Projeto de Graduação

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

02/07/2018

APLICAÇÃO DE FAST INTERRUPT NA AQUISIÇÃO DE DADOS DE UM PIG INSTRUMENTADO

Rayanne Gonçalves de Souza

PUC RIO

Projeto de Graduação

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

APLICAÇÃO DE FAST INTERRUPT NA AQUISIÇÃO DE DADOS DE UM PIG INSTRUMENTADO

Aluna: Rayanne Gonçalves de Souza

Orientador: Noemi Rodriguez

Coorientador: Miguel Freitas

Trabalho apresentado como requisito parcial à conclusão do curso de Engenharia de Controle e Automação na Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil.

Agradecimentos

Agradeço a minha família por todos os conselhos, apoios, incentivos e pela paciência em meus momentos de estresse. Agradeço especialmente a minha mãe Gisele Gonçalves pelos cafés nas madrugadas de estudo.

Aos meus orientadores Dr. Miguel Freitas, Dr. Noemi Rodriguez, Dr. Ana Lúcia e Dr. Marcelo Jimenez pela dedicação, os ensinamentos e a amizade.

A Rafael Azevedo pelos momentos de alegria e experiências compartilhadas desde o primeiro período da universidade. Sou grata pelo apoio acadêmico e emocional.

A minha melhor amiga Juliana Pinheiro pelos momentos de apoio e lazer.

A toda equipe do CPTI que contribuíram muito para meu crescimento profissional e pessoal.

A Prof. Dr. Ana Pavani pelo incentivo e pelas oportunidades de desenvolvimento profissional.

Resumo

Este projeto consiste em melhorar o atual processo de aquisição de dados de um PIG instrumentado. O sistema corrente apresenta um jitter que reduz a capacidade máxima da aquisição. Essa redução diminui a eficiência do processo de caracterização de dutos de óleo e gás. Para minimizar esse jitter, empregaremos uma Fast Interrupt no processo de demanda de aquisição.

Neste trabalho apresentamos um estudo sobre as interrupções em ARM, destacando a Fast Interrupt (FIQ) e as suas diferenças com as interrupções comuns. Além disso, levantamos uma análise comparativa do jitter para essas interrupções. Após, abordamos a criação de uma interface SPI-DMA que precisou ser desenvolvida para atender as restrições da FIQ. Associadas a essas restrições discutimos as mudanças do código e da configuração do Kernel do Linux que foram necessárias para o funcionamento da FIQ. Por último, tratamos das adaptações na arquitetura do sistema atual.

Palavras-chave: Jitter, FIQ, Kernel do Linux, Pipeline Pigging

DEPARTAMENTO
DE ENGENHARIA
ELÉTRICA

FAST INTERRUPT APPLICATION ON DATA ACQUISITION OF AN INSTRUMENTED PIG

Abstract

This project consist aims at improving the current process of data acquisition of an instrumented PIG. The current system has a jitter which reduces the maximum acquisition capability. This reduction decreases the efficiency of the characterization of oil and gas pipelines. In order to minimize the jitter, we use Fast Interrupt Request on demand for acquisition.

In this work, we present a study that focuses on ARM interrupts, emphasizing the Fast Interrupt (FIQ) and its different features in comparison to commons interrupts. Furthermore, we then conducted a comparative analysis of the jitter for these interrupts. Afterwards, we approached the creation of a SPI-DMA interface that needed to be developed to satisfy the FIQ restrictions. Associated with these restrictions, we discussed the Linux Kernel code and configuration changes that were necessary for the FIQ operation. Finally, we dealt with adaptations in the current system architecture.

Keywords: Jitter, FIQ, Linux Kernel, Pipeline Pigging

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

O setor de exploração petrolífera usa dutos para transportar petróleo e seus derivados. Dutos são tubos metálicos; podem ser terrestres, marítimos ou aéreos. O seu uso para o transporte de óleo e gás apresenta baixo consumo de energia e vantagens econômicas se comparado ao transporte rodoviário e ferroviário; são seguros e confiáveis. Entretanto, são vulneráveis à corrosão interna e externa da parede da tubulação [1].

Danos na estrutura desses tipos de dutos podem provocar desastres ambientais devido a vazamentos. Em áreas urbanas, vazamentos de óleo e gás são prejudiciais à população, um dos motivos é o risco de explosões. A sua inspeção e a manutenção são necessárias para evitar ocorrência de acidentes e, por consequência, danos econômicos. Uma ferramenta amplamente apliacada na inspeção de dutos de óleo e gás é o PIG instrumentado [1].

PIG instrumentado (*Pipeline Inspection Gauge*) é um dispositivo de inspeção que percorre o interior do duto, impulsionado pelo fluxo do fluído, para realizar medidas de caracterização dos dutos. O método de medição depende da necessidade da inspeção; este pode ser por ultrassom, perfilagem geométrica, perfilagem de temperatura, fluxo magnético e entre outros [2]. Sofwares de análise e simulação usam os dados obtidos do PIG como base para diagnosticar a condição do duto, e indicar a necessidade de manutenção.

A capacidade de sensoriamento do PIG é um parâmetro que reflete em sua eficiência. PIGs com poucos sensores fornecem informações qualitativas, em percentuais ou níveis de corrosão. Enquanto que os PIGs com números de sensores elevados permitem, com auxílio de softwares específicos, melhor caracterização e quantificação das descontinuidades [3].

A melhora da caracterização também está associada a frequência de aquisição de dados dos sensores. O sistema de aquisição realiza uma varredura de todos os canais de sensoriamento durante a movimentação do PIG. Um PIG costuma se movimentar em baixa velocidade, de 1 a 5 m/s em dutos de líquido e de 2 a 7 m/s em dutos de gás, para garantir a confiabilidade na aquisição de dados [4]. Uma baixa frequência de aquisição implica em um tempo maior para varrer todos os canais e consequentemente, em um aumento da quantidade de sensores suportados pelo sistema. Combinando a velocidade de deslocamento do PIG com esse tempo mais longo de varredura do sensores, temos uma coleta de dados menos exata, de caracterização com geometria helicoidal ao invés de circular. Portanto, quanto maior a frequência mais exata é a caracterização.

O PIG em aplicação nesse projeto é um dispositivo desenvolvido em parceria entre o Centro de Pesquisa da PETROBRAS (CENPES) e o Centro de Pesquisa em Tecnologia de Inspeção (CPTI) do Centro de Estudos em Telecomunicações (CETUC) da Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Esse é um PIG de perfilagem geométrica interna de dutos de óleo e gás capaz de atuar em linhas com variação de diâmetro e espessura. Um sensor do tipo palito permite a detecção e quantização da corrosão nesse tipo de duto. Em síntese, este sensor é um conjunto de varetas, que realizam a perfilagem geométrica do duto, e de sensores, que medem a angulação dessas varetas.

A Figura 1 mostra um conjunto de sensores palito em contato com a superfície interna de um duto. O sensor é maior que o diâmetro da tubulação. Uma vez inserido, a parede da tubulação força a movimentação do sensor para trás. Neste momento, o sensor mede a distância de referência entre o eixo central do PIG e o interior do duto. Qualquer mudança nesta posição inicial denota uma mudança na parede interna do duto, possivelmente um ponto de corrosão ou um ponto de acumulação de material [5].

Figura 1: PIG instrumentado com sensores palito em contato com a parede interna do duto [5]

O sistema de aquisição de dados deste PIG usa interrupções periódicas para uma frequência máxima de aquisição de 1024Hz. Nessa frequência a capacidade de sensoriamento é de 364 canais. O tipo de interrupção periódica desse sistema possui um atraso na demanda de aquisição entre 25μs a 158μs. Este atraso corresponde à perda no tempo de aquisição e, por consequência, redução da capacidade de sensoriamento.

O objetivo deste projeto é diminuir de forma significativa esse atraso, para aumentar a capacidade de canais, e aumentar a frequência máxima para 2048Hz a fim de obter melhor caracterização dos dutos. Para isto utilizamos um outro tipo de interrupção, denominada *Fast Interrupts*, que não apresenta este problema de atraso mas exige diversas adaptações do sistema para atender a algumas restrições peculiares.

2 A Arquitetura do Sistema

O Linux é um sistema operacional de distribuição gratuita empregado em diversas áreas tecnológicas. A indústria automotiva, aeroespacial, robótica e a automação industrial são exemplos de campos de aplicação. A natureza gratuita do seu código fonte e a sua compatibilidade com uma larga gama de arquiteturas, tornou esse sistema operacional popular na comunidade de desenvolvimento de sistemas embarcados [6].

A eletrônica embarcada do PIG em questão usa a placa Arietta G25 para embarcar o Kernel do Linux. O sistema de controle é composto por um conjunto de drivers escritos como módulos do Kernel. Módulos são pedaços de código que podem ser carregados e descarregados no Kernel sob demanda, estendendo sua funcionalidades sem a necessidade de reinicializar o sistema [7].

Na arquitetura do Linux, a memória do sistema é dividida em Espaço do Kernel e Espaço do Usuário. Definidos por:

Espaço do Kernel é onde o Kernel (ou seja, o núcleo do sistema operacional) é executado e fornece os seus serviços.

Espaço do Usuário é a parte da memória em que se executa todo programa que não seja o Kernel [8].

A Figura 2 mostra a arquitetura do Linux. A GNU C Library (glibc) do espaço do usuário fornece acesso a *system call interface* [9]. Aplicações no espaço do usuário requisitam serviços ao Kernel através de chamadas de sistema (*System Calls*), como por exemplo, acesso aos dispositivos E/S.

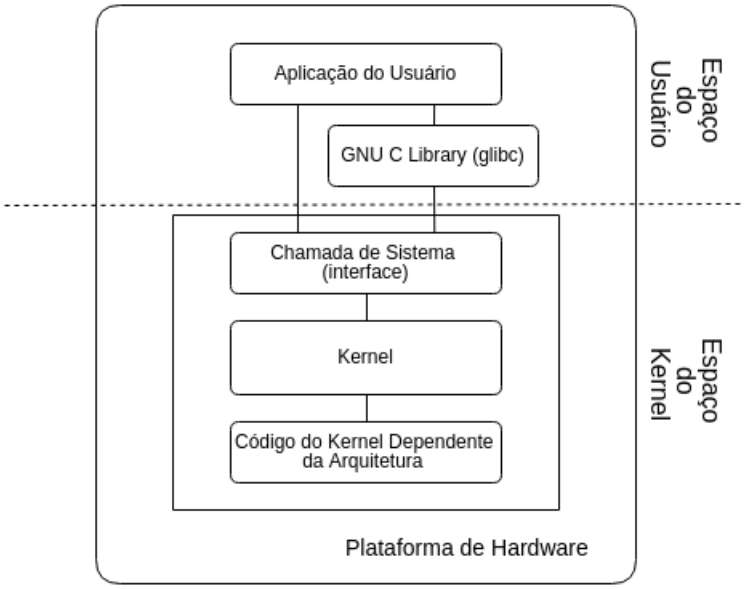

Figura 2: Arquitetura do GNU/Linux [9]

A Figura 3 mostra a arquitetura do sistema de aquisição de dados do PIG. Uma parte desse sistema roda no espaço do Kernel e a outra no espaço do Usuário. No espaço do Kernel o processador executa um módulo de Kernel principal responsável pela chamada periódica de uma interrupção (Kpig_interrupt). Enquanto que no espaço do usuário, executa um programa de acesso as FIFO que foram atualizadas a partir da execução da Kpig_interrupt.

A finalidade da função Kpig_interrupt é:

- Incremento do timestamp.
- Chamada de todos os drivers associados à aquisição através de uma varredura.
- Atualização da FIFO do módulo de Kernel principal.

A atualização da FIFO do módulo de Kernel principal corresponde ao armazenamento do timestamp corrente juntamente com o registro dos drivers que realizaram a aquisição naquele timestamp. Dessa forma atualizamos a FIFO com a data e hora em que a interrupção ocorreu. O uso do timestamp permite que o sistema saiba quando

um driver deve realizar sua aquisição. Cada driver possui seu próprio período de demanda de aquisição, isto é, a chamada de sua respectiva função através da Kpig_interrupt realiza a aquisição apenas se o período do driver for atingido.

Cada driver no espaço do Kernel dispõe de uma FIFO própria para o armazenamento dos dados adquiridos. Nesse sentido, o programa em execução no espaço do usuário (DRV_CONTROL), consulta a FIFO do módulo de Kernel principal para saber quais drivers executar no processo de acesso da FIFO_DRV. Os dados assim obtidos são armazenados em um banco de dados para a realização de processamentos que permitem sua interpretação.

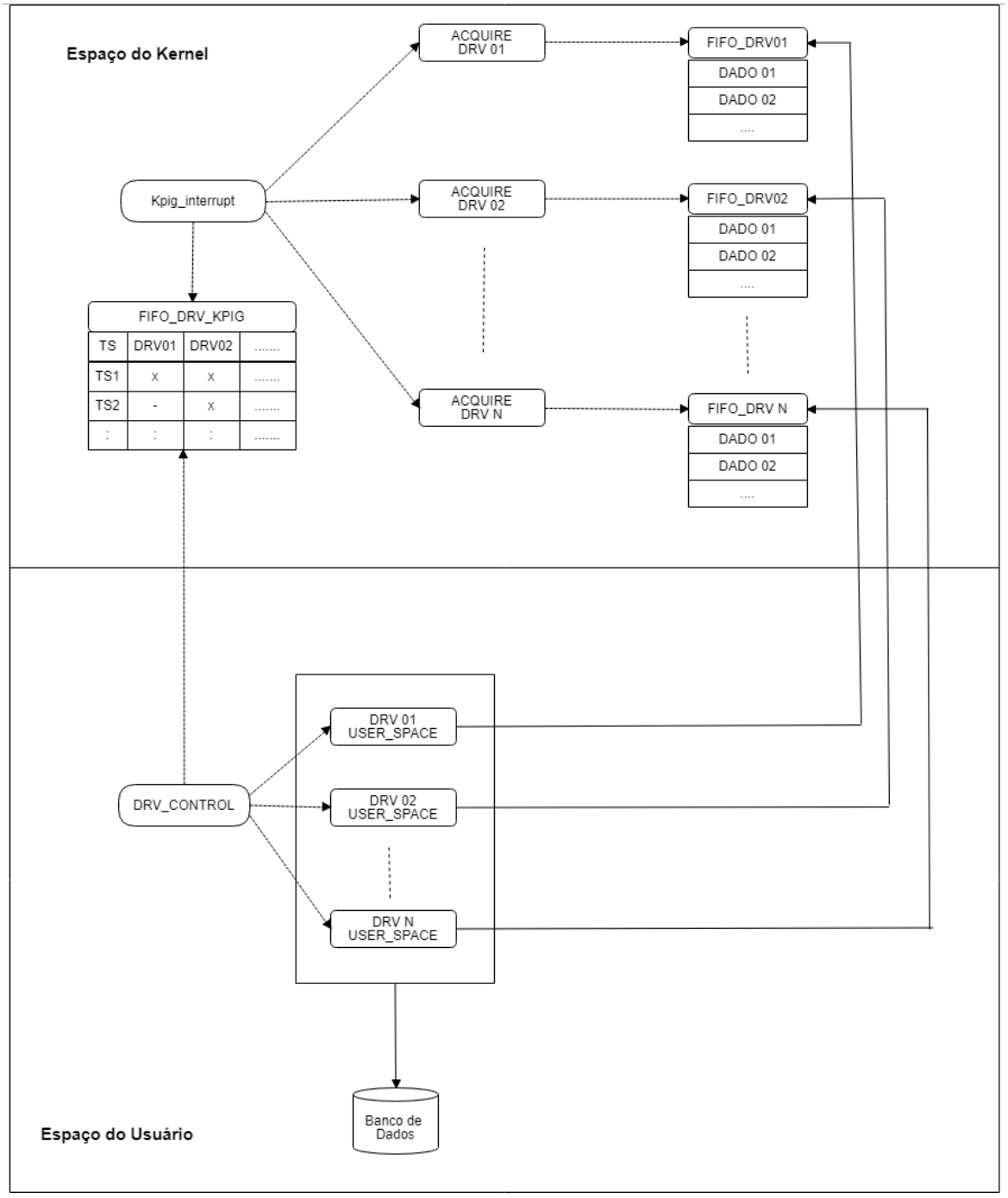

Figura 3: Arquitetura do Sistema de aquisição de dados do PIG

Nesse sistema, Kpig_interrupt é uma interrupção de hardware. As interrupções de hardware são recursos regularmente utilizados em sistemas embarcados quando alguma operação necessita de imediata atenção da CPU.

Dado que várias interrupções podem ocorrer simultaneamente, cada tipo de interrupção possui uma prioridade que indica ao processador qual tipo de interrupção deve ser atendida primeiro. Este método de prioridade permite diminuir a latência de interrupção, uma vez que a prioridade permite que interrupções de prioridade igual ou maior interrompa a rotina corrente, ou seja, o processador não perde tempo excessivo no tratamento de interrupções de prioridades menores [10]. Nesse contexto, o atual sistema de aquisição de dados emprega a interrupção PIT (Programmable Interrupt Timer) para a chamada periódica de Kpig_interrupt. A prioridade deste tipo de interrupção é a máxima entre as demais interrupções do tipo Interrupt ReQuest (IRQ).

