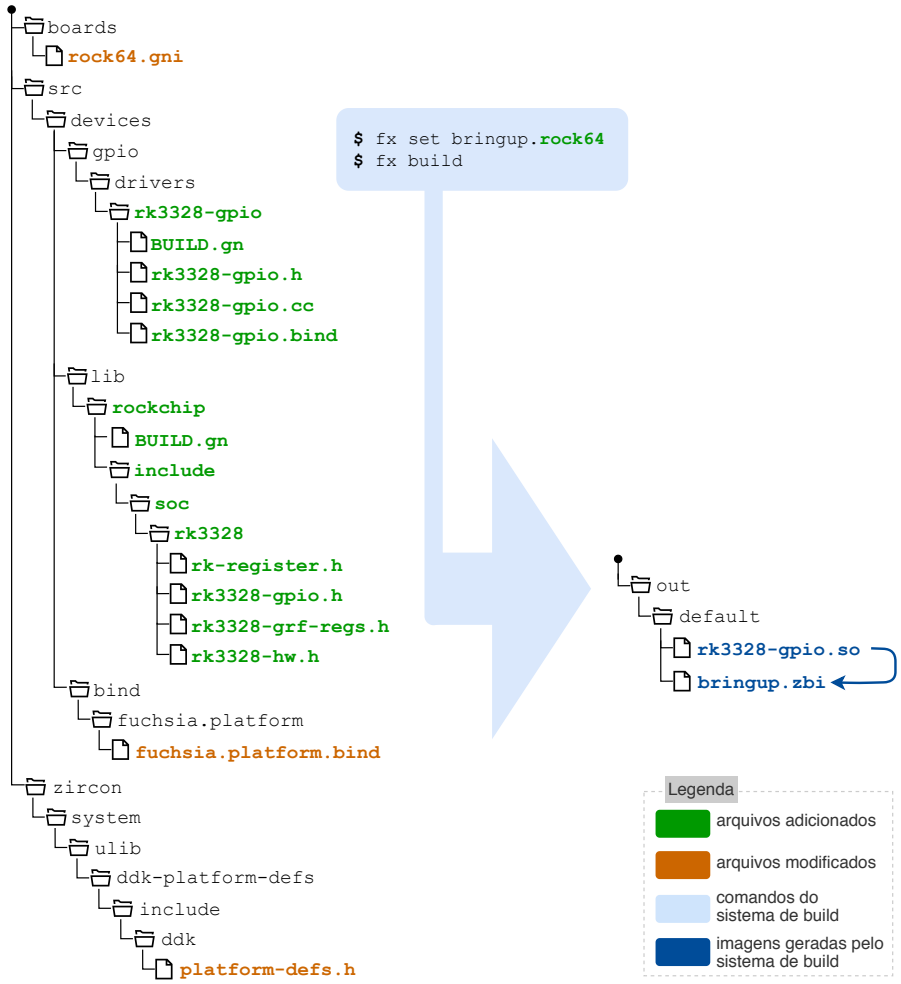
```
1 [Transport = "Banjo", BanjoLayout = "ddk-protocol"]
2 protocol GpioImpl {
3   // ...
4   /// Configures the GPIO pin for an alternate function (I2C, SPI, etc)
5   /// the interpretation of "function" is platform dependent.
6   SetAltFunction(uint32 index, uint64 function) -> (zx.status s);
7   // ...
8 };
```

A interface que esse protocolo define representa duas funcionalidades distintas. Além da funcionalidade de multiplexação, são definidos métodos associados ao controle da função de GPIO o que, como visto na seção anterior, é comumente associada com os sinais e terminologia dos pinos de um SoC. Apesar dessa associação ser comum, as funcionalidades são bem diferentes uma vez que, por si só, GPIO se refere especificamente ao uso de pinos para fins de IO através da leitura (ou escrita) de valores binários e isso não está diretamente relacionado à multiplexação. Neste trabalho apenas a parte do protocolo associado à funcionalidade de multiplexação é implementado no *driver*. Essa parte se resume a um único método — `SetAltFunction()` — cuja definição está na Listagem 6.

Esse protocolo utiliza o parâmetro `index` para identificar o pino do SoC para o qual as operações declaradas devem agir. Esse identificador é um simples número inteiro e cabe aos *drivers* relevantes usarem um mapeamento consistente. Em geral,

[3]Disponível em <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.gpioimpl/gpio-impl.fidl>

Figura 5.8: Arquivos relevantes na criação do *driver* `rk3328-gpio`.



Fonte: O Autor.

isso é feito através do uso de um arquivo de cabeçalho em linguagem C que é incluído por todos os *drivers* que utilizam esse protocolo. A mesma observação se aplica para o parâmetro `function` que é um identificador das possíveis funções multiplexadas no pino.

A partir desse protocolo, o compilador Banjo gera dois conjuntos de códigos: uma versão para ser usada por *drivers clientes* e outra para *servidores*. O *driver* `rk3328-gpio` usa a versão "servidor" que, nesse caso, corresponde à classe C++ `ddk::GpioImplProtocol<T>`. Isto é, o arcabouço C++, correspondente ao protocolo Banjo anterior, está declarado nessa classe. O código na Listagem 7 mostra como essa classe é usada pelo *driver* `rk3328-gpio`.

`Rk3328Gpio` é simplesmente o nome da classe C++ do *driver* `rk3328-gpio`. Como visto anteriormente, a classe `ddk::Device<T,/*...*/>` representa o *device protocol*. O método `GpioImplSetAltFunction(...)` representa a método `SetAlt-`

Listagem 7: Trecho do *driver* `rk3328-gpio` relacionado ao protocolo Banjo.

```
#include <fuchsia/hardware/gpioimpl/cpp/banjo.h>

class Rk3328Gpio;
using DeviceType = ddk::Device<Rk3328Gpio, /* ddk mixins */>;

class Rk3328Gpio :
    public DeviceType,
    public ddk::GpioImplProtocol<Rk3328Gpio,ddk::base_protocol> {
 public:
  //...
  static zx_status_t Create(void* ctx, zx_device_t* parent);
  //...
  zx_status_t GpioImplSetAltFunction(uint32_t index, uint64_t function);
  //...
  ddk::MmioBuffer grf_mmio_;
  //...
};
```

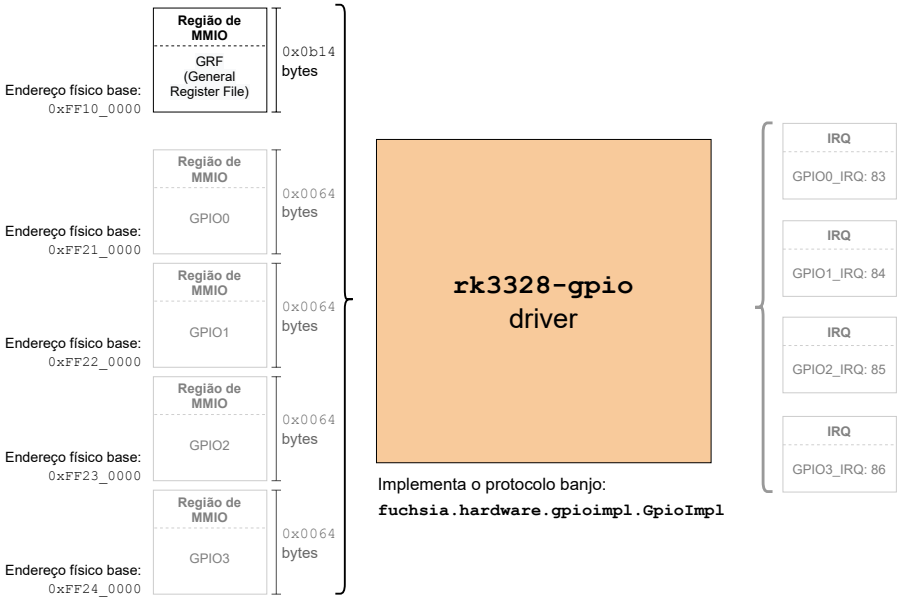
`Function()` do protocolo Banjo visto antes. Assim, para implementar o método desse protocolo, o *driver* só precisa implementar esse método `GpioImplSetAltFunction(...)`. O mixin `ddk::base_protocol` faz o protocolo `ZX_PROTOCOL_GPIO_IMPL` ser o protocolo padrão associado ao *device* publicado pelo `rk3328-gpio` quando ele executar `DdkAdd(...)` durante seu *binding*.

Para que o `rk3328-gpio` consiga realizar a tarefa de *pin-muxing*, ele precisa conseguir escrever nos registradores vistos na Seção 5.2.1. Ele poderia fazer isso indiretamente através de algum outro *driver*, mas, nesse caso, ele é o "outro *driver*". Isso significa que ele precisa obter os recursos de interface de hardware apropriados para conseguir realizar MMIO. O único meio dele fazer isso é fazendo *binding* em um *device* com protocolo `ZX_PROTOCOL_PDEV`, e isso é declarado seu *binding program* no arquivo `rk3328-gpio.bind`.

Assim, durante seu *binding*, o *driver* `rk3328-gpio` utiliza o protocolo PDEV para recuperar os recursos de MMIO oferecido pelo *driver* `platform-bus`. Os recursos que o *driver* usaria se implementasse todos os métodos do protocolo `ZX_PROTOCOL_GPIO_IMPL` estão ilustrados na Figura 5.9. Como o `rk3328-gpio` implementa apenas o método relevante para a tarefa de pin-muxing, ele acaba usando apenas o recurso de MMIO para a região GRF.

É interessante notar que os recursos provenientes do protocolo `ZX_PROTOCOL_PDEV` não contém detalhes associados ao seu recurso de hardware. Por exemplo,

Figura 5.9: Recursos de interface de hardware usadas pelo `rk3328-gpio`.



Fonte: O Autor.

a região de MMIO da região GRF que o `rk3328-gpio` utiliza não tem nenhuma informação sobre o endereço físico de memória onde essa região começa. O `rk3328-gpio` apenas sabe, por convenção, que o primeiro item da lista de MMIOs que ele recebeu corresponde à região GRF. Da mesma forma, os itens na lista de interrupções não contém o número da interrupção de hardware utilizada.

O método `Rk3328Gpio::Create(...)` implementado entre as linhas 30-97 do arquivo `rk3328-gpio.cc` (Figura 5.8) é o responsável pelo *binding* do *driver* e, como parte dessa tarefa, implementa as chamadas ao método PDEV para obter os recursos ilustrados na Figura 5.9.

Listagem 8: Classe C++ declarando um registrador usando a bibioteca `hwreg`.

```
1 class GRF_GPIO3AL_IOMUX
2       : public hwreg::RegisterBase<GRF_GPIO3AL_IOMUX, uint32_t> {
3  public:
4   DEF_ENUM_FIELD_WITH_WRITE_MASK(Gpio3A4Function, 14, 12, gpio3_a4_sel);
5
6   static auto Get() { return hwreg::RegisterAddr<GRF_GPIO3AL_IOMUX>(0x38); }
7 };
```

O último detalhe importante é relativo ao código que o *driver* usa para realizar MMIO, o qual se baseia em dois tipos de classe C++. Uma delas, a `ddk::MmioBuffer`, é proveniente da DDK e é usada para encapsular o recurso de

interface de hardware do tipo MMIO. A outra classe, que na verdade é uma pequena biblioteca C++ chamada `hwreg`, facilita a declaração de classes para definem os campos de bit dos registradores. Por exemplo, a definição do registrador `GRF_GPIO3AL_IOMUX` da Figura 5.7 ficaria como na Listagem 8.

Nesse código, `DEF_ENUM_FIELD_WITH_WRITE_MASK()` é uma macro C++ que define o campo `gpio3_a4_sel` na classe do registrador. O uso dessa macro faz com que a classe declare os métodos `set_gpio3_a4_sel()` e `get_gpio3_a4_sel()`, cuja implementação faz as operações necessárias para lidar o respectivo campo de bit do registrador. Essa macro, em especial, é uma variação das macros disponíveis na biblioteca. Ela declara os métodos de acesso aos campos de bit levando em consideração a máscara apropriada no campo `write_enable` que é comum em registradores dos SoCs da Rockchip. Esse *design pattern* para declarar e lidar com registradores é usado no código dos outros *drivers* desenvolvidos neste trabalho.

5.2.3 Carregamento do `rk3328-gpio`

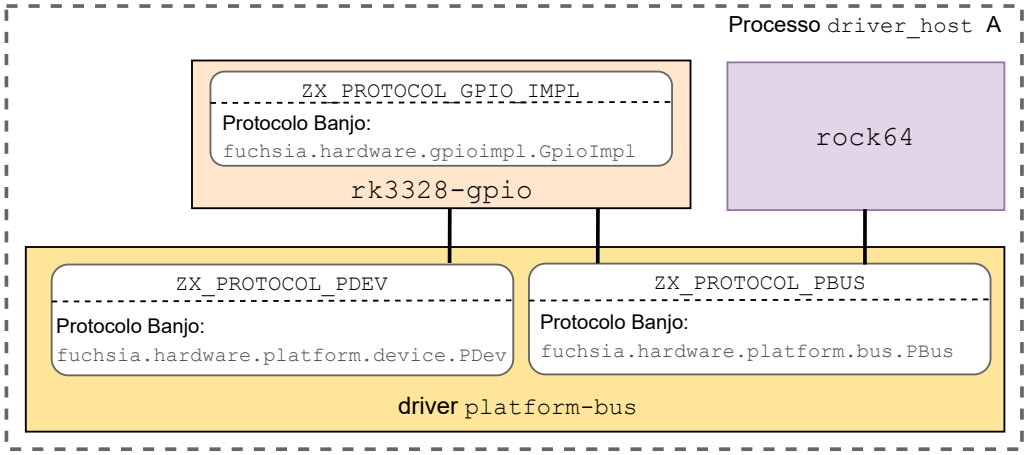
Se o Fuchsia for executado na Rock64 usando o arquivo ZBI gerado, o *driver* `rk3328-gpio` *não* será carregado. Isso acontece porque nenhum *device* para o qual ele possa fazer *binding* não foi publicado.

O *commit* disponível no Apêndice A.5 modifica o *driver* `rock64` de forma que ele publique indiretamente o *device* apropriado para o `rk3328-gpio` fazer *binding*.

O protocolo `ZX_PROTOCOL_GPIO_IMPL` que o *driver* `rk3328-gpio` implementa é considerado útil pelo *framework* de *drivers* e isso significa que o `rock64.so` pode (e vai) eventualmente usar esse protocolo. Por causa disso, o `rock64` usa o método `ProtocolDeviceAdd(...)` do protocolo PBUS (ao invés do `DeviceAdd (...)`). Isso faz com que o `rk3328-gpio` seja carregado no mesmo `driver_host` do *driver* `platform-bus` e tenha acesso ao protocolo PBUS por meio de seu *device* pai. Além disso, também é gerado um requerimento adicional no *binding* no *driver* `rk3328-gpio` o qual, além de publicar seu *device* normal, precisa registrar seu protocolo via método `RegisterProtocol(...)` do protocolo PBUS.

A Figura 5.10 ilustra os *drivers* carregados após o *boot* do Fuchsia e mostra a relação entre seus protocolos e `driver_host`s.

Figura 5.10: *Drivers* carregados incluindo o `rk3328-gpio`.



Fonte: O Autor.

5.3 *Driver* da Árvore de *clock*

O objetivo do *driver* desenvolvido nesta seção é implementar o suporte para árvore de *clock* discutida na Seção 2.9.2. A árvore de *clock* costuma ser mais complexa que o mecanismo de multiplexação de pinos porque, em geral, a síntese do sinal de *clock* usado por um dado dispositivo é feita por um circuito que pode ser configurado por diversos parâmetros. Isso requer um modelo do circuito implementado em software para que o *driver* consiga computar os parâmetros para o circuito produzir o sinal de *clock* com a frequência desejada.

Apesar desses circuitos não serem muito complexos, e de muitos dispositivos integrados utilizarem circuitos similares, as diferenças entre eles requerer um modelo próprio para cada um.

As operações essenciais de um *driver* que controla um circuito de *clock* se resumem à configurar a frequência e ligar/desligar o sinal. Porém, devido a topologia relativamente complexa em que esses circuitos estão incluídos, uma determinada operação realizada em um circuito, cujos sinais de saída estão sendo utilizados por outros circuitos filhos, pode causar uma reação em cadeia que alteraria os sinais dos circuitos filhos. Assim, a realização de uma operação em um determinado circuito precisa levar em conta as alterações correspondentes dos outros circuitos afetados na topologia. Na prática, os *drivers* usam heurísticas que simplificam as interações possíveis, como usar determinados circuitos *Phase Locked Loops* (PLLs) para algumas partes da árvore e não usar certos sinais em outros pontos.

O *driver* desenvolvido nessa seção controla apenas os circuitos que geram os sinais de *clock* para as portas seriais do SoC. Os ramos da árvore de *clock* com esses sinais não são muito profundos e isso facilita, tanto o entendimento dos registradores que controlam esses circuitos, quanto a organização e arquitetura do código fonte do *driver*.

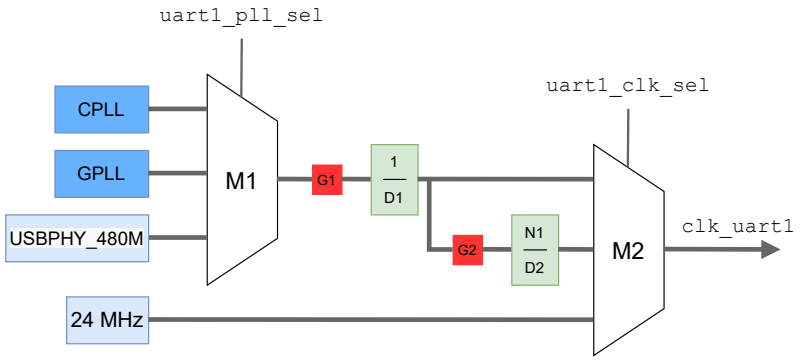
5.3.1 Descrição do hardware

As portas seriais integradas no RK3328 requerem apenas um sinal de *clock*, o qual é usado para controlar o *baudrate*. O circuito que controla a frequência desses *clocks* é um dos únicos que está disponível no manual do SoC(ROCKCHIP, 2017b).

