

**UNIVERSIDADE FEDERAL DA BAHIA.
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
TRABALHO DE GRADUAÇÃO**

LUCAS LIMA RODRIGUES

**USO DO BEAGLEBONE BLACK COMO SISTEMA DE
ADQUISIÇÃO DE DADOS**

SALVADOR-BA
DEZEMBRO/2016

LUCAS LIMA RODRIGUES

USO DO BEAGLEBONE BLACK COMO SISTEMA DE ADQUISIÇÃO DE DADOS

Este trabalho de graduação foi apresentado ao Programa de Graduação em Engenharia elétrica da Universidade Federal da Bahia, como requisito parcial para a obtenção do grau de Engenheiro eletricista.

Orientador:

Prof. Dr. Antônio Cezar de Castro Lima

SALVADOR-BA
DEZEMBRO/2016

LUCAS LIMA RODRIGUES

USO DO BEAGLEBONE BLACK COMO SISTEMA DE ADQUISIÇÃO DE DADOS

Este trabalho de graduação foi apresentado ao Programa de Graduação em Engenharia elétrica da Universidade Federal da Bahia, como requisito parcial para a obtenção do grau de Engenheiro eletricista.

Prof. Dr. Bernardo Ordoñez
Universidade Federal da Bahia
Coordenador do curso

COMISSÃO EXAMINADORA:

Prof. Dr. Antônio Cezar de Castro Lima
Universidade Federal da Bahia
Orientador

Prof. Dra. Ana Isabela Araújo Cunha
Universidade Federal da Bahia
Banca examinadora

Prof. Me. Leroy Umassi Ramos
Instituto Federal da Bahia
Banca Examinadora

RESUMO

O presente trabalho de conclusão de curso tem como objetivo verificar se o BeagleBone Black é uma solução viável para ser utilizada como parte de um dispositivo de aquisição de dados de baixo custo e que utiliza a interface USB como protocolo de comunicação. Para isso, foi realizada uma revisão bibliográfica que contempla o funcionamento deste dispositivo e as soluções encontradas na literatura para esta aplicação, e posteriormente, foram propostos alguns meios de comunicação entre o computador e o BeagleBone Black. Podemos dividir este trabalho em duas partes, a primeira relacionada aos métodos de captura de dados em tempo real e os métodos de transmissão de dados entre este dispositivo e o computador. A primeira parte contempla três experimentos encontrados na literatura, os dois primeiros utilizando conversores analógicos digitais externos e o último utilizando o conversor analógico digital interno. Já a segunda parte, contempla as implementações e testes dos métodos de transmissão de dados utilizando os protocolos UDP, HTTP e UART, respectivamente.

Palavras-chave: BeagleBone Black, Linux embarcado, dispositivo de aquisição de dados, protocolos de comunicação

ABSTRACT

The present coursework aims to verify if BeagleBone Black is a viable solution to be used as part of a low-cost data acquisition device that uses a USB interface as a communication protocol. Along this coursework, was made a literature review about this device and data-acquisition applications. Therefore, was implemented some methods of data transmission from BeagleBone Black to computer. We can divide this work into two parts, the first is related to real-time data capture methods and the second is related to programing and testing own solutions to transfer data from BeagleBone Black to computer. The first part considers three experiments found in the literature, two of them are related to the use of external analog-to-digital converters and the last one is related to use of internal analog-to-digital converter. On the second part, was made transmissions using UDP, HTTP and UART protocols, respectively.

Keywords: BeagleBone Black, embedded Linux, data acquisition device, communication protocols

SUMÁRIO

1	INTRODUÇÃO	1
1.1	Um breve histórico	1
1.2	Situação atual dos DAQs	3
1.3	Novas soluções.....	3
1.4	Objetivo deste trabalho	8
2	O BEAGLEBONE BLACK.....	10
2.1	Componentes do BeagleBone Black	10
2.2	As distribuições do BeagleBone Black	11
2.3	Instalando uma nova distribuição no BeagleBone Black.....	14
2.4	Comunicando com o BeagleBone Black	16
2.4.1	O que é o terminal	16
2.4.2	SSH e as pilhas de protocolo IP	18
2.4.3	Socket de rede	23
2.4.4	Conectando ao terminal através do SSH.....	24
2.4.5	UART e o <i>serial debug</i>	25
2.4.6	Conectando ao terminal utilizando o <i>serial debug</i>	26
2.5	Terminal do BeagleBone Black	27
2.5.1	Usuários no Linux.....	28
2.5.2	Comandos no terminal.....	29
2.5.3	Operadores.....	31
2.5.4	Arquivos no Linux	32
2.6	Programando no BeagleBone Black	32
2.6.1	Linguagens de <i>script</i> , interpretadas e compiladas.....	32
2.6.2	Linguagens orientadas a objeto.....	34
2.6.3	C++.....	35
2.6.4	Bash	36

2.6.5	JavaScript	37
2.6.6	Python.....	38
2.6.7	Cloud9 IDE	40
2.7	Configurando o BeagleBone Black	40
2.7.1	Adicionando <i>internet</i>	41
2.7.2	Controlando a frequênciā do processador.....	44
2.7.3	Alterando o endereçō de IP	45
2.8	Sistema de <i>boot</i> do BeagleBone Black e a Linux <i>Device Tree</i> .	46
2.8.1	<i>Flattened Device Tree</i> (FDT)	47
2.8.2	<i>Device Tree Overlay</i> (DTO)	48
2.8.3	Modos de Pinos do BeagleBone Black.....	50
2.8.4	Capemgr	51
2.8.5	<i>Slots</i>	52
2.8.6	uEnv.txt.....	54
2.9	BeagleBone Black e tarefas em tempo real.....	54
2.9.1	<i>Kernels</i> em tempo real.....	55
2.9.2	Beaglebone Black e microcontroladores.....	56
2.9.3	PRU-ICSS.....	56
2.10	O coprocessador PRU-ICSS.....	57
2.10.1	Arquitetura do PRU-ICSS	57
2.10.2	PRU-ICSS e a comunicação com mundo externo	60
2.10.3	Memória local e memória global	61
2.10.4	Programando o PRU.....	64
1.1.1.	Outras formas de programar a PRU	66
3	O DAQ.....	68
3.1	Métodos de captura de dados de um ADC em alta velocidade.	71
3.1.1.	Utilizando as I/O dedicadas do PRU como comunicação paralela	71
3.1.2.	Utilizando as I/O dedicadas do PRU como comunicação serial	74
A.	Comunicação SPI do MCP3008	75

B. Funcionamento do código	79
C. Resultados	81
3.1.3. Utilizando o PRUSS para salvar os dados do ADC interno	82
3.2 Métodos de transmissão de dados	84
3.2.1 Utilizando somente TCP e UDP	85
A. Protocolo TCP	85
B. Protocolo UDP	87
C. Implementação e resultados	88
3.2.2 Utilizando o protocolo HTTP	90
A. Como funciona os navegadores e a <i>web</i>	90
B. Microframework Flask	93
C. JQuery	97
D. Flot	98
E. Módulo Threading	99
F. Implementação e resultados	103
G. HTTP e MATLAB	105
3.2.3 Utilizando a conexão UART	105
A. Código	106
B. Resultados	108
C. Utilizando HTTP	109
4 CONCLUSÃO	110
5 TRABALHOS FUTUROS	111
BIBLIOGRAFIA	112

LISTA DE FIGURAS

- Figura 1: Placas PCI e PCI-Express da National Instruments (3).
- Figura 2: Exemplo de anúncio de venda casada do Kit DI-149 .
- Figura 3: Esquemático de um DAQ conectado a um computador (Próprio Autor).
- Figura 4: Diagrama do Satira AM3358A (8).
- Figura 5: O BeagleBone Black e seus componentes (9).
- Figura 6: BeagleBone controlando uma CNC e rodando uma aplicação para CNC simultaneamente (12).
- Figura 7: Gravando a imagem do Debian 7.5 de 14/05/2014 no cartão SD (Próprio Autor).
- Figura 8: Verificando a atualização do Debian 7.5 foi instalada com sucesso (Próprio Autor).
- Figura 9: Comparação entre o modelo OSI e a pilha de protocolos da internet (Próprio autor).
- Figura 10: BeagleBone Black conectado a um roteador (Próprio autor).
- Figura 11: Configurações para conectar o BBB com o computador através do PuTTy (Próprio Autor).
- Figura 12: Ligação dos cabos em uma comunicação serial UART (23).
- Figura 13: Os pacotes de dados da comunicação serial podem ser configurados (23).
- Figura 14: Conectando o cabo TTL ao BBB (25).
- Figura 15: Conectando ao BBB através do *serial debug*.
- Figura 16: Ambiente de desenvolvimento Cloud9 (Próprio autor).
- Figura 17: Compartilhando a rede de *internet* com a rede do BBB (Próprio autor).
- Figura 18: Permitindo que o BBB se conecte através do seu IP padrão (Próprio autor).
- Figura 19: Editando o arquivo “/etc/network/interfaces” pelo nano (Próprio autor).
- Figura 20: Comparação entre o PRU e um processador hipotético com *pipeline* (Próprio autor).
- Figura 21: Arquitetura do coprocessador PRU (33).
- Figura 22: Pinos de entrada e saída que podem ser acessados diretamente pelos PRUs (35).

Figura 23: Correspondência entre a memória local do PRU e a memória do Linux (9).

Figura 24: Notação utilizada para acessar palavras ou bits do registrador 2, r2 (9).

Figura 25: Processo de programação na PRU (9).

Figura 26: Concepção do DAQ para o CTAI (Próprio autor).

Figura 27: Esquemático da multiplexação de canais no ADC do BBB (Próprio autor).

Figura 28: *Shield* para Arduino ADS1015 (47).

Figura 29: Esquemático do BBB funcionando como o DAQ (Próprio Autor).

Figura 30: Exemplo de *hub* USB de 3 entradas (48).

Figura 31: Esquemático do *hub* USB funcionando como DAQ (Próprio Autor).

Figura 32: Pinos dedicados à PRU utilizado no experimento de Shabaz.

Figura 33: Diagrama de pinos do ADC08200 (50).

Figura 34: Fotografia de todo o sistema criado por Shabaz (49).

Figura 35: Esquemático completo do sistema de Shabaz (49).

Figura 36: Resultado do teste feito por Shabaz (49).

Figura 37: Diagrama de blocos do MPC3008 (51).

Figura 38: Registradores de deslocamento com *flip-flops* tipo D e *flip-flops* J-K (53).

Figura 39: Princípio de funcionamento do *shift-register* (53).

Figura 40: Sinais de comunicação do MCP3008 com o BBB (37).

Figura 41: Regras para envio de amostras no M3004, através da SPI (51).

Figura 42: Envio de uma amostra através da SPI utilizando um *clock* em pacotes de 8 *bits* (51).

Figura 43: Experimento de Molloy com o MCP3008 (37).

Figura 44: Estrutura e interação entre o ambiente Linux, o PRUSS e o ADC (37).

Figura 45: Resultado do experimento de (37).

Figura 46: Circuito com o ADS7883 (37).

Figura 47: Resultado utilizando o ADS7883 com o filtro capacitivo (a esquerda) e sem o filtro capacitivo (a direita) (37).

Figura 48: Descrição do cabeçalho do protocolo TCP (56).

Figura 49: Descrição do cabeçalho do protocolo TCP (Próprio autor).

Figura 50: Descrição do cabeçalho do protocolo UDP (58).

Figura 51: Resultado do teste utilizando o protocolo UDP (Próprio autor).

Figura 52: Programa adquirindo uma onda quadrada de 20kHz (Próprio autor).

Figura 53: Sistema operacional executando várias *threads* ao longo do tempo (Próprio autor).

Figura 54: Esquema do módulo Threading executando várias *threads* em paralelo com o programa principal (Próprio autor).

Figura 55: Resultado utilizando um aplicativo web com uma onda seinodal de 10kHz (Esquerda) e 20kHz (Direita) (Próprio autor).

Figura 56: Pinos do P8 e P9 que estão disponíveis para a comunicação UART (34).

Figura 57: Codificação proposta para enviar dados do ADC por serial (Próprio autor).

Figura 57: Resultado obtido utilizando a comunicação serial (Próprio autor).

LISTA DE TABELAS

Tabela 1: Comparação entre os *e-Linux board open source* mais usados no mercado (7).