A arquitetura ARM dispõe de dois tipos de interrupção: FIQ (Fast Interrupt Requests) e IRQs (Interrupt ReQuests). Sendo a FIQ a interrupção de maior prioridade capaz de interromper qualquer tipo de IRQ [10]. A próxima seção apresenta as interrupções no ARM.

3 As Interrupções no ARM

Uma exceção é qualquer condição que precise interromper a execução sequencial de um grupo de instruções. O reset do processador, a ocorrência de interrupção externa e falhas de acesso à memória são exemplos de exceções. A exceção é um mecanismo de tratamento de erro e de eventos externos associados ao sistema. A ARM define uma interrupção como um tipo especial de exceção [10].

Quando uma exceção ocorre o processador entra em um modo específico. A Tabela 1 mostra os modos associados a cada tipo de exceção. A especificação do modo permite que o processador conheça a forma adequada de tratar a exceção .

Exceção	Modo	Principal Função
Fast Interrupt Request	FIQ	Tratamento de uma FIQ
Interrupt Request	IRQ	Tratamento de uma IRQ
SWI and Reset	SVC	Modo protegido para sistemas operacionais
Prefetch Abort and Data Abort	abort	Tratamento de proteção à memória
Undefined Instruction	undefined	Emulação de software de coprocessadores de hardware

Tabela 1: Modos do processador para cada tipo de exceção [10]

A interrupção é definida como um evento que suspende a execução de um programa corrente para executar um código diferente, chamado de *Interrupt Service Routine* (ISR) ou handler de interrupção. Após a execução do ISR, o programa retorna ao ponto em que estava imediatamente antes da ocorrência da interrupção [11].

Nos processadores da ARM existem dois tipos de interrupção. O primeiro são interrupções causadas por um periférico externo, também conhecidas como interrupções externas; São elas FIQ (*Fast Interrupts*) e IRQ (*Interrupt Requests*). O segundo tipo é uma instrução específica que causa uma interrupção, conhecida por SWI (*Software Interrupt*).

Software Interrupt é normalmente usada para chamar rotinas privilegiadas do sistema operacional. Por exemplo, uma SWI pode ser usada por um programa para mudar sua execução no modo usuário para o modo privilegiado.

Interrupt Requests são as interrupções normais, de uso geral como por exemplo recebimento de dados de placa de rede.

Fast Interrupt Requests são reservadas para uma única fonte de interrupção. O uso da FIQ é adequado para aplicações que requerem tempo de resposta rápido. Um exemplo de aplicação é o acesso direto à memória usado para movimentar blocos de memória.

Uma interrupção externa ocorre quando um periférico envia um sinal a um pino físico conectado ao processador, também conhecido por linha de interrupção. O processador AT91SAM9G25 possui um pino para IRQ denominado por nIRQ e um pino para a FIQ denominado nFIQ.

Uma IRQ acontece quando a linha nIRQ é ativada por algum periférico externo. O processador atende este pedido de interrupção se não houver a ocorrência de uma FIQ ou de uma exceção mais prioritária como o Data Abort e o Reset. Quando o processador atende ao pedido deste tipo interrupção, ele entra no modo de operação interrupt e desabilita a IRQ.

Uma interrupção do tipo FIQ acontece quando a linha nFIQ é ativada por algum periférico externo. A FIQ é a interrupção de prioridade mais alta, então, quando o processador atende a esse pedido de interrupção ele desabilita as interrupções FIQ e IRQ pois uma IRQ não pode interromper uma FIQ.

Ao atender ao pedido de interrupção, o processador entra em um dos modos da Tabela 1, salva informações sobre o processo interrompido e executa o handler de interrupção. Os registradores que o processador deve salvar dependem do modo de interrupção.

As interrupções e exceções podem ocorrer simultaneamente mas o processador é capaz de tratar apenas uma de cada vez. Sendo assim, a ARM adota um mecanismo de prioridade no qual interrupções de prioridade mais alta podem interromper as de prioridade mais baixa. A Tabela 2 mostra a ordem de prioridade de cada tipo de exceção e interrupção.

Prioridade	Tipo de Exceção
Mais alta	Reset
	Data Abort
	FIQ
	IRQ
	Prefetch Abort
Mais baixa	Undefined instruction and SWI

Tabela 2: Níveis de prioridade [10]

a Banked Registers

Cada modo possui um conjunto de registradores de uso exclusivo conhecidos por banked register. Como mostrado na Figura 4, a IRQ possui 3 registradores de uso exclusivo, R13_IRQ, R14_IRQ e SPSR_IRQ, enquanto que a FIQ possui 8, R8_FIQ até R14_FIQ e SPSR_FIQ. A quantidade de *Banked Registers* tem influência sob o armazenamento de informações do processo interrompido. Quando o processador atende a uma IRQ, esse armazena o conteúdo dos registradores de R0_IRQ até R12_IRQ, CPSR e PC, em uma pilha. Se a interrupção é do tipo FIQ, o processador armazena o conteúdo de R0_FIQ até R7_FIQ, CPSR e PC. Essa operação é necessária para que o processador retorne a executar o processo interrompido após o atendimento da interrupção.

Os *banked registers* são cópias de registradores de mesma funcionalidades que são visíveis apenas para o modo que está relacionado. Isto é, o R8_FIQ é uma cópia do registrador R8 visível apenas para o modo FIQ.

User and System Mode	Supervisor Mode	Abort Mode	Undefined Mode	Interrupt Mode	Fast Interrupt Mode
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

Figura 4: *Banked Registers* para cada modo [12]

b Vetor de Interrupção

O vetor de interrupção permite o direcionamento ao handler. Em um cenário hipotético onde o processador atende a um pedido de interrupção do tipo FIQ, o processador entra no modo FIQ e salta para o endereço 0x1C do vetor de interrupção, veja a Tabela 3. Geralmente cada entrada do vetor de interrupção contém uma instrução de desvio para uma rotina específica [10].

Exceção	Modo	Offset Vetor de Interrupção
Reset	SVC	+0x00
Undefined	Instruction UND	+0x04
Software Interrupt (SWI)	SVC	+0x08
Prefetch Abort	ABT	+0x0c
Data Abort	ABT	+0x10
Not assigned	-	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1c

Tabela 3: Vetor de interrupção e modos do processador [10]

c Entrada e Saída do Handler de Interrupção

No processo de entrada do handler de uma interrupção o processador executa a seguinte sequência [13]:

- Armazena o endereço da instrução seguinte a interrompida;
- Copia o registrador que armazena o atual estado da CPU (CPSR) ao apropriado SPSR, que é um *banked register* de um dos modos do processador;
- Força os bits do campo mode do CPSR ao modo correspondente ao tipo de interrupção em execução;
- Força o PC a referenciar a próxima instrução do vetor de interrupção;

O processador executa o handler de interrupção após o PC referenciar o endereço no vetor de interrupção.

No processo de saída do handler, o processador executa a seguinte sequência:

- Move o registrador que armazena o endereço de retorno à função interrompida (LR) menos offset para o PC. O offset depende do tipo de interrupção.
- Copia o conteúdo de SPSR para CPSR.

A cópia de SPSR para CPSR implica no retorno do processador ao modo interrompido.

d Latência e Jitter de Interrupção

A latência de uma interrupção é o intervalo de tempo entre o requerimento da interrupção, através da ativação de uma linha de interrupção, e o início da execução do seu respectivo handler.

A latência de interrupção pode ser intensificada pelos seguintes fatores [14]:

- O processador demora múltiplos ciclos de instrução para completar a instrução corrente, imediatamente antes de atender ao pedido de interrupção.
- A interrupção pode ser interrompida por uma outra de prioridade maior, antes mesmo da primeira instrução de seu ISR ser executada pelo processador.
- Quando o sinal de requerimento é de um evento externo, primeiramente ele deve ser sincronizado com o clock do barramento/periférico.
- O processador armazena vários registradores na pilha para poder restaurar o processo interrompido.
- A desabilitação de interrupções para a execução de algumas partes do código do sistema operacional.

A latência de interrupção é um parâmetro crítico quando o sistema depende de uma resposta muito rápida a um evento. Como por exemplo, software de controle de avião autônomo. Associado a latência existe o conceito de variação da latência, denominada jitter de interrupção.

De acordo com a definição, o jitter é o conjunto de variações de pequena duração dos instantes de desvio significativos do tempo do sinal a partir das suas posições ideiais no tempo [9]. A variação da latência é um jitter. O jitter resulta de fenômenos físicos no hardware, como ruído, processamento de tarefas concorrentes, a nível hardware e software, e da passagem do código por diferentes branches. Cada instrução de desvio é potencialmente uma fonte de jitter [10].

Assim como a latência, o jitter pode ser um parâmetro crítico para sistemas que dependem do tempo de execução de uma interrupção. Por exemplo, em um sistema de aquisição de dados a 2048Hz (500μs) um jitter de 200μs compromete a aquisição pois 40% do tempo é perdido por causa do jitter.

O PIG deste projeto apresenta um jitter de interrupção mínimo de 25μs e máximo de 158μs. Este valor compromete o número máximo de canais de aquisição de dados. É desejável que a latência e a sua variação sejam a mais baixa possível para que o sistema de aquisição consiga trabalhar a uma frequência maior de aquisição e com maior número de canais.

Uma vez que o processador atende de forma mais rápida as interrupções de prioridade alta. O emprego de uma interrupção mais prioritária que a interrupção atual do sistema de aquisição reduz a latência e o jitter. Dado que a FIQ apresenta latência mais baixa que a IRQ, a próxima seção apresenta como usar uma FIQ com o Kernel do Linux a fim de possibilitar a substituição da IRQ Kpig_interrupt por uma FIQ no sistema de aquisição de dados.

4 FIQ em ARM Usando o Kernel do Linux

a Fast Interrupt Requests

Como a FIQ tem maior prioridade sobre uma IRQ, uma FIQ pode interromper uma IRQ em execução, porém uma IRQ não pode interromper a execução de uma FIQ. Assim, nenhum FIQ handler pode executar tarefas que requerem alguma IRQ ativa.

Geralmente, o Kernel do Linux não usa nenhuma FIQ o que torna o uso desta interrupção adequado para sistemas que necessitam de desempenho em tempo real, ou seja, onde a interrupção deve ser rapidamente atendida pelo processador.

No handler da FIQ deve-se evitar o uso de APIs disponíveis para o Linux pois a maioria depende internamente de schedulers e de spinlocks para gerenciar acessos concorrentes. O handler precisa conter um conjunto de tarefas que sejam de rápida execução e não causem bloqueios, uma vez que o processador deve atender e finalizar rapidamente a execução de uma FIQ. Além disso, é necessário evitar a ocorrência de *page faults* dentro do handler, dado que o Kernel pode não ter o poder de tratá-las. Nesse sentido, é imperativo alocar o contexto do handler da FIQ antes de sua execução [11].

O Kernel do Linux possui um conjunto de funções que viabilizam o registro da FIQ. Essas funções encontram-se no diretório /arch/arm/fiq.h. Em termos objetivos, o processo de inicialização de uma FIQ segue as etapas abaixo.

1. Requer a FIQ.
2. Registra a pilha a ser usada pela FIQ.
3. Registra o FIQ handler.
4. Ativa a FIQ.

Na Figura 5, a claim_fiq() representa o passo 1; a função set_fiq_regs registra a pilha, enquanto que a set_fiq_handler registra o handler e a enable_fiq ativa a FIQ.

```
static u8 fiqstack[1024];
static struct fiq_handler fh= {.name = "gpio_fiq"};
int fiq_init(void)
{

  struct pt_regs fiqregs;

  if (claim_fiq(&fh))
      return -EBUSY;

  fiqregs.ARM_sp = (u32)&fiqstack[sizeof(fiqstack)];
  set_fiq_regs (&fiqregs);

  set_fiq_handler ((void *)fiqhandler, size);

  enable_fiq (ID_FIQ);

  return 0;
}
```

Figura 5: Função de inicialização da FIQ

A função claim_fiq retorna 0 se o recurso FIQ estiver liberado para uso e retorna um valor negativo caso essa ferramenta já esteja em uso por algum outro processo. O Kernel do Linux emprega este mecanismo de proteção devido a existência de uma única FIQ no processador.

O handler de uma interrupção precisa de uma pilha para salvar os registradores do processo interrompido. Esta operação é necessária para que após o tratamento da interrupção o processador possa retornar ao conjunto de instruções suspendido.

A função de registro do handler, *set_fiq_handler*, recebe como parâmetro um ponteiro para o handler e o tamanho do seu código. Conforme visto na Tabela 3, a entrada reservada para a FIQ é a última do vetor de interrupção. Por este motivo, o handler da FIQ pode começar imediatamente neste endereço, ou seja, não é necessário o uso de uma instrução de desvio para a chamada do handler. Assim, a função *set_fiq_handler* copia o código do handler para a área de memória que começa na entrada do vetor de interrupção da FIQ.

Normalmente, escreve-se o FIQ handler em assembly, aplicando o uso de labels para identificar o endereço inicial e final. Essa estratégia permite fornecer um ponteiro inicial e um tamanho ao set_fiq_handler. Observe um exemplo do uso da set_fiq_handler.

```
handler_init:
        fiq handler
handler_end:

set_fiq_handler(&handler_init, &handler_end - &handler_init);
```

b FIQ Handler em C

O GCC viabiliza a declaração de um FIQ handler em C através do atributo *__attribute__ ((interrupt ("FIQ")))*. O uso deste atributo indica ao compilador que a função é um handler de interrupção para que o compilador gerencie o seu código de entrada (prologue) e de saída (epilogue) de forma adequada. O código de entrada e saída de um handler de interrupção diferencia-se de uma função normal devido à necessidade de armazenamento de certos registradores na pilha, e ao endereço de retorno.

Quando o processador reconhece uma interrupção, o program counter (PC) está referenciando a próxima instrução, em razão do incremento que sofre no estágio *fetch* do *pipeline* do processador. Então, para que, no fim da execução do handler de uma interrupção, a instrução interrompida possa ser retomada, o endereço de retorno deve ser o PC - 4. O compilador gerencia essa manipulação de endereço de retorno graças ao uso do atributo interrupt.

É necessário especificar o tipo de interrupção, IRQ ou FIQ, para que o compilador conheça quais registradores devem ser armazenados em uma pilha. Conforme apresentado na Seção 3a, para a FIQ é preciso armazenar apenas o conteúdo dos registradores r0-r7 pois o modo FIQ possui seu próprio banco de registradores de r8-r14.

Dada a definição do FIQ handler em C, através do atributo interrupt("FIQ"), emprega-se a função set_fiq_handler para registrar este handler. Uma vez que esta função registra apenas funções em assembly, adotou-se a seguinte estratégia.

```
u32 fiqhandler_asm[2];
fiqhandler_asm[0] = 0xE51FF004;     /* ldr pc, [pc, #-4] */
fiqhandler_asm[1] = (u32) fiqhandler_c;
set_fiq_handler ((void *)fiqhandler_asm, 8);
```

Para poder escrever o handler em C, usamos um wrapper em assembly que contém apenas o salto para o handler:

```
0x001C: ldr pc, [pc, #-4]
0x0020: .word fiqhandler_c
```

c Processo de Chamada de um FIQ Handler

Na geração do sinal FIQ, seja por hardware ou por software, o processador executa os passos abaixo.

- Armazena o valor de CPSR (*Current Program Status Register*) no SPSR_fiq (*Saved Processor Status Register* reservado para o modo FIQ).
- Armazena o valor do PC no registrador R14_fiq (*link register* reservado para o modo FIQ).
- Força a entrada no modo FIQ a partir do campo *processor mode bits* igual a M[4:0]=10001.
- Desabilitada as interrupções, setando os bits I e F do registrador CPSR.
- Força o PC a referenciar a instrução do endereço 0x1C (FIQ exception vector address)

A descrição do papel de cada registrador encontra-se no Apêndice A.

d Processo de Retorno de um FIQ Handler

No processo de retorno ao processo interrompido, o processador executa duas operações:

- Carrega CPSR com o que estava armazenado no SPSR_fiq.
- Carrega PC com o valor adequado armazenado em LR.

Há duas formas de restaurar o CPSR.

- A restauração do CPSR ocorre quando a instrução processada possui a flag S setada (Apêndice B), e o PC como registro de destinação.
- A restauração do CPSR ocorre a partir de uma instrução, no código de saída do handler, que usa uma pilha de armazenamento de registradores. A instrução abaixo exemplifica este caso.

```
LDMFD sp!,{R0–R7,pc}^
```

Onde o símbolo ^ indica que a restauração do CPSR é a partir do valor armazenado no SPSR [15]. Algumas versões do GCC geram bugs na criação do código de entrada e saída do FIQ handler. Identificamos que a FIQ executava uma função cuja instrução de retorno, gerada pelo compilador, era MOV com a flag S ativa e o PC como registrador de destino. De acordo com as informações do Apêndice B, este tipo de instrução resulta na restauração do CPSR, a partir do SPSR_fiq, provocando a saída do modo FIQ antes do término de sua execução. Para evitar que este tipo de erro ocorra, inseriu-se instruções no código de entrada e saída do FIQ handler.

