Nestor Dias Pereira Neto

Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS

Salvador

30 de abril 2022

Nestor Dias Pereira Neto

Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós Graduação em Engenharia Elétrica da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

Universidade Federal da Bahia - UFBA

Escola Politécnica

Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Wagner Oliveira

Coorientador: Paulo César

Salvador

30 de abril 2022

Nestor Dias Pereira Neto
Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS/ Nestor Dias Pereira Neto. – Salvador, 30 de abril 2022-
31p. : il. (algumas color.) ; 30 cm.

Orientador: Wagner Oliveira

Dissertação (Mestrado) – Universidade Federal da Bahia - UFBA
Escola Politécnica
Programa de Pós-Graduação em Engenharia Elétrica, 30 de abril 2022.

1. Palavra-chave1. 2. Palavra-chave2. 2. Palavra-chave3. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

Nestor Dias Pereira Neto

Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós Graduação em Engenharia Elétrica da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

Trabalho aprovado. Salvador, 24 de novembro de 2012:

Wagner Oliveira
Orientador

Professor
Convidado 1

Professor
Convidado 2

Salvador
30 de abril 2022

Resumo

Segundo a.o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: latex, abntex, editoração de texto.

Abstract

oihbqptipbõq4tnpot4photnj4yojnj4ynojp

Keywords: latex. abntex. text editoration.

Sumário

1 INTRODUÇÃO 8
1.1 Justificativa 9
1.2 Objetivos 10
1.2.1 Objetivo Geral 10
1.2.2 Objetivos Específicos 10
1.3 Organização 10

I REFERENCIAIS TEÓRICOS 11

2 CYCLONE V: SYSTEM ON A CHIP - SOC 12
2.1 Field Programmable Gate Array - FPGA 12
2.2 Hard processor ARM 12
2.2.1 Embedded Linux 12
2.3 Kit de desenvolvimento DE10-nano 12

3 ROBOT OPERATING SYSTEM - ROS 13
3.1 Sistema multiagentes 13

II DESENVOLVIMENTO 14

4 ARQUITETURA DO SISTEMA 15
4.1 Modelo cliente-servidor 15
4.2 Biblioteca de comunicação - libinterfacesocket 16

5 PACOTE ROS (CLIENTE) 18

6 SERVIDOR 21
6.1 Processo de BOOT 21
6.2 Distribuição Linux rsyocto 21
6.3 Interface socket server 24

III RESULTADOS 27

7 RESULTADOS ALCANÇADOS 28

8 CONCLUSÃO 29

9 ESTUDOS FUNTUROS 30

REFERÊNCIAS 31

1 Introdução

Nos últimos anos novas técnicas para construção de robôs tem estado em muita evidência, em especial áreas como robótica móvel e robótica colaborativa tem chamado bastante atenção dos pesquisadores. Uma das principais características dessas áreas é exigência de um alto grau de percepção do ambiente que rodeia o robô, além de uma execução mais precisa em seus movimentos, isto devido ao fato de que as atividades desempenhadas por robôs estão exigindo um nível cada vez maior de interação com as atividades desempenhadas por seres humanos.

Outro ponto que sempre teve bastante importância no desenvolvimento de novos robôs é a autonomia, tanto do ponto de vista do consumo energético quando na tomada de decisões, superar o nível de autonomia atual continua sendo um dos principais bjetivos na pesquisa e desenvolvimento de novos robôs.

As consequências desses trabalhos podem ser percebida em robôs com

2 - a busca por sistemas com um grau maior de autonomiaA robótica tem se caracterizádo pelo grande nível de percepção do ambiente e pelos sistemacomplexos de controle do movimentos

O ROS como um framework que esta se tornando o padrão no desenvolvimento de robótica

1 - Fazer um link com a complexidade dos sistemas robóticos modernos e o ROS

Agrupar inúmeros “blocos"de softwares usados em robótica, fornecer drivers para hardwares específicos (sensores e atuadores), gerenciar troca de mensagens entre os nós que fazem parte do sistema, são as função do ROS. Essas características fazem com que o ROS seja reconhecido com um pseudo sistema operacional (PYO et al., 2017). Dessa maneira o ROS se tornou muito ágil no desenvolvimento de novas aplicações para robótica. Usando nós já desenvolvidos e testados por outros desenvolvedores podemos criar novos sistemas completos apenas gerenciando esses nós na rede interna do ROS. Essa abordagem fez com que o número de pacotes para o ROS cresça em uma taxa muito rápida, desde o ano de seu lançamento, 2007, até 2012 o ROS aumentou de 1 para 3699 pacotes (YAMASHINA et al., 2005).

Com essa distribuição de tarefas através de vários nós podemos criar sistemas cada vez mais complexos, apenas inserindo novos nós na rede, essa rede é gerenciada pelo ROS Master, que é apensas mais um nó do sistema, mas com a função de ser um servidor de nome e serviços para o restante dos nós. Ele identifica os nós na rede, assim todos os nós podem se comunicar com os outros através de conexões peer-to-peer, igura 1. Para

desenvolver novas aplicações para o crescente grupo de pacotes ROS, o desenvolvedor deve respeitar os protocolos de comunicação da rede, as bibliotecas do ROS facilitam este trabalho, por já fornecer funções prontas para o desenvolvimento de novos códigos compatíveis e que possam se registrar na rede. Detalhes dos rotocolos e interno podem ser visto em (ROS, 2011a), (ROS, 2018) e (ROS, 2011b).