Tabela 2: Relação dos componentes do BBB de acordo com a figura 5 (9).

Tabela 3: Parâmetros da UART0 e suas funções (Próprio autor).

Tabela 4: Algumas opções do comando ls (26).

Tabela 5: Alguns comandos do terminal (26).

Tabela 6: Tempo de processamento para a execução de 5000 iterações do algoritmo *n-Body* em um BBB com Debian, em diferentes frequências (9).

Tabela 7: Tabela de modos dos pinos de extensão do BBB – Parte 1 (29).

Tabela 8: Tabela de modos dos pinos de extensão do BBB – Parte 2 (29).

Tabela 9: Pinos da comunicação SPI (Próprio autor).

Tabela 10: Relação de botões do aplicativo web (Próprio autor).

Tabela 11: Relação de campos configuráveis do programa (Próprio autor).

Tabela 12: Codificação dos dados para enviar por serial (Próprio autor).

LISTA DE ALGORITMOS

Algoritmo 1: Sintaxe básica dos comandos no terminal (Próprio autor).

Algoritmo 2: Executando o comando ls com diferentes opções e parâmetros (Próprio autor).

Algoritmo 3: Utilizando alguns operadores para escrever em arquivos com o comando echo (Próprio autor).

Algoritmo 4: Acendendo o User LED3 pelo terminal (Próprio autor).

Algoritmo 5: Compilando em C++ (Próprio autor).

Algoritmo 6: Código Bash para acender o User LED3 (Próprio autor).

Algoritmo 7: Exemplo usando uma Tupla, uma Lista e um Dicionário no IDLE (Próprio autor).

Algoritmo 8: Permitindo o BBB ter acesso à internet (Próprio autor).

Algoritmo 9: Alterando a frequência do processador do BBB (Próprio autor).

Algoritmo 10: Modificando arquivos com o nano (Próprio autor).

Algoritmo 11: Um trecho da FTD do BBB (9).

Algoritmo 12: Exemplo de DTO retirada de (9).

Algoritmo 13: Compilando uma DTO e salvando na pasta do capes manager (Próprio autor).

Algoritmo 14: Adicionando as DTOs UART1 e UART2 (Próprio autor).

Algoritmo 15: Adicionando suporte à UART1 e UART2 no BBB (Próprio autor).

Algoritmo 16: Parte do código da DTO utilizada por (37) para a sua aplicação de captura de dados de um ADC externo em tempo real.

Algoritmo 17: Alterando a quantidade de memória alocada para a PRUSS (Próprio autor).

Algoritmo 18: Programa para executar o binário PRU_ADC.bin (Próprio autor).

Algoritmo 19: Compilando um código assembly para o PRU (Próprio autor).

Algoritmo 20: Exemplo de código assembly para PRU (9).

Algoritmo 21: Script shell para capturar os dados da RAM e salvar em um PDF (37).

Algoritmo 22: Processo de instalação da biblioteca beaglebone_pru_adc (54).

Algoritmo 23: Exemplo em Python comentado executando o modo osciloscópio (Próprio autor).

Algoritmo 24: Outros métodos e atributos importantes da classe capture (Próprio autor).

Algoritmo 25: Código para comunicação UDP do lado do BBB (Próprio autor).

Algoritmo 26: Código para comunicação UDP do lado do PC (Próprio autor).

Algoritmo 27: Fragmentos de código do toolbox.html (Próprio autor).

Algoritmo 28: Exemplo de código usando Jinja 2 (66).

Algoritmo 29: Fragmentos do código utilizado como servidor utilizando o Flask (Próprio autor).

Algoritmo 30: Fragmentos de código do toolbox.html, exemplificando o uso de AJAX (Próprio autor).

Algoritmo 31: Sintaxe do JQuery (Próprio autor).

Algoritmo 32: Fragmento de código do toolbox.html, exemplificando o uso da biblioteca Flot (Próprio autor).

Algoritmo 33: Fragmento de código de main.py, exemplificando o uso de uma subclasse para sobrescrever métodos já predefinidos (Próprio autor).

Algoritmo 34: Código em MATLAB para enviar uma requisição e receber a resposta da aquisição (73).

Algoritmo 35: Fragmento de código utilizado para a comunicação serial utilizando Python (Próprio autor).

Algoritmo 35: Fragmento de código utilizado para a comunicação serial utilizando MATLAB (Próprio autor).

LISTA DE SIGLAS E ABREVEAÇÕES

.dst	<i>Device Tree Source</i>
.dtbo	<i>Device Tree Blob Object</i>
ADC	<i>Analog-digital converter</i>
AJAX	<i>Asynchronous JavaScript And XML</i>
ARM	<i>Advanced RISC Machine</i>
ASCII	<i>American Standard Code for Information Interchange</i>
BBB	<i>BeagleBone Black</i>
BIOS	<i>Basic Input/Output System</i>
BSON	<i>Binary JSON</i>
Capemgr	<i>Cape Manager</i>
CDC	<i>Communications Device Class</i>
CNC	Controle Numérico Computadorizado
CPU	<i>Central Processing Unit</i>
CSS	<i>Cascading Style Sheets</i>
CTAI	Centro de Capacitação Tecnológica em Automação Industrial
DAC	<i>Digital-Analog Converters</i>
DAQ	<i>Data acquisition system</i>
DC	<i>Direct Current</i>
DNS	<i>Domain Name System</i>
DSI	<i>Display Serial Interface</i>
DTO	<i>Device Tree Overlay</i>
EEPROM	<i>Electrically-Erasable Programmable Read-Only Memory</i>
eMMC	<i>embedded MultiMediaCard</i>
FAT	<i>File Allocation Table</i>
FTD	<i>Flattened Device Tree</i>
FTP	<i>File Transfer Protocol</i>
GPU	<i>Graphics Processing Unit</i>
GUI	<i>Graphical User Interface</i>
HDMI	<i>High-Definition Multimedia Interface</i>
HTTP	<i>Hypertext Transfer Protocol</i>
I/O	<i>input & output</i>
I ² C	<i>Inter-Integrated Circuit</i>

IDE	<i>Integrated Development Environment</i>
IP	<i>Internet Protocol</i>
IPC	<i>Instructions per cycle</i>
IRC	<i>Internet Relay Chat</i>
ISN	<i>Initial Sequence Number</i>
ISO	<i>International Organization for Standardization</i>
JTAG	<i>Joint Test Action Group</i>
LDO	<i>Low Dropout Regulator</i>
LED	<i>Light Emitting Diode</i>
MISO	<i>Master Input Slave Output</i>
MOSI	<i>Master Output Slave Input</i>
NI	<i>National Instruments</i>
OCP	<i>Open Core Protocol</i>
opcode	<i>operation code</i>
OSI	<i>Open Systems Interconnectons</i>
PCI	<i>Peripheral Component Interconnect</i>
PCI-e	<i>PCI-express</i>
PDA	<i>Personal Digital Assistant</i>
PDF	<i>Portable Document Format</i>
PRU	<i>Programmable Real-Time Unit</i>
PRU-ICSS	<i>Programmable Real-Time Unit and Industrial Communication Subsystem)</i>
PRUSS	<i>Programmable Real-time Unit Sub System</i>
PWM	<i>Pulse Width Modulation</i>
RAM	<i>Random Access Memory</i>
rev.	<i>revision</i>
RISC	<i>Reduced Instruction Set Computer</i>
SAR	<i>Successive Approximation Register</i>
SCR	<i>Switched Central Resource</i>
SD	<i>Secure Digital Card</i>
SO	<i>Sistema Operacional</i>
SOC	<i>System on Chip</i>
SPI	<i>Serial Peripheral Interface</i>

SSH	<i>Security Shell</i>
TCP	<i>Transmission Control Protocol</i>
TTL	<i>Transistor-transistor-logic</i>
UART	<i>Universal Asynchronous Receiver/Transmitter</i>
UDP	<i>User Datagram Protocol</i>
UEFI	<i>Unified Extensible Firmware Interface</i>
USB	<i>Universal Serial Bus</i>
UTF-8	<i>8-bit Unicode Transformation Format</i>

1 Introdução

Um sistema de aquisição de dados, ou DAQ (*Data acquisition system*), é um dispositivo, ou um conjunto deles, capaz de coletar, armazenar e distribuir uma determinada informação de tal forma que esta, posteriormente possa ser manipulada ou utilizada para entender melhor um certo fenômeno. Na prática esses sistemas são utilizados para capturar dados de uma determinada variável física de um processo, geralmente vinda de um sensor.

1.1 Um breve histórico

Inicialmente os sistemas de aquisição de dados eram dispositivos eletromecânicos que mostravam uma determinada grandeza física em um visor analógico e registravam as mesmas em papéis ou fitas magnéticas. Estes primeiros aparelhos que registravam os dados de maneira independente ficaram conhecidos como *data loggers*. Atualmente o uso de *data loggers* ainda é bastante comum, por serem mais simples e robustos. Entretanto, com o avanço da eletrônica e com o advento dos computadores, surgiram novas classes de equipamentos que capturam dados, dentre eles, os que podem ser conectados a PCs e trabalham em conjunto com *softwares* de engenharia e matemática, como MATLAB e LabVIEW, também conhecidos como *PC-based data acquisition equipment* (1).

Os primeiros *PC-based DAQs* surgiram na década de 60, inicialmente utilizado pelas grandes indústrias e centros de pesquisa. Entretanto, a partir da década de 90, com a popularização dos computadores pessoais, surgiram os primeiros DAQs na forma de placas de extensão para computadores utilizando *slots* de extensão PCI¹ (*Peripheral Component Interconnect*). Hoje em dia os *slots* PCI evoluíram para o padrão PCI-e (*PCI-express*), um barramento com elevadíssima taxa de transferência de dados, podendo chegar a 32 Gbps para o padrão PCI-e 4.0 (2). As placas PCI ainda existem, mas estão entrando em desuso, pois todas as placas-mãe atuais vêm somente com o padrão mais moderno. Na figura 1 tem-se dois modelos de DAQs na forma de placas de extensão para computadores com o padrão PCI (Esquerda) e o padrão PCI-e (Direita).

¹ Padrão de barramento interno de computadores para conectar periféricos.

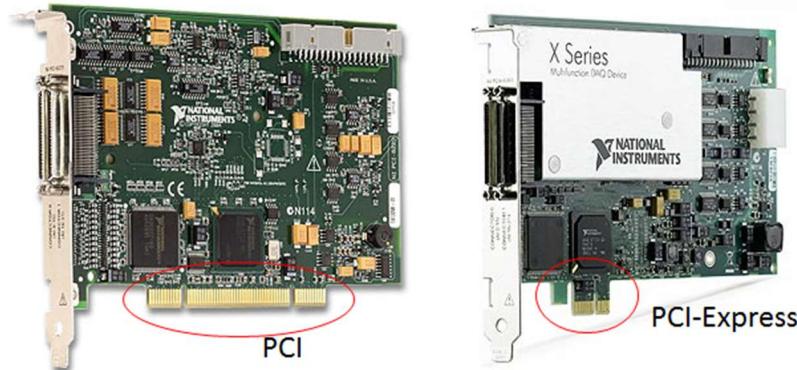


Figura 1: Placas PCI e PCI-Express da National Instruments (3).

Os modelos de DAQs na forma de extensão de placas para PCs foram bastante comuns na década de 90, porém estas classes de dispositivos podem sofrer interferências eletromagnética e eletroestática devidas às máquinas rotativas dos computadores, como *coolers*, e à própria estrutura de barramentos do computador, necessitando de um sistema de proteção contra estes tipos de interferência. Atualmente os fabricantes vêm disponibilizando sistemas de aquisição de dados em uma caixa separada do computador, garantindo um isolamento melhor em relação às interferências citadas. O fato de estar externo às máquinas de análise permite também, uma maior portabilidade e menos restrição de espaço, podendo ser utilizado em conjunto com *notebooks*, não se limitando apenas a uma máquina. Em alguns casos, ainda permite a comunicação entre DAQs a metros ou quilômetros de distância do seu PC de análise, por exemplo, a distância entre o chão de fábrica e a sala de controle.

Estas caixas separadas do computador só foram possíveis com a evolução dos protocolos de comunicação, principalmente aqueles relativos ao PC e seus periféricos externos. Os primeiros dispositivos isolados dos computadores, ou *stand-alone*, utilizavam principalmente a comunicação serial RS-232 e suas variantes, como o RS-485; este último mais frequente no meio industrial.