Na entrada armazena-se o valor do CPSR, salvo no SPSR_fiq, na pilha e o registrador corrente CPSR no SPSR_fiq. Dessa forma, caso haja alguma chamada de função dentro de um FIQ handler que possua uma instrução com a flag S setada que use o PC como registro de destinação, a restauração do CPSR a partir do SPSR não provocará a saída do modo FIQ. No código de saída, copia-se o CPSR da pilha para o SPSR possibilitando a saída adequada do FIQ handler. O trecho a seguir implementa essas instruções.

```
static void gpio_fiq_handler(void) __attribute__((interrupt("FIQ")));
static void gpio_fiq_handler(void)
{
    asm(
        "mrs r0,spsr\n\t" // antigo cpsr (do modo real, fora da fiq)
        "push {r0}\n\t"   // salva na pilha
        "mrs r0,cpsr\n\t" // copia o atual cpsr para o  r0
        "msr spsr,r0\n\t" // copia cpsr para o spsr
    );

    FIQ handler code

    asm(
        "pop {r0}\n\t"       // restaura o spsr antes de retornar da fiq
        "msr spsr,r0\n\t"
    );
}
```

e Interrupção de Tempo como Trigger da FIQ

No âmbito deste projeto é necessária uma chamada periódica ao handler de interrupção. Para garantir essa periodicidade usamos uma interrupção de tempo como trigger de hardware da FIQ.

O processador AT91SAM9G25 apresenta duas interrupções de tempo, a *Programmable Interval Timer* (PIT) e a *Timer Counter* (TC). Ambas funcionam como um contador que gera um sinal de saída ao atingir o valor pré-programado, servindo como um sinal de trigger para uma interrupção. Além disso, o processador possui a ferramenta de *Fast Forcing* no controlador de interrupção. Essa ferramenta permite que uma interrupção normal seja o trigger da FIQ. O registrador AIC_FFER (*Fast Forcing Enable Register*) torna possível, a partir do identificador do periférico, o redirecionamento de uma interrupção normal para o controle da FIQ. As informações a respeito de registradores do processador encontram-se no seu datasheet [12].

O Apêndice C mostra os identificadores dos periféricos do processador. A interrupção PIT compartilha o seu identificador com outras interrupções do sistema. Portanto, o uso deste identificador (PIT) na ferramenta *Fast Forcing* provocaria o trigger da FIQ por qualquer um dos periféricos que compartilham esse identificador. A interrupção TC não compartilha o seu identificador de periférico com outros tipos de interrupção. Nesse sentido optamos por empregar o TC como trigger da FIQ.

A utilização do TC como trigger deve assegurar que este periférico não esteja em uso como clock do sistema. A execução do comando $\mathrm{cat}\ /\mathrm{proc}/\mathrm{timer\_list}$ exibe o dispositivo usado como clock. Caso o TC esteja presente

na lista de times usados pelo sistema, em Device Drivers → Misc devices do menu de configuração do Kernel deve-se desmarcar os itens abaixo.

[ ] Atmel AT32/AT91 Timer/Counter Library

[ ] TC Block Clocksource

O periférico TC do processador é um módulo composto por três canais TC. Apenas um canal é necessário para servir de trigger da FIQ. O diagrama da Figura 6 ilustra as configurações requeridas para o uso de um contador como trigger. O código correspondente a este diagrama encontra-se no Apêndice D.

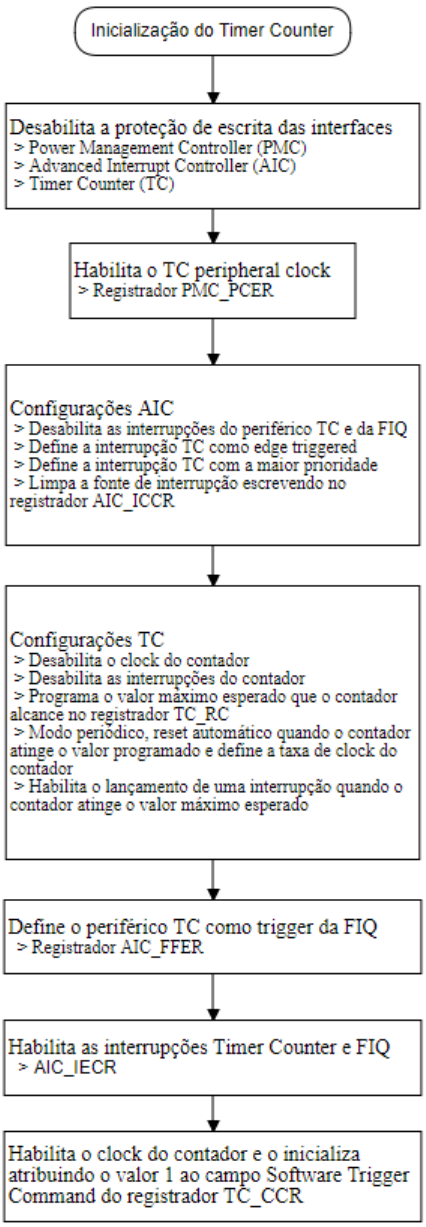

Figura 6: Programação do Timer Counter

5 Análise do Jitter das Interrupções FIQ e PIT

O objetivo do uso de uma FIQ na aquisição de dados de um A/D é reduzir o jitter da interrupção, ou seja, a variação de latência. Assim sendo, procuramos inicialmente mensurar, através do uso de um contador, o jitter das interrupções FIQ e PIT, onde a PIT é uma interrupção atualmente em utilização nessa aquisição de dados. A frequência de interrupção para ambos os tipos foi de 2kHz. Ambas interrupções possuem o mesmo handler cuja finalidade é alternar os valores de um conjunto de pinos de GPIO.

Nas medições de jitter, realizamos testes em quatro cenários distintos. Na escolha de tais cenários buscamos pela execução de comandos que requeiram um nível de atenção do processador capaz de concorrer com essas interrupções. Nesse sentido, optamos por realizar medidas nos seguintes cenários:

- Execução do comando find;
- Cópia de um arquivo;
- Leitura de um arquivo;
- Tranferência de rede com Netcat.

a Procedimento de Mensura do Jitter

O processador em aplicação possui dois módulos com 3 canais de TC. Apenas um único canal é necessário para o trigger da FIQ. Empregamos um dos canais disponíveis no processo de medição do jitter. A programação deste canal é semelhante ao mostrado no Apêndice D, excluíndo apenas as etapas referentes à interrupção FIQ. No programa de teste definimos um vetor de 5000 posições para armazenar a leitura do contador imediatamente após a chamada das interrupções FIQ e PIT. O trecho de código na Figura 7 ilustra esta estratégia.

```
void gpio_fiq_handler(void)
{
  chamou = (chamou +1)%JITTER_SIZE;
  jitter[chamou] = tc_readl(CV, TC_JITTER);

  ....
}

static void gpio_interrupt (void *private_data)
{
  chamou = (chamou +1)%JITTER_SIZE;
  jitter[chamou] = tc_readl(CV);

    ....
}
```

Figura 7: Handler com o vetor de posição de 5000 posições

Uma vez que o vetor contém o valor do contador, realiza-se a subtração consecutiva entre cada elemento do vetor a fim de obter o número de clocks decorridos entre cada chamada do handler de interrupção. Para analisar essas 5000 amostras de variação de latência, escrevemos o programa em python do Apêndice E, cuja finalidade é criar uma distribuição normal do jitter. Observa-se, neste programa, que há uma transformação desse vetor, de número de clocks, por um vetor de tempo, realizando-se a multiplicação do vetor pelo valor do clock (15.03ns). As próximas seções apresentam os gráficos de distribuição normal obtidos através desse programa para os 4 cenários mencionados.

b Medição de Jitter Durante a Execução do Comando Find

A Figura 8 apresenta a distribuição normal da variação de latência da interrupção PIT. O desvio padrão é 7.56μs com dispersão máxima de 43.32μs.

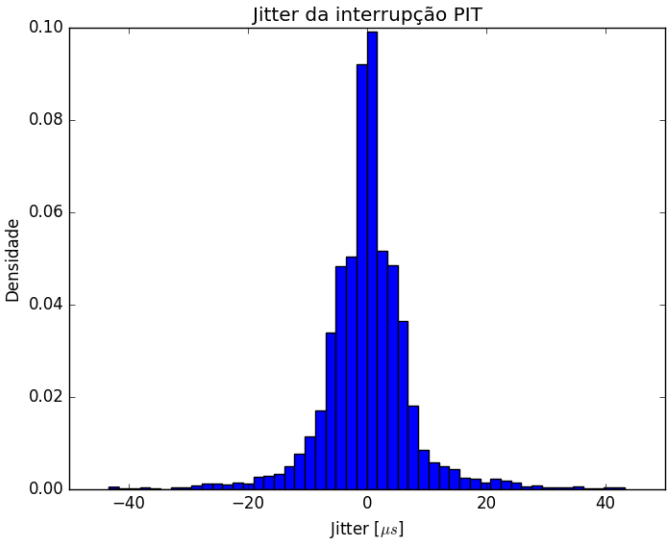

Figura 8: Distribuição normal do jitter da interrupção PIT durante a execução do comando find

Na distribuição normal do jitter da interrupção FIQ o desvio padrão é de 0.79μs. A dispersão máxima foi de -2.91μs.

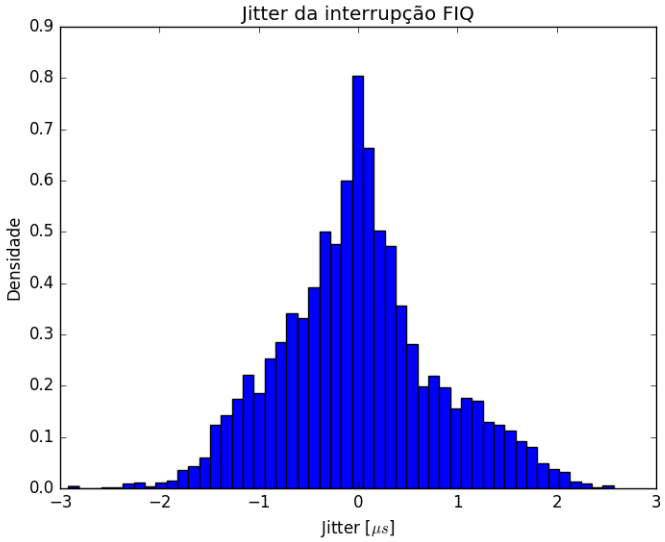

Figura 9: Distribuição normal do jitter da interrupção FIQ durante a execução do comando find

c Medição de Jitter Durante a Cópia de um Arquivo

Neste cenário realizou-se a medição de jitter durante a execução do comando $\mathrm{dd\ if{=}/dev/zero\ bs{=}2M\ count{=}100\ of{=}teste}$.

if=/dev/zero - O disco de origem a ser copiado;

bs=2M - O tamanho em bytes da leitura;

count=100 - O número de blocos de tamanho 2M da transferência;

of=teste - Arquivo de destino da cópia.

Na Figura 14, a distribuição normal do jitter da interrupção PIT possui desvio padrão de 6.14μs com dispersão máxima de 69.83μs.

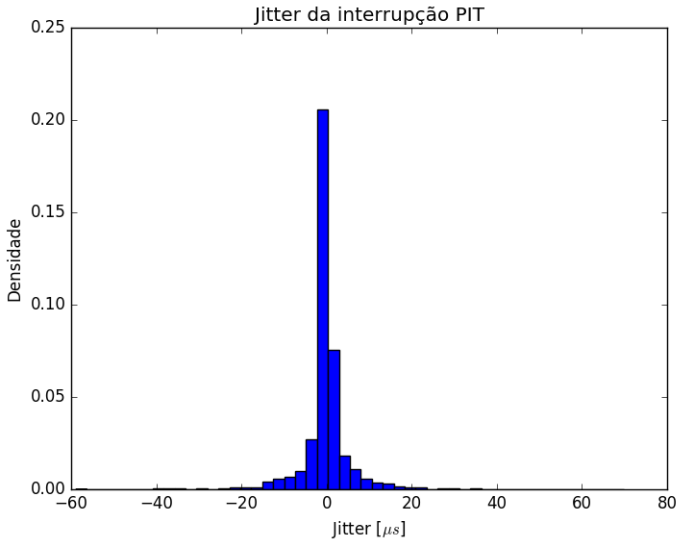

Figura 10: Distribuição normal do jitter da interrupção PIT durante a cópia de um arquivo

Na distribuição normal do jitter da interrupção FIQ o desvio padrão corresponde a 0.37μs com dispersão máxima de 2.04μs.

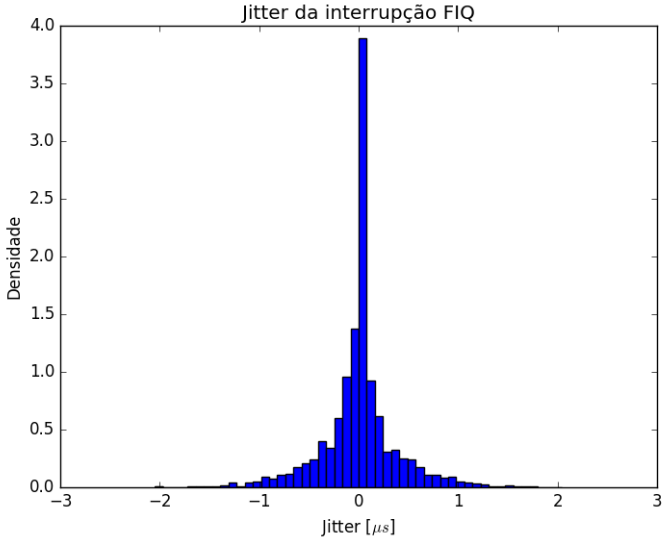

Figura 11: Distribuição normal do jitter da interrupção FIQ durante a cópia de um arquivo

d Medição de Jitter Durante a Leitura de um Arquivo

Nesta análise utilizamos o arquivo teste gerado na cópia de arquivo na Seção 5c, como arquivo de leitura. Tendo como objetivo realizar a leitura diretamente do disco, esvaziamos a cache de dados através do comando $\text{echo } 3 > /\text{proc/sys/vm/drop\_cache}$ e efetuamos a leitura durante a execução do comando $\text{dd if=teste bs=1M of=/dev/null}$.

if=teste - O arquivo a ser lido;

bs=1M - O tamanho em bytes da leitura;

of=/dev/null - O disco de destino da leitura.

A distribuição normal abaixo apresenta desvio padrão de 6.84μs com dispersão máxima de -38.00μs.

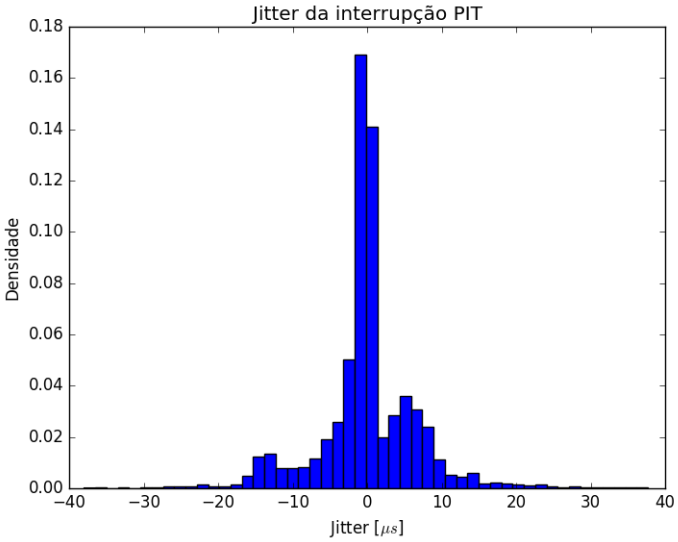

Figura 12: Distribuição normal do jitter da interrupção PIT durante a leitura de um arquivo

Na distribuição normal do jitter da interrupção FIQ possui desvio padrão de 0.50μs com dispersão máxima de 2.87μs.

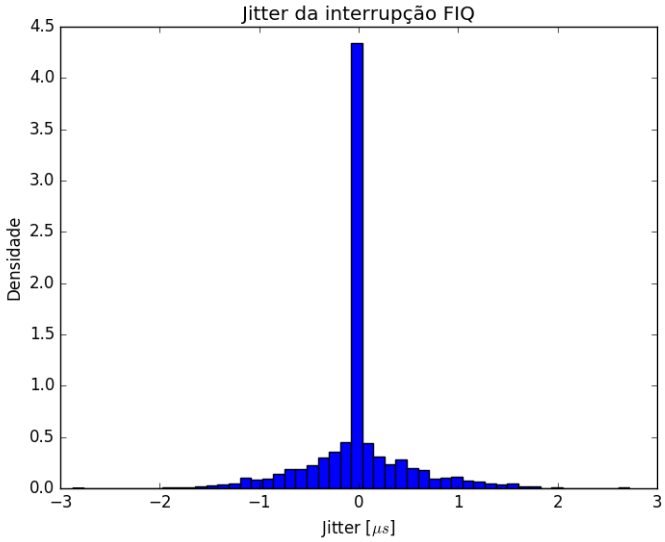

Figura 13: Distribuição normal do jitter da interrupção FIQ durante a leitura de um arquivo

e Medição de jitter Durante uma Transferência com Netcat

O Netcat é uma ferramenta de rede que permite a transferência de dados entre diferentes máquinas através de uma conexão. Nesta medição a placa eletrônica, AriettaG25 aguarda uma conexão através da porta 5000. Os dados em transferência possuem como destino o dispositivo nulo a fim de descartá-los. Executamos o comando $\mathrm{netcat\ -l\ -p\ 5000\ >\ /dev/null}$ na Arietta para aguardar uma recepção. Em um outro computador executamos o comando $\mathrm{dd\ if=/dev/zero\ bs=4M\ count=20\ |\ netcat\ 10.10.10.10\ 5000}$ para o envio de caracteres nulos em blocos de 4M através da porta 5000.