Existem três portas seriais no RK3328, e cada uma delas tem um circuito sintetizador de *clock* dedicado no SoC, o que possibilita a cada porta utilizar um *clock* de frequência distinta ao mesmo tempo. A Figura 5.11 mostra o esquemático simplificado do circuito correspondente à UART1. Os circuitos das outras UARTs têm exatamente a mesma topologia e componentes, mudando apenas os sinais que controlam os parâmetros específicos à cada um (como seletores dos multiplexadores e divisores).

Nesses circuitos, dois dos sinais de *clock* de entrada são provenientes de PLLs do SoC: *Codec PLL* e *General PLL*. O USBPHY_480M é um sinal de 480 MHz gerado por um outro componente integrado ao SoC que lida com o barramento USB e não é suportado pelo *driver* desenvolvido aqui. Isso, porém, não afeta o sinal de *clock* sintetizado pois os PLLs são suficientes para cobrir o sinal gerado pelo USBPHY_480M. O *clock* de 24 MHz, também chamado de XIN24M pelo manual, é o sinal proveniente do oscilador externo ao RK3328. Esse sinal é usado nos circuitos de *clock* da maioria dos dispositivos integrados no SoC porque ele não necessita de configuração e pode ser usado facilmente bastando configurar um multiplexador do circuito. Os componentes em verde são divisores e são responsáveis por mudar a frequência dos *clocks* vindos dos PLLs. Os componentes G1 e G2 são *gates* e são usados para desligar os sinais de *clock*, o que faz com que os divisores parem de funcionar diminuindo o consumo energético. O divisor mais à direita é denominado divisor fracionário por ter um fator multiplicativo (indicado por N1 no circuito).

Figura 5.11: Ramo da árvore de *clock* da UART1 no RK3328.



Fonte: Manual do RK3328, pág. 441 (ROCKCHIP, 2017b).

Tabela 5.1: Correspondência entre parâmetros e campos de registradores.

Parâmetro	**Nome do campo**	**Registrador**
seletor mux M1	uart1_pll_sel	CRU_CLKSEL_CON16
seletor mux M2	uart1_clk_sel	
divisor D1	uart1_pll_div_con	
multiplicador N1	uart1_frac_div_con*	CRU_CLKSEL_CON17
divisor D2	uart1_frac_div_con*	
gate G1	clk_uart1_src_en	CRU_CLKGATE_CON2
gate G2	clk_uart1_frac_src_en	

Fonte: O Autor.

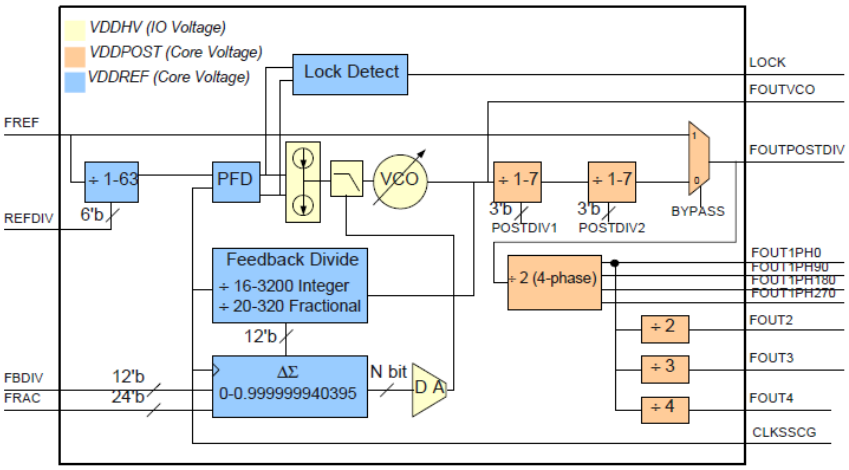
Esse circuito possui 7 parâmetros que podem ser configurados. Cada parâmetro tem um campo de bit correspondente em um registrador na região CRU. A Tabela 5.1 mostra os registradores e os nomes dos campos de bit correspondentes responsáveis pelos parâmetros do circuito. É importante notar que os registradores nessa tabela são específicos ao circuito da Figura 5.11. Isto é, seus os campos de bit controlam apenas os parâmetros do circuito da UART1. Os registradores (e campos de bit) equivalentes que controlam os circuitos das outras UARTs não são mostrados aqui mas podem ser encontrados no manual do RK3328 (ROCKCHIP, 2017b).

Como discutido na Seção 5.3, nem todo o circuito precisa ser suportado, dependendo das frequências do sinal de *clock* que se quer gerar. Por exemplo, se o seletor do multiplexador M2 no circuito da Figura 5.11 for configurado para selecionar o *clock* de 24 MHz, já seria possível utilizar a UART1. Ela ficaria restrita a funcionar com um *baudrate* limitado a um certo valor máximo, mas funcionaria.

Os PLLs do RK3328 são circuitos significativamente mais complexos, o que resulta em diversos parâmetros de configuração (Figura 5.12). Há 5 instâncias (i.e.

circuitos) de PLL no SoC: *ARM*-APLL, *New*-PLL, *DDR*-PLL, *CODEC*-PLL e *General*-PLL. Todos têm a mesma operação e tipos de parâmetros de configuração. Da mesma forma que o circuito das UARTs, os parâmetros são independentes e têm seus próprios registradores distintos.

Figura 5.12: Diagrama esquemático do circuito dos PLLs no RK3328.



Fonte: Manual do RK3328, pág. 22 (ROCKCHIP, 2017b).

Os detalhes de funcionamento desses circuitos não serão discutidos neste trabalho. Mas, em resumo, os PLLs funcionam em dois modos: inteiro e fracionário. O principal sinal de entrada é a frequência de referência ($FREF$) que é o *clock* de 24 MHz do oscilador externo ao RK3328. O principal sinal de saída é o `FOUTPOSTDIV`, que é o sinal utilizado pelos circuitos da UART discutidos anteriormente. As fórmulas que determinam a frequência desse sinal para cada modo de operação são as seguintes:

$$\frac{FREF * FBDIV}{REFDIV * POSTDIV1 * POSTDIV2} \qquad \text{(inteiro)}$$

$$\frac{FREF}{REFDIV * POSTDIV1 * POSTDIV2}(FBDIV + \frac{FRAC}{2^{24}}) \qquad \text{(fracionário)}$$

A Tabela 5.2 contém a relação entre os parâmetros das fórmulas acima e os campos de bit e registradores respectivos. Os registradores dos 5 PLLs têm nomes seguindo um mesmo padrão onde uma letra designa o PLL específico. O X na Tabela 5.2 designa as letras A, N, D, C e G que correspondem aos PLLs APLL, NPLL, DPLL, CPLL e GPLL. Todos esses registradores podem ser encontrados no manual do RK3328 (ROCKCHIP, 2017b).

Tabela 5.2: Principais parâmetros do PLL e seus registradores correspondentes.

Parâmetro	Nome do campo	Registrador
FBDIV	fbdiv	CRU_**X**PLL_CON0
POSTDIV1	postdiv1	
REFDIV	refdiv	CRU_**X**PLL_CON1
POSTDIV2	postdiv2	
DSMPD	dsmpd	
FRACDIV	fracdiv	CRU_**X**PLL_CON2

Fonte: O Autor.

5.3.2 *Driver* `rk3328-clk`

O protocolo Banjo da DDK que define a interface para os dispositivos de *clock* é o `fuchsia.hardware.clockimpl.ClockImpl`[4] reproduzido na Listagem 9.

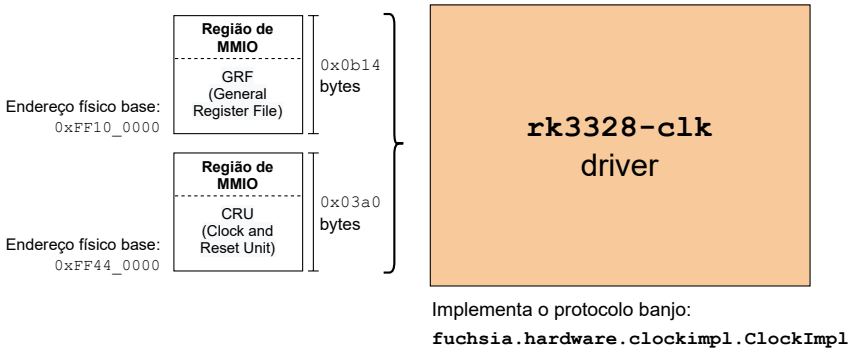
Listagem 9: Declaração do protocolo Banjo `ZX_PROTOCOL_CLOCK_IMPL`.

```
[Transport = "Banjo", BanjoLayout = "ddk-protocol"]
protocol ClockImpl {
  /// Clock Gating Control.
  Enable(uint32 id) -> (zx.status s);
  Disable(uint32 id) -> (zx.status s);
  IsEnabled(uint32 id) -> (zx.status s, bool enabled);

  /// Clock Frequency Scaling Control.
  SetRate(uint32 id, uint64 hz) -> (zx.status s);
  QuerySupportedRate(uint32 id, uint64 hz) -> (zx.status s, uint64 hz);
  GetRate(uint32 id) -> (zx.status s, uint64 hz);

  /// Clock input control.
  SetInput(uint32 id, uint32 idx) -> (zx.status s);
  GetNumInputs(uint32 id) -> (zx.status s, uint32 n);
  GetInput(uint32 id) -> (zx.status s, uint32 index);
};
```

Todos os métodos declarados nesse protocolo usam o parâmetro `id` para identificar os diferentes sinais de *clocks* no SoC, o que é a mesma forma de identificação usada pelo protocolo do *driver* de multiplexação de pinos, mudando apenas o nome do parâmetro. Vale observar que, por serem protocolos diferentes e pelos valores exatos dos identificadores serem uma definição da plataforma (i.e., com escopo deste SoC), esses identificadores estão em um espaço diferente e não têm relação uns com o outros.

[4]Disponível em <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.clockimpl/clock-impl.fidl>

Figura 5.13: Recursos de interface de hardware usadas pelo `rk3328-clk`.



Fonte: O Autor.

Esse protocolo declara diversas operações, mas, como mencionado anteriormente, as essenciais se resumem a configurar a frequência e ligar/desligar o sinal. Essas operações correspondem aos métodos `SetRate()`, `Enable()` e `Disable()`, respectivamente.

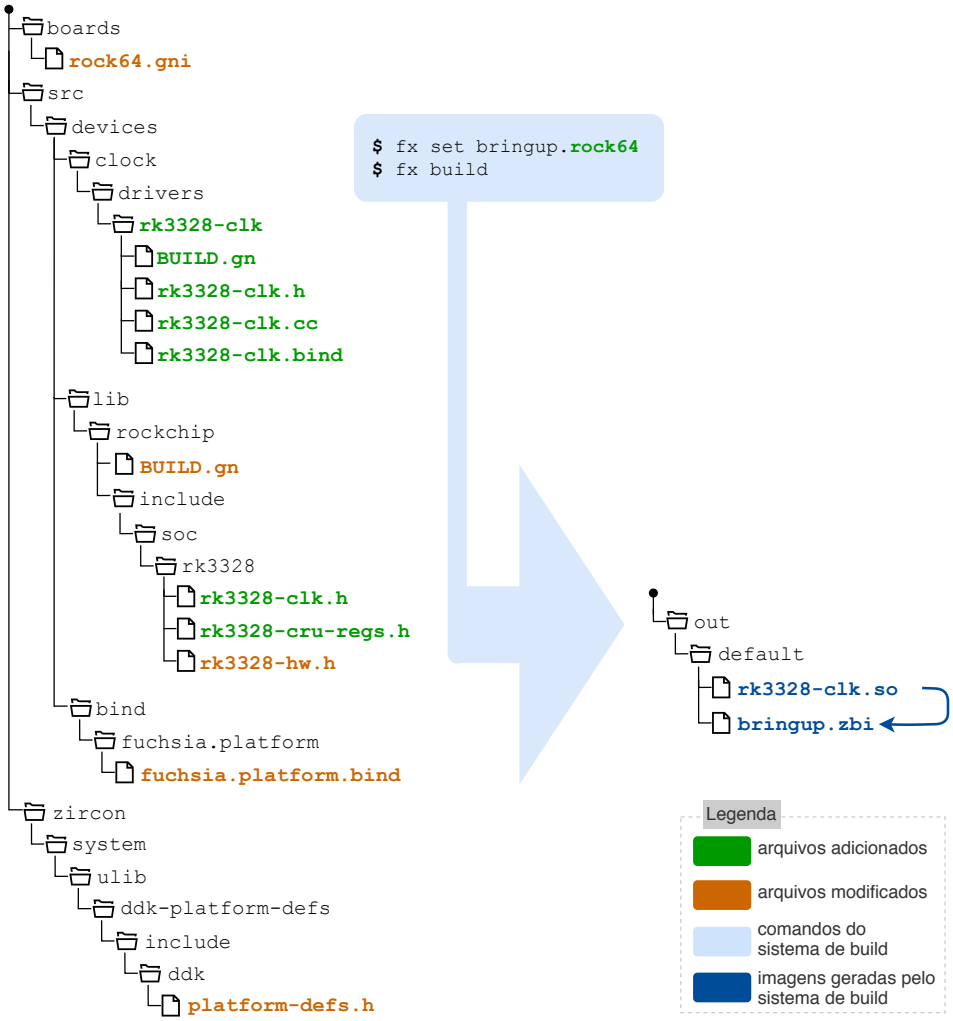
Um método importante é o `SetInput()` que permite configurar o sinal de entrada usado pelo circuito de *clock* em questão. Isso possibilita a utilização do *driver* sem a necessidade de implementar um algoritmo complexo com heurísticas e que controle a árvore de *clock* de maneira automática. O *driver* desenvolvido aqui faz uso disso e a configuração dos sinais de *clock* precisam ser feitas de maneira manual.

O *driver* foi desenvolvido em três etapas, cada uma em um *commit git* distinto. Os recursos de interface de hardware usados pelo *driver* estão ilustrados na Figura 5.13.

A primeira etapa cria o *driver* em si, e implementa o arcabouço com todos os métodos C++ correspondentes do protocolo Banjo retornando uma flag de não suportado. Os arquivos de código fonte criados nessa primeira versão estão ilustrados na Figura 5.14. O *commit git* referente a essas mudanças está disponível no Apêndice A.6. As próximas versões adicionam alguns arquivos que podem ser vistos nos *commits git* respectivos.

Durante o desenvolvimento dos *drivers*, essa primeira versão tinha o objetivo de possibilitar à UART1 funcionar, pois ela requer um sinal de *clock* para tal. A configuração desse *clock* foi implementada no *driver* de uma maneira *hardcoded* através do método na Listagem 10.

Figura 5.14: Arquivos da primeira versão do *driver* `rk3328-clk`.



Fonte: O Autor.

Listagem 10: Trecho de código configurando o *clock* da UART1 de forma *hardcoded*[6].

```
zx_status_t Rk3328Clk::SetUart1ClkTo24Mhz() {
 auto reg0 = CRU_CLKSEL_CON16::Get()
   .FromValue(0).set_uart1_clk_sel(
     CRU_CLKSEL_CON16::Uart1ClkSel::OSC_24MHZ_OUTPUT_CLK
   );
 // ...
 reg0.WriteTo(&cru_mmio_);
 // ...
 return ZX_OK;
}
```

O código da Listagem 10 é executado durante a inicialização do *driver* e, simplesmente, configura os registradores apropriados de forma que o circuito da Figura 5.11 utilize o sinal de 24 MHz externo ao SoC para o *clock* da UART1. Isso é feito escrevendo o número binário `0b10`(representado pela constante `OSC_24MHZ_-`

Figura 5.15: Campo `uart1_clk_sel` no registrador `CRU_CLKSEL_CON16`.

Bit	Attr	Reset Value	Description
31:16	WO	0x0000	write_mask write mask bits "When every bit HIGH, enable the writing corresponding bit When every bit LOW, don't care the writing corresponding bit
15:14	RO	0x0	reserved
			uart1_pll_sel clk_uart1 pll source select control register
11:10	RO	0x0	reserved
9:8	RW	0x0	uart1_clk_sel clk_uart1 clock source select control register 2'b00: select divider ouput from pll divider 2'b01: select divider ouput from fraction divider 2'b10: select 24MHz from osc input 2'b11: select 24MHz from osc input

Fonte: Manual do RK3328, pág. 55-56 (ROCKCHIP, 2017b).

`OUTPUT_CLK`) no campo `uart1_clk_sel` do registrador `CRU_CLKSEL_CON16`, o que faz com que o multiplexador M2 selecione a entrada correspondente ao *clock* de 24 MHz. Os valores possíveis desse campo do registrador estão ilustrados na Figura 5.15 obtida a partir do manual do RK3328. O campo `uart1_clk_sel` têm 2 valores possíveis que selecionam o *clock* de 24 MHz.

O *commit* que altera o *board driver* para que ele publique o *device* que faz o *driver* de *clock* ser carregado está no Apêndice A.7. A publicação do *device* ao qual o *driver* `rk3328-clk` faz *binding* será discutido logo a diante.

A segunda etapa no desenvolvimento do *driver* é a parte onde o suporte ao circuito na Figura 5.11 é de fato implementado. O *commit git* relevante está no Apêndice A.10. Esse suporte é implementado através da definição da classe `UartClk` que utiliza a biblioteca `clocktree` disponível no repositório do Fuchsia[7].