Por se tratar de um hardware configurável o FPGA é ideal para processamento digitais de sinais. O potencial que os FPGAs possuem para melhorar o

Desenvolvimento com fpga, SoC. dificuldade e maior tempo de desenvolvimento

1 - explicar a ideia do FPGA e a dificuldade do desenvolvimento

O potencial que os FPGAs possuem para melhorar o desempenho de sistemas que utilizam processamento digitais de sinal é conhecido já algum tempo, as possibilidades de paralelismo, criação de estruturas de DSP dedicadas à aplicação, são recursos muito interessantes que a possibilidade do hardware configurado ferecem. Em contra partida as facilidade de desenvolvimento encontradas em aplicações que fazem uso de softwares não são encontradas nas mesmas proporções no mundo do ardware, sendo assim:

Juntar fpga com robótica através do ros dando foco na facilidade de desenvolvimento

2 - Fala sobre o FPGA escolhido

explicar a parte da comunicação entre o computador/ros e o SoC melhorar a comunicação, comunicação eficiente, pacote pronto e de fácil integração com qualquer sistema ros, O mais genérico possível para se enquadrar a qualquer projeto é que o desenvolvedor tenha interesse em incluir um FPGA ao sistema

- **Como estabelecer a comunicação entre o ROS e um sistema de processamento auxiliar embarcado em um FPGA?**

Este problema é o que o trabalho vai tentar responder, podendo assim outros pesquisadores possam usar os benefícios do uso do hardware dedicado integrados ao benefícios que o ROS fornecem aos sistemas robóticos.

1.1 Justificativa

Nos últimos anos novas técnicas para construção de robôs tem sido bastante estudadas, em especial uma área que tem sido bastante explorada é a robótica móvel. A principal características que tem sido buscada é cada vez fornecer mais autonomia ao sistemas robóticos o que torna seus softwares cada vez mais complexos, o que aumenta a

necessidade do uso de processadores muito mais poderosos, consequentemente aumentando muito o consumo de energia. Entretanto a busca por mais autonomia, diz respeito também às baterias, que são as fontes de energia da maioria dos robôs móveis, o que provoca uma verdadeira briga entre poder de processamento e baixo consumo.

Sendo assim, o FPGA pode ser uma ótima alternativa para solucionar os problemas de aumento do poder de processamento em conjunto com baixo consumo de energia. Meyer-Baese (2007) descreve algumas vantagens dos FPGAs modernos para uso em processamento digitais de sinais, como as cadeias de fast-carry usadas para implementar MACs de alta velocidade e o paralelismo tipicamente encontrado em dedign implementados em FPGA. Por essas características o FPGA necessita de frequências menores de trabalho para alcançar desempenho equivalente ou superior às soluções baseadas em processadores, tornando a dissipação de energia consideravelmente menor.

1.2 Objetivos

1.2.1 Objetivo Geral

Desenvolver uma solução para estabelecer comunicação entre *Field Programmable Gate Array - FPGA*, configurado como um co-processador de vídeo.

1.2.2 Objetivos Específicos

- Estudar os assuntos relevantes ao projeto: Verilog HDL, embedded linx, Cyclone V, TCP/IP Stack, ROS;
- Conhecer com detalhes os protocolos da rede TCP/IP usada para comunicação interna dos nós e serviços ROS;
- Implementa distribuição embedded linux para processador ARM;
- Estabelecer comunicação entre o ROS e o Cyclone V, através da tecnologia Gigabit Ethernet;
- Testar aplicações de processamento de vídeo em hardware em conjunto com ROS;
- Avaliar a performance com a inclusão do FPGA ao sistema.

1.3 Organização

No primeiro caítulo...

Parte I

Referenciais teóricos

2 Cyclone V: System on a Chip - SoC

2.1 Field Programmable Gate Array - FPGA

2.2 Hard processor ARM

2.2.1 Embedded Linux

2.3 Kit de desenvolvimento DE10-nano

3 Robot Operating System - ROS

3.1 Sistema multiagentes

Parte II

Desenvolvimento

4 Arquitetura do sistema

Para conseguirmos estabelecer a comunicação entre o computador e o SoC precisaremos efetuar programação de sockets e bibliotecas específicas para trabalho em redes, desenvolver um pacote ROS para disponibilizar os dados recebidos através da interface de rede para os outros pacotes ROS do sistema robótico, além de um programa rodando no HPS do SoC para estabelecer esta comunicação entre a interface de rede da placa De10-nano e a aplicação sendo executada no FPGA. Já a aplicação que estará embarcada no FPGA contido no SoC deverá ser descrita por alguma linguagem de descrição de hardware, como por exemplo, verilog ou VHDL.

Todas essas etapas descritas anteriormente são necessárias para a construção completa do sistema proposto, o que torna o desenvolvimento da solução completa um desafio devido às diferentes ferramentas de software e hardware necessárias para sua conclusão. Tendo em vista este problema, a solução foi idealizada para conter o maior grau de modularidade possível, ou seja, cada uma dessas etapas será tratada com um projeto independente, apenas tendo cuidado para garantir a correta comunicação entre cada uma delas.

A grande vantagem que esse abordagem traz ao projeto é a possibilidade futura, de tanto a continuação do desenvolvimento como da manutenção do sistema, serem realizados por profissionais com background nas diferentes áreas envolvidas, sem a necessidade de se envolver no desenvolvimento de outros módulos. Sendo assim, um profissional especialista em descrição de hardware poderia se dedicar apenas à concepção da solução embarcada no FPGA, sem a necessidade possuir conhecimento em programação de redes.