Na Figura 14, a distribuição normal do jitter da interrupção PIT possui desvio padão de 28.46μs com dispersão máxima de 91.71μs.

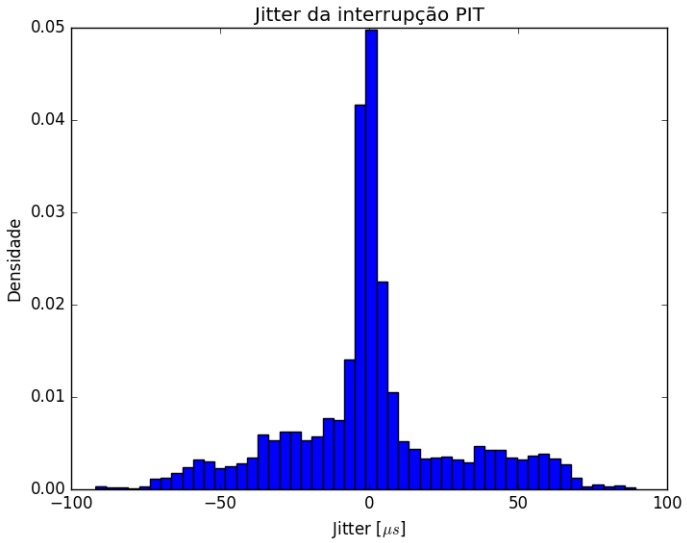

Figura 14: Distribuição normal do jitter da interrupção PIT uma transferência com Netcat

Na distribuição normal do jitter da interrupção FIQ o desvio padrão de 0.50μs com dispersão máxima de 3.57μs.

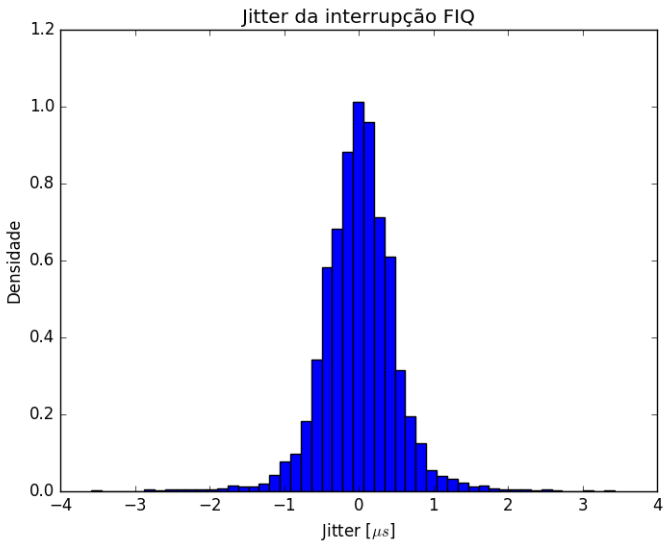

Figura 15: Distribuição normal do jitter da interrupção FIQ durante uma transferência com Netcat

f Comparação dos Resultados

A Tabela 4 resume os resultados da interrupção PIT e FIQ. O pior cenário para a interrupção PIT ocorreu na transferência de um arquivo cujo desvio padrão correspondeu a 28.46μs e à dispersão máxima a 91.71μs. O pior cenário para a FIQ também ocorreu para a tranferência de um arquivo cujo desvio padrão correspondeu a 0.50μs e à dispersão máxima a 3.57μs.

Teste	Desvio Padrão PIT (μs)	Desvio Padrão FIQ (μs)	Dispersão Máxima PIT (μs)	Dispersão Máxima FIQ (μs)
Find	7.56	0.79	43.32	2.91
Cópia de arquivo	6.14	0.37	69.83	2.04
Leitura de arquivo	6.84	0.50	38.00	2.87
Transferência de arquivo	28.46	0.50	91.71	3.57

Tabela 4: Desvio padrão e dispersão máxima do jitter em μs das interrupções PIT e FIQ

Supondo uma aquisição de dados do A/D, o período disponível para cada aquisição seria correspondente a 500μs, uma vez que a frequência de chamada do handler de ambas interrupções foi de $2kHz$. Ao considerar o pior caso para a interrupção PIT, transferência de arquivo, tem-se uma perda de 18.34% do tempo de aquisição devido ao jitter. Entretanto, ao considerar o mesmo cenário de transferência de arquivo para a FIQ, a perda de tempo de aquisição devido ao jitter é de 0.71%. Portanto, o uso da FIQ apresenta um resultado muito melhor, havendo uma redução da perda de 17.63%. Consequentemente, o uso da FIQ se revela mais vantajoso que o uso da PIT.

Os altos valores de jitter para interrupção PIT são esperados pois o Kernel desabilita as interrupções em alguns trechos de código com o objetivo de manter a consistência de suas estruturas de dados. Um exemplo disto está no código do escalonador de tarefas, a função scheduler(), presente em kernel/sched/core.c. O aumento de latência observado nos testes possivelmente está associado ao acréscimo de trechos de código com desabilitação de interrupções. Portanto, é esperado que certas tarefas aumentem a latência.

Na Tabela 4, observou-se uma pequena variação da latência da FIQ comparada com os resultados da PIT. Entretanto, esperava-se obter um valor de jitter praticamente nulo, mesmo com execução das cargas de teste, pois a princípio, o código do Kernel não desabilita a FIQ e esta é mais prioritária que as IRQs. Então presume-se que este jitter, com valor máximo de 3.57μs, esteja relacionado com operações de hardware realizadas pelo processador que atrasam ligeiramente o atendimento da FIQ.

6 Processo de Aquisição dos Dados do A/D

Na arquitetura descrita em 2, a interrupção Kpig_interrupt chama o handler do driver de aquisição de dados dos sensores do PIG. Este handler é responsável por inicializar a transferência desses dados para memória do sistema através do uso de um protocolo de comunicação associado a um dispositivo de acesso direto a memória, conhecidos respectivamente como protocolo SPI e dispositivo de DMA.

a Serial Peripheral Interface

A SPI (Serial Peripheral Interface) é um barramento serial síncrono do tipo full-duplex, ou seja, a transmissão e recepção ocorrem ao mesmo tempo. Quando uma aplicação requer transmissão de dados em apenas um sentido, o lado receptor deve enviar dados para que ocorra a transmissão. A interface SPI usualmente está presente em aplicações de comunicação com conversores A/D (analógico/digital) e memória flash. Por ser uma comunicação síncrona, a transmissão de cada bit depende de um sinal de clock. Este protocolo possui uma estrutura do tipo Master-Slave, onde existe apenas um master e um ou mais slaves. Sua estrutura física é composta basicamente por quatro pinos:

- Serial Clock (SPCK): Pino em que o master indica a taxa de transferência de dados e a sincronia da troca de dados nos pinos MOSI e MISO.
- Master Out Slave In (MOSI): Pino em que ocorre a transmissão de dados do master para o slave (linha controlada pelo master).
- Master In Slave Out (MISO): Pino em que ocorre a transmissão de dados do slave para o master (linha controlada pelo slave).
- Chip Select (CS): Pino permite a escolha de um slave pelo master.

O sinal de clock (SPCK) é um sinal aleatório gerado pelo Master cuja finalidade é sincronizar a amostragem e transmissão dos dados. A definição dos parâmetros de polaridade (CPOL) e fase (NCPHA) do clock determinam quando ocorrerá a amostragem. O CPOL indica a polaridade do sinal SPCK no estado inativo. Quando a fase é zero, tem-se a captura dos dados na primeira transição do clock,e quando a fase é um, a captura ocorre na segunda transição. Para CPOL=0 e NCPHA=0, a amostragem ocorre na primeira transição do clock, ou seja, na transição positiva e para CPOL=0 e NCPHA=1 a amostragem ocorre na segunda transição do clock, ou seja, na transição negativa. A Figura 16 ilustra este comportamento para todas as combinações possíveis de fase e polaridade.

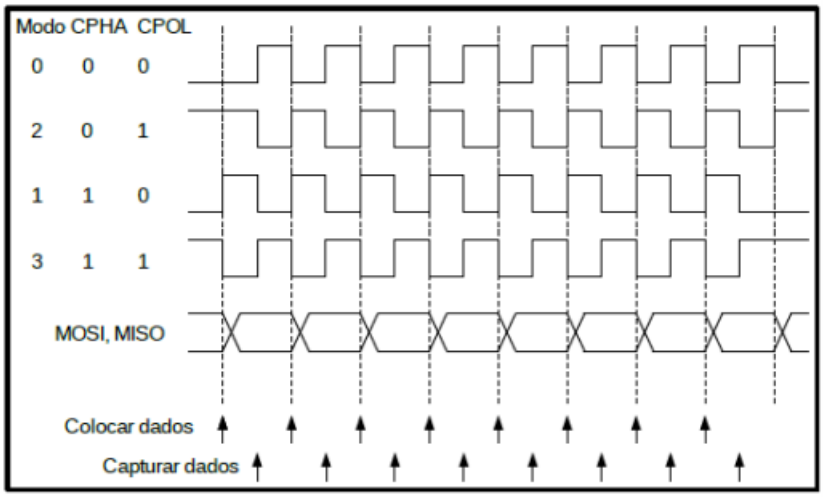

Figura 16: Modos de transferência [16]

Na interface SPI com múltiplos slaves, todos os slaves compartilham os pinos de clock, MISO e MOSI. Os pino CS permitem a seleção do slave que recebe ou envia dados. A Figura 17 ilustra este tipo de configuração.

A SPI no modo master gera internamente o sinal de clock (SPCK). Nessa geração é necessário informar, na programação, o baud de transferência. Ao habilitar a SPI, a transferência se inicia quando o processador escreve

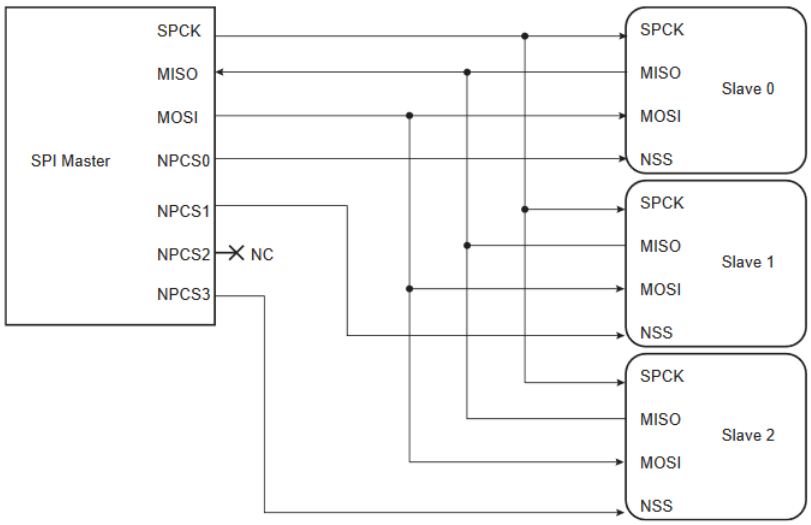

Figura 17: Configuração um master e múltiplos slaves [12]

no registro de transferência da SPI. Tal escrita inicializa imediatamente a transferência do dado no barramento da SPI. Enquanto que na linha MOSI ocorre a transmissão do dado, na linha MISO ocorre a captura de um dado, de acordo com as configurações de polaridade e fase da Figura 16. Normalmente não é desejável que o registro de recepção receba um novo dado antes da operação de leitura do dado presente. No microprocessador em questão, a interface SPI permite, através de uma flag, informar que a transferência de um novo dado não pode ocorrer enquanto o dado presente no registro de recepção não for lido. Sendo assim, esta flag assinala que o conteúdo do registrador de transmissão deve manter-se na espera de uma transmissão.

Uma flag encarregada de informar se o registro de transmissão está cheio (possui um dado para iniciar uma transmissão), ou vazio, indica ao processador se uma nova transmissão pode ocorrer.

No contexto do uso de uma FIQ para aquisição, foi necessário programar em hardware a interface SPI para viabilizar seu uso dentro de um handler FIQ, dado que neste handler devemos evitar o uso de APIs disponíveis para o Linux. Isto porque essas APIs dependem de operações bloqueantes como schedulers e spinlocks de gerenciamento de acessos concorrentes.

b Programação em Hardware da SPI

A programação da SPI segue o fluxograma da Figura 18. Além das configurações básicas, como definição de pinos, modo de fase e polaridade, o número de bits de transferência e entre outros, outras duas configurações são necessárias:

Proteção a overrun no buffer de recepção - Indica que a inicialização de cada transmissão ocorre apenas se o registro de recepção da SPI, SPI_RDR, estiver vazio, ou seja, se não houver algum dado ainda não lido.

Desativação do mode fault detection - Quando a SPI atua no modo master em um ambiente multi-master, um master externo pode tentar selecionar esta SPI como um slave, causando um erro de mode fault.

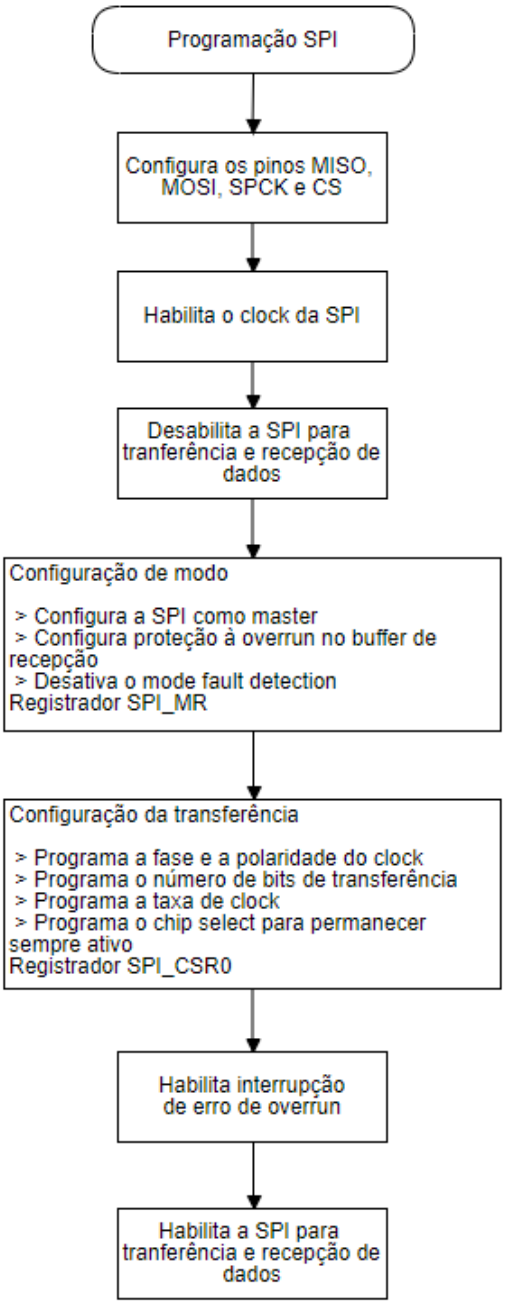

Figura 18: Programação da SPI

c Acesso Direto à Memória

Acesso Direto à Memória (DMA - *Direct Memory Access*) é o nome dado à ação de copiar dados de um recurso a outro sem o envolvimento da CPU nesse processo. A CPU gerencia a operação de DMA através de um dispositivo eletrônico conhecido por controlador de DMA (DMAC) [17]. O DMA permite que dispositivos de E/S acessem a memória diretamente, sem o uso do processador. Por dispensar a CPU da realização de transferências de dados entre um dispositivo de hardware e a memória do sistema, o uso do DMA é comum.

A compreensão do funcionamento do DMAC exige a definição dos elementos abaixo [18].

Barramento de dados - Conjunto de linhas que transmitem um dado. Cada linha transmite um bit; assim barramentos de 16 linhas transmitem um dado de 16 bits. Este barramento é bi-direcional, ou seja, há entrada ou saída de dados dependendo se a operação é leitura ou escrita.

Barramento de endereço - Conjunto de linhas de endereçamento de dispositivos que indicam a fonte ou o destino dos dados trasmitidos pelo barramento de dados. Um barramento de 16 linhas endereça $2^{16} = 16K$. Este barramento é unidirecional pois ou os dados fluem para memória ou para os dispositivos de E/S.

A Figura 19 ilustra o funcionamento geral do DMAC. O processador configura o DMAC para a transferência de dados, informando o sentido da transferência (memória para periférico, periférico para memória), o tamanho da palavra de transferência, a quantidade de transferências consecutivas a realizar e o canal de DMA da transferência (cada periférico está associado a um canal, isto é, um canal para SPI, um outro para o UART). Após essa configuração, a transferência de dados se inicia quando o DMAC envia o sinal de requerimento do barramento (BR - bus request) ao processador, requisitando o controle do barramento. Em resposta, o processador termina suas tarefas em execução e passa o controle do barramento ao DMAC através do sinal de concessão do barramento (BG - bus grant). Posteriormente à recepção do sinal BG, o DMAC toma o controle do barramento e realiza a transferência de acordo com as configurações realizadas pelo processador. O DMAC e a memória são endereçados pelo barramento de endereços. O endereçamento do controlador de DMA ativa as linhas de seleção de registro (RS - Register Selects) e seleção de dispositivo (DS - Device Select) para que o controlador execute operações de leitura ou escrita no local de memória selecionado. O DMAC emite o sinal de acknowledge quando o sistema inicia a transferência e o barramento de dados transfere dados entre o periférico externo e a memória. O DMAC emite um sinal de interrupção ao processador ao terminar a transferência de todas as palavras [19].