A `clocktree` tem o intuito de auxiliar na modularização e facilitar a implementação dos algoritmos para a automatização do controle de diferentes circuitos de *clock* discutidos anteriormente. Porém, a funcionalidade com os algoritmos que auxiliam na automatização ainda não está concluída e o *driver* desenvolvido aqui utiliza essa biblioteca apenas como um arcabouço[8].

A classe `UartClk` é definida via herança a partir da classe `BaseClock` proveniente pela biblioteca `clocktree` e os métodos da `UartClk` são bem similares aos métodos do protocolo `fuchsia.hardware.clockimpl.ClockImpl`. Entretanto, é importante observar que a classe `UartClk` não representa (ou implementa) um *driver Fuchsia* no sentido em que a classe `Rk3328Clock` faz. Essa semelhança entre

[7] <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/devices/clock/lib/clocktree/>
[8] Mais informações disponíveis em <https://fxrev.dev/361789> e <https://fxrev.dev/361792>

métodos é devido à forma como as classes da biblioteca `clocktree` são definidas e tem o intuito apenas de facilitar a divisão do *driver* em módulos que controlam circuitos individuais.

A classe `UartClk` suporta as três UARTs do RK3328. O código na Listagem 11 mostra os métodos que fazem a interação com o hardware e escrevem nos registradores do SoC.

Listagem 11: Métodos que suportam o *clock* das UARTs no `rk3328-clk`[9].

```
1 void SetGatingOfDividersBranches(bool integer_branch_enable, bool
  ↪  fractional_branch_enable);
2 void SetPllMuxSelector(const UartPllSel selector);
3 void SetBranchMuxSelector(const UartClkSel selector);
4 void SetIntegerDivider(uint8_t denominator);
5 void SetFractionalDivider(uint16_t numerator, uint16_t denominator);
```

O principal método da classe `UartClk` é o `SetRate()` definido entre as linhas 285-323 do arquivo `uart-clk.cc` (Apêndice A.10). Ele implementa a lógica que escolhe os parâmetros adequados para que o circuito gere o *clock* com a frequência informada no parâmetro.

É interessante notar que, sem o suporte aos PLLs (e ao USBPHY_480M), o circuito de *clock* das UARTs na Figura 5.11, teoricamente, só poderia se configurado para usar o *clock* de 24 MHz, pois os outros sinais do multiplexador M2 são todos provenientes de componentes não suportados nesse ponto e a frequência de seus sinais de *clock* podem não ser conhecidos *a priori*. Poderia existir uma situação onde o sinal do CPLL ou GPLL tivessem uma frequência conhecida e fixa. Nesse caso, o *driver* de *clock* da UART poderia contar com isso para funcionar.

A terceira etapa no desenvolvimento do *driver* é o suporte aos PLLs. O *commit git* associado está no Apêndice A.12. O suporte é feito através da classe `PllClk` definida no arquivo `pll-clk.h`. Dado que essa classe herda da classe `BaseClock`, ela possui os mesmos tipos de métodos públicos que a classe `UartClk`. A classe `PllClk` suporta os PLLs apenas no modo inteiro e aplica a fórmula correspondente para tal.

5.3.3 Carregamento do `rk3328-clk`

De forma similar ao *driver* `rk3328-gpio`, o protocolo implementado pelo *driver* de *clock* também é considerado útil pelo *framework* de *drivers* e o *device* ao qual

ele faz *binding* precisa ser criado pelo `rock64` usando o método `ProtocolDeviceAdd()` do protocolo PBUS. Isso também faz com que o `rk3328-clk` precise registrar seu protocolo no PBUS através do método `RegisterProtocol()`.

A diferença importante aqui é que ao solicitar a criação do *device* via protocolo PBUS para o *driver* `platform-bus`, o *driver* `rock64` especifica um parâmetro adicional que faz com que o *driver* `platform-bus` publique um *device* adicional quando o `rk3328-clk` registrar seu protocolo no PBUS. Isso é uma funcionalidade do *driver* `platform-bus` que não foi discutida anteriormente, mas está relacionada à discussão da Seção 3.8.6.

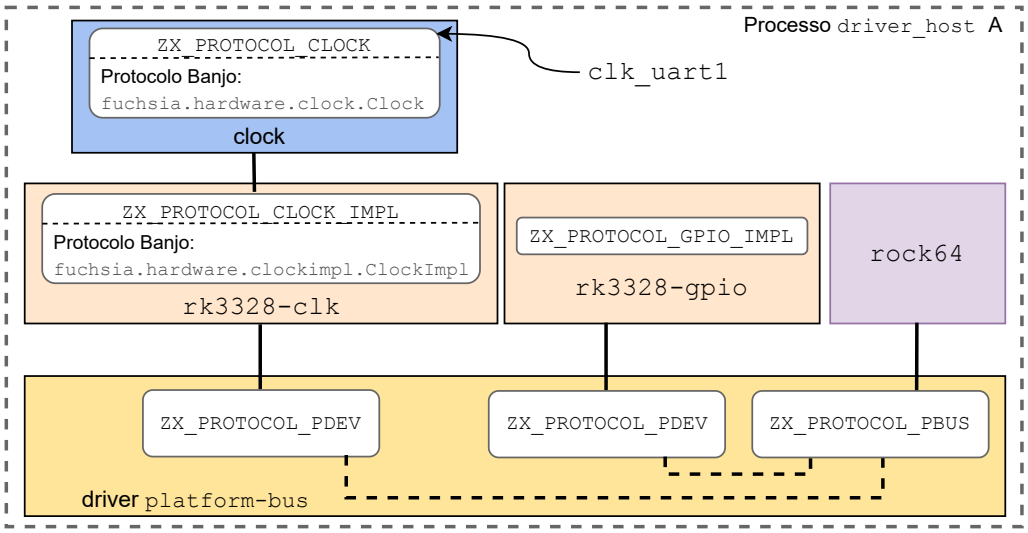
Esse parâmetro adicional que o `rock64` informa contém uma lista de `id`s que correspondem aos `id`s nos métodos do protocolo Banjo `fuchsia.hardware.clockimpl.ClockImpl`, visto anteriormente. Quando o `rk3328-clk` regista seu protocolo via PBUS, o *driver* `platform-bus` publica um *device* com protocolo `ZX_PROTOCOL_CLOCK` contendo essa lista informada pelo `rock64`. Isso faz com que um *driver* apropriado para esse *device* seja buscado. O *driver* carregado é denominado `clock` e é um driver genérico, já disponível no repositório do Fuchsia, e incluído automaticamente no *build*[10]. Esse *driver* publica um *device* para cada `id` da lista anterior, o que significa que cada *device* encapsula os sinal de *clock* referente ao circuito suportado. Isso possibilita que esses sinais de *clock* podem ser acessados individualmente, pois há um *device* representando cada um.

Esse mesmo padrão poderia ter sido usado com o *driver* `rk3328-gpio`, mas não foi adotado por simplicidade e por não ser um padrão utilizado pelos *drivers* no repositório do Fuchsia. A Seção 6.2.2, no próximo capítulo discute o uso dessa funcionalidade pelo *driver* da porta serial.

A Figura 5.16 ilustra a relação entre os *drivers* instanciados e seus respectivos protocolos. O *driver* `rock64` especifica um único `id` que corresponde ao sinal gerado pelo circuito sintetizador do *clock* da UART1. A mudança que faz isso está na linha 38 do arquivo `rock64-clk.cc` e foi feita no *commit* do Apêndice A.10.

[10]O código desse *driver* está disponível em <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/devices/clock/drivers/clock/>

Figura 5.16: *Drivers* carregados incluindo o `rk3328-clk` e o driver genérico.



Fonte: O Autor.

6 PORTE III: *DRIVER* DE PERIFÉRICO

O *driver* deste capítulo representa um dispositivo da camada mais alta no diagrama da Figura 2.1. Isso significa que o dispositivo que ele suporta não é específico ao SoC da Rock64. Assim, ele poderia ser usado em outras placas com processadores ARM e o *driver* desenvolvido aqui funcionaria do mesmo jeito.

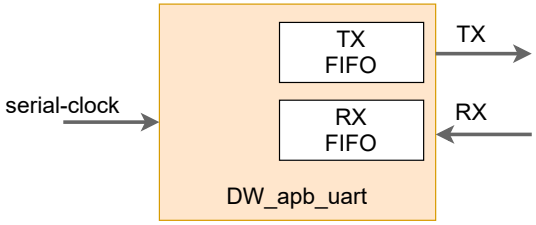
Esse capítulo começa fazendo uma introdução ao funcionamento do dispositivo UART e da parte do *driver* que de fato controla o dispositivo. Porém, como discutido em diversos pontos desse trabalho, um dispositivo dificilmente funciona de forma isolada e depende de outros dispositivos ou subsistemas do hardware. Assim, o objetivo principal deste capítulo é mostrar como essas dependências são modeladas e suportadas pelo *framework* de *drivers* do Fuchsia e, consequentemente, pelos *drivers* desenvolvidos no trabalho.

Esse *driver* ilustra como a *composição no nível do hardware* pode ser modelada e tratada por *drivers* no Fuchsia. Como já discutido, os dispositivos de hardware, frequentemente, não operam sozinhos e necessitam de recursos e funcionalidades implementadas por outros dispositivos. Essas relações de requerimentos precisam ser levadas em conta para todas as funcionalidades do dispositivo poderem ser utilizadas. Consequentemente, os *drivers* precisam modelar isso de alguma forma, e o *driver* desenvolvido aqui ilustra isso através dos *composite devices*.

6.1 Descrição do hardware

Os dispositivos de hardware por trás das portas serias são as *Universal Asynchronous Receiver-Transmitter* (UART). O nome não é muito sugestivo quanto à função exata do dispositivo, mas as UARTs são usadas para transmitir e receber bits que são geralmente agrupados em octetos. Assim, a interface de alto nível do dispositivo consiste no envio e recebimento de bytes. As UARTs do RK3328 são instâncias do mesmo dispositivo de hardware desenvolvido pela Synopsys cujo nome oficial é DW_apb_uart. Por um lado, essa UART é específica para processadores ARM devido ao fato de funcionar através de um barramento específico da ARM. Entretanto, ela foi desenvolvida para ser compatível com uma família de portas seriais conhecidas como 16550 (dentre vários nomes). Isso significa que, teoricamente, a interface de software dos dispositivos dessa família é suportada pela DW_apb_uart,

Figura 6.1: Principais sinais da DW_apb_uart.



Fonte: O Autor.

o que quer dizer que os registradores são iguais, possuindo os mesmos campos de bit e a mesma semântica associada. Há outros aspectos, além do conjunto de registradores, que definem a interface do dispositivo, como os tipos de interrupções, mas isso está ligado essencialmente à operação do dispositivo e à semântica associada ao conjunto de registradores e não será discutido aqui.

Apesar de ser relativamente simples, uma UART possui diversos pinos que podem ser usados dependendo do modo de operação suportado. Do ponto de vista das funcionalidades que o *driver* deste capítulo implementa, os pinos relevantes da DW_apb_uart são o TX, RX e o sinal de *clock* (Figura 6.1).

Os pinos TX e RX, *transmissor* e *receptor*, respectivamente, são os mais importantes, pois é através deles que a comunicação serial propriamente dita acontece. Como já mencionado no capítulo anterior, a UART usa o pino *serial-clock* para controlar a taxa de transmissão (e recepção) da linha serial (i.e., através dos pinos TX e RX). Na terminologia usada em protocolos de comunicação serial, essa taxa é denominada *baudrate*.

As UARTs da família 16550 possuem 12 registradores. Uma característica importante dessa interface é que alguns desses registradores compartilham o mesmo endereço. Isso foi feito, historicamente, para minimizar o espaço de endereçamento utilizado. Com esse esquema, são usadas 7 posições de memória para acessar 13 registradores. Esse compartilhamento é possível, primeiramente, devido uma multiplexação que define qual registrador está ativo em um dado endereço. O controle dessa multiplexação é feito através de um campo de bit chamado *Divisor Latch Access Bit* (DLAB) presente em no *Line Control Register* (LCR). O segundo detalhe, que torna isso possível, é uma característica do mecanismo de MMIO o qual, de um determinado ponto de vista, pode ser visto como um protocolo de troca de mensagens. Um dispositivo consegue distinguir entre uma operação de escrita e leitura realizada nos endereço mapeados para seus registradores. Essa distinção permite ao

Tabela 6.1: Registradores da DW_apb_uart.

Offset	**Registrador**	**Tipo de Acesso**	**Condição**
	RBR	R	DLAB=0
	THR	W	DLAB=0
0x00	DLL	R/W	DLAB=1
0x04	DLH	R/W	DLAB=1
0x0c	LCR	R/W	None

Fonte: O Autor.

dispositivo interpretar operações de escrita provenientes do barramento de memória como mensagens cujo conteúdo é escrito em determinado registrador. Da mesma forma, uma operação de leitura para o mesmo endereço pode ser respondida com o conteúdo de um registrador diferente do anterior.

O envio de dados para transmissão na UART é feito através da escrita de um byte no registrador *Transmitter Holding Register* (THR). E o dado recebido (se houver algum) fica armazenado no *Receiver Buffer Register* (RBR). Adicionalmente, pode-se notar que o tipo de acesso deles são diferentes. O THR só pode ser escrito e o RBR só pode ser lido. Como comentado anteriormente, a UART usa a própria operação de escrita (ou leitura) na posição de memória desses registradores (feita via MMIO) para diferenciar e determinar qual registrador vai ser utilizado. Supondo que o bit DLAB=0, uma simples escrita na primeira posição de memória vai escrever somente no registrador THR e não vai tocar no registrador RBR. E vice-versa.

A UART possui um divisor de *clock* interno de 16 bits que possibilita diminuir a frequência do sinal de *clock* recebido pelo pino `serial_clock`. Os bytes mais (e menos) significativos são armazenados nos registradores *Divisor Latch High* (DLH) e *Dividor Latch Low* (DLL), respectivamente. Desde que o bit DLAB esteja configurado para `1`, esses registradores podem ser acessados para leitura e escrita. Assim, a UART pode diminuir o *baudrate*, mas não consegue aumentar além de um certo limite. Dependendo do *baudrate* máximo que se quer utilizar, e sabendo a frequência do sinal `serial_clock`, é perfeitamente possível utilizar a UART sem o *driver* do circuito sintetizador do sinal `serial_clock`.

6.2 *Driver* `dw-apb-uart`

O *driver* foi desenvolvido como um *fork* do *driver* `uart16550` já disponível no repositório do Fuchsia[1]. Isso foi feito pelos seguintes motivos:

1. O *driver* `uart16550` usa o conjunto padrão de registradores da família 16550. Esses registradores têm uma largura de 8 bits, enquanto que os da DW_apb_uart são estendidos para 32 bits. Além disso, alguns campos de bit deles não são exatamente iguais aos registradores da DW_apb_uart.
2. O *driver* se baseia no protocolo Banjo `fuchsia.hardware.acpi.Acpi` para obter os recursos de interface de hardware. Esse protocolo define uma interface similar, mas não exatamente igual a do protocolo `fuchsia.hardware.platform.device.PDev`.
3. O *driver* não utiliza o protocolo de *clock* para descobrir em tempo de execução a frequência do sinal `serial_clock` fornecido para a UART. É usado um valor pré-definido (*hardcoded*) de frequência.

Todos esses pontos podem ser resolvidos através do uso de códigos condicionais (p.ex, `#ifdef`s, etc.) para definir diferentes versões dos métodos C++ e usar instâncias diferentes de variáveis membro em cada caso. Isso foi até feito inicialmente, mas o código ficou mais complicado em termos de tamanho e lógica e, na época, o *driver* não funcionava devido a uma certa peculiaridade DW_apb_uart. Assim, foi decidido fazer uma versão do *driver* independente e específica para o DW_apb_uart.

O protocolo Banjo para portas seriais é o `fuchsia.hardware.serialimpl.SerialImpl`[2] que corresponde ao identificador `ZX_PROTOCOL_SERAL_IMPL` da DDK. Os principais operações do protocolo estão ilustrados na Listagem 12.

O *driver* da DW_apb_uart foi desenvolvido em duas etapas. A primeira etapa implementa o *driver* sem o suporte ao aumento do *baudrate*. A segunda etapa implementa o *driver* levando em conta o sinal de *clock* provido pelo SoC e o *driver* correspondente.

[1] Esse *driver* está disponível em <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/devices/serial/drivers/uart16550/>

[2] <https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/banjo/fuchsia.hardware.serialimpl/serial-impl.fidl>

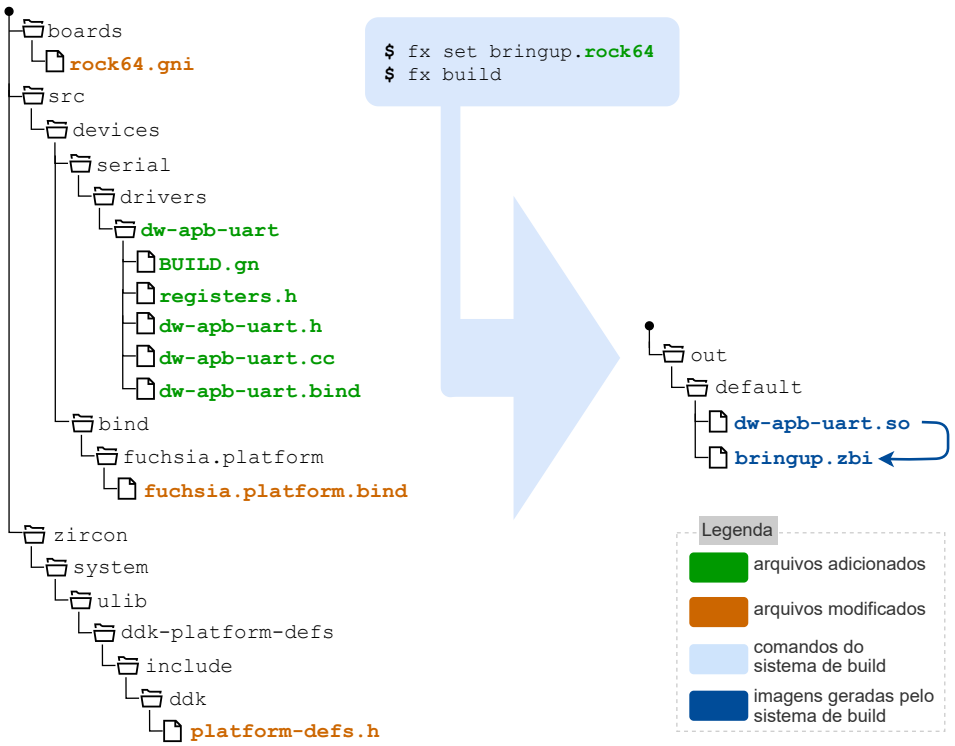
Listagem 12: Trecho da declaração do protocolo Banjo `ZX_PROTOCOL_SERIAL_IMPL`.

```
[Transport = "Banjo", BanjoLayout = "ddk-protocol"]
protocol SerialImpl {
    // ...
    /// Configures the given serial port.
    Config(uint32 baud_rate, uint32 flags) -> (zx.status s);
    // ...
    Read() -> (zx.status s, [Buffer] vector<uint8>:MAX buf);
    Write([Buffer] vector<uint8>:MAX buf) -> (zx.status s, uint64 actual);
    // ...
};
```

6.2.1 Primeira versão

Os arquivos de código fonte associados a essa versão estão ilustrados na Figura 6.2. O *commit git* correspondente está disponível no Apêndice A.8.