4.1 Modelo cliente-servidor

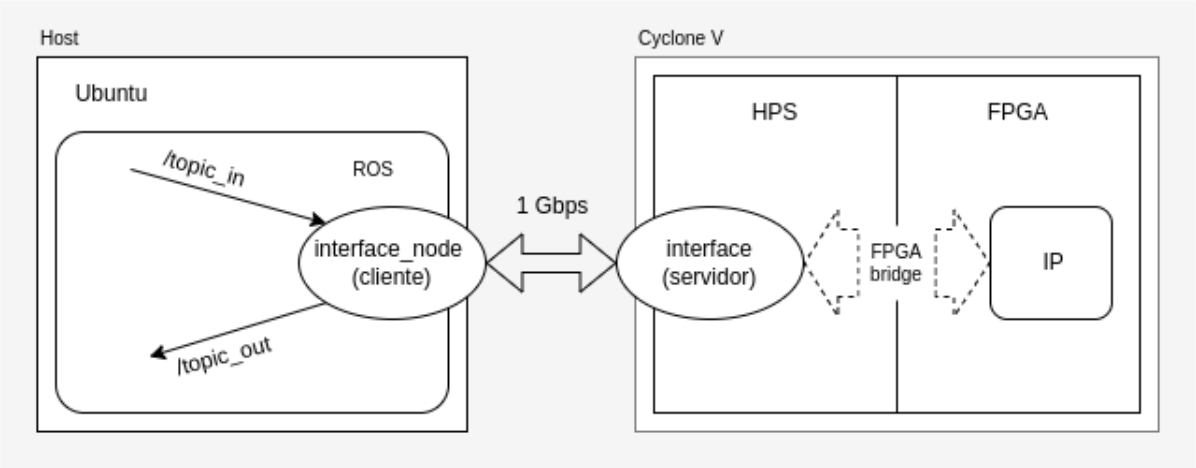
A comunicação entre o host, rodando o ROS, e a placa DE10-nano será estabelecida através de uma rede gigabit ethernet ponto a ponto, ou seja, o host e o SoC estarão conectados diretamente entre si. Desta maneira é possível obter o melhor desempenho da rede, alcançando as maiores taxas de transmissão de dados. Com o meio de comunicação definido é preciso definir também a arquitetura da comunicação, uma boa alternativa é o modelo cliente-servidor.

O modelo cliente-servidor é caracterizado por possuir uma estrutura que permite dividir o trabalho computacional entre os participantes da comunicação, isto é, entre o servidor, que é o encarregado de disponibilizar os recursos e serviços, e o cliente, que realiza as solicitações para os serviços disponíveis. Desta maneira tanto o cliente quanto

o servidor foram tratados como módulos independentes durante o desenvolvimento do trabalho. O uso do modelo cliente-servidor contribui de forma significativa para que o sistema alcance o máximo de modularização, essa abordagem facilita, entre outras coisas, a depuração e manutenção do código, o que proporciona mais agilidade e simplicidade no processo de desenvolvimento da solução.

Na Figura 1 podemos ter uma visão global do sistema, nela podemos ver cada etapa da comunicação.No lado do host, está instalado o ROS, nele também é onde o cliente será executado, assim sendo o cliente fica responsável por ler o tópico de entrada, fornecido por outro nó do sistema, realizar uma solicitação ao servidor enviando os dados já lidos. O servidor, por sua vez, aceita a solicitação do cliente, recebe os dados e os envia à aplicação embarcada no FPGA que os devolve após seu processamento. Para completar o ciclo o servidor retorna os dados processados ao cliente, que por sua vez, disponibiliza os dados já processados através do tópico de saída.

Figura 1 – Arquitetura geral



Fonte: do autor

4.2 Biblioteca de comunicação - libinterfacesocket

Para manter o padrão do desenvolvimentos dos códigos tanto do cliente quanto do servidor, foi desenvolvida uma classe, que fornece os métodos para a abertura da comunicação, além de métodos para envio e recebimento das mensagens através da rede gigabit ethernet. Essa classe foi desenvolvida como um módulo a parte e compilada como uma biblioteca estática, sendo assim, a partir do momento em que os métodos de comunicação estiverem testados e validado tanto o código do cliente quanto o do servidor poderão fazer uso desta biblioteca, eliminando assim a necessidade de reescrever uma parte do código código. Outra vantagem nessa abordagem é que ao manter o código desassociado

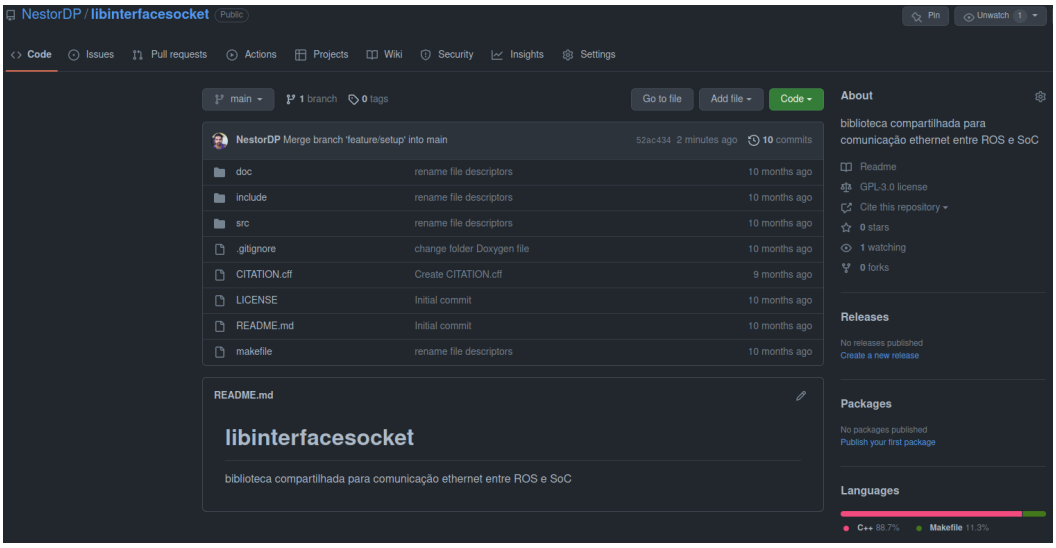
tanto do servidor como do cliente, nós possibilita fazer alterações ou correções de bugs, sem necessariamente realizar alterações nos códigos do servidor ou do cliente.