Com o tempo, surgiram protocolos de comunicação com alta largura de banda, como a USB (*Universal Serial Bus*), a partir da versão 2.0, a Ethernet e redes sem fio de alta velocidade como Wi-Fi, permitindo o surgimento de *data acquisitions stand-alone* de alta velocidade e vários canais. Nas indústrias há, ainda, muitos dispositivos que utilizam os padrões de conexão mais antigos. Entretanto, nos escritórios, onde a troca de equipamentos é bem mais frequente, aparelhos com protocolos antigos estão cada vez mais escassos. Portanto, para

DAQs que funcionam em escritórios e centros de pesquisa, utilizar padrões de comunicação modernos é uma necessidade devida, principalmente, à compatibilidade, pois está cada vez mais raro encontrar computadores com porta serial, por exemplo.

1.2 Situação atual dos DAQs

Historicamente os sistemas de aquisição de dados foram criados para atender à necessidade de grandes indústrias e centros de pesquisa. Estes tipos de equipamentos, por sua vez, nunca foram baratos, pois há uma demanda por equipamentos de alta qualidade, com condições de operação mais severas, para os quais há clientes com grande poder de compra. Por isso, os maiores fabricantes destes dispositivos entregam soluções com preços pouco convidativos e, quase sempre, com *software* e *hardware* proprietários.

Para exemplificar, a NI (National Instruments), grande fabricante do ramo, vende DAQs que variam de R\$2.087,88 à R\$63,540,00, segundo o site oficial da loja no Brasil (3). Com o uso de *software* e *hardware* proprietários, é comum muitos fabricantes praticarem venda casada, onde, para ter uma experiência completa o cliente deve comprar diversos outros produtos além do de interesse. É possível observar esta prática no anúncio da DAQ Instruments, reproduzido na figura 2, onde o interessado deve comprar o pacote do DAQ e as licenças de softwares. Vale mencionar que, sozinho, o aparelho custa \$59,00 e o pacote \$244,00, ou seja, mesmo com o desconto de \$109,00, o custo total é maior que o triplo do custo do equipamento sozinho.



Figura 2: Exemplo de anúncio de venda casada do Kit DI-149 .

1.3 Novas soluções

Na topologia padrão de um DAQ conectado a um computador, é necessário converter o sinal analógico, geralmente vindo de um sensor, para dados digitais, de tal forma que possa ser reconhecido pelo computador. Esses dados posteriormente serão enviados através de uma interface de comunicação

entre o computador e o DAQ. Para que tudo isso seja possível, existe uma CPU (*Central Processing Unit*) capaz de gerenciar os dados de entrada do ADC (*Analog-digital converter*) e a interface de comunicação. Um esquemático desta topologia pode ser visto na figura 3.

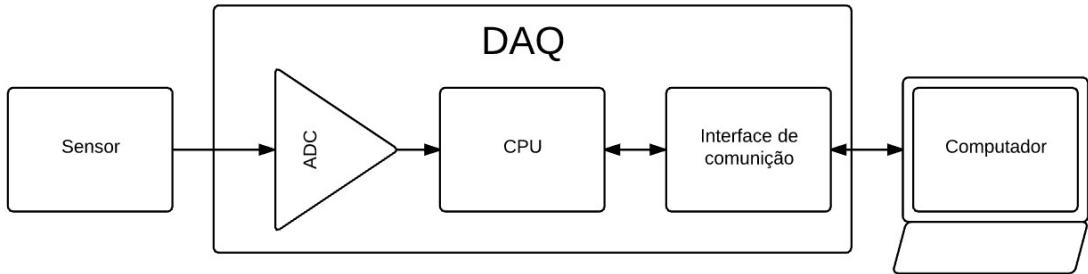


Figura 3: Esquemático de um DAQ conectado a um computador (Próprio Autor).

Na verdade, a topologia do DAQ representado na figura 3 é algo bastante comum na eletrônica embarcada, por isso, já vem embutida em alguns sistemas embarcados. Sistemas embarcados são dispositivos feitos para um propósito específico, desde o controle de um elevador ao gerenciamento de um instrumento de aquisição de dados. Nestes sistemas é comum o uso de microcontroladores, que são computadores dedicados àquela tarefa exclusiva. Muitos microcontroladores, além da CPU e memória, vêm com diversos periféricos como sistemas de comunicação com o meio externo, ADCs, DACs (*Digital-Analog Converters*) embutidos em um único encapsulamento. Essa integração de diversos periféricos em um único *chip* deixa o projeto mais simples e barato. Por isso, é cada vez mais comum os fabricantes disponibilizarem sistemas completos em um único encapsulamento. Este conceito é conhecido como *System on Chip*, os famosos SOCs, que estão embarcados nos mais diversos dispositivos, até mesmo em aparelhos telefônicos celulares.

Os primeiros microcontroladores surgiram no final do século passado. Inicialmente eles não eram muito acessíveis, tanto na questão de preço quanto na complexidade de programação e desenvolvimento. Muitos deles tinham que ser programados em *assembly*² por causa da baixa memória disponível. Entretanto, com o tempo, os novos modelos vieram com mais memória, mais poder de processamento e periféricos mais avançados. Junto a isso, surgiram novas

² Linguagem de programação de baixo nível onde cada palavra representa o código de alguma instrução de uma máquina que usa uma determinada arquitetura.

ferramentas e compiladores, facilitando bastante o desenvolvimento nestes dispositivos para uso específico.

Uma grande revolução neste sentido ocorreu em meados de 2005, quando um grupo de pesquisadores da Itália propuseram um sistema embarcado de baixo custo, fácil utilização e código aberto, conhecido como Arduino. O Arduino é uma plataforma de prototipagem completa incluindo *hardware* e *software* para isto. Do lado do *hardware*, além do microcontrolador, ela vem com todos os circuitos externos para o seu funcionamento, incluindo fonte de alimentação, regulador de tensão oscilador externo, entrada USB e pinos de extensão (4). Do lado do *software* o Arduino dispõe de uma linguagem de programação que abstrai os elementos de *hardware* do microcontrolador. Portanto é possível programar na plataforma sem precisar alterar nenhum valor de registradores diretamente no código, pois operações de baixo nível como estas são feitas em *background* pelo *firmware* pré-instalado na memória do microcontrolador.

O sucesso da placa de desenvolvimento veio principalmente do fato de ser código aberto e da facilidade de utilização, que atraiu diversos entusiastas, “hobbistas” e profissionais da área, tornando-se uma das maiores comunidades, se não a maior, de usuários de sistemas embarcados no momento em que esta monografia está sendo escrita. De uma forma geral, o Arduino, além de aumentar o número de pessoas interessadas em eletrônica, aumentou a quantidade de projetos e a velocidade com que eles eram feitos, possibilitando a criação de diversas invenções como o tênis que amarra sozinho (5) e impressoras 3D.

Hoje existem diversos fabricantes que fazem módulos de expansão para o Arduino conhecidos como *shields*. Existem *shields* para diversas aplicações: controle de motores relés, conexão Ethernet, Wi-fi, Bluetooth, dentre outros. Com os módulos e o Arduino em mãos, o usuário só precisa de conectores, criatividade e programação para criar ideias fantásticas como as citadas no parágrafo anterior, sem a necessidade de fazer as placas de circuito impresso, que dão tanto trabalho, na fase de desenvolvimento.

Depois que o Arduino se tornou popular, diversas organizações e empresas resolveram criar suas placas de desenvolvimento *open source*. Na mesma época em que o Arduino foi criado, surgiram os primeiros *smartphones* baseado em telas *touch-screen*. Junto a eles, surgiram também os SOCs com

processadores ARM³ (*Advanced RISC Machine*) e GPUs⁴ (*graphics processing unit*) de baixo consumo que permitiu aliar o baixo consumo, a baixa dissipação de calor, o tamanho portátil e o alto desempenho em um único *chip*. Em 2008 a Texas Instruments, uma empresa privada que desenvolve diversos produtos semicondutores, como microcontroladores e SOCs com arquitetura ARM para celulares, resolveu criar a BeagleBoard em parceria com a Digikey e a Element14, duas empresas do ramo de varejo eletrônico nos EUA. A ideia era demonstrar o poder do SOC OMAP3530 em uma placa de desenvolvimento do tamanho de um cartão de crédito, capaz de rodar uma distribuição Linux portada para ARM, incluindo os aplicativos preparados para este sistema operacional. A empreitada era uma das pioneiras do ramo e poderia ser promissora se não fosse pelo preço de U\$125,00 que não a fez decolar.

Os dispositivos baseados em e-Linux⁵ (*Embedded Linux*) só se tornaram populares quando a fundação Raspberry Pi resolveu criar um produto de mesmo nome por apenas 35 dólares, em fevereiro de 2012. A ideia da fundação era criar uma plataforma de baixíssimo custo que ajudasse as crianças a aprender a programar (6). Hoje o Raspberry Pi é de longe a *e-Linux board* mais vendida e, também, com a maior comunidade de usuários.

Junto ao sucesso do Raspberry, diversas empresas e fundações criaram suas plataformas *open source* com Linux embarcado, ou atualizaram suas antigas soluções, como foi o caso da empresa criadora do BeagleBoard. Assim, aproveitando a onda dos computadores *single board*, a Texas Instruments criou o BeagleBone Black. O grande diferencial deste computador em relação ao Raspberry Pi não é o poder de processamento: o BBB (BeagleBone Black) tem 65 pinos de extensão, um microcontrolador auxiliar para programação em tempo real, o PRU (*Programmable Real-Time Unit*), e muito mais periféricos que o Raspberry Pi. Para efeito de comparação, na tabela 1, tem-se a comparação entre BeagleBone Black, Arduino Yun, uma das versões do Arduino com Linux embarcado, e o Raspberry Pi.

³ Arquitetura de processador de 32 bits de alto poder de processamento, baixo consumo e baixa dissipação de calor que permitiu surgimento de *smartphones* como iPhone.

⁴ Tipo de processador especializado em processar gráficos. Hoje em dia as GPU estão cada vez mais versáteis, permitindo processar outras tarefas como equações matemáticas, física e inteligência artificial (74).

⁵ *Embedded Linux*, ou Linux embarcado, é o termo utilizado para distribuições Linux portadas para ARM com o intuito de funcionar em sistemas embarcados.

	Arduino Yun	BBB	Raspberry Pi
CPU	MIPS32	ARM Cortex-A8	ARM1176
CPU Freq.	400Mhz	1Ghz	700Mhz
Microcontrolador	ATmega32U4	PRU	Não tem
μ Controlador Freq	16Mhz	200Mhz	-
RAM	64Mb	512Mb	512Mb
GPU	Não tem	PowerVR SGX530	Broadcom VideoCore IV
Memória interna	16Mb	4GB	Não tem
Memória externa	Micro SD	Micro SD	SD
I/O	20	65	17
Ethernet	10/100 Mbit	10/100 Mbps	10/100 Mbps
ADC	12x 10bits, 0-5V	7x 12bits 0-1,8V	Não tem
PWM	7x	8x	1x
UART	2x	4x	1x
SPI	1x	2x	2x
I ² C	1x	2x	1x
USB host	1x	1x	2x
USB client	1x	1x	Não tem
Video	Não tem	Micro HDMI	HDMI, RCA, DSI
Audio	Não tem	Micro HDMI	HDMI, P2
Preço	\$75	\$55	\$35

Tabela 1: Comparação entre os e-Linux board open source mais usados no mercado (7).

Pela tabela, percebe-se que o BBB é o único concorrente que avalia o alto poder de processamento com a grande quantidade de periféricos e I/O (*input & output*). A quantidade elevada de I/Os permitiu a fabricante do BeagleBone criar o conceito de placas de expansão para a e-Linux board conhecidas como *capes*, semelhante aos *Shields* disponíveis para Arduino.

Quando lançado, o BeagleBone Black fez um relativo sucesso, mas que o seu antecessor, principalmente por causa da nova política de preço, seu microcontrolador integrado, a quantidade generosa de I/Os e seus periféricos. Entretanto, isso não foi o suficiente para desbancar o Raspberry Pi. Parte disso é devido à forma de como as e-Linux boards são utilizadas. A maioria das

aplicações dos *single board computers* está na área da computação, como na criação de pequenos servidores, centrais de emulação, *media centers*, câmeras de vigilância, dentre outras. Para isso, as poucas portas do Raspberry Pi são suficiente – quando há a necessidade de alguma aplicação em tempo real, é comum utilizá-lo em conjunto com algum dispositivo dedicado, como um Arduino, por exemplo.