Figura 6.2: Arquivos da primeira versão do *driver* `rk3328-clk`.

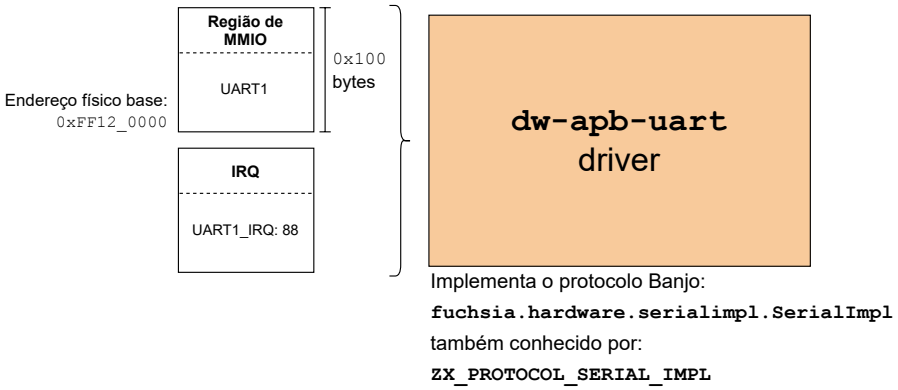


Fonte: O Autor.

Todos os métodos do protocolo `ZX_PROTOCOL_SERAL_IMPL` são implementados, embora o método `Config(...)` não consiga aumentar o *baudrate* e também precisa ter a frequência do sinal `serial-clock` pré-definida (i.e., *hardcoded*) no código[3].

[3]Isso está declarado na variável `kDefaultBaudRate` na linha 32 do arquivo `dw-apb-uart.cc`

Figura 6.3: Recursos de interface de hardware usados pela primeira versão do *driver*.



Fonte: O Autor.

Os recursos de interface de hardware que essa versão usa estão representados na Figura 6.3. É interessante notar o uso de uma interrupção de hardware pois *drivers* desenvolvidos nos capítulos anteriores não utilizam[4].

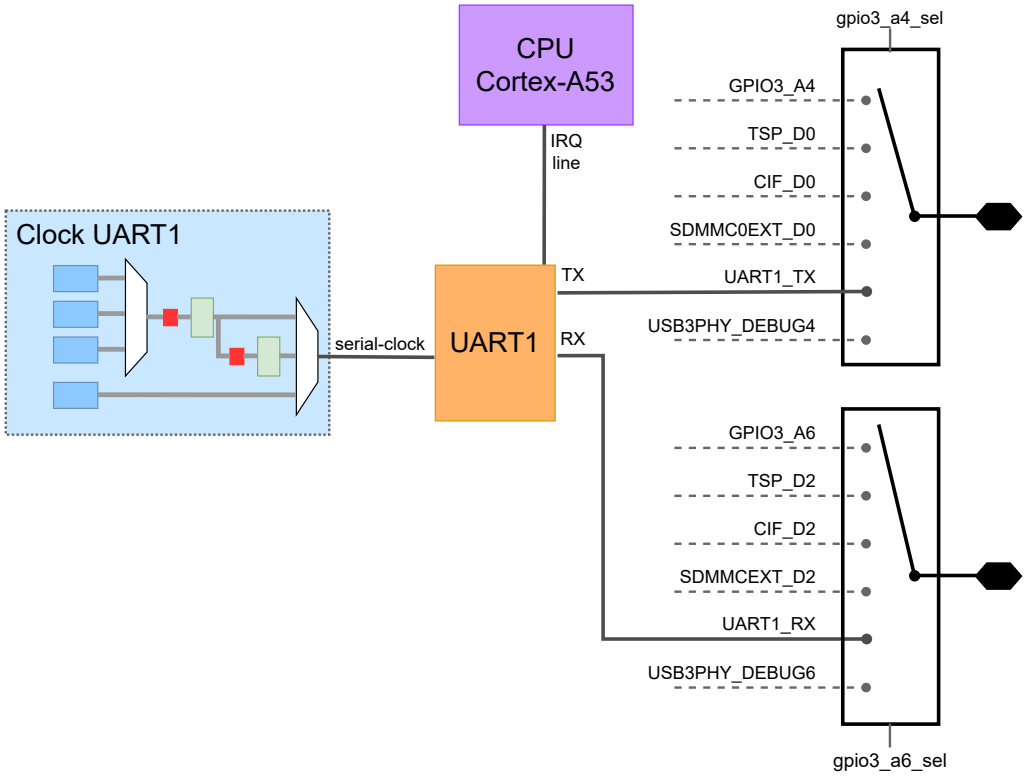
Esses recursos são especificados pelo *board driver* `rock64` em um *commit git* próprio disponível no Apêndice A.9. O `rock64` faz isso através do método `DeviceAdd(...)` do protocolo PBUS o que faz com que o `dw-apb-uart` seja instanciado em processo `driver_host` próprio. A Figura 3.3, usada como exemplo na Seção 3.8.6, ilustra a relação entre os *drivers* e `driver_host`s que acontece aqui. Isso mostra que o `dw-apb-uart` faz *binding* em um *device* com protocolo `ZX_PROTOCOL_PDEV` e publica um *device* com protocolo `ZX_PROTOCOL_SERIAL_IMPL` o qual é usado pelo driver genérico para fazer *binding*. Esse *driver* genérico chamado `serial` é quem implementa o protocolo FIDL por meio do qual a porta serial pode ser acessada por outros componentes do Fuchsia.

6.2.2 Segunda versão

A segunda etapa no desenvolvimento do `dw-apb-uart` implementa uma mudança de forma a usar a funcionalidade de *composite devices* do *framework* de *drivers* do Fuchsia discutido na Seção 3.8.7. Isso permite a um *driver* fazer *binding* em vários *device* ao mesmo tempo onde cada um representa o protocolo implementado pelo dispositivo usado pelo *driver* em questão.

[4]Uma versão do *driver* `rk3328-gpio` que implementasse todo os métodos do protocolo `ZX_PROTOCOL_GPIO_IMPL` também necessitaria de acesso à interrupções de hardware.

Figura 6.4: Alguns dos componentes de hardware usados pela UART1 do RK3328.



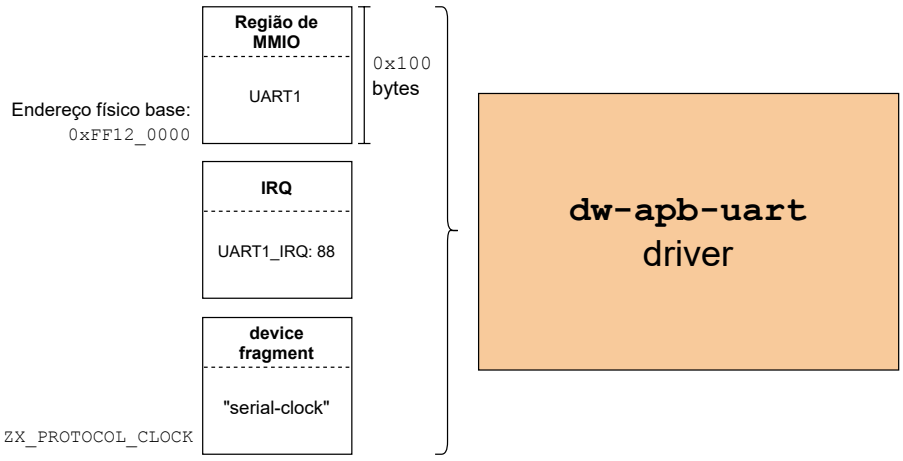
Fonte: O Autor.

A Figura 6.4 ilustra os componentes e sinais de hardware relevantes usados pela DW_apb_uart correspondente a UART1 do RK3328. A ideia aqui é possibilitar ao *driver* `dw-apb-uart` ter acesso ao protocolo do *driver* da árvore de *clock*, o qual contém um módulo que suporta o circuito que sintetiza o sinal `serial-clock`. Esse novo recurso de hardware (dessa vez proveniente do *composite device* e não do protocolo PDEV) está ilustrado na Figura 6.5.

É interessante notar que o *driver* `dw-apb-uart` não precisa (e não deve) saber o `id` do *clock* usado no protocolo Banjo `fuchsia.hardware.clockimpl.ClockImpl` que identifica o *clock* da UART1. Ele apenas utiliza o fragmento identificado pela string `serial-clock` contido no *composite device* ao qual faz *binding* e supõe que o *device* por trás desse fragmento de fato implemente o protocolo `ZX_PROTOCOL_CLOCK` no sinal de *clock* correto. Como visto na Seção 5.3.3 é o *driver* `rock64` que orquestra esses parâmetros e identificadores.

O trecho de código que o *driver* `dw-apb-uart` usa para recuperar o fragmento utilizado está na Listagem 13 e usa a função `device_get_fragment_count()` da DDK para testar se existem *devices* do tipo fragmento no *device* pai ao qual o *driver* está fazendo *binding*. Se houver, o protocolo que implementam é acessado da mesma

Figura 6.5: Recursos de hardware usadas pela segunda versão do `dw-apb-uart`.



Fonte: O Autor.

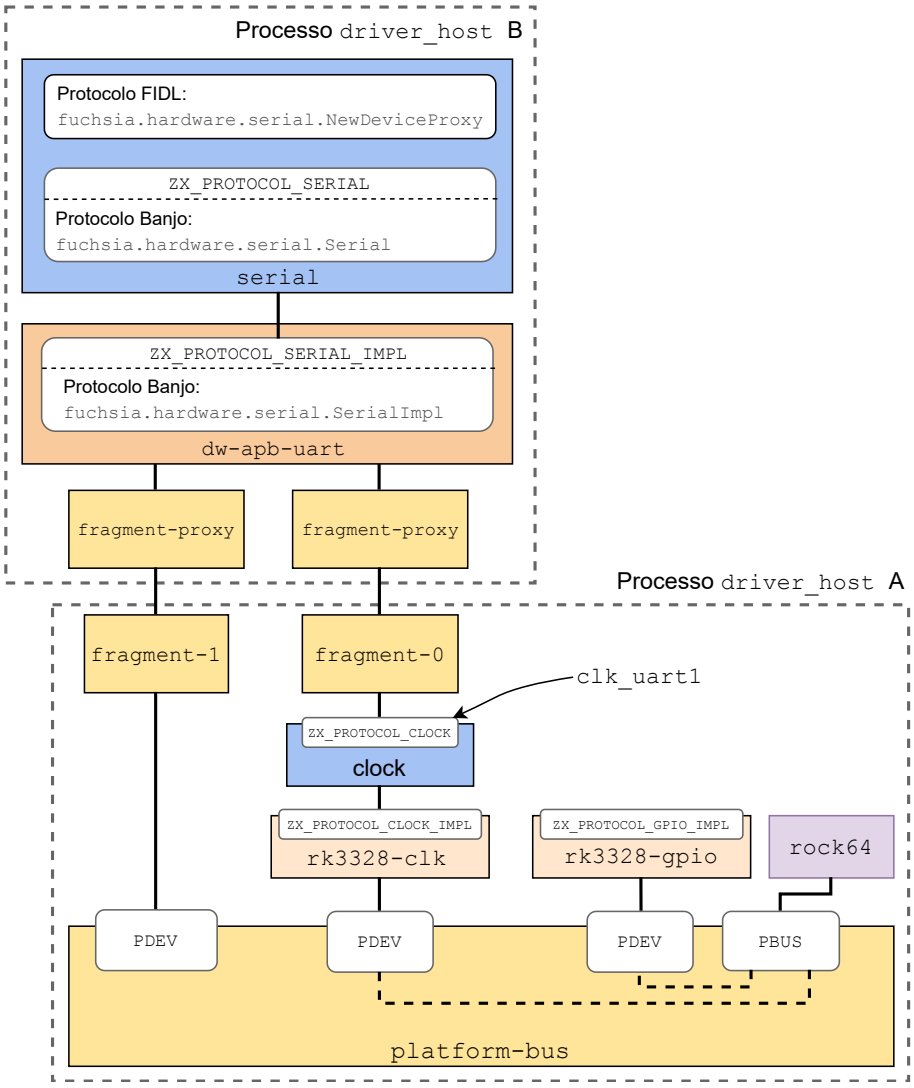
forma — i.e., usando a versão cliente do código gerado pelo compilador Banjo para o protocolo em questão — com a única diferença sendo a passagem de uma *string* C que especifica o fragmento para o qual a conexão vai ser criada.

Listagem 13: Recuperação dos fragmentos do *composite device* pelo `dw-apb-uart`.

```
ddk::PDev pdev;
std::optional<ddk::ClockProtocolClient> serial_clock

if (device_get_fragment_count(parent) > 0) {
 // ...
 ddk::ClockProtocolClient clock(parent, "serial-clock");

 if (clock.is_valid()) {
   serial_clock.emplace(clock);
 } //...
 //...
}
```

A Figura 6.6 ilustra todos os drives instanciados e seus respectivos processos `driver_hosts`. Os *drivers* `fragment` e `fragment-proxy` são fornecidos pelo *framework* de *drivers* e são instanciados automaticamente quando *composite devices* são usados. Eles têm um papel similar ao *driver* `platform-bus-proxy` na Figura 3.3, que é possibilitar a comunicação entre diferentes `driver_hosts`. Isso é necessário porque o protocolo Banjo, neste momento, suporta apenas apenas comunicação no mesmo processo. Esses *drivers* de fragmento têm suporte aos diversos protocolos do *framework* de *drivers* incluindo o protocolo `ZX_PROTOCOL_CLOCK` usado pelo *driver* `dw-apb-uart`.

Figura 6.6: *Drivers* e seus `driver_host`s.



Fonte: O Autor.

6.3 Testando o `dw-apb-uart`

Como mencionado na Seção 4.3, o produto '`bringup`' inclui um *shell* executando em um console criado por um componente do Fuchsia. É possível testar o *driver* desenvolvido aqui usando o comando `echo` que é um *builtin* do *shell* da seguinte maneira:

```
echo Hello World! > /dev/class/serial/000
```

Nessa linha de comando, `/dev` é diretório do *Device filesystem* discutido na Seção 3.8.8. Esse diretório é usado pelo componente `console-launcher` mencionado

na Seção 4.3[5]. O caminho `class/serial/000` se refere ao *driver* genérico `serial` carregado em cima do *driver* `dw-apb-uart` (Figura 6.6).

O *driver* `serial` sobrescreve as funções `open()`, `read ()` e `write()` cujas implementações padrão são normalmente fornecidas pelo `driver_host`. O subdiretório `class/serial/` na linha de comando anterior é um *handle* de diretório o qual fala o protocolo FIDL `fuchsia.io.Directory`. Este último têm a mensagem `Open()`[6]. Quem implementa o protocolo FIDL por trás do *handle* `class/serial/` é o `driver_manager`, o qual, ao receber a mensagem `Open()` com parâmetro `"000"`, retorna um *handle* apontando para o *device* publicado pelo *driver* `serial`. Isso causa a execução da função `open()` nesse *driver*, o que é mediado pelo `driver_host` no qual o *driver* executa. O *handle* referente ao arquivo `"000"` fala o protocolo `fuchsia.io.File` que é implementado pelo `driver_host` o qual direciona as mensages aos métodos correspondentes do *driver* em questão. O *shell* então envia a mensagem `Write()` nesse *handle*, o que executa a função `write()` no *driver* `serial`. Esta função, nesse *driver* faz com que a função `Write()` do *driver* `dw-apb-uart` seja chamada. Esse última parte do fluxo de execução está ilustrado na parte final do diagrama da Figura 3.2.

Outra forma de testar o *driver* é através da seguinte linha de comando:

```
dd if=/dev/class/serial/000 bs=1
```

O fluxo de execução é o mesmo, a diferença é que, no final, o programa `dd`[7] executado pelo *shell* fica enviando mensagens FIDL `Read()` no *handle* por trás do arquivo `000`. Esse comando bloqueia e fica imprimindo os bytes que chegam à porta serial.

Há ainda outra forma de testar, dessa vez utilizando o programa `serial-test`, que utiliza a mensagem `GetClass()` do protocolo FIDL `fuchsia.hardware.serial.Device`, que também é implementado pelo *driver* `serial` genérico. A partir disso, o `serial-test` usa as mensagens de `Read()` e `Write()` do protocolo FIDL `fuchsia.io`
`.File` para interagir com a porta serial.