A programação da biblioteca foi realizada com base em sockets. Sockets são um caminho para conectar processos em uma rede de computadores. A conexão através de sockets entre nós em uma rede independe do protocolo. Um nó da rede ouve uma determinada porta para um IP específico esperando por o pedido de conexão do segundo nó, assim a conexão entre dois processos é estabelecida. O servidor é o nó que aguarda o pedido ser enviado pelo cliente.

A programação de sockets em C++ possibilita um alto nível de otimização da comunicação entre os processos, principalmente por se tratar de um modelo cliente-servidor onde só existirá a comunicação entre o servidor e apenas um cliente. Após implementar a comunicação entre o servidor e o cliente, poderá ser testadas novas técnicas de para otimizar o desempenho da rede possibilitando o aumento da taxa de transferência de dados entre o servidor e o cliente.

O código fonte da biblioteca pode ser encontrado no repositório no github (NETO, 2021a), que pode ser visto na figura 2, onde podemos observar a estrutura de arquivos da blibioteca. Vale frizar que, a libinterfacesocket possui um makefile para realizar o processo de compilação de forma automática. Assim podemos de forma simplificada compilar e instalar a biblioteca tanto no sistema do host onde será executado o cliente, quanto no sistema do HPS embarcado no SoC, onde o servidor estará rodando.

Figura 2 – Repositório libinterfacesocket



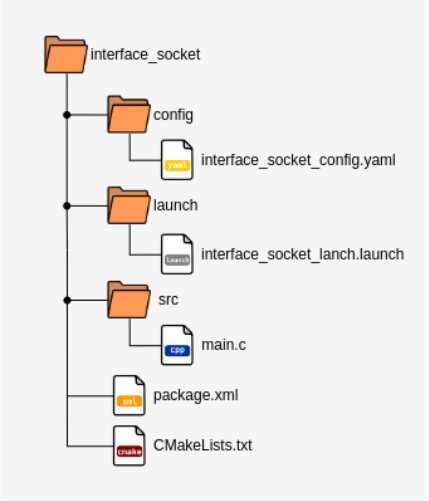
Fonte: do autor

5 Pacote ROS (cliente)

Como já foi mencionado anteriormente o cliente é um pacote ROS, sendo assim, deve-se levar em consideração, os conceitos de programação do framework ROS em seu desenvolvimento. As técnicas específicas de programação ROS usadas no cliente serão descritas com detalhes neste capítulo, além da explicação do seu funcionamento interno e do uso da libinterfacesocket.

No desenvolvimento de aplicações com ROS deve ser respeitada a estrutura de diretórios de um pacote ROS. Pacotes são a maneira com que os softwares são organizados no ROS, eles podem conter desde nós, que são a unidades de processamento do ROS, até mesmo bibliotecas ou módulos de softwares de terceiros. Os pacotes devem seguir uma estrutura padrão, por este motivo o código fonte do cliente foi organizado como é mostrado na Imagem 3 abaixo.

Figura 3 – Estrutura de diretórios pacote cliente



Fonte: do autor

A seguir será apresentada uma breve descrição de cada item do pacote:

config/interface_socket_config.yaml: Arquivo com o qual o usuário pode mudar alguns parâmetros de configuração da comunicação, como IP do servidor ou alguns outros parâmetros do tópico que será lido. Essa mudança pode ocorrer sem a necessidade de recompilar o código fonte, isso pode dar flexibilidade ao usuário do pacote durante o desenvolvimento de uma nova aplicação, que poderá ser configurada sem alterações no código fonte do nó.

launch/interface_socket.launch: Arquivo responsável chamar a execução do nó, além de carregar os parâmetros presentes no arquivo config.yaml no servidor de parâmetro

do ROS.

src/main.cpp: Código fonte do executável, ou seja, o código fonte do único nó deste pacote.

package.xml: Manifesto do pacote é o arquivo que define as propriedades do pacote, como por exemplo, nome do pacote, autor, e dependências. Deve estar presente em todos os pacotes ROS (ROS, 2019).

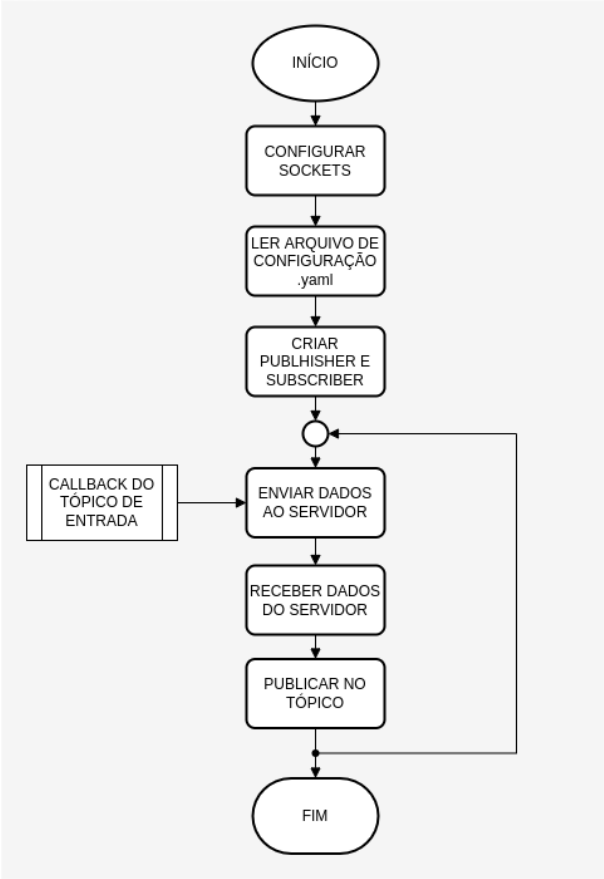
CmakeLists.txt: Contém instruções e diretivas para configuração do processo de compilação do pacote.