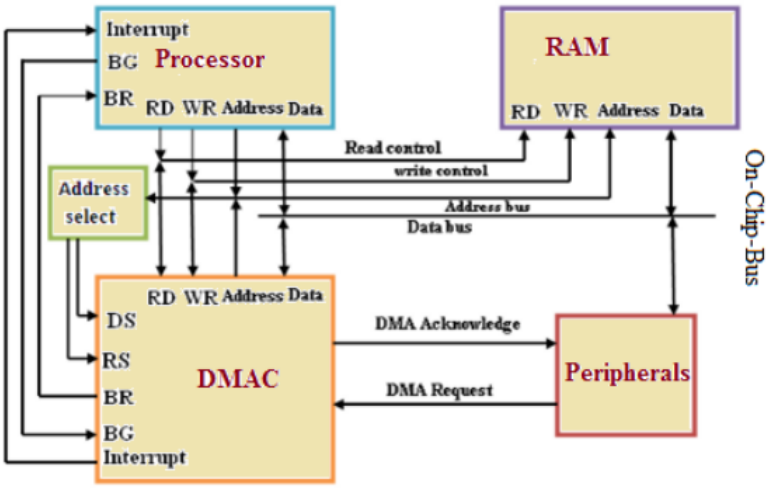

Figura 19: Método de transferência de DMA [19]

Normalmente as transferências de E/S procedem em blocos de palavras. O DMAC possui um contador responsável pela contagem do número de palavras transmitidas, permitindo que o controlador conheça o fim da transmissão. Enquanto o contador não atinge o valor desejado, o DMAC incrementa o endereço de acesso a memória.

De maneira geral a inicialização do DMAC pela CPU requer as seguintes informações.

- O canal a ser usado.
- O sentido da transferência.

- O tamanho da transferência (byte, halfword, word)
- A quantidade de transferências a serem realizadas.
- O endereço de origem e de destino.

Neste projeto, o DMA está em aplicação com uma SPI a fim de realizar transferências de dados do conversor A/D para a memória. Uma vez que a SPI é bi-direcional, são necessários dois canais de DMA, um para a transferência e outro para a recepção. A tabela do Apêndice F discrimina os periféricos capazes de se comunicarem com o DMA do processador em uso. Os canais de identificação 1 e 2 correspondem, respectivamente, ao canal de transmissão e de recepção da SPI. Nas seções 6d e 6e apresenta-se a aplicação desses identificadores, respectivamente, na habilitação dos canais de recepção e transmissão.

Ao canal de recepção do DMA atribui-se a recepção de dados provenientes do conversor A/D. Ao canal de transmissão atribui-se a transmissão de um buffer com valores nulos (fictícios). O DMAC acessa na memória o buffer de transmissão e transmite cada elemento do buffer para o registro de transmissão da SPI. Cada escrita no registro de transmissão da SPI inicializa a transferência de acordo com a descrição presente na Seção 6a. À medida que ocorre a transmissão de um buffer nulo pelo canal de transmissão, o canal de recepção transmite o dado presente no registro de recepção da SPI para um buffer de destino presente na memória. Dessa forma, o canal de recepção apresenta fluxo do periférico para memória, ao passo que o canal de transmissão apresenta fluxo da memória para o periférico.

A imagem abaixo explicita o sistema de aquisição de dados em aplicação. Nota-se o uso de um CPLD, *Complex Programmable Logic Device* ou Dispositivo Complexo Lógico Programável, em comunicação com o conversor A/D. O CPLD é um dispositivo que permite a construção de circuitos digitais. No âmbito deste projeto, o papel do CPLD limita-se ao de uma ponte de comunicação entre o conversor e a SPI. Neste sistema transmite-se os dados do conversor A/D ao CPLD, que por sua vez os transfere pela linha MISO da SPI. Sendo assim, os dados chegam ao DMA atraveś do seu canal de recepção para o armazenamento em um buffer na memória.

Os próximos tópicos apresentam o processo de programação da transferência e da recepção de um buffer via DMA.

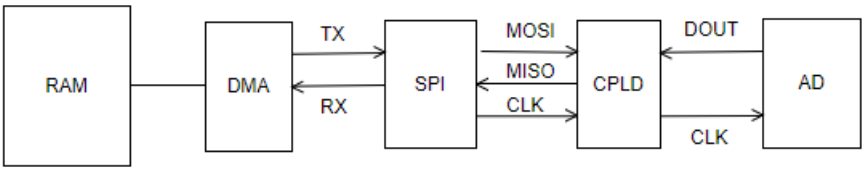

Figura 20: Ligação SPI-DMA e A/D

d Configurando a Recepção via SPI-DMA

A programação da recepção via SPI-DMA segue o fluxograma da Figura 21.

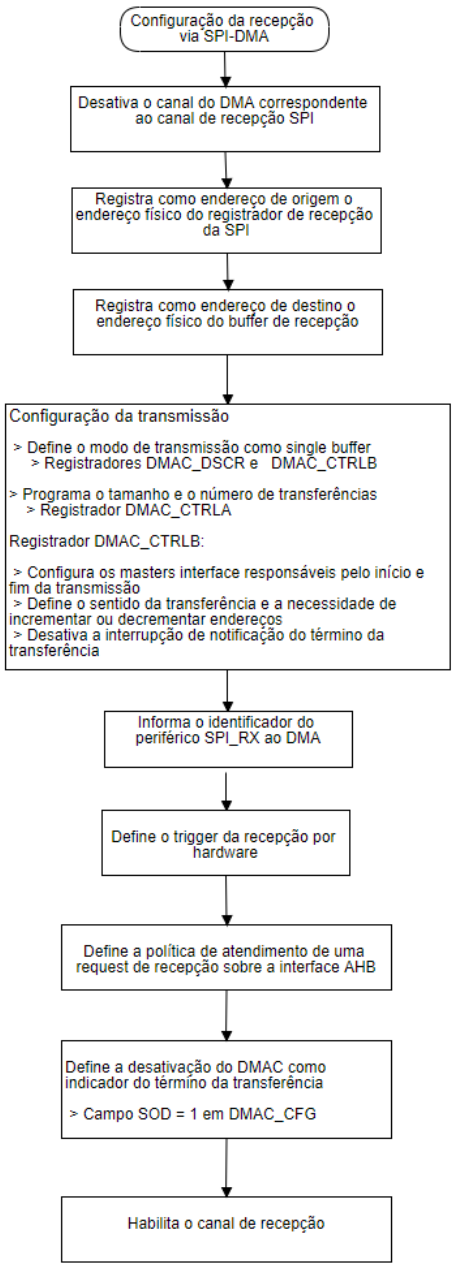

Figura 21: Configuração da recepção via SPI-DMA

A interface de DMA do processador em aplicação, apresenta dois masters, AHB-Lite interface 0 e AHB-Lite interface 1. Estes dois masters são responsáveis por realizar leituras e escritas sobre a interface de DMA. Nessa aplicação, o DMA deve se comunicar com a memória e com um periférico. Sendo assim, definimos que o master 0 é a interface de comunicação entre a memória e o DMA e o master 1 é a interface de comunicação entre o DMA e o periférico. Nesse contexto, é necessário informar qual master atua sobre os recursos de origem e qual master atua sobre os recursos de destino da transferência. Nos campos SIF (*Source Interface Selection Field*) e DIF (*Destination Interface Selection Field*) do registrador DMAC_CTRLB, efetua-se esta distinção. Uma vez que na recepção tem-se como origem dados provenientes de um periférico, ao campo SIF atribuímos o master 1. Nesse mesmo sentido, sendo a destinação da transferência um buffer alocado na memória, ao campo DIF atribuímos o master 0.

A recepção usa, como endereço de origem, o endereço do registrador de recepção da SPI (SPI_RDR) e, como endereço de destino, o endereço do buffer de recepção alocado na memória, portanto o sentido da transmissão é de um periférico para uma área de memória (PER2MEM), com o endereço de origem sendo fixo e o endereço de destino incrementado (buffer).

Na transferência de um dado do periférico para memória, o DMAC armazena o dado em uma FIFO do canal de recepção do DMA (RX-FIFO), para depois o transferir ao barramento AHB de acesso à memória. O barramento AHB deve acessar a fifo de recepção quando esta apresentar o dado disponível. No campo FIFOCFG do registrador DMAC_CFG, definimos que um requerimento de acesso a fifo, proveniente do barramento AHB, é atendido apenas se houver dado disponível para um único acesso [18].

O código correspondente a esta programação encontra-se no Apêndice G.

e Configurando a Transferência via SPI-DMA

A programação da transferência via SPI-DMA segue o fluxograma da Figura 22.

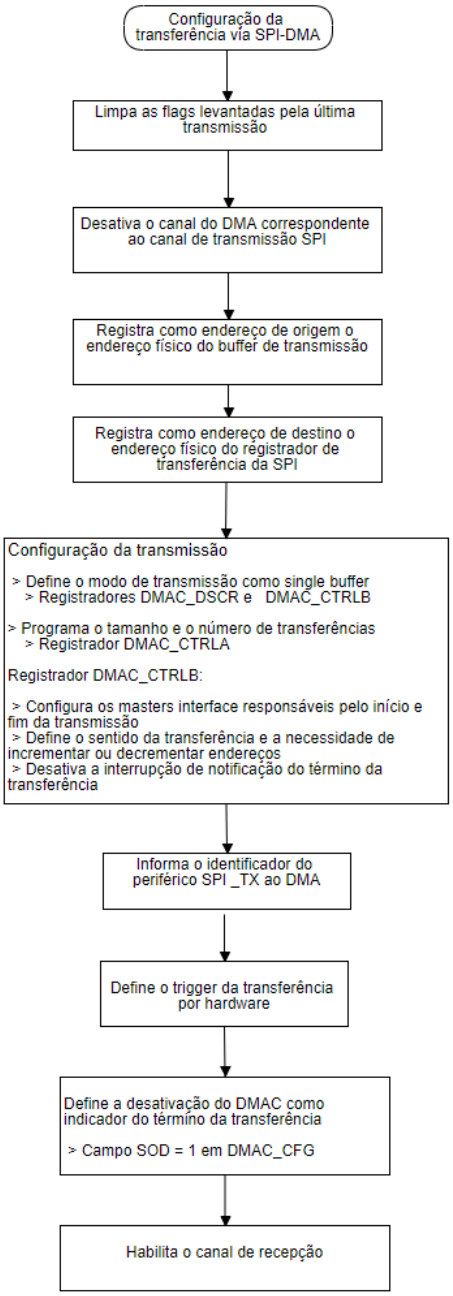

Figura 22: Configuração da transferência via SPI-DMA

Na seção anterior, definiu-se o master 0 como a interface de comunicação entre a memória e o DMA e o master 1 como a interface de comunicação entre o DMA e o periférico. No canal de transmissão, tem-se como origem um buffer de transmissão, então deve-se atribuir o master 0 ao SIF. Sendo a destinação um periférico, ao campo DIF atribui-se o master 1.

A transmissão usa como endereço de origem, o endereço do buffer de transmissão e, como endereço de destino, o endereço do registrador de transmissão da SPI (SPI_TDR), portanto o sentido da transmissão é de uma área da memória para um periférico (MEM2PER), sendo o endereço de origem incrementado (buffer) e o endereço de destino fixo.

O código correspondente a esta programação encontra-se no Apêndice H.

f Alocação do DMA Buffer

Os dispositivos de E/S possuem endereços do tipo bus address. Quando um dispositivo executa uma operação de DMA para ler ou escrever na memória, os endereços nessas operações são bus addresses. Em alguns sistemas esses endereços são idênticos aos endereços físicos da CPU. Quando não, o sistema possui uma IOMMU (*Input–Output Memory Management Unit*) para realizar mapeamentos entre um endereço físico e o bus addresses [20]. Na arquitetura do microprocessador em uso, bus address é equivalente ao endereço físico da CPU.

Posto que o DMA trabalha no bus address space, o Kernel do Linux fornece uma API para alocar o buffer envolvido na operação de DMA.

Para acessar esta API é necessário incluir o arquivo <linux/dma-mapping.h>.

O trecho de código abaixo, exemplifica o uso desta API.

```
dma_addr_t dmaphys;
uint16_t *dmabuf;

dmabuf = kmalloc(buffer_size, GFP_KERNEL | GFP_DMA);
if (!dmabuf) {
                    printk(KERN_INFO "kmalloc dmabuf failed.\n");
                    return -1;
}

dmaphys = dma_map_single(NULL, dmabuf, buffer_size, DMA_BIDIRECTIONAL);
if (dma_mapping_error(NULL, dmaphys)) {
printk(KERN_INFO "dma_map_single failed.\n");
            return -1;
 }
```

A flag GFP_KERNEL indica que a alocação de memória é no espaço do Kernel.

A flag GFP_DMA indica que a alocação deve ocorrer na zona de memória DMA-capable memory. A existência desta zona de memória é devida incapacidade de alguns dispositivos de E/S operarem com 32-bits de endereçamento. Uma alocação nesta área garante o acesso do DMA a esses dispositivos de endereçamento limitado. Em outras palavras, um dispositivo de E/S com limitação de endereço de 16-bit deve usar esta flag para que ocorra somente alocação de memória endereçável em 16-bits.

A função dma_map_single mapeia uma parte da memória virtual do processador para que possa ser acessado pelo dispositivo de E/S e retorna o endereço de DMA da memória [21]. A chamada de dma_mapping_error() verifica se houve algum erro no mapeamento.

Quando a operação de DMA termina é necessário desfazer o mapeamento usando a função dma_unmap_single().

g Interface SPI DMA

A partir do processo de programação apresentado acima desenvolveu-se uma interface para a programação da transmissão e da recepção via SPI-DMA.

Primeiro inicializa-se a tranferência com função spi_transfer explicita abaixo.

```
void spi_transfer(struct spi_dma_transfer*);
```

Onde a struct spi_dma_transfer possui a seguinte definição.

```
struct spi_dma_transfer {

    dma_addr_t tx_dmaphys;
    dma_addr_t rx_dmaphys;
    u16 buffer_size;
    u8 transfer_width;
    u8 clock_phase;
    u8 clock_polarity;
    u8 bits_per_word;
    u32 speed_hz;
};
```

tx_dmaphys - Buffer de transmissão alocado em uma região de memória acessível pelo controlador de DMA.

rx_dmaphys - Buffer de transmissão alocado em uma região de memória acessível pelo controlador de DMA.

buffer_size - Número de transferências.

transfer_width - Tamanho da transferência realizada pelo controlador de DMA. As macros BYTE, WORD e HALF_WORD estão disponibilizas no arquivo spi_dma.h.

clock_phase - Fase do clock.

clock_polarity - Polaridade do clock.

bits_per_word - Número de bits da palavra.

speed_hz - Frequência de transferência em Hz.

O usuário deve chamar as funções spi_dma_init e spi_dma_exit respectivamente, nas funções init e exit do módulo de Kernel.

A função spi_dma_init mapeia endereços base de acesso aos registros de GPIO, SPI e DMA. A spi_dma_exit remove os mapeamentos de memória que foram necessários e desabilita o clock da SPI e do DMA.

As funções reception_done e transfer_done informam, respectivamente, o término de uma recepção e de uma transferência.

7 Influência de alocação na execução da FIQ

O módulo de kernel que programa a FIQ e o TC é do tipo *loadable kernel module*. É um código de extensão que pode ser inserido e retirado do Kernel em execução, sem a necessidade de fazer um reboot do sistema. O sistema operacional aloca este módulo na área de memória denominada por *Kernel Module Space*. A função vmalloc realiza esta alocação adotando a técnica *Demand Paging*, que consiste em adiar ao máximo possível a alocação de uma *page frame*, isto é, tardá-la até que o processo demande o acesso de uma página que não está na RAM. Este tipo de demanda causa a exceção *page fault* que é tratada pelo Kernel. Assim, o código alocado no *Kernel Module Space* gera, de forma imprevisível, *page faults* que comprometem a execução de certas funções dentro do handler da FIQ. Quando uma *page fault* ocorre dentro de uma FIQ, o sistema sofre um *Kernel Panic*, isto é, encontra-se em estado de impossível recuperação pois o Kernel não é capaz de interromper a FIQ para o tratamento dessa exceção [22].

Na estratégia *Demand Paging*, apenas o processo que requeriu a página sofre atualização da *page table*. Dessa forma, não há atualização simultânea das páginas de todos os processos. Portanto, uma troca de contexto pode gerar *page fault* quando um novo processo requer o endereço virtual que gerou a atualização da *page table* do processo anterior. No cenário da FIQ, mesmo que o processo interrompido por ela possua todas as *pages tables* requeridas pela FIQ, existe a possibilidade de ocorrência de *page fault* pois a atualização das *pages tables* de um processo não implica em uma atualização para todos os processos. Além disso, o processo a ser interrompido pela FIQ é imprevisível, portanto a FIQ requerer acesso a área de memória alocada por vmalloc é inviável devida possibilidade de ocorrências de *page faults*.

Para que não ocorra *page fault* dentro do handler da FIQ, toda memória requerida pelo módulo deve ser alocada com kmalloc pois esta função realiza alocação contínua de memória, retornando um endereço virtual que está diretamente mapeado à um endereço físico; isto é, não há a necessidade de atualização de páginas sob demanda.