Outro motivo para o BeagleBone Black não ter desbancado o Raspberry, foi o fato do primeiro ter sido lançado tarde. Em 2013, quando foi lançado, a comunidade do Pi já estava grande e já existiam outros concorrentes, como a Cubie Board. Além disso, as *capes*, que poderiam ser um grande diferencial, são caras e sem muita variedade. Mesmo assim, o BBB é um dos *singles board computers* mais utilizados, mas está longe de ser o mais popular e, ainda, o conceito das *capes* não logrou êxito comercial, como a fabricante previa.

1.4 Objetivo deste trabalho

O CTAI (Centro de Capacitação Tecnológica em Automação Industrial) é um centro da UFBA com laboratórios destinados ao desenvolvimento de atividades de pesquisa e desenvolvimento. Neste centro houve a demanda de novos dispositivos de aquisição de dados. O método tradicional seria adquirir estes aparelhos através de uma fabricante tradicional como a National Instruments, mas com o baixo custo da eletrônica embarcada, fez-se esta opção, pois além de economizar, poderá gerar novas tecnologias relativas aos DAQs, até então, das quais o Brasil é carente.

Embora o BeagleBone Black tenha os seus contras, ele ainda é uma das melhores opções para construir um protótipo de um DAQ moderno. Esta plaquinha contempla um ADC de alta qualidade em relação aos outros da categoria, uma CPU de alto desempenho e interfaces de comunicação modernas, acessíveis e com elevada largura de banda.

Este trabalho, então, propõe provar que um BeagleBone Black pode ser utilizado como um dispositivo de aquisição de dados acessível aos computadores de escritórios e com taxa de aquisição e número de canais comparáveis a alguns modelos de DAQ dos fabricantes tradicionais. Para este trabalho, será utilizada a porta USB disponível no BBB como interface de comunicação com o mundo externo. Em trabalhos futuros, os conceitos apresentados neste trabalho podem servir como base para a construção de um dispositivo de aquisição de dados mais

completo, capaz de trabalhar com os acelerômetros analógicos do CTAI da mesma forma que os que seriam adquiridos pela NI.

2 O BeagleBone Black

Nesta seção será apresentado os fundamentos necessários, relativos ao BeagleBone Black, para a execução dos experimentos no capítulo 3. No meio do percurso serão feitas as configurações necessárias para a execução dos mesmos experimentos.

2.1 Componentes do BeagleBone Black

No capítulo 1 foi introduzido o BeagleBone Black como um minicomputador de arquitetura ARM. Esta seção irá mostrar os componentes da placa e os integrados ao SOC Sitara AM335x. Na figura 5 tem-se uma foto de frente e verso da placa de circuito impresso do BBB, identificando os componentes, e na tabela 2 identifica a função de cada componente na placa. Para complementar na figura 4 mostra os componentes integrados ao SOC Sitara AM3358A, este último disponível na rev. (revision) 3 deste produto.

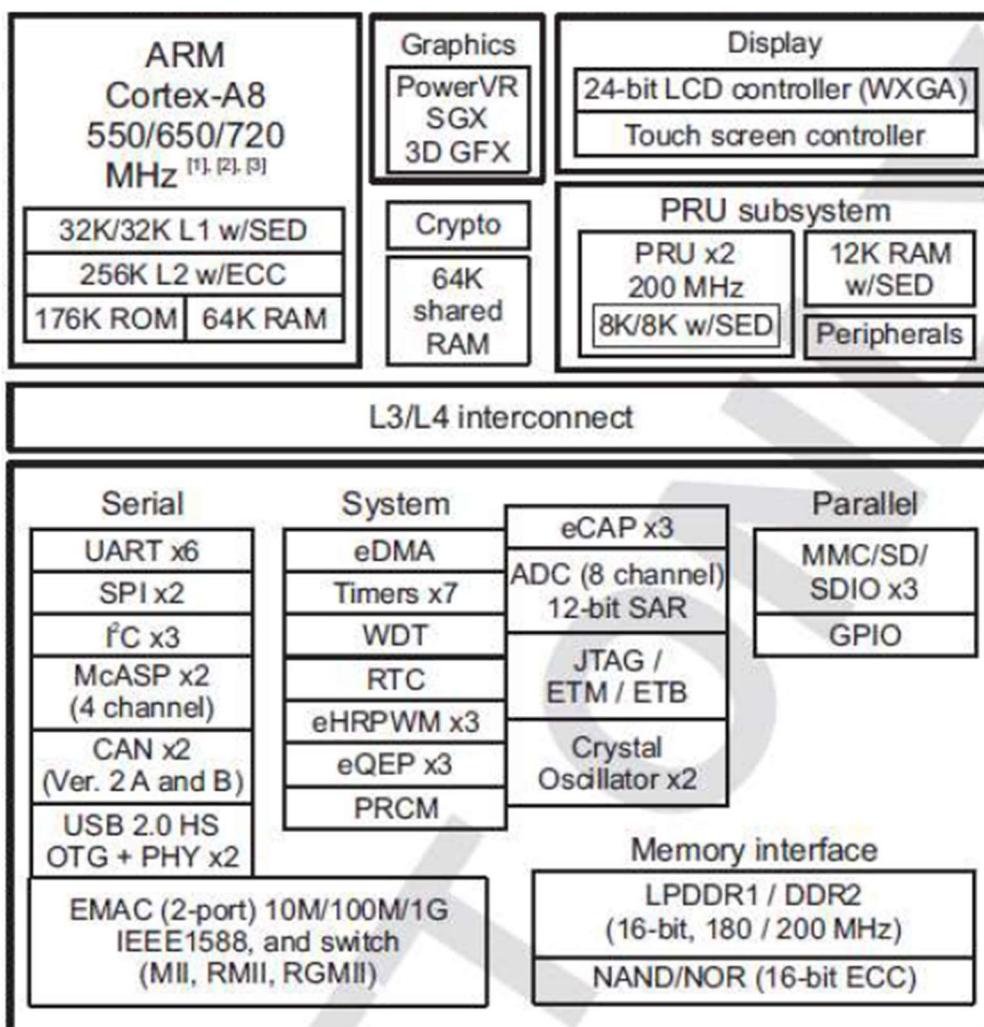


Figura 4: Diagrama do Satira AM3358A (8).

No.	Componente	BBB
1	Satara AM335x	SOC do BBB contendo diversos componentes integrados incluindo CPU, GPU e PRU
2	HDMI <i>framer</i>	Converte o controlador de LCD do AM335x
3	Memória RAM	512MB DDR3
4	eMMC	4GB de memória de armazenamento interno
5	TPS65217C	Regulador de potência sofisticado com 4 reguladores de tensão LDO ⁶ controlado por I ² C
6	Ethernet PHY	PowerVR SGX530
7	7x LEDs	4GB
8	<i>Push buttons</i>	Micro SD
9	Micro HDMI	65
10	Ethernet RJ45	10/100 Mbps
11	5V DC	7x 12bits 0-1,8V
12	Slot micro SD	8x
13	UART	4x
14	USB 2.0 <i>client</i>	2x
15	USB 2.0 <i>host</i>	2x
16&17	Expansões P8 e P9	1x
18	JTAG	1x
19	Conector de bateria	\$55

Tabela 2: Relação dos componentes do BBB de acordo com a figura 5 (9).

2.2 As distribuições do BeagleBone Black

Como foi dito nos capítulos anteriores, o BeagleBone Black é um minicomputador completo, capaz de rodar sistemas operacionais portados para a arquitetura ARM. Atualmente a comunidade do BeagleBone desenvolveu o *port*⁷ para Android e as distribuições Linux, Debian, Ubuntu, Ångström e Arch Linux.

⁶ Regulador de tensão capaz de regular a voltagem de saída quando a tensão de estrada está próxima da tensão de saída

⁷ *Port*, no que tange os *softwares*, se trata do processo de adaptar um código criado para uma plataforma computacional de modo que o resultado final seja compatível com outras plataformas (75).

As primeiras versões do BBB, mais especificamente as revisões 1 e 2, vinham com Ångström instalado por padrão. Esta última uma distribuição criada exclusivamente para sistemas embarcados, *tablets*, PDAs, *set top boxes*, roteadores e outros dispositivos com SOCs *mobile* (10). Nesta época era bastante comum os usuários remover o Ångström e instalar uma distribuição mais conhecida, como o Ubuntu. O Ångström não tem raízes comuns com as distribuições mais populares no *desktop*, isto dificultava o aprendizado e o desenvolvimento, principalmente se a pessoa não tem contato muito contado com o Linux, que representa boa parcela dos compradores do BBB.

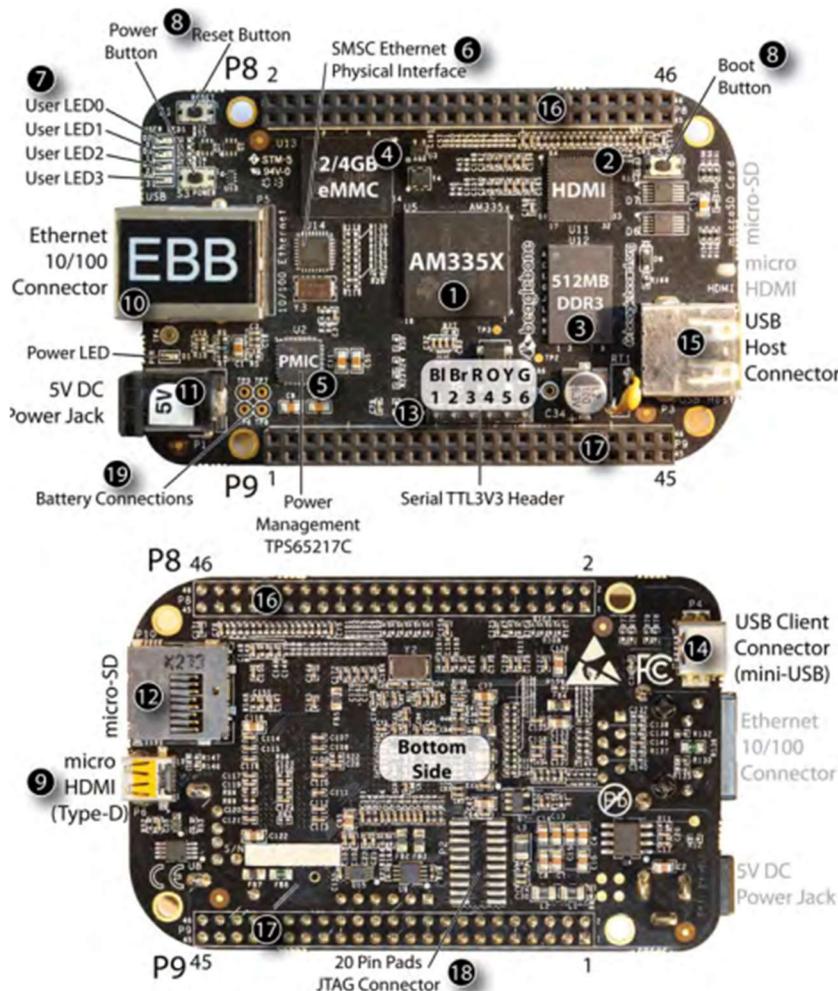


Figura 5: O BeagleBone Black e seus componentes (9).

Percebendo a popularidade de tutoriais ensinando como trocar de distribuição, a Texas passou a incluir o Debian a partir da revisão 3 do BeagleBone Black, lançada em 2014 e até o momento a mais recente. Uma outra adição desta versão foi o aumento da memória eMMC de 2GB para 4GB. Estas duas adições foram fundamentais para tornar o BBB mais atraente para ser usado

como o computador. O espaço extra permitiu a instalação de aplicativos adicionais, além dos que já viam com o sistema operacional, principalmente aqueles mais pesados, como Java, Open Office, reprodutores de vídeo e editores de imagem.

Com o Debian, o BeagleBone ficou muito mais amigável para os novatos, principalmente aqueles que tiveram pouco contato com o Linux. A maioria dos comandos seriam iguais àqueles utilizados no computador, por exemplo, para instalar ou atualizar programas pode-se utilizar o comando *apt*, no Ångström o comando equivalente é outro.