Este pacote disponibiliza apenas um executável, ou seja, apenas um nó que tem a finalidade de ler um tópico, enviar através da interface de rede os dados brutos contidos na mensagem do tópico de entrada, receber de volta esses dados processados pelo servidor, montar uma mensagem ROS e publicar essa mensagem através do tópico de saída. O processo descrito anteriormente é relativamente simples, como pode ser visto no fluxograma simplificado da Figura 4

Este pacote disponibiliza apenas um executável, ou seja, apenas um nó que tem a finalidade de ler um tópico, enviar através da interface de rede os dados brutos contidos na mensagem do tópico de entrada, receber de volta esses dados processados pelo servidor, montar uma mensagem ROS e publicar essa mensagem através do tópico de saída. O processo descrito anteriormente é relativamente simples, como pode ser visto no fluxograma simplificado da Figura 4

Existem alternativas prontas para fazê-lo, como por exemplo o ROS serial, pacote ROS desenvolvido para permitir comunicação entre o ROS e outros dispositivos que possuem uma porta serial ou uma interface de rede [8]. A escolha por desenvolver um novo pacote para executar a mesma função se fez necessário pela necessidade de alta taxa de transferência de dados entre o ROS e o SoC para que seja aceitável a utilização do SoC com a finalidade de acelerar o processamento por hardware. Desenvolvendo um novo pacote de comunicação podemos extrair o máximo de desempenho da rede, como por xemplo, escolhendo o melhor protocolo (UDP ou TCP) e transferindo apenas os dados para o processamento da informação. Todos os códigos do cliente podem ser encontrados no repositório disponível em [9].

Figura 4 – Fluxograma pacote cliente



Fonte: do autor

6 Servidor

No modelo cliente-servidor, o servidor é o elemento responsável por executar uma ação somente após um pedido realizado pelo cliente, sendo assim, o servidor deve estar sempre preparado para responder à uma solicitação de serviço. Deste modo, além da capacidade de se comunicar com o cliente, o servidor deve oferecer algum serviço de interesse do cliente.

Neste trabalho o programa servidor disponibiliza ao cliente a interface com o FPGA presente no SoC, logo, ao receber o pedido do cliente, o servidor deve ser capaz de receber os dados vindos do cliente à unidade de processamento embarcada no FPGA e em seguida devolver esses dados já processados ao cliente. Entretanto, antes mesmo de iniciar o desenvolvimento do código do servidor, devemos compilar uma distribuição linux e configurar o processo de boot desse sistema no processador ARM presente no SoC. Estas etapas no desenvolvimento do servidor serão descritas neste capítulo.

6.1 Processo de BOOT

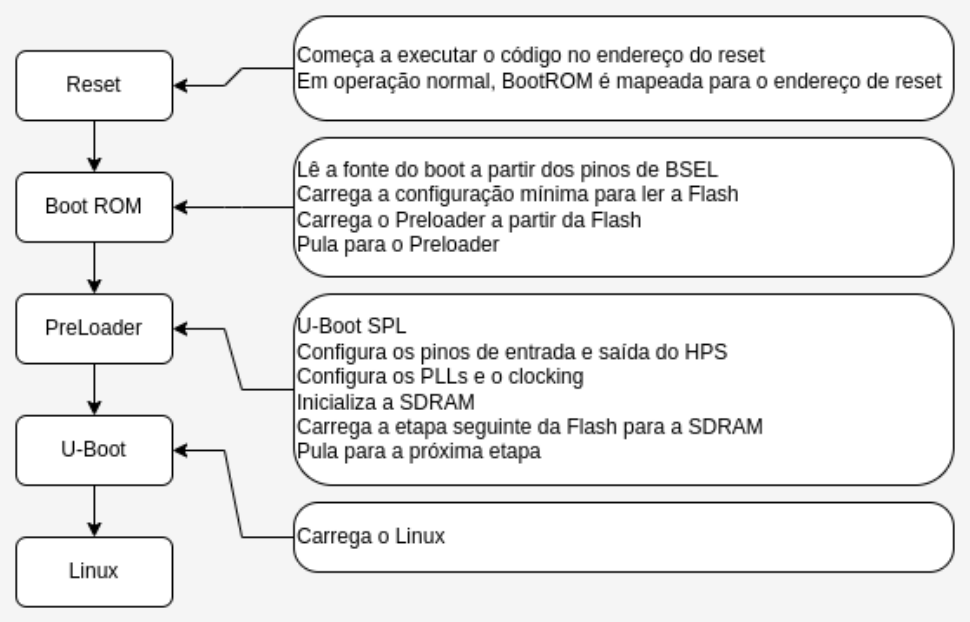
O fluxo de boot do linux na placa é resumido na Figura 5. O primeiro elemento de software é o *Boot ROM* que está gravado de fábrica internamente no dispositivo. Os arquivos de *PreLoader*, *U-boot* e do sistemas de aquivos do linux são salvos em um cartão de memória micro SD. O *Secondary Program Loader - SPL*, conhecido como *PreLoader*, é executado a partir da Boot ROM. Ele é responsável por configurar o sistema para que o *bootloader* (U-boot) possa ser executado. A Intel fornece uma ferramenta chamada BSP editor que, a partir de arquivos que descrevem o hardware, podem gerar o PreLoader para o projeto específico (ROCKETBOARDS.ORG, 2015).