Neste trabalho a operação de mudança do *baudrate* da porta serial foi testada

[5]Esse uso pode ser visto na linha 36 em <https://cs.opensource.google/fuchsia/fuchsia/+/master:src/bringup/bin/console-launcher/meta/console-launcher.cml;l=36>

[6]<https://cs.opensource.google/fuchsia/fuchsia/+/main:sdk/fidl/fuchsia.io/io.fidl;l=490>

[7]Essa versão do programa `dd` para o Fuchsia está disponível em <https://cs.opensource.google/fuchsia/fuchsia/+/master:src/storage/bin/dd>

indiretamente configurando diferentes frequências o *clock* da UART1 através do *driver* `rock64` que é um cliente do driver `rk3328-clk`. Um desses testes pode ser visto no arquivo `rock64-clk.cc` no *commit git* do Apêndice A.12.

7 CONSIDERAÇÕES FINAIS

Este trabalho buscou documentar os princípios, abstrações e mecanismos relacionados aos diversos componentes de hardware e software necessários para se compreender o funcionamento dos *drivers* do sistema operacional Fuchsia necessários para suportar a operação básica um periférico na Rock64.

Apesar da dificuldade inicial para realizar o *boot*, o *kernel* Zircon funcionou surpreendemente bem desde o primeiro *boot* completo. Considerando que o Zircon utiliza uma arquitetura de *microkernel* e que a arquitetura ARMv8 padroniza os principais componentes relacionados ao processador, isso não é algo tão inesperado.

Além do *boot* do *kernel* foram desenvolvidos 4 *drivers*, o que é relativamente pouco para o uso prático da Rock64. Entretando, dois desses *drivers* controlam os dois subsistemas principais do RK3328 que são essenciais para que a maioria dos outros *drivers* venham a ser desenvolvidos. Os drivers desenvolvidos neste trabalho não estão completos, mas já contém uma estrutura que pode ser facilmente estendida à medida que os *drivers* de outros dispositivos são desenvolvidos.

Existe ainda um outro aspecto importante no suporte à dipositivos que é uso de *Direct Memory Access* (DMA) a qual pode ser utilizada para operar muitos tipos de dispositivos de hardware com um maior *throughput*. Este trabalho, porém, não tratou desse assunto. Ainda assim, o *kernel* Zircon suporta as primitivas necessárias para implementar tal suporte[1].

[1] https://fuchsia.dev/fuchsia-src/concepts/drivers/driver_development/dma

REFERÊNCIAS

ARM. **ARM Architecture Reference Manual - Armv8, for Armv8-A architecture profile**. 2019. Disponível na Internet: <https://developer.arm.com/documentation/ddi0487/latest/>. Accessado em: Abril 2021.

ARM. **ARM Cortex-A Series Programmer's Guide for ARMv8-A**. [s.n.], 2019. Disponível na Internet: <https://developer.arm.com/documentation/den0024/latest/>. Accessado em: Abril 2021.

ARM. **Arm Generic Interrupt Controller - Architecture Specification - GIC architecture version 3 and version 4**. 2021. Disponível na Internet: <https://developer.arm.com/documentation/ihi0069/latest/>. Accessado em: Abril 2021.

AUTHORS, T. F. **Architecture Support**. 2021. Disponível na Internet: <https://web.archive.org/web/20210126120341/https://fuchsia.dev/fuchsia-src/get-started/get_fuchsia_source>. Accessado em: Abril 2021.

BELLONI, A. **Porting Linux on an ARM board**. 2015. Disponível na Internet: <https://bootlin.com/pub/conferences/2015/captronic/captronic-porting-linux-on-arm.pdf>. Accessado em: Abril 2021.

CLEMENT, G. **Your newer ARM64 SoC Linux check list!** 2016. Disponível na Internet: <https://elinux.org/images/1/18/ARM64_SoC_Linux_Support_Check-List.pdf>. Accessado em: Abril 2021.

CO., L. F. R. E. **Boot option**. 2019. Disponível na Internet: <https://web.archive.org/web/20200625113651/http://opensource.rock-chips.com/wiki_Boot_option>. Accessado em: Abril 2021.

CORTEX-A53 - Technical Reference Manual. ARM, 2013. Disponível na Internet: <https://developer.arm.com/docs/ddi0500/g>. Accessado em: Abril 2021.

MARX, A.-L. **The Zircon Kernel A Consideration of a Microkernel Approach and its Effects on Driver Development**. Dissertation (Master) — University of Pforzheim, Pforzheim - Germany, 2019.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design ARM Edition: The Hardware Software Interface**. [S.l.]: Morgan kaufmann, 2016.

PINE64. **ROCK64 Pi-2 Bus**. 2020. Disponível na Internet: <https://files.pine64.org/doc/rock64/ROCK64_Pi-220_and_Pi_P5+_Bus.pdf>. Accessado em: Abril 2021.

PINE64. **ROCK64 Wiki page**. 2020. Disponível na Internet: <https://web.archive.org/web/20210418011918/https://wiki.pine64.org/wiki/ROCK64>. Accessado em: Abril 2021.

ROCKCHIP, E. F. C. L. **RK3328 datasheet**. [s.n.], 2017. Disponível na Internet: <https://rockchip.fr/RK3328%20datasheet%20V1.2.pdf>. Accessado em: Abril 2021.

ROCKCHIP, E. F. C. L. **Rockchip RK3328 Technical Reference Manual Part1**. 2017. <http://opensource.rock-chips.com/images/9/97/Rockchip_RK3328TRM_V1.1-Part1-20170321.pdf>. Accessado em: Abril 2021.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. [S.l.]: John Wiley & Sons, 2018.

APÊNDICE A — LISTA DE *COMMITS GIT*

As seções a seguir listam os *commits* realizados durante o trabalho. Cada *commit* têm uma URL a partir da qual é possível ver o *diff* correspondente. Alguns capítulos fazem mais de um *commit*.

A Tabela A.1 contém um sumário com links para os *commits*.

Tabela A.1: Resumo dos *commits*.

***Commit* nro**	**Descrição**	**Link p/ Seção**	**Link externo**
01	Sobrescrita endereço FDT boot-shim	A.1	Link 01
02	Adiciona *board* no sistema de *build*	A.2	Link 02
03	Versão simples do *board driver*	A.3	Link 03
04	Primeia versão *driver* `rk3328-gpio`	A.4	Link 04
05	`rock64` "carrega"o *driver* `rk3328-gpio`	A.5	Link 05
06	Primeira versão *driver* `rk3328-clk`	A.6	Link 06
07	`rock64` "carrega"o *driver* `rk3328-clk`	A.7	Link 07
08	Cria o *driver* `dw-apb-uart`	A.8	Link 08
09	`rock64` "carrega"o *driver* `dw-apb-uart`	A.9	Link 09
10	Suporta *clocks* das UARTs no `rk3328-clk`	A.10	Link 10
11	Adapta `dw-apb-uart` para usar *clock*	A.11	Link 11
12	Suporta *clock* dos PLLs no `rk3328-clk`	A.12	Link 12

Fonte: O Autor.

O *commit* imediatamente anterior ao primeiro *commit* (i.e., o *commit* base) tem SHA `12575a48fb333e7871bf58fb153f33030463d775` e está disponível nas seguintes URLs:

- https://fuchsia.googlesource.com/fuchsia/+/12575a48fb333e78 71bf58fb153f33030463d775
- https://cs.opensource.google/fuchsia/fuchsia/+/12575a48fb333 e7871bf58fb153f33030463d775

A.1 Sobrescrita do endereço da FDT no boot-shim

https://github.com/csrpi/fuchsia/commit/178e8793fb9234bb549c341633d87c0fb0edeec4

```
[boot-shim] Allows overriding FDT address passed by bootloader

The `BOARD_FDT_MEMORY_ADDRESS` macro allows the overriding of the FDT
memory address passed by the bootloader.

This is convenient because it enables one to set this address
needed by the Zircon boot-shim in cases when the board bootloader
firmware (sucha as u-boot) doesn't do that in a easy way.

This can be done in a per-board basis as this macro can be defined
in the `boot-shim-config.h` associated with the board.

Change-Id: Ie2c6a0da6668b8de73a779fab238629f38c04c13
```

A.2 Adiciona a Rock64 no sistema de *build*

https://github.com/csrpi/fuchsia/commit/c4cbcb80893db40ff1f8ce25fbf97a33a40541f6

```
[bringup] Bringup of Rock64 board with the RK3328 SoC

Adds the `rock64` board to the build system.

Booting in a Rock64 board through an U-Boot with BL31 firmware
from ARM's TF-A lands into a shell. Just the debug UART (uart2)
is supported.

No board driver or anything else is available as of this commit.

Change-Id: I0519be705afc3e36f635539f822a9c042923f7c4
```

A.3 Adiciona um *driver* vazio para a placa

https://github.com/csrpi/fuchsia/commit/323867c8410b63e85234e26b201fe683307da7cb

```
[rock64] Create a simple board driver for the Rock64

The driver still doesn't bind/load any other driver, but the
bind library for PINE64 was added and used in the board driver
binding.

Other than that, this just tests the binding of the board
driver itself.

Change-Id: I096bb78fb484ed6919b7752d2ffd4b2794ec4106
```

A.4 Cria *driver* simples para *pin muxing*

https://github.com/csrpi/fuchsia/commit/012cf6f9f5c28766e1c169e26fa6471ac381e979

```
[rk3328-gpio] Create simple gpio driver for the RK3328 SoC

Adds a GPIO implementation driver for the Rockchip RK3328 SoC.
The driver implements just the
`fuchsia.hardware.gpioimpl.GpioImpl\SetAltFunction` protocol method. And
    ↪ it
implements it just for the `GPIO3_A4` and `GPIO3_A6` pins, because
they are the ones needed to make the RK3328 uart1 work.

No interrupts are supported, neither actual GPIO pin IO.

Besides that, it's created the `fuchsia.rockchip` bindc library.

Change-Id: I1a4222a16baf2a1d338b301f64f88691b4a97911
```

A.5 *Driver* da placa carrega *driver* de *pin muxing*

https://github.com/csrpi/fuchsia/commit/f489c7083eadd96bd6225cb80e74d74a5601c359

```
[rock64][rk3328-gpio] Board driver binds rk3328-gpio driver

Change the rock64 board driver so that the `rk3328-gpio` gets binded.

The initialization of the the child drivers is now made in its own
thread so that the board driver binding is performed without blocking.

Change-Id: I8002c4b88c3beab0075dad9a824ad4a7d08952ff
```

A.6 Primeira versão do *driver* da árvore de *clock*

https://github.com/csrpi/fuchsia/commit/31049ce965c2af945a1f7baeb1fee0a56816e3ae

```
[rk3328-clk] Create simple clock driver for RK3328 SoC

The driver manages just one clock: the UART1's clock.
And that is set up to the XIN24MHz oscillator in a hardcoded manner
when the driver is binding.

All other banjo `fuchsia.hardware.clockimpl.ClockImpl` protocol methods
    ↪ just
return `ZX_ERR_NOT_SUPPORTED`.

Change-Id: I2e51fb7666e83906ef1450615129d3647f0ee19f
```

A.7 *Driver* da placa carrega *driver* da árvore de *clock*

https://github.com/csrpi/fuchsia/commit/1b08ed8af9ab29ca655efe28db22f370ecb522c3

```
[rock64][rk3328-clk] Board driver binds rk3328-clk driver

Change-Id: I037598cbb48dceb248983445966b1e31a4f9cf0e
```

A.8 Cria *driver* `dw-apb-uart`

https://github.com/csrpi/fuchsia/commit/d391abdb58a74f255343a3bf404949109f2874fe

```
[dw-apb-uart] Create driver for DesignWare's APB UART

As the DesignWare's APB UART is based on the NS16550, this driver
is based on the uart16550 already available on the tree.

Change-Id: I9de4d2239447ac46eca8370e02ad8d7a26ee8f6e
```

A.9 *Driver* da placa carrega *driver* da porta serial

https://github.com/csrpi/fuchsia/commit/5ff19819eb32a251fc0631305ac40c10d5a8612b

```
[rock64][dw-apb-uart] Board driver binds dw-apb-uart driver for uart1

The board driver does the pin-muxing using the GPIO driver in order for
the uart1 serial port to work.

Uart1 is the other available UART pin on the Rock64 board pinout.

Change-Id: I82a64570bf47e4d08c5baefeb62668eff4a56e72
```

A.10 Suporte ao *clock* da porta serial na árvore de clock

https://github.com/csrpi/fuchsia/commit/1d556952d705b8e8fedfeb69ad82773ff3884393

```
[rk3328-clk][uart-clk] Adds UART clock support in the RK3328 clock driver

This also changes the `rock64` board driver in order to specify
a clock device corresponding to the UART1 clock. This indirectly
causes a device to be published for the clock signal.

Change-Id: Ibaaa7fc026363453f737f4b6590fced145110d0f
```

A.11 Adapta *driver* da porta serial para usar o *clock*

https://github.com/csrpi/fuchsia/commit/d7c8b94d482ffd6a2aaf99deca0224a2ac8bdb89

```
[dw-apb-uart][rock64] Adapt serial driver for binding to composite device

With this, the serial driver can make use of the serial clock
exported/published by the `rk3328-clk` driver, which it uses in order to
correctly set its baud rate config.

Change-Id: I3fd24cb2e83c4457fa5db69dc3ddff10f6be33a2
```

A.12 Adiciona suporte aos PLLs na árvore de *clock*

https://github.com/csrpi/fuchsia/commit/2f015594dcb4c13aa51eee305ad561078f1f9b7f

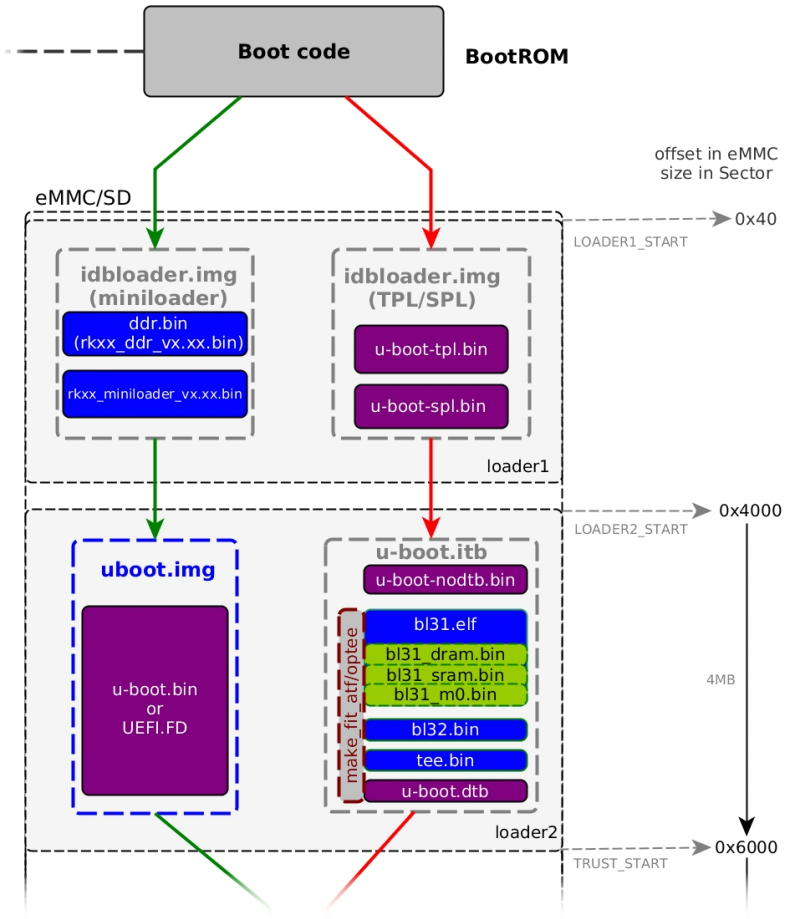
```
[rk3328-clk][pll-clk] Adds PLL clock support in the RK3328 clock driver

Change-Id: I6c045b042822413858d150ca8589415f816bec08
```

APÊNDICE B — COMPILAÇÃO DO FIRMWARE NECESSÁRIO PARA O *BOOT*

Como mencionado na Seção 4.2.1, um dos problemas encontrados para realizar o *boot* do Fuchsia fez necessário obter uma versão do firmware da Rock64 onde o *bootloader* U-Boot incluísse o comando `dcache` (e opcionalmente o `icache`).

O firmware da Rock64 usado neste trabalho é gerado a partir do U-Boot e do Trusted Firmware-A (TF-A) da ARM. Ambas versões usadas aqui são *upstream*, o que significa que são as versões oficiais e não um *fork* específico de algum fabricante ou comunidade de usuários/entusiastas.

Figura B.1: Firmwares alternativos para *boot* a partir de cartão SD/eMMC no RK3328.



Fonte: Adaptado de (CO., 2019).

Os firmwares padrão disponíveis para o *boot* a partir de um cartão SD/eMMC no RK3328 estão ilustrados no diagrama da Figura B.1. O fluxo de *boot* da esquerda utiliza imagens binárias fornecidas pela Rockchip, que é a fabricante do RK3328. As

imagens de firmware do fluxo à direita são obtidos a partir dos projetos mencionados anteriomente e representa as imagens utilizadas neste trabalho[1].

O diagrama na Figura B.1 ilustra o mesmo fluxo de execução dos estágios de *bootloaders* da Figura 2.5. A diferença é que aqui estão ilustrados as imagens (ou arquivos) dos estágios de *bootloaders* e as posições de um cartão SD/eMMC onde estão inicialmente armazenados.

No diagrama da Figura B.1, a imagem (ou arquivo) `idbloader.img` é um termo próprio da Rockchip e designa o estágio BL2. O `idbloader.img` do fluxo da direita é gerado a partir de módulos do U-Boot. O arquivo `u-boot.itb` é a imagem padrão gerada na compilação do U-Boot e contém, além da versão completa do U-Boot, o BL31. Esse último contém o *Secure Monitor* o qual executa o serviço PSCI mencionado na Seção 2.7. O PSCI é requerido pelo *kernel* Zircon para fazer o *boot*.