O trecho de código da Figura 23 é a função responsável pela alocação do módulo de Kernel na área de memória *Kernel Module Space*.

```
#ifdef CONFIG_MMU
void *module_alloc(unsigned long size)
{
  void *p = __vmalloc_node_range(size, 1, MODULES_VADDR, MODULES_END,
        GFP_KERNEL, PAGE_KERNEL_EXEC, 0, NUMA_NO_NODE,
        __builtin_return_address(0));
  if (!IS_ENABLED(CONFIG_ARM_MODULE_PLTS) || p)
    return p;
  return __vmalloc_node_range(size, 1,  VMALLOC_START, VMALLOC_END,
        GFP_KERNEL, PAGE_KERNEL_EXEC, 0, NUMA_NO_NODE,
        __builtin_return_address(0));
}
#endif
```

Figura 23: Função de alocação do módulo de Kernel

Alteramos a função acima para fazer a alocação do módulo com kmalloc, conforme mostra a Figura 24.

```
#ifdef CONFIG_MMU
void *module_alloc(unsigned long size)
{
  return kmalloc(size, GFP_KERNEL);
}
#endif
```

Figura 24: Função de alocação do módulo de Kernel adaptada com kmalloc

Uma vez que mudamos a área de alocação, a função de liberação de memória também necessita de alteração. Por isso, é necessário alterar a função module_memfree presente em /kernel/module.c:

```
void __weak module_memfree(void *module_region)
{
  vfree(module_region);
}
```

Dado que a alocação atual é feita através da utilização da função kmalloc, devemos usar a função kfree para liberar a memória:

```
void __weak module_memfree(void *module_region)
{
  kfree(module_region);
}
```

No código do Apêndice D, a escrita e a leitura nos registradores do periférico são possíveis através das macros tc_writel e tc_readl. A variável tc_base armazena o endereço virtual de acesso ao periférico, entregue pela função de mapeamento ioremap. A função ioremap cria um mapeamento dinâmico para o endereço físico passado como argumento e aloca um endereço virtual.

```
#define tc_writel(value, reg, tc)\
  __raw_writel((value), tc_base + tc + ATMEL_TC_##reg)

#define tc_readl(reg, tc)\
  __raw_readl(tc_base + tc + ATMEL_TC_##reg)
```

Figura 25: Definição das macros tc_writel e tc_readl

A variável tc representa o offset do canal escolhido, uma vez que o módulo do TC possui três canais (Seção 4e).

A execução do handler da FIQ com o mapeamento dinâmico do ioremap mostra-se instável pois a área de memória de retorno desta função corresponde a mesma área de memória retornada pela função vmalloc. Portanto, a escolha de não usar mapeamento dinâmico se justifica pelas mesmas razões de inviabilidade do uso do vmalloc. Uma possível solução para este problema é o emprego do mapeamento estático.

O mapeamento estático consiste em definir e associar de forma estática, um endereço virtual ao endereço físico do periférico. O Kernel realiza este tipo de mapeamento no momento do boot do sistema, garantindo desta forma que este endereço virtual seja sempre visível durante toda execução do sistema.

Quando para um dado endereço físico existe um endereço virtual proveniente de um mapeamento estático, a função ioremap atua retornando este endereço virtual já definido e não realiza o mapeamento dinâmico. Logo, a função ioremap é válida na obtenção do endereço virtual do mapeamento estático.

a Mapeamento Estático de Periféricos

Geralmente o Kernel realiza o mapeamento de memória através de um vetor da seguinte estrutura (a definição está em arch/arm/include/asm/mach/map.h):

```
struct map_desc {
  unsigned long virtual;
  unsigned long pfn;
  unsigned long length;
  unsigned int type;
};
```

Onde,

virtual - Endereço virtual referente a definição do map_desc;

pnf - *Page frame number* associado ao endereço físico do periférico;

length - Tamanho da memória mapeada;

type - Tipo de memória do mapeamento.

No desenvolvimento deste projeto, os periféricos em uso são SPI, TC e PMC (*Power Management Controller*). Sendo assim, definimos o vetor de map_desc do mapeamento estático como:

```
#define AT91_SYSC_PHYS  UL(0xffffc000)
#define AT91_SPI1_PHYS  UL(0xf0004000)
#define AT91_TC1_PHYS   UL(0xf800c000)
static struct map_desc at91_iomap_desc[] __initdata = {
  {
    .virtual  = ITCM_OFFSET,
    .pfn    = __phys_to_pfn(AT91_SYSC_PHYS),
    .length   = SZ_16K,
    .type   = MT_DEVICE,
  },
  {
```

```
    .virtual  = ITCM_OFFSET + SZ_16K,
    .pfn      = __phys_to_pfn(AT91_SPI1_PHYS),
    .length   = SZ_16K,
    .type     = MT_DEVICE,
  },
  {
      .virtual  = ITCM_OFFSET + 2*(SZ_16K),
        .pfn    = __phys_to_pfn(AT91_TC1_PHYS),
      .length   = SZ_16K,
    .type     = MT_DEVICE,
  }
};
```

Na escolha do endereço virtual estático devemos assegurar que este não será mapeado para uma outro endereço físico durante a execução de um processo. Por esse motivo, escolhemos como endereço virtual um endereço na região *ITCM mapping area*.

A memória TCM (*Tightly-Coupled Memory*) possui localização próxima a CPU para reduzir a latência de acesso. Geralmente empregamos esta região de memória para armazenar dados que requerem alta velocidade de acesso, como por exemplo dados para tarefas executadas em tempo-real. Uma vez que esta é uma área de memória reservada e sem utilização, optamos por utilizá-lo como endereço virtual do mapeamento estático [23].

No processo de boot do Linux, o Kernel realiza o mapeamento estático. A estrutura machine_desc carrega as informações relacionadas à inicialização do Kernel na placa hospedeira:

```
#define DT_MACHINE_START(_name, _namestr)    \
static const struct machine_desc __mach_desc_##_name   \
__used                \
__attribute__((__section__(".arch.info.init"))) = { \
  .nr    = ~0,        \
  .name    = _namestr,

#endif
```

Normalmente, a inicialização da estrutura machine_desc, através da macro acima, está no arquivo de inicialização da placa. Este arquivo costuma estar presente em uma pasta denominada mach-xxx. No âmbito deste projeto, este arquivo possui o nome at91sam9.c e está em arch/arm/mach-at91.

No arquivo at91sam9.c, alteramos a inicialização da macro DT_MACHINE_START, acrescentando a função at91_map_io de mapeamento estático:

```
DT_MACHINE_START(at91sam9x5_dt, "Atmel AT91SAM9")
  /* Maintainer: Atmel */
  .init_machine = at91sam9x5_dt_device_init,
  .dt_compat  = at91sam9x5_board_compat,
  .map_io   = at91_map_io,
MACHINE_END
```

Na função at91_map_io, a iotable_init realiza o mapeamento através da chamada da função create_mapping:

```
static void __init at91_map_io(void)
{
  /* Map static IO for FIQ module usage */
  iotable_init(at91_iomap_desc, 2);
}
```

Conforme abordado, a chamada para o mapeamento estático segue o seguinte fluxo [24].

- Em /arch/arm/kernel/setup.c a execução da função setup_arch() realiza a chamada da paging_init;
- A função paging_init executa a devicemaps_init;
- A devicemaps_init realiza a chamada da função at91_iomap_desc através do map_desc registrado a partir da macro DT_MACHINE_START executando o trecho mdesc->map_io();

8 Adaptação da Arquitetura do Sistema para o uso da FIQ

O sistema de aquisição da Figura 3 possui um conjunto de drivers que podem ser divididos em dois grupos:

- Drivers de obtenção dos dados dos sensores palito e do odômetro (instrumento de medição de distância percorrida).
- Drivers de controle, como por exemplo driver gerador de log, drivers de comunicação serial e de aceleração do instrumento.

Nesse projeto, o jitter é um parâmetro crítico na aquisição de dados. Então, o uso da FIQ é mais adequado para a chamada periódica dos drivers de aquisição. Adaptar todos os outros drivers para atender as restrições da FIQ seria uma tarefa árdua e desnecessária pois não apresenta vantagens significativas. Por esse motivo atribuímos a chamada dos drivers de aquisição à FIQ e os demais drivers à Kpig_interrupt. A Tabela 5 discrimina o papel de cada tipo de interrupção.

Kpig_FIQ_interrupt	Kpig_interrupt
• Incremento do timestamp. • Chamada da função de aquisição de dados dos sensores palito e do odômetro através de uma varredura. • Atualização da FIFO módulo de Kernel principal.	• Chamada periódica dos outros drivers. • Atualização da FIFO do módulo de Kernel principal.

Tabela 5: Discriminação do papel das interrupções Kpig_FIQ_interrupt e Kpig_interrupt

O handler Kpig_interrupt é de uma interrupção periódica do tipo PIT. O padrão do Kernel do Linux é usar essa interrupção como clock do sistema, então nessa arquitetura foi necessário configurar o Kernel para substituir o PIT pelo TC na função de clock do sistema. Na arquitetura adaptada para o uso da FIQ, o processador usa o TC como trigger da FIQ então a interrupção PIT volta a ser o clock do sistema. Uma vez que as duas interrupções de tempo estão em uso, decidimos usar a interface RTC (*Real-Time Clock*) para a chamada periódica do Kpig_interrupt.

O RTC é um relógio de hardware cuja principal função é manter a contagem do tempo. Esse dispositivo funciona independente do sistema operacional. Normalmente possui uma bateria própria de alimentação para que continue operando mesmo com o processador desligado. O Kernel do Linux disponibiliza uma interface RTC que permite o seu uso para geração de interrupções periódicas.

A Figura 26 mostra a arquitetura proposta para o uso da FIQ. Em comparação com a arquitetura atual, o timestamp é incrementado pela FIQ pois desejamos usá-la com frequência maior que a RTC. Os handles de interrupção da FIQ e da RTC acessam a mesma FIFO. O handler Kpig_interrupt desabilita a FIQ ao acessar a FIFO para que não haja erro de timestamp no registro de seus drivers na FIFO.

A próxima seção explicita o uso do RTC na geração da interrupção periódica.

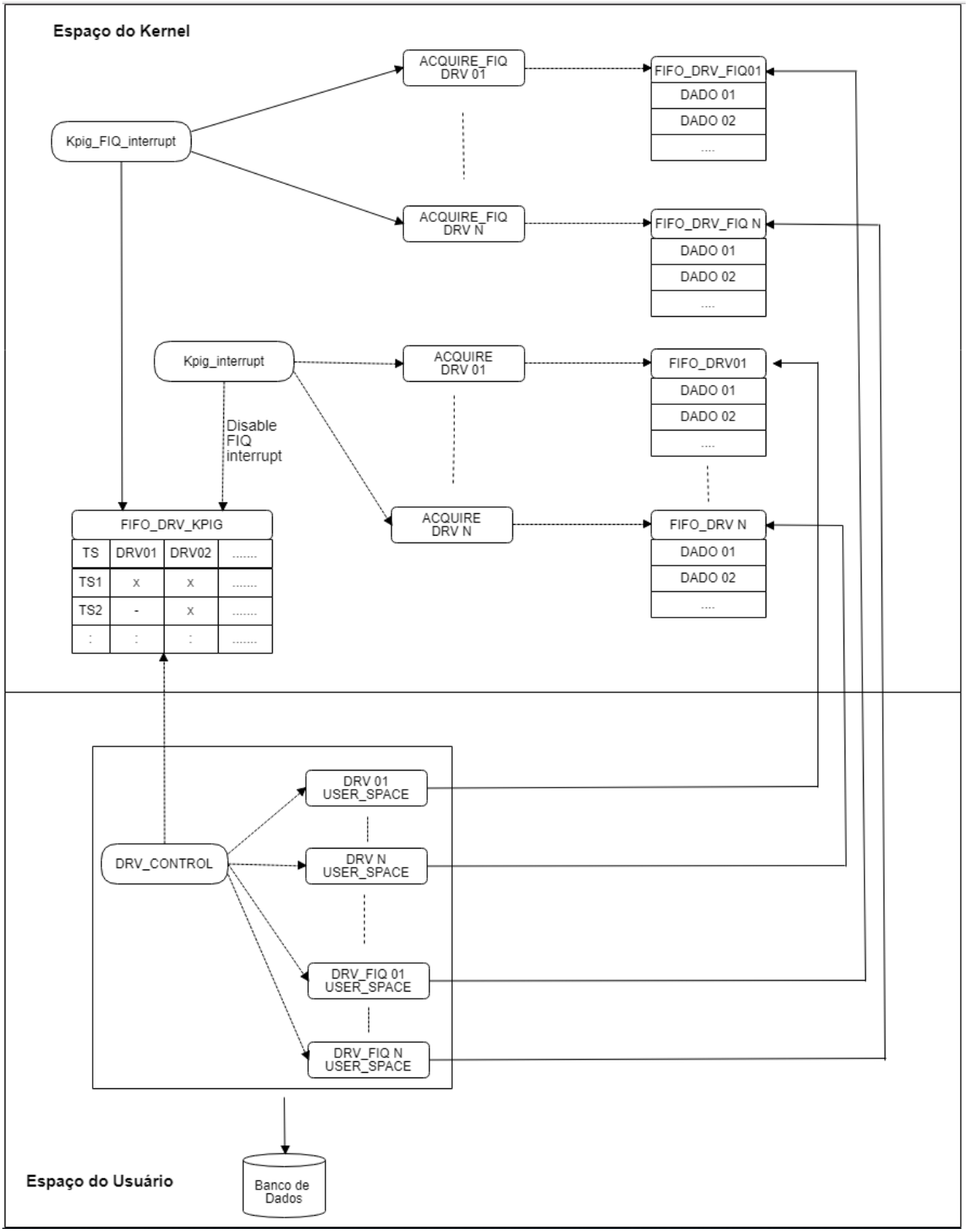

Figura 26: Arquitetura do sistema de aquisição de dados do PIG adaptada para o uso da FIQ

a O Uso do RTC na Geração de Interrupção Periódica

No trecho de código da Figura 27 , a estrutura de dados rtc_task referencia o handler de interrupção rtc_interrupt_handler.

A função rtc_class_open retorna a estrutura rtc_device necessária para o registro do handler como uma interrupção periódica.

A rtc_irq_register associa a interrupção rtc_interrupt_handler ao dispositivo rtc0. Enquanto que, a rtc_irq_set_freq registra a frequência 2^n Hz para a geração da IRQ.

A partir da função rtc_irq_set_state podemos habilitar, com state 1, ou desabilitar, com state 0, a geração de interrupção periódica com a frequência registrada pela rtc_irq_set_freq.

```
#include <linux/rtc.h>

static rtc_task_t m_rtc_task = {
  .func = rtc_interrupt,
  .private_data = NULL,
};

static void rtc_interrupt_handler (void *private_data)
{
    RTC handler code
}

struct rtc_device mrtc_device = rtc_class_open("/dev/rtc0");
rtc_irq_register(m_rtc_device, &m_rtc_task);
rtc_irq_set_freq(m_rtc_device, &m_rtc_task, rtc_freq);
rtc_irq_set_state(m_rtc_device, &m_rtc_task, 1);
```

Figura 27: Uso do RTC para a geração de interrupção periódica

A função rtc_irq_set_freq permite registrar no máximo 128Hz pois este é o valor padrão definido pelo Kernel. Na atual arquitetura de aquisição, o handler Kpig_interrupt é uma função periódica de 1024Hz. Para adotarmos a RTC como geradora dessa interrupção periódica é preciso modificar esse limite de 128Hz para 1024Hz. Essa alteração é possível modificando o parâmetro HZ_FIXED de 128 para 1024 no arquivo Kconfig presente em arch/arm:

```
config HZ\_FIXED
        int
        default 200 if ARCH\_EBSA110
        default 128 if SOC\_AT91RM9200  % Alterar aqui para 1024
        default 0
```

9 Resultados

Para avaliar o uso da FIQ no processo de aquisição, usamos o software PigManager de gerenciamento do PIG, desenvolvido pelo CPTI. A Figura 28 ilustra a interface com o usuário dessa ferramenta.

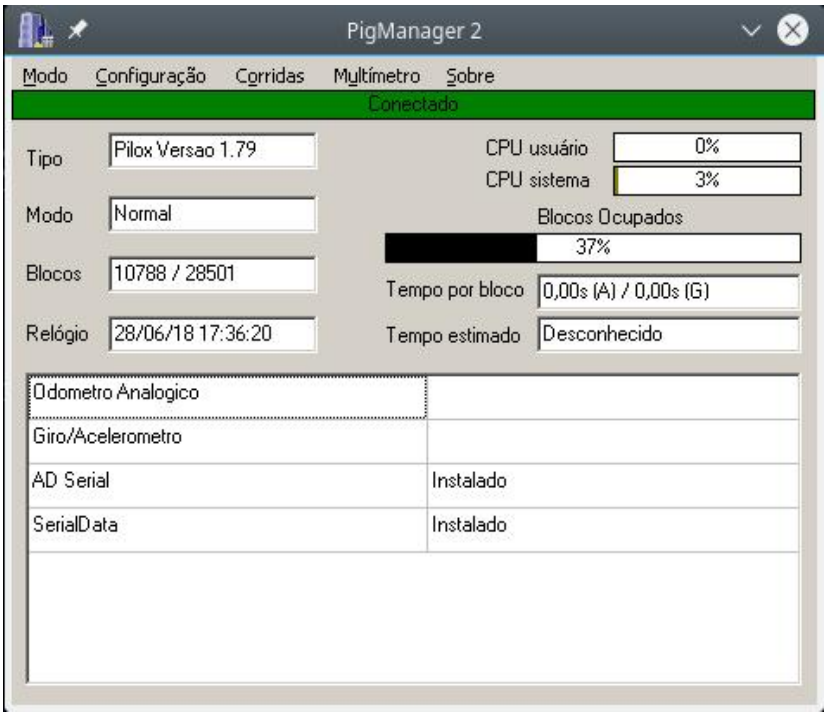

Figura 28: Software PigManager

Este software possui três modos de funcionamento: Parado, Normal e Multímetro. O modo Parado é o modo de configuração que permite mudar o número de canais e a taxa de aquisição de dados dos sensores palito e do odômetro. Quando o PigManager está nesse modo, não há aquisição de dados. A aquisição começa quando mudamos o modo para Normal ou Multímetro. No modo Normal o sistema de aquisição grava os dados adquiridos em blocos no SD Card e os disponibiliza para baixar. No modo Multímetro não há gravação dos dados dos sensores. Este modo apenas permite uma visualização gráfica em tempo real da leitura dos sensores.