A etapa seguinte ao *PreLoader* é o *bootloader*. Nessa fase do boot todas as questões de baixo nível do SoC, como por exemplo, os clocks, pinos e SDRAM, já foram inicializados e estão prontos. O objetivo do bootloader, é obter essas informações do sistema e fazer com que ele funcione até o ponto onde o linux possa ser iniciado. Outra função importante do U-boot em um SoC Intel é programar o FPGA (ROCKETBOARDS.ORG, 2015).

6.2 Distribuição Linux rsyocto

Com todos os arquivos necessários para o boot do sistema já gerados a partir das ferramentas de desenvolvimento da Intel, poderemos escolher nosso sistema operacional. O sistema escolhido foi uma distribuição linux desenvolvida exclusivamente para SoCs da

Figura 5 – Boot linux embarcado



Fonte:(ROCKETBOARDS.ORG, 2015)

Intel com fpga integrado, como pode ser observado na descrição da distribuição disponível em (ROBIN, 2022):

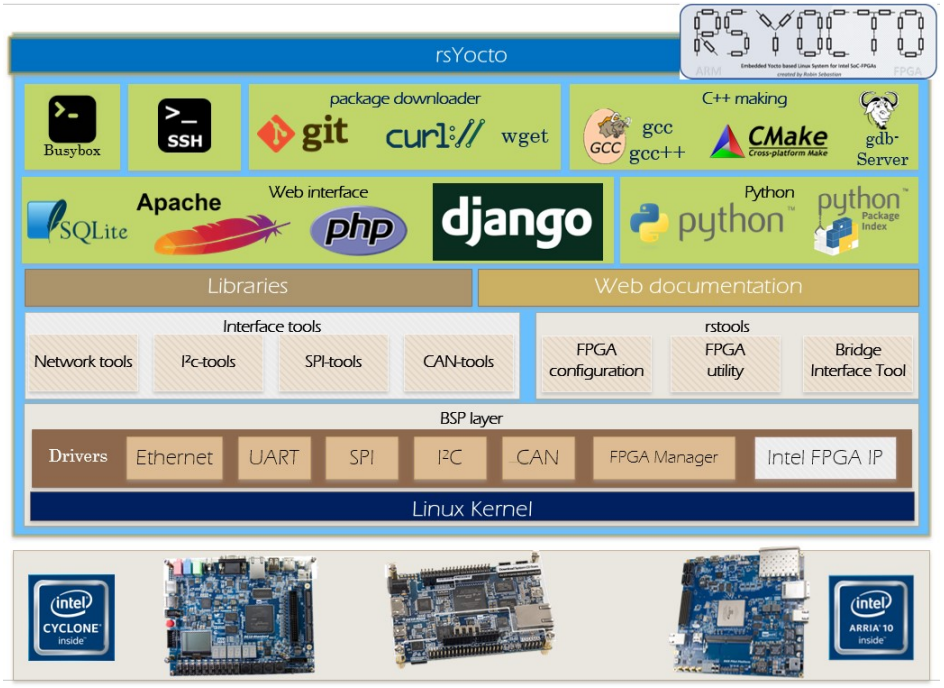
> "**rsyocto** is an open source Embedded Linux Distribution designed with the Yocto Project and with a custom build flow to be optimized for Intel SoC-FPGAs (Intel Cyclone V and Intel Arria 10 SX SoC-FPGA with an ARM Cortex-A9) to achieve the best customization for the strong requirements of modern embedded SoC-FPGA applications."

O rsyocto faz uso do Kernel Linux **linux-socfpga 5.11** (ALTERA-OPENSOURCE, 2022) e possui um conjunto de ferramentas necessárias para ajudar a simplificar o uso e desenvolvimento de aplicações desenvolvidas para os SoCs com FPGA integrado da Intel. Além destas ferramentas o rsyocto dispõe de drivers para todos os periféricos de comunicação integrados SoC, como por exemplo, a interfaces I2C e CAN, e para todas as interfaces entres o HPS e o FPGA. O rsyocto oferece um conjunto de simples aplicações que podem ser executadas em linha de comando, que possibilitam executar novas configurações no FPGA, usando o FPGA Manager, até mesmo ler e escrever a interface ARM AXI-Bridge que permite interagir com o FPGA.

Essas aplicações simplificam a comunicação com o FPGA a partir de simples comandos que podem ser executados a partir do terminal, através de todas as interfaces disponíveis, estes comandos são: **lwhps2fpga**, que permite ações de leitura e escrita no barramento Lightweight HPS-to-FPGA-Bridge, o **hps2fpga** para leitura e escrita no barramento HPS-to-FPGA-Bridge, o **hps2sdram**, que proporciona acesso à interface da

SDRAM e os **gpi** e **gpo** para acesso aos sinais de uso geral. Na Figura 6 podemos ter uma visão geral de todas as ferramentas disponíveis na distribuição linux *rsyocto*.