B.1 Obtenção do código fonte dos firmwares

O repositório git oficial do U-Boot e do TF-A estão disponíveis nos seguintes endereços:

- <https://github.com/u-boot/u-boot>
- <https://github.com/ARM-software/arm-trusted-firmware>

O código fonte deles pode ser baixado de várias formas, mas é recomendado utilizar a ferramenta git porque algumas versões recentes do U-Boot podem não funcionar. A seguinte sessão do *shell* no GNU/Linux mostra os comandos para para baixar esses repositórios.

```
$ mkdir rk3328-firmware

$ cd rk3328-firmware/
$ git clone 'https://github.com/ARM-software/arm-trusted-firmware'
$ git clone 'https://github.com/u-boot/u-boot'

# Os prximos comandos so necessrios para usar
# uma verso especfica do U-Boot.
$ cd u-boot
$ git pull --tags
$ git checkout tags/v2020.10 -b v2020.10-release
```

[1] É válido observar que é possível obter os mesmos firmwares a partir de diferentes forks. Ou seja, há uma certa *fragmentação* dos firmwares disponíveis.

B.2 Ferramentas necessárias para a compilação

Devido ao fato do U-Boot e do BL31 do TF-A serem escritos em linguagem C e seus sistemas de *build* utilizarem ferramentas padrão da GNU e do *kernel* Linux (`make`, `kconfig`, etc.), os programas necessários para compilá-los podem ser instalados facilmente em um sistema operacional GNU/Linux.

Porém, é importante lembrar que os firmwares compilados vão executar no SoC da Rock64, assim é necessário instalar o compilador apropriado para a arquitetura ARMv8. Para o caso do Ubuntu — que é a distribuição Linux utilizada neste trabalho — os seguintes comandos são suficientes:

```
$ sudo apt install build-essential gcc-aarch64-linux-gnu swig
```

B.3 Compilar U-Boot e o BL31 do TF-A

Depois de ter feito os passos da seção anterior (em especial o *checkout* de uma *release* específica do U-Boot), a sequência de comandos a seguir pode ser usada:

```
# Lembre de estar no diretrio `rk3328-firmware/'
$ cd rk3328-firmware

$ cd arm-trusted-firmware
$ make CROSS_COMPILE=aarch64-linux-gnu- PLAT=rk3328 DEBUG=1 bl31
# A compilao anterior gera o seguinte arquivo:
# build/rk3328/release/bl31/bl31.elf

$ cd ../u-boot
$ make CROSS_COMPILE=aarch64-linux-gnu- rock64-rk3328_defconfig
# O comando anterior gera o arquivo `.config' na raiz do diretrio
```

O arquivo `.config` gerado pelo último comando na sequência contém as diversas configurações que fazem o U-Boot gerar as images específicas para a Rock64. É necessário alterar o `.config` para fazer o comando `dcache` ser compilado e incluído no U-Boot. Para isso, basta procurar nesse arquivo a variável `CONFIG_CMD_CACHE` e mudá-la para `"y"`.

Feito isso, o seguinte comando gera as imagens finais:

```
$ make CROSS_COMPILE=aarch64-linux-gnu- \
      BL31=../arm-trusted-firmware/build/rk3328/debug/bl31/bl31.elf \
      all \
      u-boot.itb
```

Os principais arquivos gerado pelo último comando são o `u-boot.itb` e `idbloader.img` na raiz do diretório do U-Boot.

B.4 Gravando o firmware no cartão (micro) SD

É necessário gravar as imagens geradas anteriormente em um cartão SD, ou eMMC, de forma a conseguir executá-los na sequência de *boot* da Rock64.

O arquivo `/dev/sdb` a seguir se refere ao cartão SD inserido no computador, o qual pode variar entre diferentes distribuições Linux. Cuidado para não gravar **no dipositivo errado**!

Os valores `64` e `16384` correspondem aos offsets `0x40` e `0x4000` na Figura B.1, respectivamente.

```
$ sudo dd if=idbloader.img of=/dev/sdb seek=64
$ sudo dd if=u-boot.itb of=/dev/sdb seek=16384
$ sync # garante que os arquivos foram gravados
```

APÊNDICE C — AMBIENTE DE DESENVOLVIMENTO E MATERIAIS UTILIZADOS

O ambiente de desenvolvimento utilizado foi uma distribuição derivada do Ubuntu, mas qualquer outra distribuição GNU/Linux poderia ser usada desde que os programas necessários fossem instalados[1].

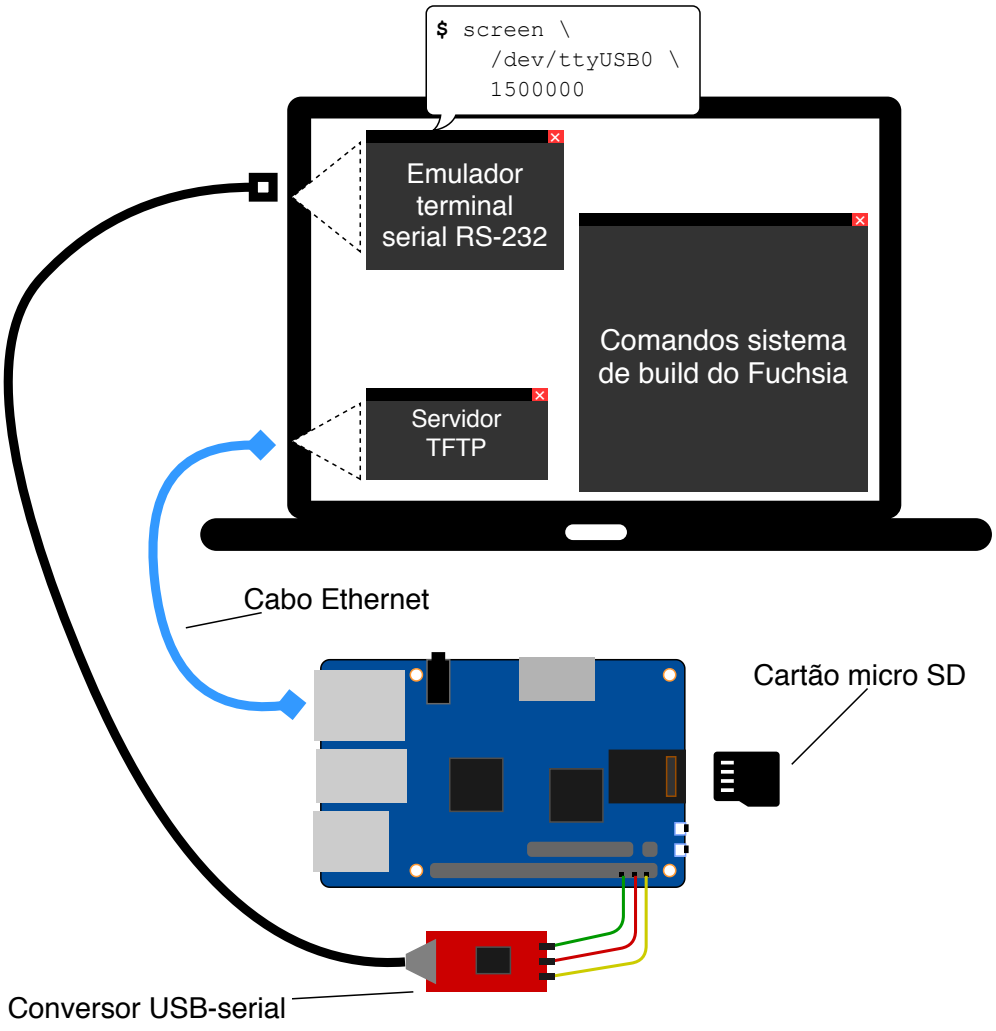
O código fonte do Fuchsia é bem grande e instruções sobre como baixá-lo estão disponíveis em (AUTHORS, 2021). Os pré-requesitos para baixar (e compilar) o código são o `git`, `curl` e `unzip` que podem ser instalados no Ubuntu. Todos os binários do *toolchain* necessário para compilar o Fuchsia estão incluído no download do repositório.

O diagrama na Figura C.1 ilustra a Rock64 e as duas principais conexões físicas com o computador de desenvolvimento. O conversor USB-serial possibilita o acesso ao *shell* do U-Boot, o que é essencial para a realização do *boot* do Fuchsia como feito na Seção 4.2. O cabo *Ethernet* também é essencial, pois é através dele que os arquivos ZBI, boot-shim e a FDT vazia são carregados na memória RAM da Rock64.

Devido à forma como este trabalho foi feito, é necessário ter o `screen` e o `dnsmasq` instalados. Ambos costumam já vir instalados em distribuições GNU/-Linux derivadas do Debian. O `dnsmasq` implementa um servidor TFTP, o qual é necessário para enviar os arquivos para a Rock64 através do link *Ethernet*. O `screen` é um emulador de terminal que permite utilizar o conversor USB-serial para acessar tanto o *shell* do U-Boot onde o script da Listagem 2 é enviado, quanto o *shell* do Fuchsia.

[1]Nas listas de email do Fuchsia, eventualmente acontecem problemas no *build* em outras distribuições e também no macOS. Mas esses problemas são transientes e logo são resolvidos pelos desenvolvedores do Fuchsia. O Windows, até o momento, não é suportado.

Figura C.1: Interconexões entre a Rock64 e o computador de desenvolvimento.



Fonte: O Autor.

Porte do microkernel Zircon para um Single Board Computer

César Pastorini, Marcelo Johann

{caspastorini,johann}@inf.ufrgs.br

Instituto de Informatica – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Resumo: *Sistemas operacionais de dispositivos como smartphones, tablets e smart TVs costumam ser baseados no kernel Linux. Esses dispositivos são baseados em System on Chips cujo suporte no kernel Linux geralmente é implementado pelo próprio fabricante e distribuídos em pacotes conhecidos como Board Support Packages. As modificações feitas no kernel não costumam ser documentadas de maneira sistemática e os artigos acadêmicos geralmente são baseados em SoCs sujo suporte no kernel Linux já foi implementado. Esse trabalho propõe a análise e implementação das mudanças necessárias para portar o microkernel Zircon do sistema operacional Fuchsia para um Single Board Computer. É feita uma breve análise do microkernel e de ferramentas e técnicas úteis para realização do trabalho.*

Abstract: *Operating systems of devices such as smartphones, tablets and smart TVs are usually based on the Linux kernel. These devices are based on Systems on Chips whose Linux kernel support is usually implemented by the manufacturer itself and distributed as Board Supported Packages. Modifications made to the Kernel are not usually documented in a systematic manner and academic papers are usually based in a kernel whose support is already present for the hardware being used. This work proposes the analysis and implementation of the changes required to port the Zircon microkernel from the Fuchsia operating system to a Single Board Computer. A brief analysis of the microkernel is made and some useful tools and tecniques for such a work are considered.*

1 Introdução

Sistemas embarcados complexos como smartphones, tablets, smart TVs, roteadores e smart home devices são baseados em System on Chips (SoCs). A grande maioria dos sistemas operacionais para o usuário final desses sistemas embarcados são baseados no kernel Linux [1, 2, 3, 4] e, por conta disso, esse é um dos primeiros softwares a ser portado para um novo SoC.

O kernel Linux se tornou um software estável e maduro implementando vários mecanismos para facilitar o porte para outras arquiteturas, SoCs e dispositivos. Como na prática o Linux é importante para o uso de um novo dispositivo, normalmente o próprio fabricante faz as modificações necessárias no kernel. Entre essas mudanças estão o desenvolvimento de drivers e modificações no próprio kernel [5].

O sistema operacional Fuchsia [6], diferentemente do Linux é baseado em um microkernel, o qual implementa uma API para o nível de aplicação mais moderna e não baseada em POSIX [7]. Por não ser tão maduro, e também não suportar tantas arquiteturas e dispositivos quanto o Linux, esse sistema representa uma boa oportunidade para investigar as mudanças necessárias para efetuar um porte para um sistema embarcado moderno.

Tendo isso em mente, esta proposta de Trabalho de Conclusão de Curso tem como objetivo analisar o trabalho necessário para portar o microkernel Zircon para um Single Board Computer (SBC) com um SoC ARM. Para isso, será analisada a arquitetura do microkernel Zircon, kernel do Fuchsia. Também será analisada a arquitetura dos SoCs com processadores ARM, bem como os requisitos para software de baixo nível (*bare metal*) para tornar o uso de um SoC possível na prática.

1.1 Systems on Chips (SoC)

SoCs são dispositivos que além do processador contém uma variedade de outros componentes de hardware como controlador de memória, coprocessadores, GPU, conversores, modems e outros dispositivos de entrada e saída. Todos esses dispositivos são integrados em um único chip. A Figura 1 contém um exemplo de um SoC e seus dispositivos integrados.

O hardware de dispositivos para consumidores finais como smartphones, tablets e smart TVs geralmente são baseadas em SoCs. Esses componentes possuem um alto nível de integração o que torna possível um alto poder computacional e um baixo consumo energético, estabelecendo, assim, a base de smartphones e dispositivos afins [9]. A arquitetura e plataforma de hardware desses sistemas embarcados é definida principalmente pela arquitetura dos processadores contidos no SoC [10, 11].

Um ponto em comum dos SoCs da maioria dos sistemas embarcados, como smartphones e tablets, é que estes utilizam processadores que implementam a arquitetura ARM [11]. Por exemplo, a grande maioria dos modelos de smartphones dos fabricantes que mais vendem atualmente utilizam SoCs com processadores ARM Cortex A [12, 13, 14].

Além do conjunto de instruções (ISA), a arquitetura ARM também especifica alguns componentes padrão como gerenciador de interrupções (GIC) [15], *system generic timer* e gerenciador de memória (MMU) [11]. Essas especificações de componentes fundamentais facilitam o desenvolvimento e a portabilidade de softwares de baixo nível mesmo havendo uma grande variedade de processadores e SoCs que implementam a arquitetura ARM. Além disso, a arquitetura ARM utiliza o método de Memory Mapped IO (MMIO) para fazer a entrada e saída nos processadores [11]. No final, um SoC com processadores que implementam a arquitetura ARM expõe os componentes do sistema em um espaço de memória único; a memória RAM, por exemplo, é apenas mais um componente que ocupa uma certa região nesse espaço.

A Figura 2 mostra o mapa de memória do SoC da Figura 1. No caso desse SoC, o software iria interagir com o dispositivo de *watchdog* por meio de leituras e escritas comuns nas posições de memória entre os endereços `FF1A0000` e `FF1AFFFF`. A semântica de cada posição desse intervalo de memória depende do dispositivo específico de

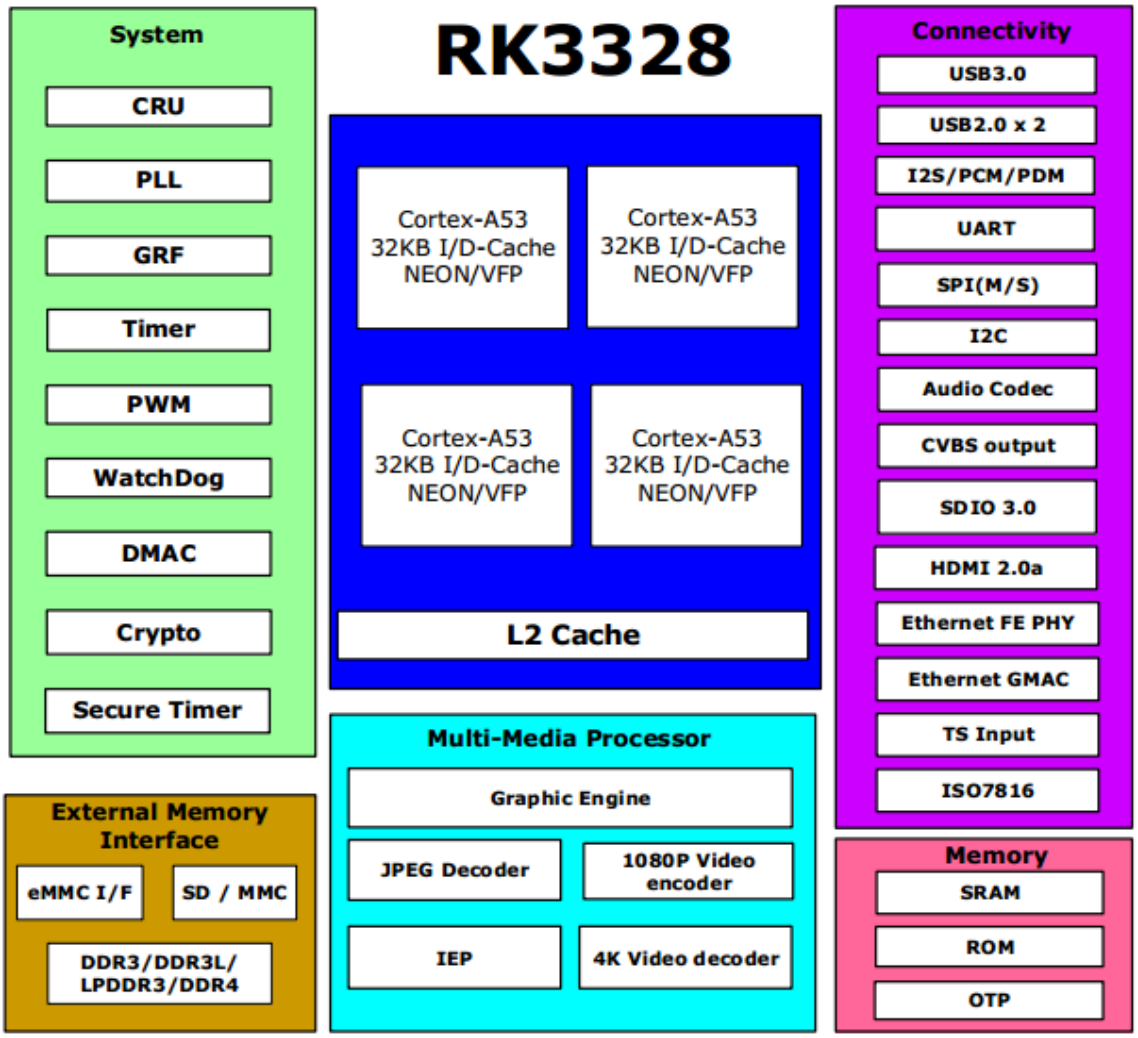


Figura 1: Componentes do SoC RK3328 da Rockchip (retirado de [8]). Além da CPU de 4 núcleos, há muitos outros dispositivos de IO integrados ao chip. No caso desse SoC, a memória RAM (*dynamic* RAM) não está integrada no chip.

watchdog que esse SoC utiliza.