A nova distribuição facilitou, também, a execução de aplicativos com interface gráfica, ou GUI (*Graphical User Interface*), baseados nas *toolchains* mais populares para Linux Desktop, o Qt e GTK+. *Toolchains* são um conjunto de ferramentas que permitem o desenvolvimento de interface gráficas. Ao desenvolver um aplicativo em Qt ou GTK+, por uma questão memória, nem todas as bibliotecas são incluídas no empacotamento do programa. Assim é comum programas terem dependências relativo a *toolchain* que o criou. Portanto, com o Debian a criação de aplicativos fica muito parecida com o que já se fazia no desktop, além disso, o *port* de aplicativos x86 para ARM se torna facilitado. No Ångström é comum outras *toolchains*, como Bitbake e EmbToolkit (11).

O uso de aplicativos com interface de *desktop* é bastante popular em e-Linux boards para servir como interface homem-máquina. Na figura mostra o exemplo de uma máquina CNC (Controle Numérico Computadorizado) controlada por um BeagleBone em conjunto com a *cape* K9 CNC I/O. Para controla-la a fabricante da *cape* criou um aplicativo com interface gráfica para Linux, onde o usuário é capaz de visualizar o processo e interagir com a CNC.

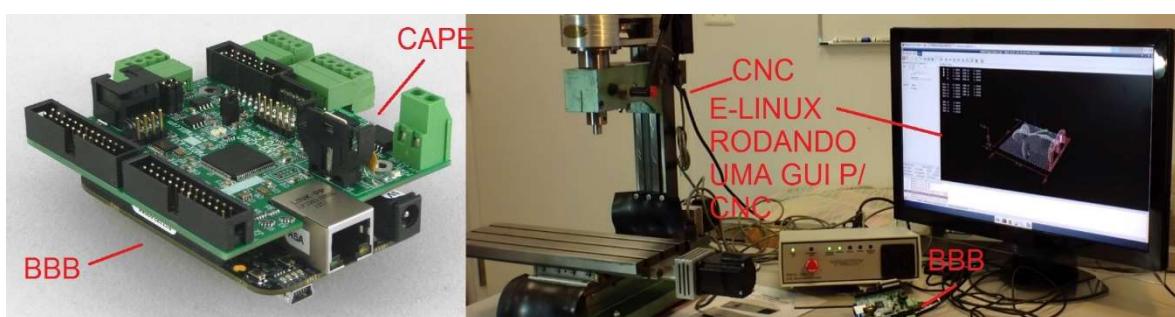


Figura 6: BeagleBone controlando uma CNC e rodando uma aplicação para CNC simultaneamente (12).

2.3 Instalando uma nova distribuição no BeagleBone Black

Na seção 2.2 falou-se da importância de utilizar a distribuição Debian. Portanto, se o BeagleBone Black adquirido for anterior a rev. 3, deve-se instalar o Debian 7.5 Wheezy de 14/05/2014. Caso o BBB venha com uma versão superior a esta é recomendado fazer o *downgrade* para versão 7.5, principalmente, se for a versão 8 Jessie ou superior. A versão 7.5 foi projetada pensando, também, nas revisões anteriores a rev. 3, por isso, ocupa um pouco menos de 2GB, sobrando aproximadamente 2GB dos 4GB da eMMC (Para as rev. 3 ou superiores). Esse espaço extra será suficiente para instalar novos programas, módulos⁸ e gerar grandes arquivos de dados, se necessário.

Um outro motivo para fazer o *downgrade* é que neste trabalho e a maioria das bibliografias encontrada na literatura, faz-se o uso desta versão do Debian, portanto, se for utilizada uma versão diferente, pode haver casos em que determinados procedimentos não sejam os mesmos desta monografia, dificultando a reprodução dos experimentos.

A primeira coisa que deve ser feita para instalar uma nova distribuição é fazer o *download* do sistema operacional, que pode ser baixado na página que contém as últimas imagens do BeagleBone Black, disponível em (13). Caso o leitor desta monografia esteja lendo-a muito tempo depois, existem uma alternativa no domínio oficial do Debian disponível em (14), ou ainda, no site do e-Linux, disponível em (15).

No site das últimas imagens para BBB existem duas versões do Debian 7.5 14/15/2014. Uma delas é para utilizar o sistema operacional por um cartão SD (*without flashing the eMMC*), enquanto a outra grava o sistema operacional na eMMC (*eMMC flasher*). É preferível que o sistema operacional seja instalado na memória interna do BeagleBone, pois, além desta ter uma taxa de leitura e escrita maior do que o cartão SD, fica-se com o *slot* SD livre para outras ocasiões.

O formato do arquivo baixado é **.img.xz** que é um formato de compactação bastante comum no Linux. Caso o usuário esteja utilizando o Windows talvez seja necessário fazer o *download* de alguma ferramenta capaz de descompactar este tipo de arquivo. Uma sugestão de descompactador capaz de trabalhar com este formato é o 7-Zip, disponível em (16), gratuito, *open source* e sem propagandas.

⁸ O nome correto para *drivers* no Linux é módulo, ou module, em inglês. Entretanto, o termo *driver* também é utilizado por causa dos usuários de Windows.

Enquanto a imagem do sistema operacional está sendo baixada, deve-se fazer o *download* de outro programa para gravar a imagem no cartão de memória. Uma sugestão dada pela Wiki do e-Linux (17) é o Win32 Disk Imager, disponível em (18). Quando ambos os arquivos estiverem baixados, descompacte a imagem do Debian que está no formato **.img.xz** e irá obter um arquivo **.img**. Insira o cartão SD no leitor de cartão do seu computador, abra o Disk Imager e selecione a letra correspondente a partição do cartão de memória, selecione a imagem do Debian e clique em gravar como mostra na figura 7. Irá aparecer uma caixa de mensagem explicando que esta operação pode danificar os dados do cartão, clique em yes e espere o processo ser finalizado.

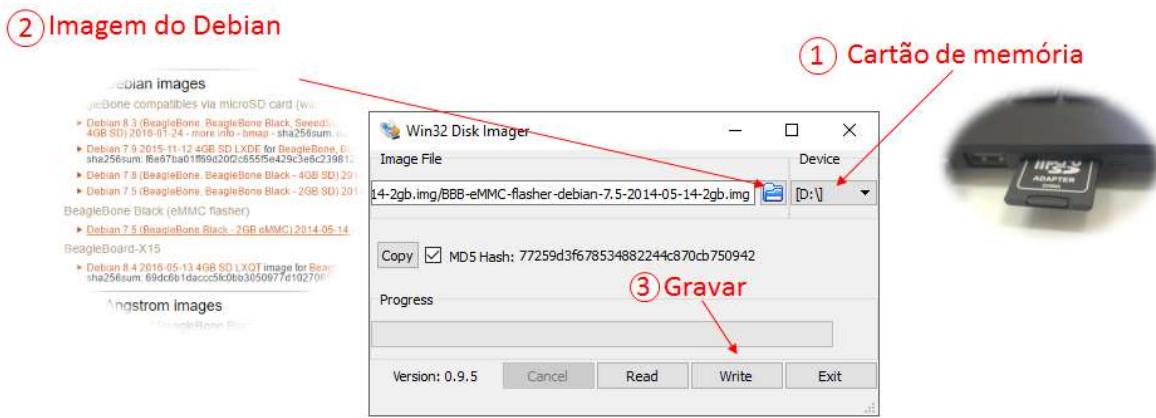


Figura 7: Gravando a imagem do Debian 7.5 de 14/05/2014 no cartão SD (Próprio Autor).

Depois do processo de gravação finalizar com sucesso, retire o cartão SD do leitor e com o BBB ainda desligado, insira o microSD no *slot* apropriado (Item 12 na figura 5). Antes de ligar a placa, mantenha pressionado o *boot switch* (Item 8 na figura 5), e ainda, com o botão pressionado, conecte à USB *client* do BeagleBone (Item 14 na figura 5) e ao computador ou alguma fonte de alimentação. Depois dos *users* LEDs (Item 7 na figura 5) começarem a piscar solte o *boot switch*, a partir daí o sistema operacional será instalado na eMMC. O processo dura cerca de 30 a 40 minutos, durante este tempo não deslique o BeagleBone. Quando a instalação estiver concluída os 4 *user* LEDs irão se apagar.

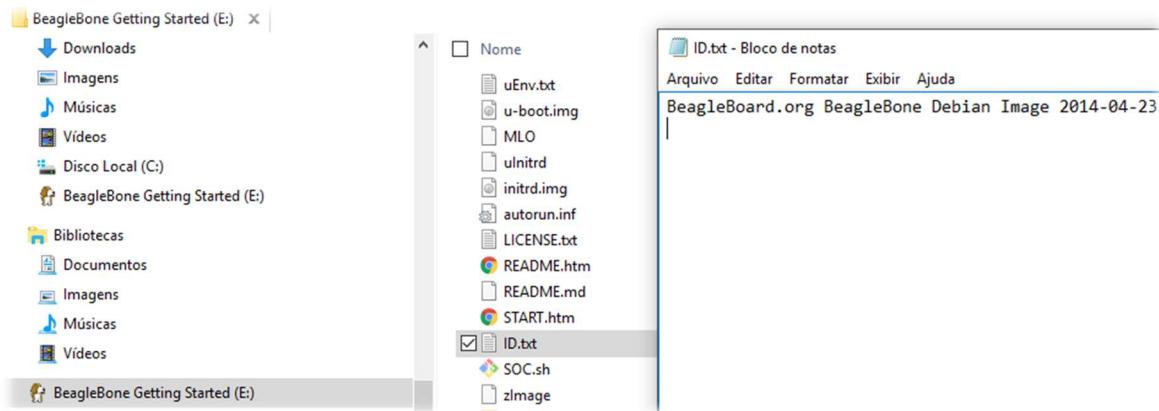


Figura 8: Verificando a atualização do Debian 7.5 foi instalada com sucesso (Próprio Autor).

Neste momento desconecte o cabo USB da fonte de alimentação e, posteriormente, retire o cartão SD, para, então, conectar novamente o BBB o computador. Será montada uma partição FAT com o nome *BeagleBone Getting Started*. Clique nela e abra o arquivo *ID.txt* com o bloco de notas. Se aparecer o texto, “*BeagleBoard.org BeagleBone Debian Image 2014-04-23*”, a atualização da distribuição foi realizada com sucesso.

2.4 Comunicando com o BeagleBone Black

O BeagleBone Black *vanilla*, ou seja, da forma como veio de fábrica, não tem nenhuma interface com usuário além de alguns *push-buttons* e LEDs (*Light Emitting Diode*) indicadores (Itens 7 e 8 na figura 5), e isso não é o suficiente para programar ou adicionar alguma função à placa de desenvolvimento, a menos que usuário use o BeagleBone ligado a um monitor mouse e teclado.

Para interagir com o *board computer* sem a necessidade desses aparelhos é necessário se comunicar com o BBB através de um computador hospedeiro. Existem diversas formas de fazer esta comunicação, mas esta seção irá focar no protocolo SSH e no *serial debug* através dos pinos UART0 (Item 13 na figura 5). Primeiro será falado do protocolo SSH.

2.4.1 O que é o terminal

Nos primórdios da informática não existia *mouse*, nem sistema operacional com interface gráfica, janelas e menus. Portanto a interação homem-máquina era feita por comandos que, por sua vez, executavam aplicativos e faziam operações, como entrar em diretórios, copiar arquivos, escrever documentos e até mesmo programar. Este tipo de interação ficou conhecido como linha de comando e era bastante popular em sistemas mais antigos como o DOS. Com o surgimento do

Windows, a Microsoft passou a desestimular o uso da linha de comando em detrimento à interface de janelas em conjunto com o *mouse*. Foi a partir daí que a informática começou a se popularizar, pois a interface de janelas era mais intuitiva e, por isso, atraiu a atenção das massas.