Inicialmente observamos que o incremento do relógio da interface mostrada na Figura 28 opera corretamente. Esse foi o primeiro indicador do funcionamento da FIQ pois o incremento do relógio é o incremento do timestamp. Caso o processador não executasse a FIQ, não haveria incremento do timestamp e o relógio estaria parado na interface do PigManager.

Para validar a leitura dos sensores, usamos o Modo Multímetro com a placa Arietta conectada a um sensor palito e variamos manualmente a inclinação do palito a partir da posição horizontal até atingir a inclinação máxima que deve corresponder a 3.2 volts na leitura.

As Figuras 29 e 30 mostram, respectivamente, 0 volts como leitura do palito para posição horizontal e 3.2 volts como leitura para a inclinação máxima. Assim, confirmamos o funcionamento da interface SPI-DMA na realização de transferências de dados do conversor A/D para a memória.

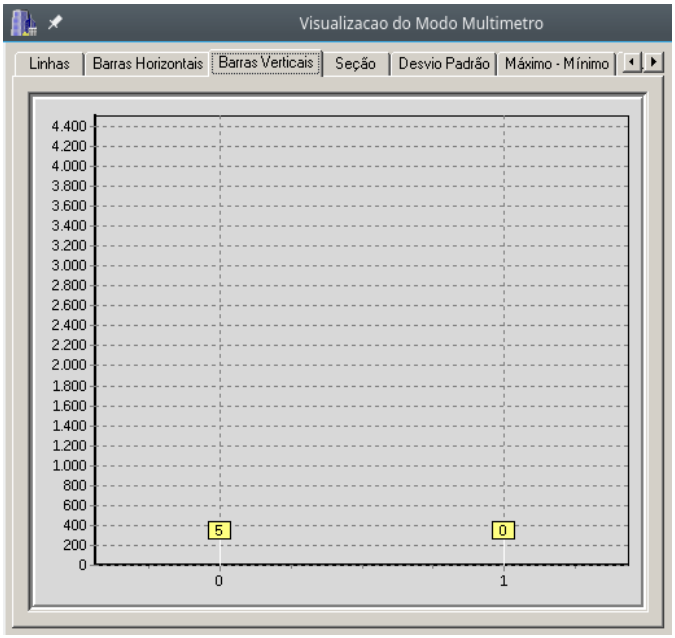

Figura 29: Leitura do sensor palito para a posição horizontal

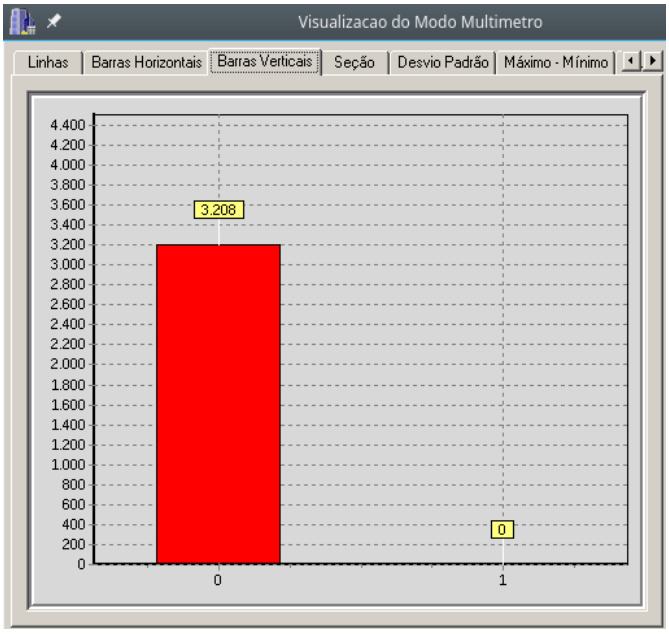

Figura 30: Leitura do sensor palito para a inclinação máxima

Após a confirmação do valor de leitura do sensor procuramos determinar a quantidade máxima de canais suportados para 512Hz, 1024Hz e 2048Hz. Na Figura 31 temos a interface de configuração da quantidade de canais e da frequência de aquisição. Realizamos a configuração de canais alterando os valores da grade principal. Alteramos a frequência de aquisição no campo Frequência, destacado em vermelho. A interface exibe o número total de canais resultantes dessa configuração no campo Total, também destacado em vermelho.

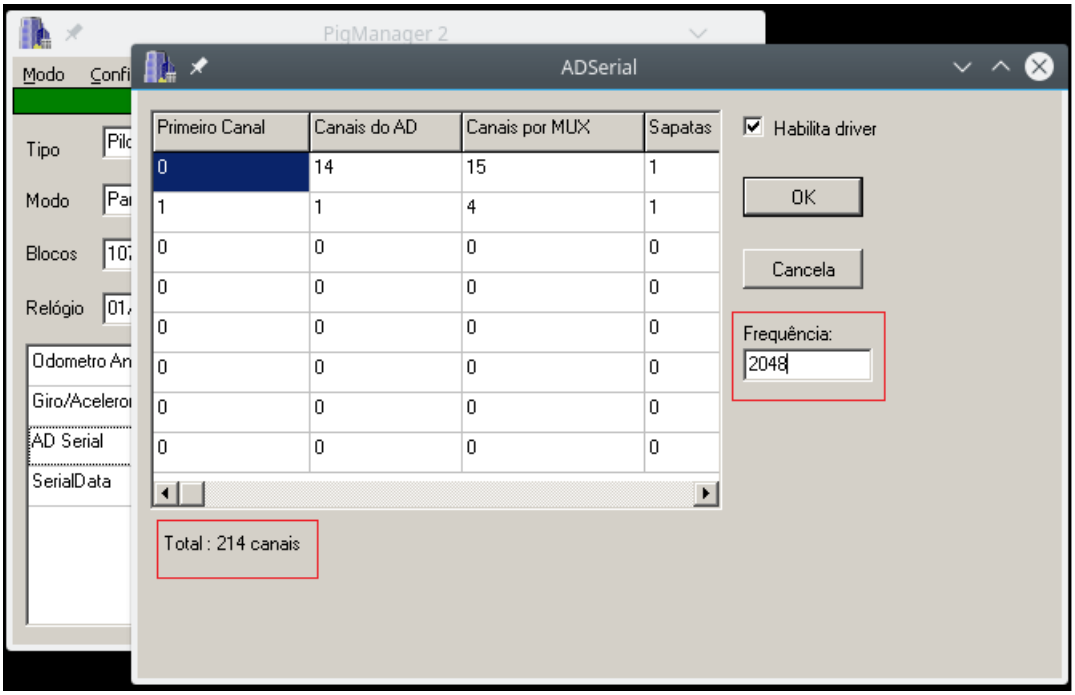

Figura 31: Configuração da quantidade de canais e da frequência de aquisição

O driver de aquisição exibe uma mensagem de erro quando o conversor A/D não consegue converter todos os dados a tempo. Isso ocorre quando a quantidade de canais que configuramos ultrapassa a quantidade máxima de canais suportada pelo sistema. Para chegarmos ao número máximo de canais para cada frequência adotamos a seguinte estratégia.

1. Começamos com 80 canais que é uma quantidade suportada por todas as três frequências de aquisição.
2. Passamos para o modo Parado.
3. Fixamos a frequência de aquisição.
4. Aumentamos o número de canais em uma unidade.
5. Passamos o software para o modo Normal.
6. Caso não seja exibida nenhuma mensagem de erro na aquisição, passamos para o modo Parado e voltamos ao passo 4. Caso alguma mensagem de erro seja exibida, atingimos a capacidade máxima de canais.

Através deste método, chegamos aos resultados da Tabela 6. Estes resultados mostram o desempenho da aquisição para um conversor A/D de 500KHz, que por precaução o sistema usa a 480kHz. Assim, a capacidade máxima do A/D para uma aquisição a 512Hz e sem latência é 480kHz/512 = 937 canais.

Frequência (Hz)	Máximo de Canais PIT	Máximo de Canais FIQ	Máximo Ideal
512	812	911	937
1024	364	447	468
2048	-	214	234

Tabela 6: Número de canais máximo para as interrupções PIT e FIQ

Comparando o número máximo de canais suportados para as interrupções PIT e FIQ, podemos afirmar que o uso da interrupção FIQ aumentou em 12.2% a capacidade de aquisição em 512Hz e em 22.8% a capacidade de aquisição a 1024Hz.

A coluna máximo ideal corresponde a capacidade máxima de aquisição caso o sistema usasse 100% do tempo de aquisição para adquirir dados, ou seja, sem perda de tempo por causa do jitter ou da latência. A partir da comparação dos resultados das interrupções PIT e FIQ com a capacidade máxima ideal, criamos a Tabela 7.

Frequência (Hz)	Tempo de Aquisição PIT (%)	Tempo de Aquisição FIQ (%)
512	86.7%	97.2%
1024	77.8%	95.5%
2048	-	91.5%

Tabela 7: Tempo usado para a aquisição para as interrupções PIT e FIQ

A partir da Tabela 7, podemos inferir que para uma taxa de 1024Hz, o uso da PIT garante que em 86.7% do tempo disponível, o sistema está adquirindo dados enquanto que o uso da FIQ garante aquisição em 97.2% do tempo.

10 Conclusões

A interrupção FIQ mostrou-se adequada para aplicações cuja a latência e/ou o jitter são parâmetros críticos de um sistema, uma vez que a FIQ possui uma prioridade de atendimento elevada. No sistema operacional Linux, o seu uso implica em uma alteração no código do Linux de alocação do módulo de Kernel a fim de evitar a ocorrência de *page faults*. A FIQ é mais prioritária que o código do Linux, então o Kernel não pode tratar uma ocorrência desse tipo exceção.

Associado à impossibilidade de ocorrências de *page faults*, temos o problema de acesso aos registradores de periféricos dentro da FIQ. Para programar periféricos como SPI, DMA, GPIO e outros, necessitamos do endereço base desses periféricos. Obtemos este endereço a partir de uma função *ioremap* que realiza um mapeamento dinâmico. Este endereço de retorno é suscetível à ocorrência de *page faults*. Por esse motivo, é necessário realizar o mapeamento estático de todos os periféricos que a FIQ acessa.

A estratégia adotada para escrever o handler da FIQ em C consiste em usar um wrapper, em assembly, contendo o salto para o handler em C. Não é aconselhável que a FIQ execute funções de APIs disponibilizadas pelo Kernel do Linux pois a maioria usa operações bloqueantes para gerenciar acessos concorrentes. A FIQ deve ser um processo rápido e sem bloqueios. O programador precisa estar certo de que a API demandada não possui nenhuma operação bloqueante para usá-la dentro da FIQ. Geralmente é necessário reprogramar/programar certos elementos para conseguir as mesmas funcionalidades das APIs desejadas mas sem operações bloqueantes. Nesse projeto, programamos a comunicação SPI-DMA em hardware pois API de SPI-DMA do Linux apresenta esse problema de possuir operações bloqueantes.

O emprego da FIQ no processo de aquisição possibilitou o uso de uma frequência de aquisição maior, que aliada a redução do jitter, permitiu o aumento da capacidade máxima de canais que o sistema suporta. Aumentamos em 12.2% e 22.8%, respectivamente, a capacidade máxima para uma aquisição a 512Hz e a 1024Hz. Além disso, garantimos para 1024Hz que o sistema está 95.5% do tempo aquisitando dados. Este valor, comparado ao valor da PIT, corresponde a um aumento de 22.8% do aproveitamento do tempo de aquisição. Dessa forma, podemos inferir que alcançamos o objetivo deste projeto.

A Registradores em ARM

A arquitetura ARM dispõe dos seguintes registradores.

- 13 general-purpose registers R0-R12.
- R13 - Stack Pointer (SP).
- R14 - Link Register (LR).
- R15 - Program Counter (PC).
- CPSR - Current Program Status Register
- SPSR - Saved Program Status Register

O papel do registrador CPSR é controlar e armazenar o estado atual da CPU. Este registrador possui dois campos cruciais de informação que são as flags e os bits de controle.

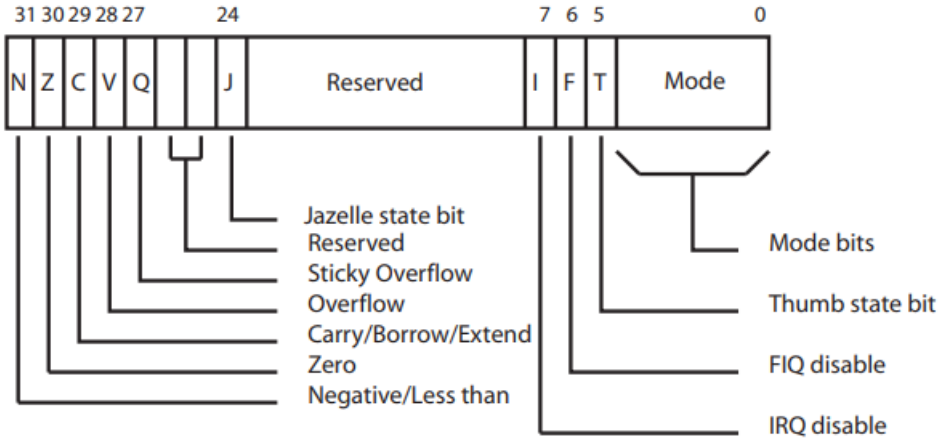

Figura 32: Registrador CPSR [12]

As flags representam informações de uma operação recentemente realizadas na ALU. Cada flag possui o seguinte significado.

N:1 - Resultado negativo.

Z:1 - Resultado nulo.

C:1 - Carry na operação de adição.

C:0 - Borrow na operação de subtração.

V:1 - Overflow.

Q:1 - Saturação de operações aritméticas e de multiplicação necessárias em Processamento Digital de Sinal, como por exemplo, QADD, QDADD, QSUB, QDSUB, SMLAxy, and SMLAWy.

Os bits de controle controlam a ativação e a desativação de interrupções e o modo de operação do processador.

I:1 - Interrupção IRQ desabilitada.

F:1 - Interrupção FIQ desabilitada.

T:1 - Modo Thumb.

T:0 - Modo ARM.

M[4:0]

b10000 - User mode

b10001 - FIQ mode

b10010 - IRQ mode

b10011 - Supervisor mode

b10111 - Abort mode

b11011 - Undefined mode

b11111 - System mode

O registrador SPSR possui campos semelhantes ao CPSR pois sua função é armazenar o valor do CPSR na detecção de uma exceção ou interrupção.

B Instruções de processamento de dados

As instruções de processamento de dados são classificadas como lógicas ou aritméticas. As operações lógicas (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) executam a ação lógica em todos os bits correspondentes do operando para produzir o resultado. As operações aritiméticas (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) tratam cada operando como um inteiro de 32 bits. A Figura 33 lista todas as instruções de processamento de dados.

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2(operand1 is ignored)
BIC	1110	operand1 AND NOT operand2(Bit clear)
MVN	1111	NOT operand2(operand1 is ignored)

Figura 33: Instruções de processamento de dado [15]

A Figura 34, apresenta a codificação das instruções de processamento.

Em uma operação com instruções de processamento de dados, se o registrador de destinação Rd é R15 (PC) e o flag S na instrução não está setada, o resultado da operação é colocado em R15 e o CPSR não é afetado. Mas se Rd é R15 e a flag S está setada, o resultado da operação é colocado em R15 e o SPSR correspondente ao modo atual é movido para o CPSR, restaurando atomicamente o PC e o CPSR [15].