Figura 6 – Overview do rsyocto



Fonte:(ROBIN, 2022)

A seguir são listadas algumas características da distribuição linux rsyocto que fazem dela uma excelente escolha para ser utilizada neste trabalho:

- Embedded Linux specially developed for Intel SoC-FPGAs
- Linux Kernel 5.11 (Source)
- Full usage of the Dual-Core ARM (ARMv7-A) Cortex-A9 with
- FPGA Fabric configuration during the boot and with a single Linux command
- All Bridge Interfaces between the HPS and FPGA are enabled and ready for use!
- Tools to interact with the FPGA Fabric via the ARM AXI HPS-to-FPGA bridges
- HPS Hard IP components (I²C-,SPI-, CAN-BUS or UART) are routed to FPGA
- USB Host support with test tools (e.g. lsusb)
- Full Linux Network stack with dynamic and static iPv4 is supported
- OpenSSH-Server starts automatically during boot

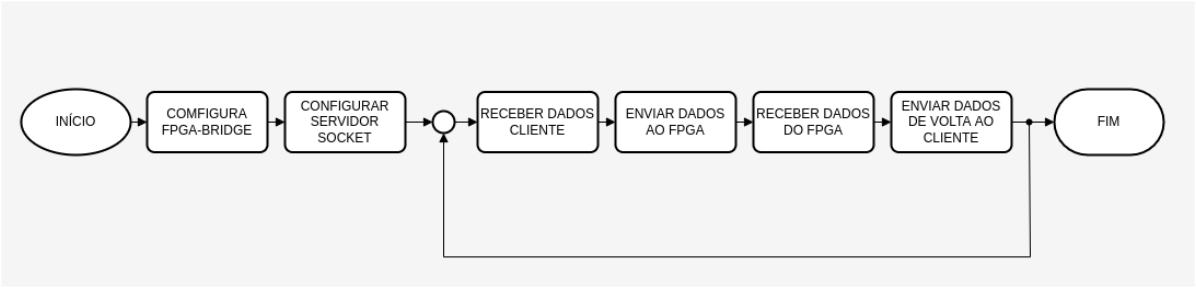
- gcccompiler 9.3.0; glibc and glib-2.0(The GNU C Library) cmake 3.16.5
- Python 3.8 Python3-dev and Python-dev
- git 2.31, wget 1.20.3, curl 7.69.1
- opkg package manager enables to add packages from different Linux Distributions.
- Changing the running FPGA-Configuration of the FPGA-Fabric

6.3 Interface socket server

Com o linux já devidamente configurado, ele já pode ser inicializado no processador do SoC, assim podemos iniciar o desenvolvimento do servidor. No primeiro momento devemos realizar o download do código fonte da biblioteca de comunicação, a libinterfacesocket, que já foi detalhada anteriormente, o código fonte pode ser baixado em (NETO, 2021b). Com os arquivos já no sistema de diretórios do SoC devemos realizar a compilação e a instalação da libinterfacesocket, só assim o servidor terá acesso a suas funcionalidades.

Logo que é inicializado o servidor mantém uma porta aberta para estabelecer uma conexão com o cliente e assim que a conexão é estabelecida, o cliente já pode começar a enviar os dados. Ao receber os dados do cliente, o servidor envia-os ao FPGA que os processa e os devolve ao servidor para que ele possa reenviar os dados já processados ao cliente. Um fluxograma simplificado desse processo pode ser visualizado na Figura 7.

Figura 7 – Fluxograma simplificado do servidor



Fonte: Elaborado pelo autor

O acesso do servidor ao FPGA é feito através de mapeamento de endereços do linux, os SoCs Intel possuem uma arquitetura de memória mapeada. Desta forma, além da biblioteca desenvolvida durante a pesquisa, um outro arquivo cabeçalho é necessário para facilitar o acesso aos endereços de memória do HPS. Quando uma instância do processador ARM é incluída no projeto do QSys Designer, é gerado um arquivo.sopcinfo no momento em que o projeto é compilado. Podemos usar esse arquivo como entrada da ferramenta *sopc-create-header-files* presente no SoC EDS para gerar um novo arquivo com extensão *.h*, que lista os endereços base de todos os módulos (IP) incluídos no FPGA.

Portanto, o arquivo cabeçalho gerado disponibiliza o *offset* do endereço em que cada periférico está localizado para cada uma das FPGA-bridges disponíveis. Deste modo o servidor pode enviar ou receber dados ao periférico conhecendo o endereço da FPGA-bridge em que este periférico está conectado e o seu nome, sem a necessidade de se conhecer exatamente o seu endereço. Na Figura 8 é apresentado uma parte do arquivo cabeçalho gerado pela ferramenta sopc-create-reader-files, neste trecho são realizadas algumas definições que facilitam o acesso ao IP dobro, que já está instanciado no FPGA. Este módulo IP foi usado para testar o processo completo de comunicação entre o tópico ROS até o FPGA.

Figura 8 – Definição de endereço presente no arquivo hps_0.h

```
1  /*
   * Macros for device 'Dobro_0', class 'Dobro'
3  * The macros are prefixed with 'DOBRO_0_'.
   * The prefix is the slave descriptor.
5  */
   #define DOBRO_0_COMPONENT_TYPE Dobro
7  #define DOBRO_0_COMPONENT_NAME Dobro_0
   #define DOBRO_0_BASE 0x38
9  #define DOBRO_0_SPAN 4
   #define DOBRO_0_END 0x3b
```

Fonte: Elaborado pelo autor

Para interagir com FPGA usando o recurso de memória mapeada, a primeira coisa que devemos fazer é abrir o dispositivo de memória do sistema. Em distribuições linux o acesso à memória física do sistema é realizado através do dispositivo /dev/mem, esse dispositivo funciona como um arquivo caracteres que representa uma imagem da memória principal do computador (KERRISK, 2021). Por se comportar como um arquivo no sistema, podemos no programa abri-lo como abriríamos qualquer arquivo de caracteres, com a funções *open*, como pode ser visto na linha 2 da Figura 9.