Com o tempo as pessoas se acostumaram com o ambiente gráfico do Windows e a linha de comando ficou em abandono, não apenas pelos seus usuários, mas também pela própria Microsoft, cujo o principal objetivo era deixar o ambiente de janelas cada vez mais rico e moderno. Por motivos de legado, a empresa de Bill Gates disponibilizou o programa *cmd.exe* ou Prompt de Comando. Ele é um emulador de DOS onde é possível executar boa parte das operações do sistema operacional antigo. Contudo, como não houve atualizações do DOS durante décadas, o Prompt de Comando é ultrapassado, porque as novas tecnologias foram implementadas apenas na interface gráfica, tornando-o bastante pobre em relação ao ambiente de janelas. Entretanto, isto só é verdade no Windows, seus grandes concorrentes, Linux e Mac, não abandonaram a linha de comando.

O Linux e Mac são baseados em um ancestral comum, o Unix. Por isso, sua base de arquivos e a forma de como são organizados são parecidas. Uma das principais semelhanças desses sistemas operacionais é o uso da mesma linguagem de linha de comando, o Shell. Esta linguagem, ao contrário do Prompt Comando, é bastante completa, sendo capaz de fazer quase todas as operações do sistema operacional, as vezes até mais operações que a interface gráfica. O programa que executa os comandos Shell é chamado de terminal.

Por muito tempo o Linux só permitia a instalação de programas e *drivers* através da linha de comando. Uma tarefa um tanto complicada para usuários comuns. Isso, teoricamente, afastou as pessoas comuns deste sistema operacional, chegando ao ponto de muitos atribuírem a baixa adoção do SO (Sistema Operacional) à ausência de uma interface gráfica tão completa quanto a do Windows.

Com o tempo isso mudou e hoje o Linux permite fazer quase tudo, incluindo a instalação de *drivers* e aplicativos pela interface gráfica. Porém, é possível fazer o mesmo pela linha de comando e isso é uma vantagem enorme do Linux e Mac em relação ao Windows, o freguês pode escolher a maneira de interagir com o sistema.

Existem algumas tarefas que são muito mais rápidas e práticas de serem feitas através do terminal, embora não sejam tão intuitivas quanto, um grande exemplo disso é a automação de tarefas. Com o Shell é possível criar rapidamente pequenos macros ou *scripts* capazes de automatizar tarefas chatas e repetitivas. Além disso, por utilizar apenas textos para enviar e mostrar informações, esta é uma forma de interação que consome poucos recursos da máquina. E ainda, o uso exclusivo de texto facilita o compartilhamento de artigos e tutoriais relacionados a interações com o sistema, principalmente se eles forem mais complexos como alterar parâmetros de configurações, pois basta o usuário copiar os comandos do tutorial, colar no terminal e apertar *Enter*. O mesmo tutorial utilizando interface gráfica necessitaria de alguns textos explicativos e capturas de tela, por isso que a maioria dos tutoriais para Linux, disponíveis em livros e na internet, são feitos através da linha de comando, embora seja perfeitamente possível fazer o mesmo pela interface gráfica. O lado ruim dessa prática é que o usuário leigo muitas vezes não tem noção dos processos necessários para a execução de tal tarefa e, por isso, não aprende o porquê ou como reproduz o mesmo sem copiar e colar a sequência de comandos. Por causa disso, muitos usuários recém-chegados no Linux pensam que está é a única forma de fazer determinada tarefa, e acabam abandonando a plataforma por pensar que o sistema operacional não foi feito para “seres humanos”.

2.4.2 SSH e as pilhas de protocolo IP

A sigla SSH significa *Secure Shell*, ou seja, uma forma de enviar comandos com criptografia, ou forma segura, de uma máquina para outra. Este protocolo foi criado, inicialmente, para facilitar o acesso remoto de máquinas dentro de servidores (9). Para exemplificar, imagine que um engenheiro necessita instalar um novo aplicativo de um servidor a quilômetros de distância. Com o SSH o engenheiro por de iniciar uma conexão, ou seção, SSH entre o seu computador e uma das máquinas do servidor para, posteriormente, escrever os comandos de instalação do aplicativo, tudo através do terminal da sua máquina.

O fato do SSH ser baseado em somente texto torna a comunicação leve, no sentido de transmitir e receber poucos dados, e fácil de ser implementada. Além disso, traz consigo todas as vantagens que o Shell oferece em relação ao ambiente de janelas. Tudo isso é evidente quando se compara o esforço necessário para fazer uma comunicação remota através de interface gráfica. Para

que isso seja possível, é necessário, no mínimo, fazer a captura de tela, mouse e teclado do computador remoto, enquanto com SSH, só precisa os dados digitados no terminal sejam enviados no formato de texto criptografado.

O protocolo SSH é orientado a IP (*Internet Protocol*), ou seja, faz parte do mesmo padrão de comunicação utilizado pela internet, também conhecido como pilha de protocolos TCP/IP. Este modelo surgiu como um projeto de comunicação do exército americano (19). Com o tempo o projeto se expandiu e tornou-se a *internet* como conhecemos hoje.

Atualmente o TCP/IP é baseado num modelo simplificado do padrão de referência da ISO (*International Organization for Standardization*) para sistemas de comunicação, o OSI (*Open Systems Interconnections*). O modelo da ISO é baseado em 7 camadas hierárquicas. Cada camada é responsável por executar determinada função e é especificada através de protocolos de comunicação. Estes, por sua vez, podem ser ou não compatíveis com os protocolos de outras camadas. Quando tomado em conjunto os protocolos de diferentes camadas, denominamos pilhas de protocolos. A pilha de protocolos da internet é chamada de TCP/IP. Ela é composta obrigatoriamente de 5 das 7 camadas do padrão OSI (Figura 9). As outras duas camadas, dependendo da aplicação, podem existir, mas são opcionais.

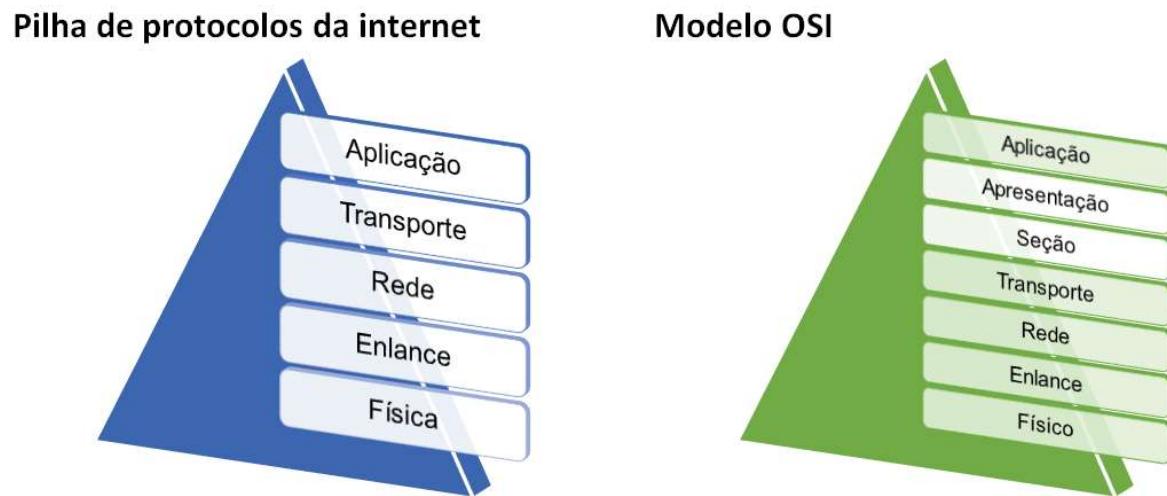


Figura 9: Comparação entre o modelo OSI e a pilha de protocolos da internet (Próprio autor).

A camada com maior grau de abstração do TCP/IP é a aplicação. Nela contém os protocolos responsáveis por oferecer os serviços às máquinas ou ao usuário final. O SSH é um exemplo de protocolo desta camada, ele oferece o serviço de acesso remoto aos computadores. Outros exemplos de serviços

dessas camadas são: enviar arquivos (Protocolo FTP, BitTorrent), navegar na web (Protocolo HTTP) e comunicar-se através de chat (Protocolo IRC). Estes protocolos geralmente são implementados via *software*, junto com a camada de transporte.

A camada de transporte é responsável por transportar as mensagens da camada de aplicação. Existem dois protocolos possíveis para esta camada, o TCP (*Transmission Control Protocol*), que veio primeiro e também o responsável pelo acrônimo TCP/IP. E o UDP (*User Datagram Protocol*), que veio depois, devido a necessidade de aumentar a velocidade de transmissão em aplicações em tempo real. É importante ressaltar que alguns protocolos da camada de aplicação foram feitos para ser usado somente com o protocolo TCP, ou somente com o protocolo UDP, ou ambos. A maioria dos protocolos de aplicação são definidos sob o protocolo TCP. São exemplos de serviços baseados em TCP a navegação web (HTTP) e a transferência de arquivos (FTP). Serviços definidos sob o protocolo UDP podem ser protocolos de jogos online, vídeo conferência e voIP⁹ (*Voice over IP*).

A camada de rede é responsável por movimentar os pacotes de dados de uma máquina para outra. Esta camada tem dois componentes principais, o primeiro deles é o protocolo IP, que define o famoso conjunto de 4 números separados por pontos, os endereços de IP. O protocolo de IP é único e é ele quem caracteriza os sistemas que utilizam a pilha de dados TCP/IP. O outro componente da camada de rede são os protocolos de roteamento. Estes protocolos só fazem sentido em redes maiores, como as redes locais ou a rede mundial de computadores. Como, na nossa aplicação, estamos interessados apenas na comunicação ponto a ponto entre o computador e o BBB, utilizando a USB, esta segunda parte da camada de rede é irrelevante. Neste caso, o endereço de IP padrão do BeagleBone é 192.168.7.2. O componente roteamento só se torna relevante se o BBB for conectado à rede local pela porta Ethernet (Item 10 da figura 5), através de um roteador.

A camada de enlace de dados, que, por sua vez, significa, ligação ou *link* de dados, define as especificações de transmissão, recepção, controle de fluxo, opcionalmente, a correção de erros que venham a ocorrer na camada física. As

⁹ Tecnologia de comunicação que utiliza o *Internet Protocol* para transferir chamadas telefônicas e serviços de voz de uma maneira geral.

especificações desta camada estão intimamente ligadas ao meio em que se propagam. A família de protocolo DSL, por exemplo, é muito comum nas bandas largas comerciais, pois foi projetado para utilizar as linhas telefônicas como meio de transmissão. Já o protocolo Wi-Fi foi projetado para ser utilizado em redes locais cujo o meio de transmissão é o ar, assim tem suporte a senhas de acesso, correção de erro e tem um limite de usuários simultâneos menor do que protocolos para redes sem fio maiores, como o LTE, padrão utilizado na *internet* móvel dos celulares. Nas redes locais com fio é comum utilizar o protocolo Ethernet na camada de enlace.

Por fim, a camada física é o próprio meio de transmissão, ou seja, o ar, cabo coaxial, linhas telefônicas. Como dito anteriormente, é comum que os padrões da camada de enlace seja implementado em conjunto com a camada física, por isso, os circuitos integrados de rede geralmente já implementam as três camadas mais básicas (Rede, enlace e físico) via *hardware*. Por exemplo, as placas de rede dos computadores pessoais especificam que deve ser utilizado como camada física cabos UTP com conectores RJ45 em conjunto com o protocolo Ethernet na camada de enlace e o protocolo IP na camada de rede. O CI SMSC do (Item 6 da figura 5) segue esta mesma especificação e é utilizado em conjunto com o conector Ethernet 10/100 (Item 10 da figura 5). Neste caso o BeagleBone deve ser conectado a um roteador como mostra na figura 10. Como não estamos utilizando a porta USB, o endereço de IP da placa de desenvolvimento será diferente de 192.168.7.2.