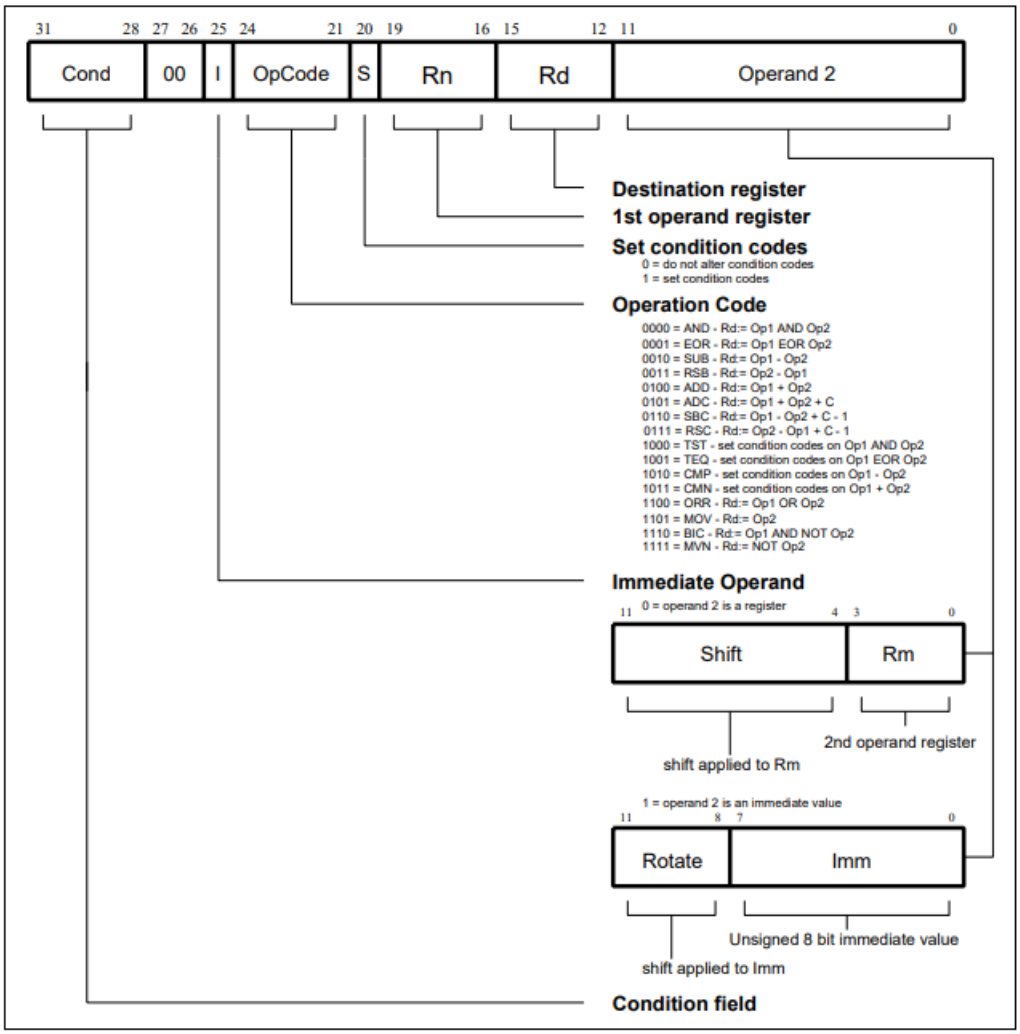

Figura 34: Codificação das instruções de processamento de dados [15]

C Peripheral Identifiers

Instance ID	Instance Name	Instance Description	External interrupt	Wired-OR interrupt
0	AIC	Advanced Interrupt Controller	FIQ	
1	SYS	System Controller		DBGU, PMC, SYSC, PMECC, PMERRLOC, RTSC, SHDWC, PIT, WDT, RTC
2	PIOA, PIOB	Parallel I/O Controller A and B		
3	PIOC, PIOD	Parallel I/O Controller C and D		
4	SMD	Soft Modem Device		
5	USART0	Universal Synchronous Asynchronous Receiver Transmitter 0		
6	USART1	Universal Synchronous Asynchronous Receiver Transmitter 1		
7	USART2	Universal Synchronous Asynchronous Receiver Transmitter 2		
8	USART3	Universal Synchronous Asynchronous Receiver Transmitter 3		
9	TWI0	Two-Wire Interface 0		
10	TWI1	Two-Wire Interface 1		
11	TWI2	Two-Wire Interface 2		
12	HSMCI0	High Speed Multimedia Card Interface 0		
13	SPI0	Serial Peripheral Interface 0		
14	SPI1	Serial Peripheral Interface 1		
15	UART0	Universal Asynchronous Receiver Transmitter 0		
16	UART1	Universal Asynchronous Receiver Transmitter 1		
17	TC0, TC1	Timer Counter Channel 0, 1, 2, 3, 4, 5		
18	PWM	Pulse Width Modulation Controller		
19	ADC	ADC Controller		
20	DMAC0	DMA Controller 0		

Figura 35: Peripheral Identifiers [12]

D Programação Timer Counter

```
static void tc_init(void)
{
  unsigned mask;

  /* Disables write protection */
  pmc_write_protection(AT91_PMC_WPDIS);
  aic_write_protection(AIC_WPMR_WPDIS);
  tc_write_protection(TC_WPMR_WPDIS);

  /* Enables TC peripheral clock */
  at91_pmc_write(AT91_PMC_PCER, (1 << TC_ID));

  /* Disables TC and FIQ interrupts */
  aic_write((1 << TC_ID) | (1 << AT91_ID_FIQ), AT91_AIC_IDCR);

  /* Sets edge triggered mode INT_EDGE_TRIGGERED and highest priority level */
  aic_write(((1 << 5) | 7), AT91_AIC_SMR(17));

  /* Clears TC and FIQ interrupts */
  aic_write((1 << TC_ID) | (1 << AT91_ID_FIQ), AT91_AIC_ICCR);

  /* Disables TC_FIQ clock */
  tc_writel(ATMEL_TC_CLKDIS, CCR, TC_FIQ);

  /* Disables TC_FIQ interrupt events*/
  tc_writel(0xFF, IDR, TC_FIQ);

  /* Periodic mode, reset when RC value is achieved */
  mask = ATMEL_TC_WAVE | ATMEL_TC_WAVESEL_UP_AUTO | ATMEL_TC_TIMER_CLOCK4;
  tc_writel(mask, CMR, TC_FIQ);

  /* Sets RC value */
  tc_writel(0xFF, RC, TC_FIQ);

  /* Enables RC compare Interrupt */
  tc_writel(ATMEL_TC_CPCS, IER, TC_FIQ);

  /* Enables Fast forcing */
  aic_write((1 << TC_ID), AT91_AIC_FFER);

  /* Enables TC and FIQ interrupts */
  aic_write((1 << TC_ID) | AT91_ID_FIQ, AT91_AIC_IECR);

  /* Enable TC_FIQ device */
  tc_writel(ATMEL_TC_CLKEN | ATMEL_TC_SWTRG, CCR, TC_FIQ);
}
```

E Programação do gráfico do jitter

```
from pylab import *
from ast import literal_eval

clk_period = 0.01503759

with open('jpit_find_cmd.txt') as inp:
    for line in inp:
        v = literal_eval(line)

v = array(v)

v2 = (v - average(v))*clk_period
plot(v2)
show()

plt.hist(v2, 50, normed=1)
plt.xlabel(r'Jitter [$\mu s$]')
plt.ylabel('Densidade')
plt.title('Jitter da interrupcao')
show()

print("maximo v2 e media")
print(max(v2))
print(average(v2))
```

F Tabela do número de identificação dos canais de DMA

Table 30-2. DMAC 1 Channel Definition

Instance Name	Transmit/Receive	DMA Channel Number
HSMCI1	RX/TX	0
SPI1	TX	1
SPI1	RX	2
SMD	TX	3
SMD	RX	4
TWI1	TX	5
TWI1	RX	6
ADC	RX	7
DBGU	TX	8
DBGU	RX	9
UART1	TX	10
UART1	RX	11
USART2	TX	12
USART2	RX	13
USART3	TX	14
USART3	RX	15

Figura 36: Tabela do número de identificação dos canais de DMA [12]

G Programação da recepção via SPI-DMA

```
static void spi_dma_rx(struct spi_dma_transfer *spi_dma)
{
    unsigned mask;

    /* Disables SPI_RX */
    mask = AT_DMA_DIS(DMA_ID_SPI1_RX);
    dmac_writel(CHDR, mask);

    dmac_writel(SADDR2, SPI_RDR_PYS_ADDR); // RDR physical address
    dmac_writel(DADDR2, spi_dma->rx_dmaphys);
    dmac_writel(DSCR2, 0);

    /**
     * Transfer Width for the Source
     * Transfer Width for the Destination
     * Buffer Transfer Size
     */
    mask = ATC_SRC_WIDTH(spi_dma->transfer_width);
    mask |= ATC_DST_WIDTH(spi_dma->transfer_width);
    mask |= ATC_BTSIZE(spi_dma->buffer_size);
    dmac_writel(CTRLA2, mask);


    /** Config DMAC_CTRLB2
     * Source transfer done by second DMA Master Interface
     * Destination transfer done by second DMA Master Interface
     * PER2MEM as flow control
     * Fixed address for the source
     * Incremental address for the destination
     * Fetch from memory
     */
    mask = ATC_SIF(AT_DMA_PER_IF) | ATC_DIF(AT_DMA_MEM_IF);
    mask |= ATC_FC_PER2MEM | ATC_IEN;
    mask |= ATC_SRC_ADDR_MODE_FIXED | ATC_DST_ADDR_MODE_INCR;
    mask |= ATC_DST_DSCR_DIS;
    mask |= ATC_SRC_DSCR_DIS;
    dmac_writel(CTRLB2, mask);


    /** Config DMAC_CFG2
     * Sets source peripheral identifier
     * Master interface locked by a buffer transfer
     * Hardware Selection for the Source
     * STOP ON DONE activated
     */
    mask = ATC_SRC_PER(DMAC_PER_ID_RX);
    mask |= ATC_SRC_H2SEL;
    mask |= ATC_SOD | ATC_FIFOCFG_ENOUGHSPACE;
    dmac_writel(CFG2, mask);


    /* Enables SPI_RX */
    dmac_writel(CHER, AT_DMA_ENA(DMA_ID_SPI1_RX));
}
```

H Programação da transmissão via SPI-DMA

```
static void spi_dma_tx(struct spi_dma_transfer *spi_dma)
{
    unsigned mask;

    /* Reads status register */
    dmac_readl(CHSR);

    /* Cleaning any pending interrupts */
    dmac_readl(EBCISR);

    /* Disables SPI1_TX */
    mask = AT_DMA_DIS(DMA_ID_SPI1_TX);
    dmac_writel(CHDR, mask);

    /* Config source and destination address */
    dmac_writel(SADDR1, spi_dma->tx_dmaphys);
    dmac_writel(DADDR1, SPI_TDR_PYS_ADDR); // TDR physical address
    dmac_writel(DSCR1, 0);

    /**
     * Transfer Width for the Source
     * Transfer Width for the Destination
     * Buffer Transfer Size
     */
    mask = ATC_SRC_WIDTH(spi_dma->transfer_width);
    mask |= ATC_DST_WIDTH(spi_dma->transfer_width);
    mask |= ATC_BTSIZE(spi_dma->buffer_size);
    dmac_writel(CTRLA1, mask);


    /** Config DMAC_CTRLB1
     * Source transfer done by second DMA Master Interface
     * Destination transfer done by firt DMA Master Interface
     * MEM2PER as flow control
     * Incremental address for the source
     * Fixed address for the destination
     * Fetch from memory
     */
    mask = ATC_SIF(AT_DMA_MEM_IF) | ATC_DIF(AT_DMA_PER_IF);
    mask |= ATC_FC_MEM2PER | ATC_IEN;
    mask |= ATC_SRC_ADDR_MODE_INCR | ATC_DST_ADDR_MODE_FIXED;
    mask |= ATC_DST_DSCR_DIS;
    mask |= ATC_SRC_DSCR_DIS;
    dmac_writel(CTRLB1, mask);


    /** Config DMAC_CFG1
     * Sets destination peripheral identifier
     * Hardware Selection for the Destination
     * Master interface locked by a buffer transfer
     * STOP ON DONE activated
     */
    mask = ATC_DST_PER(DMAC_PER_ID_TX);
    mask |= ATC_DST_H2SEL;
    mask |= ATC_SOD | ATC_FIFOCFG_LARGESTBURST;
    dmac_writel(CFG1, mask);


    /* Enables SPI1_TX */
    dmac_writel(CHER, AT_DMA_ENA(DMA_ID_SPI1_TX));

}
```

I Interface SPI-DMA

```
#ifndef SPI_DMA_H
#define SPI_DMA_H

#include"spi-atmel2.h"

#define WORD        2
#define HALF_WORD   1
#define BYTE        0

struct spi_dma_transfer {

    dma_addr_t tx_dmaphys;
    dma_addr_t rx_dmaphys;
    u16 buffer_size;
    u8 transfer_width;
    u8 clock_phase;
    u8 clock_polarity;
    u8 bits_per_word;
    u32 speed_hz;
};

/* Mapeia e desabilita os registros base para a configuracao da spi */
int spi_dma_init(void);

/* Remove o mapeamento realizado e desabilita o clock da spi e do dma */
void spi_dma_exit(void);

/* Inicializa a transferencia o buffer de transmissao e insere os dados recebidos no buffer
 * de recepcao segundo as informacoes contidas na struct spi_dma_transfer
 */
void spi_transfer(struct spi_dma_transfer*);

/* Retorna 0 caso a transferencia tenha terminado.
 * Retorna EAGAIN caso contrario
 */
int transfer_done(void);

/* Retorna 0 caso a recepcao tenha terminado.
 * Retorna EAGAIN caso contrario
 */
int reception_done(void);

/* Retorna zero caso nenhum erro tenha ocorrido.
 * Retorna ECOMM caso contrario.
 * */
int get_overrun_error_status(void);

#endif
```

Referências

[1] G. F. B. J. T. B. M. d. A. P. W. K. Eduardo Ratton, Garrone Reck, “Sistemas de transportes tt046,” http://www.dtt.ufpr.br/Sistemas/Arquivos/TT046_Aula%2014.pdf, 2015, accessed: 30-06-2018.

[2] J. S. Medeiros, “ Medição e Modelagem da Resposta de um Sensor de PIG Perfilométrico sob Diferentes Solicitações Dinâmicas,” p. 53, 2013.

[3] C. S. C. Carlos Henrique Francisco de Oliveira, “Pig instrumentado da petrobr£s resultados e perspectivas,” http://www.aaende.org.ar/ingles/sitio/biblioteca/material/pdf/cote158.pdf, Aug. 2002, accessed: 11-06-2018.

[4] T. T. Nguyen, S. B. Kim, H. R. Yoo, and Y. W. Rho, “Modeling and simulation for pig flow control in natural gas pipeline,” *KSME International Journal*, vol. 15, no. 8, pp. 1165–1173, Aug 2001. [Online]. Available: https://doi.org/10.1007/BF03185096

[5] G. B. H. de Azevedo, “In-system programmer for hall effect sensors for high-resolution geometric sensor pigs; an application in hardware and software,” 2017, final project presented as a requirement for attainment of the Bachelor’s degree in Computer Engineering.

[6] F. Proctor and W. Shackleford, “Timing studies of real-time linux for control,” vol. 1, 01 2001.

[7] P. J. Salzman and O. Pomerantz, “The linux kernel module programming guide,” C. . P. J. Salzman, Ed., 2001, 2003-04-04 ver 2.4.0.

[8] “Kernel space definition,” The Linux Information Project, 2005.

[9] M. T. Jones, “Anatomy of the linux kernel,” https://www.ibm.com/developerworks/library/l-linux-kernel/index.html, Jun 2007, accessed: 27-05-2018.

[10] C. W. Andrew N. Sloss, Dominic Symes and J. Rayfield, “"arm system developer’s guide",” in "*Designing and Optimizing System Software*". Elsevier Inc, 2004, ch. 9.

[11] *Real-time performance using FIQ interrupt handling in SPEAr MPU*, 1st ed., STMicroelectronics, 1 2010, appl. Note AN3129.

[12] *Atmel | SMART ARM-based Embedded MPU*, Microchip Technology, Aug 2015, sAM9G25 Datasheet.

[13] A. F. M. Abdelrazek, “Exception and interrupt handling in arm,” Universität Stuttgart, Report, 2006.

[14] *Measuring Interrupt Latency*, 1st ed., NXP Semiconductors, 4 2018, appl. Note AN12078.

[15] “Arm architecture reference manual,” https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf, ARM, 2005, accessed: 05-06-2018.

[16] P. M. F. Sampaio, “Protocolo spi,” https://adeetc.thothapp.com/classes/SE1/1213i/LI51D-LT51D-MI1D/resources/1100, 2012, accessed: 25-06-2018.

[17] “Axi dma debug guide,” https://www.xilinx.com/Attachment/DMA_debug_guide_rev2.0.pdf, May 2015, accessed: 28-05-2018.

[18] S.-S. Editora, *Sistemas Digitais*. SENAI-SP Editora, Dec 2015.

[19] A. Aljumah and A. Ahmed, “Amba based advanced dma controller for soc,” vol. 7, 03 2016.

[20] J. J. David S. Miller, Richard Henderson, “Dynamic dma mapping guide,” https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt, Jul. 2017, accessed: 28-05-2018.

[21] J. E. Bottomley, “Dynamic dma mapping using the generic device,” https://www.kernel.org/doc/Documentation/DMA-API.txt, Sep. 2017, accessed: 28-05-2018.

[22] M. C. Daniel Bovet, *Understanding the Linux Kernel*. O’Reilly Media, Dec 2005.

[23] “Arm926ej-s technical reference manual,” http://ww1.microchip.com/downloads/en/DeviceDoc/ARM_926EJS_TRM.pdf, ARM, 1 2004, accessed: 05-06-2018.

[24] *i.MX51 Board Initialization and Memory Mapping Using the Linux Target Image Builder (LTIB)*, 1st ed., NXP Semiconductors, 3 2010, appl. Note AN3980.