1.2 Plataformas de desenvolvimento e Single Board Computers (SBCs)

Na prática um SoC não é suficiente para um sistema de desenvolvimento completo. São necessários outros periféricos e toda a infraestrutura de componentes elétricos, pinos, conectores etc. Por causa disso, ao lançar um novo SoC, os fabricantes utilizam uma placa de desenvolvimento com todos os componentes que normalmente estariam integrados ao produto final para o qual o SoC foi projetado. Por exemplo, para o SoC Snapdragon 855 da Qualcomm é disponibilizado o kit [17] (Figura 3) e a MediaTek disponibiliza o [18].

Paralelo à isso, existem as *Single Board Computers* (SBCs), utilizadas para desenvolvimento por estudantes e entusiastas de hardware e software, além de usos comerciais. Elas são muito mais acessíveis e, por conta disso, costumam ter uma comunidade associ-

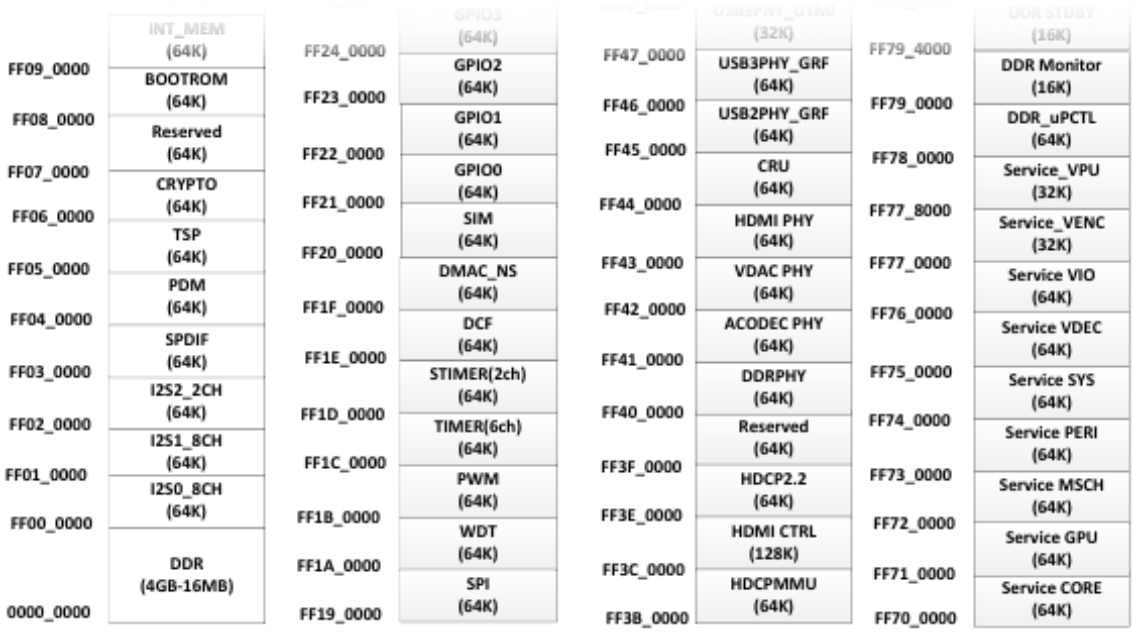


Figura 2: Parte do mapa de memória do SoC RK3328 da Rockchip (adaptado de [16]). No caso desse SoC, a memória RAM começa no endereço zero e vai, no máximo, até `0xFFFF0000` (4 GiB).

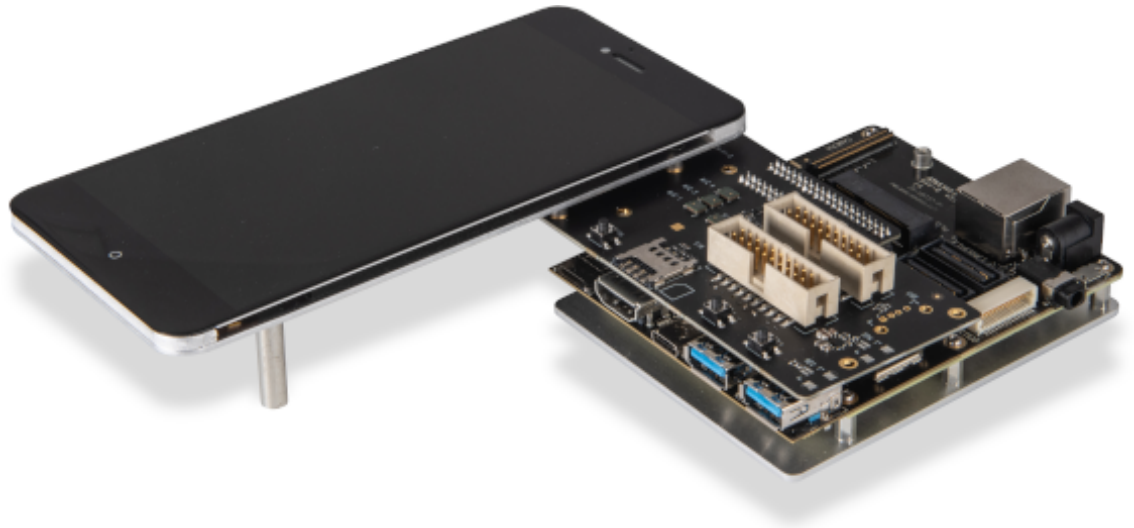


Figura 3: Kit de desenvolvimento com um SoC Snapdragon 855 e display LCD com painel touch conectado (retirado de [17]).

dada. Os SBCs também costumam ser baseados em SoCs que implementam a arquitetura ARM e, assim, possuem uma arquitetura muito semelhante aos kits de desenvolvimento dos dispositivos para usuários finais relacionados. Talvez o exemplo mais conhecido seja o Raspberry Pi (Figura 4).

Em relação ao software de baixo nível, o fabricante também disponibiliza um porte do kernel Linux, junto com outras modificações e drivers em um conjunto conhecido como *Board Support Package* (BSP). Com o kernel Linux do BSP disponível, é possível

Figura 4: Raspberri Pi 4 Model B, baseado no SoC BCM2711 da Broadcom, é a versão mais recente dessa linha de Single Board Computers.

portar outros softwares. Por exemplo, para portar o Android, é aplicado um conjunto adicional de patches que implementam as funcionalidades específicas do Android no kernel Linux [1, 19].

Porém, não é sempre que esses placas de desenvolvimento para SoCs utilizados em produtos comerciais (como smartphones) são disponibilizadas ao público. Nesses casos os fabricantes apenas disponibilizam esses kits de desenvolvimento para equipes internas ou parceiros. Além disso, esses SoCs e kits de desenvolvimento não costumam ter muita documentação disponível ao público, havendo apenas o código fonte do kernel Linux para tal.

2 Introdução ao Fuchsia

O sistema operacional Fuchsia ficou conhecido em 2016 através de um conjunto de repositórios git públicos. O primeiro commit no repositório é de junho de 2016. O microkernel do Fuchsia era inicialmente chamado Magenta e mais tarde foi renomeado para Zircon[1]. O Zircon é um *fork* do Little Kernel (LK), um pequeno sistema operacio-

[1]Commit em que a mudança de nome foi realizada:
<https://fuchsia.googlesource.com/fuchsia/+/f3e2126c8a8b2ff64ca6cb7818f0606ceb5f889a>

nal para sistemas embarcados usado, por exemplo, como bootloader padrão do Android [20]. Ao longo do desenvolvimento, o *fork* foi sendo extendido com abstrações comuns à sistemas operacionais de propósito geral como threads e processos, além das *features* desenvolvidas para o Fuchsia propriamente. Entretanto, atualmente o Zircon ainda tem alguns remanescentes da arquitetura de código do LK [21]. Ele suporta processadores de 64 bits ARM e x86_64 até o momento [22].

O Fuchsia é um sistema com um modelo de segurança baseada no conceito de *capabilities*. As *capabilities* são representadas através da associação de um conjunto de *rights* à um *handle*. *Handles* são utilizados para representar os diversos objetos (threads, processos, memória virtual, canais de comunicação IPC, etc.) que o kernel implementa [23, 24]. Por conta disso, a grande maioria das chamadas de sistema (*syscalls*) do Zircon agem sobre *handles* que referenciam objetos do kernel [25].

Objeto do kernel	**Chamada de sistema (*syscall*)**

Channel	`zx_channel_create()`
	`zx_channel_read()`
	`zx_channel_write()`
	`zx_channel_call()`

Socket	`zx_socket_create()`
	`zx_socket_read()`
	`zx_socket_write()`
	`zx_socket_shutdown()`

Process	`zx_process_create()`
	`zx_process_read_memory()`
	`zx_process_write_memory()`
	`zx_process_start()`
	`zx_process_exit()`

Thread	`zx_thread_create()`
	`zx_thread_read_state()`
	`zx_thread_write_state()`
	`zx_thread_start()`
	`zx_thread_exit()`

Virtual Memory Object (VMO)	`zx_vmo_create()`
	`zx_vmo_read()`
	`zx_vmo_write()`
	`zx_vmo_op_range()`
	`zx_vmo_set_cache_policy()`

Timer	`zx_timer_create()`
	`zx_timer_set()`
	`zx_timer_cancel()`

Tabela 1: Alguns objetos que o kernel implementa junto com as chamadas de sistema correspondentes. Cada chamada de sistema contém um link apontando para a documentação associada.

Sistemas operacionais baseados em microkernels fazem uso extensivo de Inter Pro-

cess Communication (IPC). Para facilitar a utilização deste mecanismo, foi criada a *Fuchsia Interface Definition Language* (FIDL), que é uma *Interface Definition Language* (IDL) para abstrair e facilitar a implementação de IPC. Essa IDL, e o compilador associado, fazem a geração automática de código para que seja possível fazer IPC de uma maneira fácil e robusta mesmo que os programas (e os processos correspondentes) tenham sido escritos em linguagens de programação diferentes [26] (Figura 5).

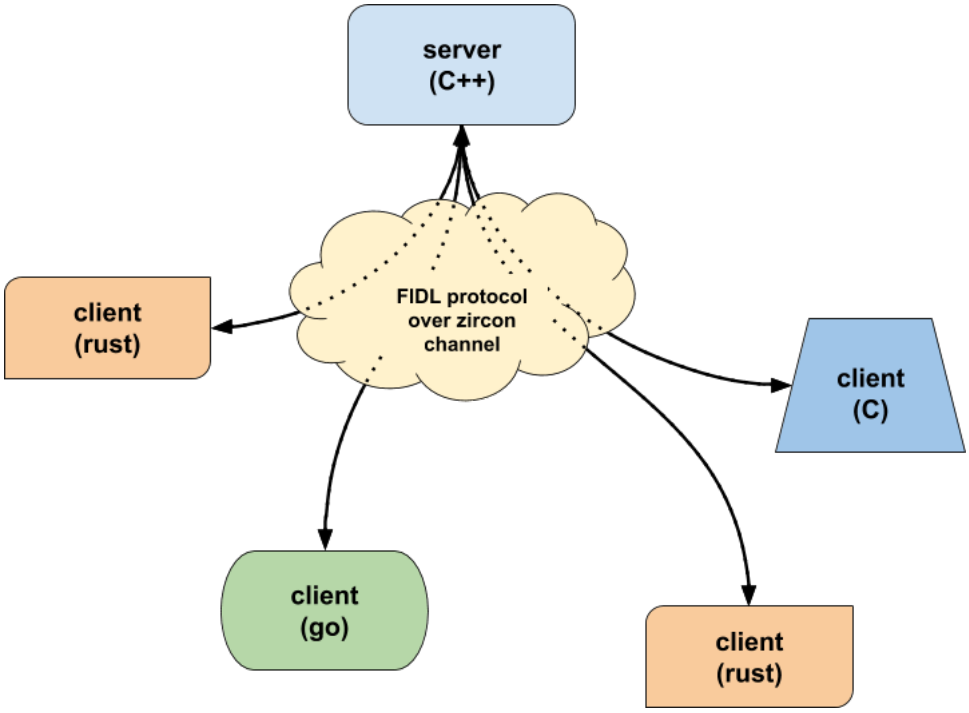


Figura 5: Diagrama ilustrando a utilização de FIDL para realizar IPC. O protocolo especificado através da linguagem FIDL, junto com o compilador e *bindings* associados, faz possível a comunicação simples e robusta entre processos mesmo que os programas correspondentes sejam escritos em linguagens de programação distintas (retirado de [27]).

2.1 Código fonte

O código fonte do Fuchsia[2] contém a maioria dos componentes do sistema como kernel, drivers e aplicações que implementam serviços fundamentais (ex.: pilha TCP/IP, sistemas de arquivos, primitivas para interface de usuário). O projeto utiliza a ferramenta Jiri[3] para gerênciar os múltiplos sub-repositórios git contidos no repositório principal. Além do código fonte, o Jiri também faz o download dos *toolchain*s suportados (GCC e Clang) e também do emulador QEMU.

O kernel, Zircon, é escrito em C, C++17 e assembly. É utilizado um subconjunto estrito do C++; não sendo permitido, por exemplo, o uso da biblioteca padrão de templates (STL), *exceptions* e *virtual inheritance* [28].

[2]Disponível em <https://fuchsia.googlesource.com/fuchsia/>
[3]Disponível em <https://fuchsia.googlesource.com/jiri>

Por ser baseado em um microkernel e ter uma arquitetura modular, os diversos serviços do Fuchsia são divididos em componentes. Além disso, como mencionado, utilizando a FIDL para realizar IPC permite que os componentes sejam escritos em, tecnicamente, qualquer linguagem de programação [29]. Com exceção do kernel, a maioria dos componentes são escritos em C++ ou Rust, mas também é utilizado Dart para os componentes de interface gráfica e Go para alguns outros componentes.

A forma de fazer o download do código fonte é através de um *script*[4] que, dentre outras coisas, baixa o Jiri que, por sua vez, faz o clone do repositório `git` principal junto dos múltiplos sub-repositórios git contidos no principal [30].

2.2 Sistema de *build*

A ferramenta de build utilizada no repositório é o GN e o Ninja que, juntos, são semelhantes ao GNU `make` [31]. É utilizado um extenso conjunto de arquivos `.gn` especificando as dependências entre arquivos e as várias informações relevantes ao build. Porém, o build em si é executado através de chamadas ao script `fx` ao invés de chamar o `gn` e `ninja` diretamente.

O sistema de build é dirigido por duas configurações fundamentais que, combinadas, especificam o conteúdo dos artefatos finais do build [32]. A primeira, `board`, especifica a dispositivo e plataforma de hardware na qual os binários da imagem final devem executar. Assim, essa configuração acaba definindo a arquitetura dos binários gerados e também seleciona o conjunto específico de drivers básicos para o hardware alvo. Um valor possível dessa configuração é '`qemu-arm64`', que significa que o código gerado é para a arquitetura ARM de 64 bits e o dispositivo de hardware é uma máquina virtual emulada pelo QEMU. A segunda configuração, `product`, especifica o conjunto de softwares de aplicação para serem incluídos no build e imagem final. Por exemplo, o produto '`bringup`' incluí o conjunto mínimo possível de softwares para que o sistema consiga *bootar* no dispositivo de hardware especificado pela `board`.

Assim, além de algumas ferramentas para usar no host, e dos próprios componentes do Fuchsia, o sistema de build também gera os artefatos finais que serão executados no hardware alvo. Destes arquivos, o mais importante é um arquivo no formato ZBI (*Zircon Boot Image*[5]). O arquivo ZBI é um *container* para uma variedade de itens de dados. Entre eles há um item para o kernel, o qual contém a imagem binária do kernel.

Outro arquivo importante é o `boot-shim` que funciona como um estágio subsequente a um *bootloader*. Uma vez carregados na memória RAM do dispositivo, ele tem a função de extrair a imagem binária do kernel do arquivo ZBI e fazer um *jump* para ele iniciando, assim, a execução do Zircon.

[4] Ver seção 'Download Fuchsia source' em
<https://fuchsia.dev/fuchsia-src/get-started/get_fuchsia_source#download-fuchsia-source>
[5] Arquivo de cabeçalho em linguagem C com a especificação do formato ZBI:
<https://fuchsia.googlesource.com/fuchsia/+/master/zircon/system/public/zircon/boot/image.h>

3 Proposta do Trabalho de Graduação

Apesar de existir uma certa quantia de artigos na literatura sobre o porte de sistemas de baixo nível, como o kernel de um sistema operacional, muitas dessas publicações focam apenas no kernel Linux. Elas descrevem o processo de obtenção, aplicação dos patches e compilação de um kernel cujo suporte à arquitetura alvo já foi implementado [33, 34]. Além disso, esses artigos foram publicados a um certo tempo e não há muitos exemplos para processadores e SoCs ARM mais modernos como os utilizados em Single Board Computers e dispostivos como smartphones.