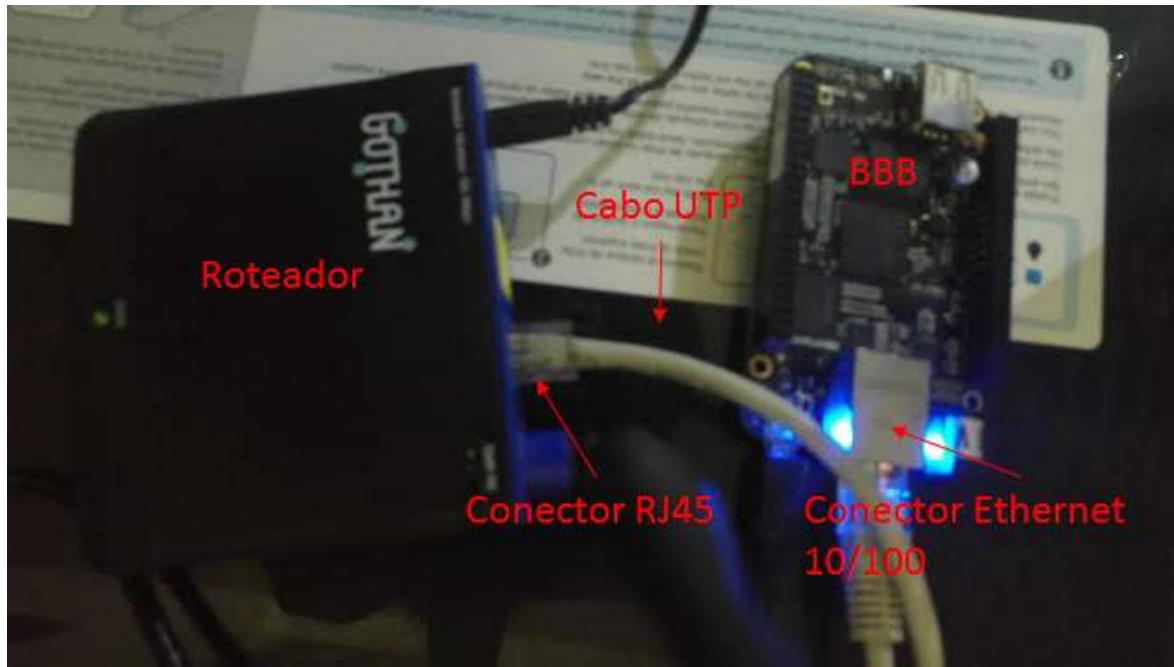


Figura 10: BeagleBone Black conectado a um roteador (Próprio autor).

Um outro cenário possível é a utilização do protocolo Ethernet sobre a porta USB (*Ethernet over USB*). A porta USB é bastante utilizada para conectar periféricos a computadores e muitas vezes estes periféricos deve utilizar protocolos definidos para funcionar exclusivamente sobre IP, por isso, é comum que impressoras, celulares, *smartphones* e dispositivos embarcados que utilizam a porta USB para se comunicar com o computador, implementem as camadas de enlace e rede via *software* para que seja possível utilizar os serviços disponibilizados por IP.

No caso de dispositivos embarcados com Linux isso é quase uma obrigação, visto que o próprio Linux já vem com driver USB-eth, que implementa os protocolos Ethernet e IP sobre a interface física da USB, criando uma rede local entre cliente e hospedeiro, no nosso caso computador e BeagleBone. A partir daí quase todos os serviços e protocolos disponibilizados por IP estão disponíveis, incluindo aí a comunicação remota via SSH. No Windows é importante que os *drivers*, disponibilizados em (20), estejam atualizados, pois ambos os lados (Computador e BeagleBone Black) devem estar preparados para este tipo de conexão.

Pelo fato do protocolo Ethernet não ser um padrão definido na comunicação USB, e ainda ser implementado via *software*, é normal que o desempenho teórico fique abaixo dos 480Mbps teórico da USB 2.0. Segundo o

usuário *jons34yp* do site de perguntas e respostas, superuser.com (21), foi possível atingir a média de 90Mbps em uma comunicação entre um *smartphone* com Android e um computador. Para este trabalho taxas de transferências próximas a 2Mbps são mais do que o suficiente.

2.4.3 Socket de rede

Imagine que você esteja jogando uma partida *online* com seu amigo ao mesmo tempo que comunica com ele através um aplicativo de áudio conferência como Skype. Nesta situação dois processos, ou aplicações, do seu computador estará enviando e recebendo informações para o mesmo IP. Em outras palavras, fica complicado saber se a informação que chega daquele IP é relativa ao jogo ou ao sinal de áudio, se não houver alguma outra informação além do endereço de IP.

Essa informação adicional é conhecida como porta. O jogo online que você e seu amigo estão jogando se comunica através de uma porta enquanto a comunicação via Skype ocorre em outra porta. Em outras palavras, uma porta é um endereço único de um determinado processo em sua máquina. Com a combinação do endereço de IP com o endereço de porta, então tem-se um endereço único de um determinado processo, conhecido como *socket*. Com os *sockets* fica fácil de saber se aquele pacote de dados é para o Skype ou para o jogo online.

Nos computadores existem dois endereços de portas um para processos que se comunicam via TCP e outro para processos que se comunicam via UDP. As portas são endereços de 16 bits e cada processo comunicante do sistema operacional só pode ocupar um número de porta por vez, ou seja, o número máximo de processos comunicantes no sistema operacional é 65535 para os processos que se comunicam via UDP e 65535 para processos que se comunicam via TCP, totalizando 131070 processos simultâneos.

Para evitar que aplicações essenciais tentem utilizar a mesma porta, gerando conflitos. As portas até 1024 são de uso específico do sistema operacional e não podem ser utilizadas por qualquer aplicação. Dentre estas portas específicas o protocolo HTTP usa a porta 80, SSH usa a porta 22, Skype porta 81 e assim por diante. O jogo online, como não é uma aplicação oficial, deve usar uma porta maior que 1024.

2.4.4 Conectando ao terminal através do SSH

O Linux, Mac e sistemas operacionais baseados no Unix, já vem com suporte nativo ao SSH, que pode ser acessado através do terminal. Entretanto, usuários do Windows devem baixar um cliente SSH como PuTTY, disponível em (22). Antes de iniciar o processo de comunicação é importante atualizar os *drivers* do BeagleBone, disponível em (20).

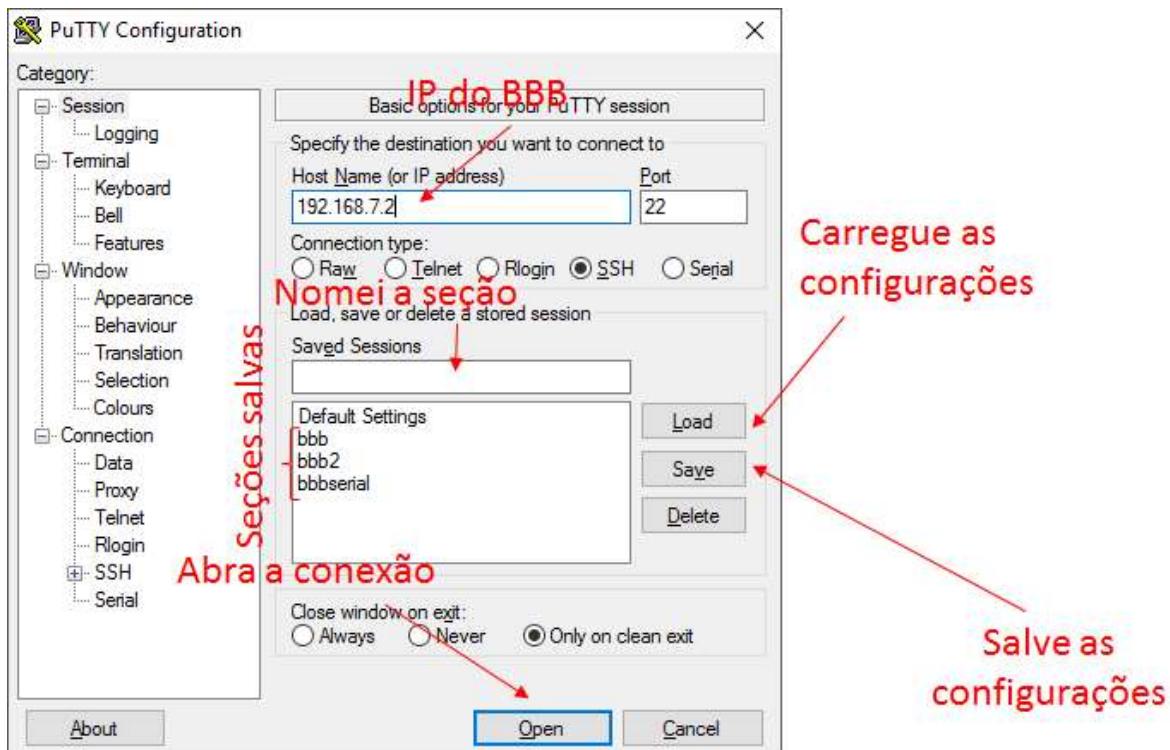


Figura 11: Configurações para conectar o BBB com o computador através do PuTTY (Próprio Autor).

Depois da atualização feita e o PuTTY baixado, abra este aplicativo e no campo *Host Name (or IP address)* escreva o IP do BeagleBone que por padrão é 192.168.7.2. Deixe as outras opções nos valores padrões como mostra a figura 11. Por fim, clique em *Open* para iniciar uma conexão SSH entre o computador e o BeagleBone Black. Uma janela de terminal se abrirá, inicialmente, pedindo para que o usuário entre com um *login* e senha. Mais detalhes de como operar o terminal do BeagleBone Black podem ser vistos na seção 2.5.

Existe a possibilidade de criar configurações pré-definidas neste aplicativo. Para isso, preencha os campos e selecione as opções de acordo como o desejado. Depois no campo *Saved Sections* escreva o nome dessa configuração e clique em *Save*. Caso o usuário, por algum motivo, feche o PuTTY e abra-o novamente, os campos do aplicativo serão restaurados para os seus valores

padrões. Se houver alguma comunicação pré-definida é possível carrega-la rapidamente apertando o botão *Load*.

2.4.5 UART e o *serial debug*

Os protocolos da *internet* são muito susceptíveis a ataques de maliciosos, geralmente por vírus. Para evitar possíveis ataques os *firewalls* bloqueiam a maior parte das portas de conexões TCP e UDP, podendo tornar a comunicação SSH indisponível em alguns casos. Em outros casos pode ser que os *drivers* não funcione corretamente e a conexão por IP entre o BBB e computador não seja estabelecida.

O acrônimo UART significa *Universal Asynchronous Receiver/Transmitter*, é um tipo de conexão ponto a ponto assíncrona, ou seja, que não precisa do *clock* para sincronizar os dados. Para que a conexão seja estabelecida são necessários pelo menos dois fios, o RX responsável por receber os dados e o TX responsável por enviar os dados. Fios adicionais podem ser úteis, como um fio para igualar o referencial de terra (GND) entre os dois dispositivos de comunicação. Na figura 12 têm-se uma representação de uma comunicação serial entre dois dispositivos, incluído o GND. Observe que o RX de um dispositivo é conectado no TX do outro e vice-versa.

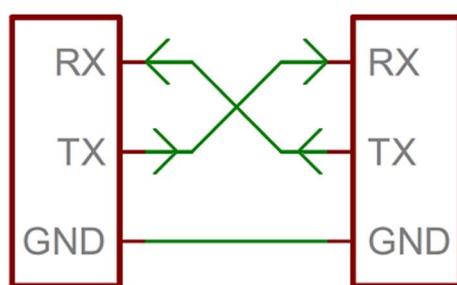


Figura 12: Ligação dos cabos em uma comunicação serial UART (23).

O protocolo UART tem pacotes pequenos que podem ser configurados de acordo com a aplicação (Figura 13). Para que a conexão funcione ambos os dispositivos devem ser configurados com os mesmos parâmetros. Estes parâmetros inclui as especificações dos pacotes e a quantidade de pacotes por segundo (*baud rate*), podendo variar o tamanho da palavra (*Data bits*), quantidade de bits de fim (*Stop bits*) e correção de erro por paridade (*Parity bit*), que podem ser vistos na tabela 3.

Parâmetro	Valores permitidos	UART0	Função
Data bits	5 a 9	8	Determina a quantidade de bit de dados enviado em um pacote
Stop bits	1 a 2	1	Quantidade de bits de para determinar o fim do pacote
Parity	N/O/M/S	N	Bit de paridade (Correção de erro)
Speed (Baud rate)	Vários	115200	Velocidade de transmissão dada em pacotes por segundo
Flow Control	None	None	O controle de fluxo não é suportado

Tabela 3: Parâmetros da UART0 e suas funções (Próprio autor).



Figura 13: Os pacotes de dados da comunicação serial podem ser configurados (23).