Em seguida podemos usar o descritor de arquivo gerado como retorno da função open, para criar um ponteiro, que dará acesso à memória principal do sistema. Para gerarmos este ponteiro, podemos usar a função mmap, ela cria um mapeamento virtual no espaço de endereço passado como argumento da função. Nos dois primeiros argumentos da função mmap, como pode ser visto na linha 10 da Figura 9, podemos definir o espaço de endereços para o mapeamento. No código apresentado este espaço se inicial no endereço zero, NULL, e termina na constante HPS_TO_FPGA_AXI_SPAN (0x3C000000), que guarda o valor do alcance máximo do espaço de endereço das interfaces entre o HPS e o FPGA. O ultimo argumento da função mmap é o endereço base da Lightweight HPS-to-FPGA-Bridge (0xC0000000), representado pela constante HPS_TO_FPGA_AXI_BASE.

Figura 9 – Memória mapeada

```
// Open up the /dev/mem device
devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);
if(devmem_fd < 0) {
        perror("devmem open");
        exit(EXIT_FAILURE);
}

// mmap() the entire address space of the Lightweight
// bridge so we can access our custom module
lw_bridge_map = (uint32_t *)mmap(
                                NULL,
                                HPS_TO_FPGA_AXI_SPAN,
                                PROT_READ|PROT_WRITE,
                                MAP_SHARED,
                                devmem_fd,
                                HPS_TO_FPGA_AXI_BASE );

if(lw_bridge_map == MAP_FAILED) {
        perror("devmem mmap");
        close(devmem_fd);
        exit(EXIT_FAILURE);
}
```

Fonte: Elaborado pelo autor

Agora que já existe um ponteiro para a região de memória onde se encontra a interface de comunicação com o FPGA, podemos enviar e receber dados para o IP instanciado no FPGA. Para isto, vamos usar este ponteiro para o endereço da HPS-to-FPGA-Bridge, somado com o offset presente no arquivo hps_0.h. Consequentemente, será possível interagir com o IP em um programa escrito em linguagem C, da mesma maneira que interagimos com um ponteiro comum. O bloco de código da Figura 10 ilustra este processo de escrita e leitura de dados em um IP através de um ponteiro

Figura 10 – Memória mapeada

```
// Set the dobro_map to the correct offset within the RAM
uint32_t * dobro_map = 0;
dobro_map = (uint32_t*)(lw_bridge_map + DOBRO_0_BASE);

*dobro_map = 56;
valor = *dobro_map;
```

Fonte: Elaborado pelo autor

Parte III

Resultados

7 Resultados Alcançados

Para validar a comunicação foram idealizados dois ensaios: o primeiro com o objetivo de testar o fluxo completo de troca de dados entre um nó ROS e uma aplicação embarcada no FPGA; o segundo ensaio teve o objetivo de testar um fluxo grande de dados trafegando entre o ROS e o SoC.

No primeiro ensaio foi descrito um circuito simples para calcular o dobro de um número, em seguida a *HPS-to-FPGA Lightweight* bridge foi utilizada para realizar a comunicação entre o servidor rodando no HPS e o FPGA. No nó cliente, rodando no ROS, foram configurados dois tópicos, um para o número a ser enviado e outro para receber o resultado calculado no FPGA, da mesma maneira que mostra o esquema da Figura

Com o objetivo de testar a sobrecarga de dados na rede, para o segundo ensaio foi realizado o envio de uma stream de vídeo com resolução de 1280x720 em 15 fps. Ao receber os dados o servidor os reenvia ao cliente que o republica em um tópico. Neste ensaio os dados não foram processados no FPGA.

Os resultados alcançados para os dois ensaios foram considerados satisfatórios. Apesar de, na transferência das imagens, ter sido observado um pequeno *delay* entre a imagem enviada e a recebida de volta.

8 Conclusão

O objetivo de estabelecer uma comunicação com alta taxa de transferência entre o framework de robótica ROS e um SoC com FPGA integrado foi alcançado, possibilitando o uso de aceleração por hardware de maneira mais ágil em projetos de robótica.

9 Estudos funturos

Por se tratar de um trabalho interdisciplinar que abrange áreas como programação de rede, compilação e configuração de linux embarcado, desenvolvimento com o ROS e com SoC, trabalhos futuros podem dar ênfase a alguma dessas áreas específicas, otimizando alguns pontos do projeto para se alcançar maiores taxas de transferência, tornar o sistema ainda mais genérico e fácil de ser usados por outros grupos de trabalho que tenham o interesse de adicionar processamento por hardware em seus projetos de robótica.

Referências

ALTERA-OPENSOURCE. *linux-socfpga*. 2022. Disponível em: <https://github.com/altera-opensource/linux-socfpga>. Citado na página 22.

KERRISK, M. *Linux Programmer's Manual - mem, kmem, port - system memory, kernel memory and system ports*. [S.l.], 2021. Disponível em: <https://man7.org/linux/man-pages/man4/mem.4.html>. Acesso em: 20 julho 2022. Citado na página 25.

NETO, N. P. *Biblioteca projeto interfacesocket*. 2021. Disponível em: <https://github.com/NestorDP/libinterfacesocket>. Citado na página 17.

NETO, N. P. *interface_socket_server*. 2021. Disponível em: <https://github.com/NestorDP/interface_socket_server>. Citado na página 24.

ROBIN, S. *rsyocto*. 2022. Disponível em: <https://github.com/robseb/rsyocto>. Citado 2 vezes nas páginas 22 e 23.

ROCKETBOARDS.ORG. *Embedded Linux Beginners Guide*. 2015. Disponível em: <https://rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide>. Acesso em: 5 outubro 2021. Citado 2 vezes nas páginas 21 e 22.

ROS. *catkin/package.xml*. 2019. Disponível em: <http://wiki.ros.org/catkin/package.xml>. Acesso em: 21 setembro 2021. Citado na página 19.