Nesse contexto do porte de um kernel, as Single Board Computers são ideais pois são muito semelhantes em termos de arquitetura à smartphones e dispositivos afins. Elas geralmente possuem um tempo de vida de produto muito maior que os kits de desenvolvimento específicos de um determinado SoC e também possuem documentação aberta e abrangente que incluí esquemáticos elétricos da placa e manual dos SoCs. Além disso, as Single Board Computers também costumam ter suporte de bootloaders mais completos como o u-boot, o que facilita o ciclo de desenvolvimento para o porte de um kernel. E, como mencionado na Seção 1.2, a principal vantagem é a comunidade associada, que pode fornecer uma ajuda essencial e que também será beneficiada com o suporte de um novo kernel de sistema operacional.

Com isso em mente, esse trabalho propõe investigar e realizar as mudanças necessárias para portar o kernel Zircon do sistema operacional Fuchsia para um Single Board Computer com SoC que implemente a arquitetura ARM de 64 bits (`armv8` ou `arm64` na nomenclatura Linux). Exemplos podem ser o Raspberry Pi (Figura 4) ou a Rock64 (Figura 6) [35]. Esse trabalho, se assemelha ao porte do sistema operacional Choices apresentado em [36]. A diferença importante é que o kernel Zircon já suporta a arquitetura ARM de 64 bits, o que significa que será necessário fazer as mudanças para que este suporte um novo SoC e, correspondentemente, uma nova Single Board Computer.

3.1 Métodos

Como o trabalho proposto é o porte do Zircon para um novo dispositivo de hardware, é natural que seja definido uma nova `board` no sistema de build do Fuchsia (Seção 2.2). O novo valor possível para essa configuração indicará o Single Board Computer escolhido para o porte. Já a configuração `product` não necessita da definição de novo valor ou modificação, pois o item '`bringup`' serve exatamente para o caso de um porte; isto é, para o suporte mínimo em um dado hardware.

Também será muito útil poder utilizar a UART (*Universal Asyncronous Receiver-Transmiter*) do dispositivo de hardware. A maioria dos sistemas embarcados costumam ter esse dispositivo de entrada e saída (ver, por exemplo, a parte de '*Connectivity*' no diagrama do SoC da Figura 1). A UART faz com que o código executando no dispositivo consiga enviar caracteres para um pino na placa, o que torna possível instrumentar o código. Os chips de UART nesses sistemas embarcados costumam ter uma operação simples para o envio de mensagens; depois de uma pequena configuração, a operação consiste basicamente na escrita do caractere ASCII em algum registrador mapeado na memória. Porém, para fazer essa comunicação na prática é necessário utilizar um con-

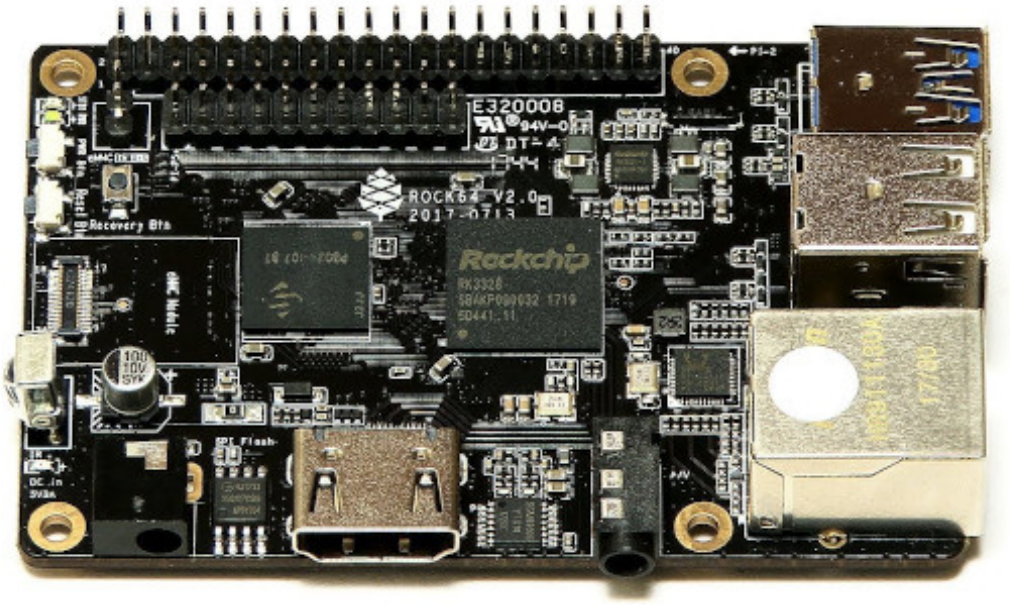


Figura 6: Rock64, Single Board Computer com SoC RK3328 da Rockchip.

versor UART-USB, já que atualmente os computadores PC não costumam vir com portas seriais. Um exemplo de tal conversor é o da Figura 7.

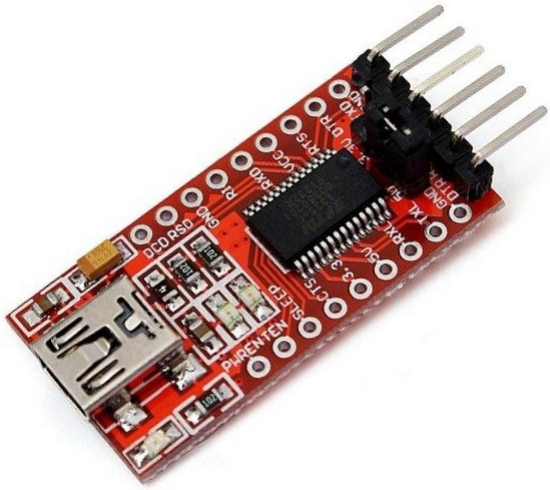


Figura 7: Exemplo de conversor USB-serial para fazer *debugging* do software executando no sistema embarcado [37, 38].

Outra ferramenta importante será o emulador QEMU, um emulador para diversas arquiteturas e dispositivos de hardware [39]. Idealmente, seria possível realizar o porte para o hardware alvo utilizando uma emulação desse hardware no QEMU. Porém, não existe uma emulação das Single Board Computers mais atuais. Além disso, não necessariamente um kernel que funcione na máquina emulada funcionaria na máquina real; podem existir imperfeições na emulação como bugs e mecanismos simplesmente não emulados.

Apesar disso, o QEMU é muito útil porquê existe a máquina `virt` que é de especial interesse para esse projeto por emular uma máquina ARM de 64 bits e ser ativamente suportada pelos desenvolvedores do Fuchsia (há uma `board` específica para essa máquina do QEMU). Com isso, será possível utilizar os mecanismos de *debugging* do QEMU para estudar o funcionamento do código [40].

O uso do `u-boot` também será muito útil pois, além de ser suportado pela maioria dos Single Board Computers, ele implementa um *shell* com diversos comandos. Um desses comandos permite o download de arquivos para a memória RAM via protocolo TFTP através do link Ethernet[6]. Ele também suporta a execução de scripts, isto é, uma sequência de comandos. Então a ideia é desenvolver um script que agilize o ciclo de testes no hardware pois o processo normal, através da gravação de arquivos em cartão micro SD, é mais lento.

O u-boot também tem comandos que lidam uma estrutura de dados chamada FDT (*Flattened Device Tree*), que contém diversas informações sobre os componentes de hardware e a topologia em que estão conectados uns aos outros. Essa estrutura de dados (e formato associado) é um padrão[7] utilizado por kernels de outros sistemas operacionais como Linux e FreeBSD para obter essas informações sobre a plataforma em que estão rodando. Como o boot-shim e o Zircon também fazem uso dela, ter um bootloader com suporte a isso pode ser importante.

```
boot_shim: hi there!
Setting RAM base and size device tree value: 0000000040000000 0000000080000000
Kernel at 0000000048000000 to 00000000481d4000 reserved 0000000000072940
ZBI at 0000000048000000 to 000000004a4e1e40
Splitting kernel len 0000000000161680 from ZBI len 00000000023807a0
Kernel to 000000004a4f0000
ZBI to 0000000048000000
Kernel container length 00000000001616a0 ZBI container length 0000000002380780
Entering kernel at 000000004a58c52c with ZBI at 0000000048000000
[00000.000] 00000:00000> PMM: boot reserve add [0x4a4f0000, 0x4a6c3fff]
[00000.000] 00000:00000> mem_arena.base 0x40000000 size 0x8000000
[00000.000] 00000:00000> overriding mem arena 0 base from FDT: 0x40000000
[00000.000] 00000:00000> overriding mem arena 0 size from FDT: 0x80000000
[00000.000] 00000:00000> detected GICv3
[00000.000] 00000:00000> PSCI version 0.2
[00000.000] 00000:00000> arm generic timer freq 62500000 Hz
[00000.039] 00000:00000> cntpct_per_ns: 00000000.1000000000000000
[00000.039] 00000:00000> ns_per_cntpct: 00000010.0000000000000000
[00000.039] 00000:00000> test_time_conversion_check_result:221: FAIL, off by 72057594037927936
[00000.039] 00000:00000> reserving ramdisk phys range [0x48000000, 0x4a380fff]
[00000.039] 00000:00000> PMM: boot reserve add [0x48000000, 0x4a380fff]
[00000.082] 00000:00000> PMM: boot reserve marking WIRED [0x48000000, 0x4a380fff]
[00000.085] 00000:00000> PMM: boot reserve marking WIRED [0x4a4f0000, 0x4a6c3fff]
```

Figura 8: Saída de log do `boot-shim` e Zircon na máquina '`virt`' do QEMU durante o boot.

Finalmente, observando o log de boot do Zircon na máquina virtual do QEMU (exemplo na Figura 8) e realizando um estudo preliminar do código fonte, é possível ver que a inicialização do mesmo se dá através da execução de uma sequência de *hooks*. Esses hooks, que são funções C/C++, são chamados em uma ordem especícica para inicializar

[6] Ver comando `tftpboot` em <https://www.denx.de/wiki/view/DULG/UBootCmdGroupDownload>
[7] Mais informações disponíveis em <https://www.devicetree.org/>

subsistemas e componentes de hardware fundamentais para a operação do kernel. Alguns deles, principalmente os primeiros, são específicos da arquitetura (nesse caso, `arm64`). A Tabela 2 mostra esses hooks na ordem em que são executados. Os hooks mostrados nessa tabela são, na verdade *labels* das funções C/C++ que são de fato executadas. Os hooks também podem ser diferentes dado um kernel compilado para outras arquiteturas, como `x86-64`.

1. `global_prng_seed`	14. `pmm`
2. `late_update_reg_procs`	15. `dpc`
3. `elf_build_id`	16. `arm64_perfmon`
4. `version`	17. `debuglog`
5. `arm_resource_init`	18. `platform_dev_init`
6. `console`	19. `kcounters`
7. `platform_init_pre_thread`	20. `ktrace`
8. `platform_postvm`	21. `finalize_root_resource_filter`
9. `pmm_fill`	22. `kernel_shell`
10. `percpu_heap_init`	23. `userboot`
11. `global_prng_thread_safe`	24. `pager_init`
12. `global_prng_reseed`	25. `scanner_init`
13. `libobject`	

Tabela 2: Hooks executados durante a inicialização do Zircon na máquina virtual '`virt`' com arquitetura ARM de 64 bits do QEMU.

4 Conclusão

Sistemas embarcados complexos como smartphones, tablets e smart TVs são baseados em SoCs. A grande maioria utiliza sistemas operacionais baseados no kernel Linux, incluindo o Android. O suporte a esses SoCs é, em geral, implementado pelos próprios fabricantes e as modificações necessárias costumam não ser documentadas em uma forma sistemática. Os trabalhos acadêmicos costumam utilizar SoCs cujo suporte ao kernel Linux já foi implementado. Esse trabalho propõe suprir essa lacuna de conhecimento, investigando e documentando as mudanças necessárias no microkernel Zircon do sistema operacional Fuchsia para que este suporte um Single Board Computer moderno.

Referências

1 SHANKER, A.; LAI, S. Android porting concepts. In: IEEE. *2011 3rd International Conference on Electronics Computer Technology*. [S.l.], 2011. v. 5, p. 129–133.

2 ALAM, I.; KHUSRO, S.; NAEEM, M. A review of smart tv: Past, present, and future. In: IEEE. *2017 International Conference on Open Source Systems & Technologies (ICOSST)*. [S.l.], 2017. p. 35–41.

3 JANCZUKOWICZ, E. Firefox os overview. 2013.

4 ROKU, I. *Development environment overview*. 2020. Disponível em: <https://developer.roku.com/en-gb/docs/developer-program/getting-started/architecture/dev-environment.md>.

5 SEMINAR "Porting Linux on an ARM board". 2015. Disponível em: <https://bootlin.com/pub/conferences/2015/captronic/captronic-porting-linux-on-arm.pdf>.

6 AUTHORS, T. F. *Fuchsia overview*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/concepts>.

7 AUTHORS, T. F. *Fuchsia's libc*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/concepts/system/libc>.

8 ROCKCHIP. *RK3328 Wiki*. 2017. Disponível em: <http://opensource.rock-chips.com/wiki_RK3328>.

9 BENNETT, P. *The why, where and what of low-power SoC design*. 2004. Disponível em: <https://www.eetimes.com/the-why-where-and-what-of-low-power-soc-design/>.

10 FURBER, S. B. *ARM system-on-chip architecture*. [S.l.]: pearson Education, 2000.

11 ARM Architecture Reference Manual - Armv8, for Armv8-A architecture profile. ARM, 2019. Disponível em: <https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf>.

12 COUNTERPOINT. *Global Smartphone Market Share: By Quarter*. 2019. Disponível em: <https://www.counterpointresearch.com/global-smartphone-share/>.

13 STATCOUNTER. *Mobile Vendor Market Share Worldwide Nov 2018 - Nov 2019*. 2019. Disponível em: <https://gs.statcounter.com/vendor-market-share/mobile>.

14 STATISTA. *Global market share held by leading smartphone vendors from 4th quarter 2009 to 3rd quarter 2019*. 2019. Disponível em: <https://www.statista.com/statistics/271496/global-market-share-held-by-smartphone-vendors-since-4th-quarter-2009/>.

15 ARM Generic Interrupt Controller - Architecture Specification - GIC architecture version 3 and version 4. ARM, 2015. Disponível em: <https://static.docs.arm.com/ihi0069/e/Q1-IHI0069E_gic_architecture_specification_v3.1_19_01_21.pdf>.

16 Fuzhou Rockchip Electronics Co., Ltd, 2017. Disponível em: <http://opensource.rock-chips.com/images/9/97/Rockchip_RK3328TRM_V1.1-Part1-20170321.pdf>.

17 TECHNOLOGIES, I. Q. *Snapdragon 855 Mobile Hardware Development Kit*. 2019. Disponível em: <https://developer.qualcomm.com/hardware/snapdragon-855-hdk>.

18 LABS, M. *MediaTek X20 Development Board*. 2019. Disponível em: <https://labs.mediatek.com/en/platform/mediatek-x20#HDK>.

19 PUNDIR, A. *Android Common Kernel and Out of Tree Patchset*. Disponível em: <https://www.youtube.com/watch?v=pD8koS4kwFE>; <https://schd.ws/hosted_files/elciotna18/ac/Android%20common%20kernel%20and%20out%20of%20tree%20patchset.pdf> .

20 SWETLAND, B. et al. *Introduction - littlekernel/lk Wiki*. 2020. Disponível em: <https://github.com/littlekernel/lk/wiki/Introduction>.

21 AUTHORS, T. F. *Zircon and LK*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/concepts/kernel/zx_and_lk>.

22 AUTHORS, T. F. *Architecture Support*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/concepts/architecture/architecture_support>.

23 AUTHORS, T. F. *Zircon Handles*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/concepts/objects/handles>.

24 AUTHORS, T. F. *Zircon Kernel objects*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/concepts/objects/objects>.

25 AUTHORS, T. F. *Zircon System Calls*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/reference/syscalls>.

26 AUTHORS, T. F. *FIDL: Fuchsia Interface Definition Language*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/languages/fidl>.

27 AUTHORS, T. F. *FIDL tutorial*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/languages/fidl/tutorials/overview>.

28 AUTHORS, T. F. *C++ in Zircon*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/languages/c-cpp/cxx>.

29 AUTHORS, T. F. *Guide to bringing a new language to Fuchsia*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/languages/new>.

30 AUTHORS, T. F. *Getting started with Fuchsia*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/getting_started>.

31 AUTHORS, T. F. *The Fuchsia build system*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/build/overview>.

32 AUTHORS, T. F. *The Fuchsia build system*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/build/overview>.

33 PANDIT, V.; DESSAI, S.; CHAUDHARI, S. Development of bsp for arm9 evaluation board. *International Journal of Reconfigurable and Embedded Systems*, IAES Institute of Advanced Engineering and Science, v. 4, n. 3, 2015.

34 KUMAR, K. E.; KAMARAJU, M.; YADAV, A. K. Porting and bsp customization of linux on arm platform. *International Journal of Computer Applications*, Foundation of Computer Science, v. 975, p. 8887, 2013.

35 PINE64. *ROCK64 Single Board Computer*. 2019. Disponível em: <https://wiki.pine64.org/index.php/ROCK64>.

36 DAVID, F. M. et al. *Porting Choices to ARM Architecture Based Platforms*. [S.l.], 2007.

37 CHIP, F. *FT232R USB UART IC Datasheet*. 2020. Disponível em: <https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf>.

38 FELIPEFLOP. *Placa FTDI FT232RL Conversor USB Serial*. 2020. Disponível em: <https://www.filipeflop.com/produto/placa-ftdi-ft232rl-conversor-usb-serial/>.

39 BELLARD, F. Qemu, a fast and portable dynamic translator. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USA: USENIX Association, 2005. (ATEC '05), p. 41.

40 AUTHORS, T. F. *Debug the kernel using QEMU*. 2020. Disponível em: <https://fuchsia.dev/fuchsia-src/development/emulator/qemu>.