O BeagleBone tem 5 portas UART, sendo que a UART0 já vem pré-configurada pelo sistema operacional e tem os parâmetros definidos na tabela 3. Seu uso exclusivo para comunicação serial com propósito de *debug* e está disponível pelos pinos Serial TTL3V3 Header (Item 13 da figura 5). O protocolo USB especifica a conexão serial através da classe CDC (*Communications Device Class*). Existem cabos ou conversores que já fazem a conversão da UART para a conexão serial especificada pelo protocolo USB, estes cabos são conhecidos como USB Serial TTL (*Transistor-transistor-logic*). Eles têm de um lado um conector USB macho e do outro lado alguns fios coloridos que devem ser conectados diretamente aos pinos Serial TTL3V3 Header do BeagleBone Black. A coloração dos fios varia de fabricante para fabricante, por isso, antes de fazer as ligações é importante saber a função de cada fio. Na Wiki do E-Linux, disponível em (24), têm as principais colorações utilizadas.

2.4.6 Conectando ao terminal utilizando o *serial debug*

Verificada a função de cada fio do conversor TTL, conecte os fios aos pinos do Serial TTL3V3 Header como mostra a figura 14. A ordem de conexão depende da marca do conversor, mas de uma maneira geral deve-se conectar o GND do cabo no pino 1, RX no pino 4 e o TX no pino 5.

Depois de conectado clique com o botão direito no menu iniciar e selecione gerenciador de dispositivos e procure por Portas (COM e LPT) e estará listado o cabo TTL, anote o número da porta COM. Com o número em mãos abra o PuTTY selecione *Serial* e preencha os campos de acordo com a figura 15 e clique em *Open*. A partir daí uma janela de terminal se abrirá e será possível controlar o BeagleBone remotamente semelhante ao protocolo SSH.

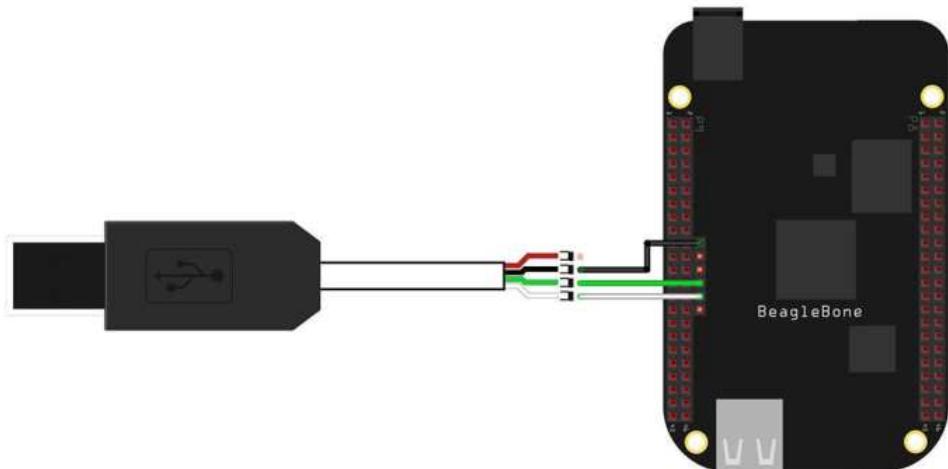


Figura 14: Conectando o cabo TTL ao BBB (25).

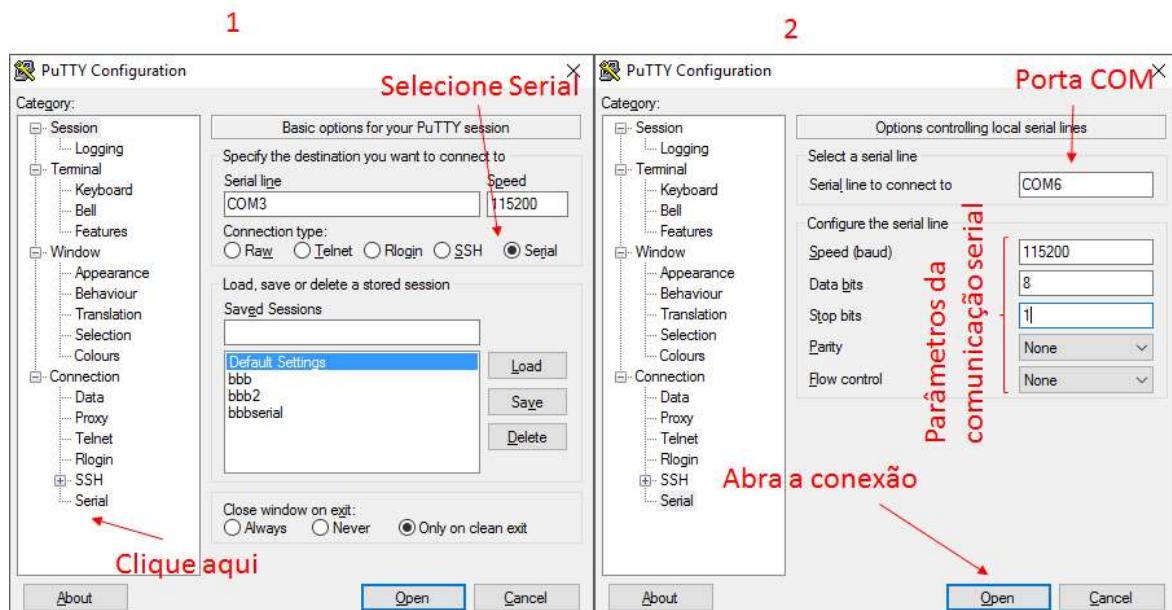


Figura 15: Conectando ao BBB através do *serial debug*.

2.5 Terminal do BeagleBone Black

Na seção 2.4 foi explicado o que é o terminal e como conectá-lo através do PuTTY utilizando o protocolo SSH ou o *serial debug*. Nesta seção será explicado como funciona o terminal.

2.5.1 Usuários no Linux

No Linux existem, basicamente, três tipos de usuários. O primeiro tipo são os usuários comuns. Esses usuários são equivalentes ao que temos no Windows. A função deles é permitir que diferentes pessoas compartilhe o mesmo computador, mas tenha arquivos e configurações pessoais diferentes. Sendo assim, cada usuário comum tem uma pasta `/home/<nome_do_usuário>`. Nela estão guardadas as configurações e arquivos pessoais de cada pessoa, muito parecido com que se tem no Windows.

No BeagleBone o usuário comum é o ***debian***, a senha é ***temppwd***. É possível criar mais usuários, mas, ao contrário do ambiente *desktop*, isso não é muito trivial em sistemas projetados para tarefas específicas, como este da monografia.

Os usuários comuns têm acesso limitado a arquivos fora da pasta sua ***/home***. Portanto este tipo de usuário não pode fazer operações que depende de manipulações fora desta pasta, como instalar arquivos através do comando ***apt-get***. Para isso, existe um outro tipo de usuário, o ***root***.

O *root*, ou super usuário, é equivalente ao administrador do sistema. O *root* é único, portanto não pode existir mais de um super usuário na mesma máquina. O seu *login* é sempre o próprio nome, ***root***, e a senha pode variar, mas no caso do BeagleBone Black é em branco.

Normalmente não é recomendado acessar o sistema operacional através do *root*. Pois, se por acaso este usuário executar algum programa malicioso, este pode ter acesso completo ao sistema e causar problemas. Por isso, alguns sistemas operacionais para *desktop* não permitem que as pessoas entrem diretamente com o *root*. Ao invés disso, o indivíduo deve acessar o sistema como um usuário comum e, se necessário, ter acesso temporário ao ***root*** através do prefixo ***sudo*** antes do comando que se deseja executar. Feito isso o terminal irá pedir a senha do ***root*** e ao apertar *Enter* o comando será executado como super usuário. Após a execução do comando o sistema devolve novamente o acesso ao usuário comum.

Uma outra maneira de acessar o ***root*** quando se está logado através de um usuário comum, é através do comando ***su***. A diferença deste comando para o pré-fixo ***sudo*** é que, o primeiro dá acesso permanente ao super usuário ***root***, enquanto o outro, o acesso se dá apenas durante a execução do comando. Isso é

útil quando precisa fazer várias operações como ***root***. Somente o super usuário pode executar este comando, portanto deve-se escrever ***sudo su***.

O terceiro tipo são os usuários do sistema. Diferente dos outros dois, essas contas não se conectam. São usadas apenas para propósitos específicos do sistema operacional, por exemplo, o usuário, ***lp*** (*line printer*), manipula as solicitações de impressão. Na prática, estes usuários trabalham de forma transparente e nada deve ser feito com eles.

2.5.2 Comandos no terminal

Geralmente os comandos no Linux segue a sintaxe do algoritmo 1. Assim além do comando em si, é possível adicionar opções e parâmetros. Os parâmetros podem ser obrigatórios ou não, dependendo do comando, enquanto as opções são opcionais, pois, caso não seja passado opções o comando, será executado com os valores padrões.

\$ comando [opções] [parâmetros]

Algoritmo 1: Sintaxe básica dos comandos no terminal (Próprio autor).

Para ficar mais simples o entendimento será exemplificado um exemplo com o comando ***ls***. Este comando tem a função de exibir conteúdo de arquivos e diretórios. Como parâmetro, o comando recebe o diretório ou arquivo em que se deseja analisar e as opções mais diversas. Sendo alguma delas mostrada na Tabela 4.

Opção	Função
-a	Exibe arquivos ocultos
-A	Não exibe diretórios
-l	Listagem detalhada
-d	Exibe o diretório especificado e não o conteúdo
-i	Exibe o número de índice (<i>l-node</i>) dos arquivos

Tabela 4: Algumas opções do comando ***ls*** (26).

Observe que todos as opções começam com “-“. Este indicador é feito para diferenciar parâmetros de opções. Tudo que começa com “-“ é uma opção e o resto são parâmetros. É possível executar um comando com mais de uma opção, por exemplo, ao escrever ***-al*** será exibido os arquivos ocultos de uma determinada pasta de maneira detalhada.

```
$ ls
ExploringBB
$ ls -l
total 4
drwxr-xr-x 15 root root 4096 Sep  4 18:43 ExploringBB
$ ls -al
total 12
drwxr-xr-x  3 root root 4096 Sep  4 18:42 .
drwxr-xr-x 34 root root 4096 Oct 25 12:08 ..
drwxr-xr-x 15 root root 4096 Sep  4 18:43 ExploringBB
$ ls ExploringBB
chp02 chp05 chp08 chp10 chp12 extras License.txt
chp03 chp06 chp09 chp11 chp13 library README.md
```

Algoritmo 2: Executando o comando *ls* com diferentes opções e parâmetros (Próprio autor).

Comando	Sintaxe	Função
<i>cd</i>	\$ cd [diretório]	Seleciona diretório
<i>more</i>	\$ more [opções] [arquivo]	Lista conteúdo de arquivo aos poucos, útil para arquivos grandes
<i>echo</i>	\$ echo [opções] [texto] \$ echo	Exibe um texto na tela do terminal
<i>apt-get</i>	[opções][operação] [pacote]	Instala, remove e faz outras funções de gerencialmente de arquivos
	\$ shutdown [opções]	
<i>shutdown</i>	[hora] [mensagem de alerta]	Desliga o sistema operacional
<i>clear</i>	\$ clear	Limpa a tela do terminal
<i>reboot</i>	\$ reboot	Reinicia a máquina
<i>cp</i>	\$ cp [arquivo] [pasta]	Copia arquivos
<i>exit</i>	\$ exit	Desloga do usuário atual

Tabela 5: Alguns comandos do terminal (26).

O algoritmo 2 exemplifica o uso do comando *ls* com diferente opções e parâmetros numa pasta qualquer do sistema. Veja que o uso de parâmetro não é obrigatório, neste caso o comando irá listar o conteúdo da pasta atual, entretanto, ao escrever o nome da pasta de interesse é exibido o conteúdo da pasta, como

mostra no final do algoritmo. Além do comando ***ls*** existem muitos outros. A tabela 5 lista alguns dos mais utilizados.

2.5.3 Operadores

É possível utilizar alguns operadores para mudar algumas opções nos comandos, como redirecionar a entrada ou saída. Por exemplo, imagem que por algum motivo deseja-se escrever um determinado arquivo usando o comando ***echo***. Por padrão este comando escreve no console de saída do terminal, mas é possível direcionar esta saída para o arquivo em que se deseja escrever, basta utilizar o operador de redirecionamento de saída, **>**. Se por acaso utilize-se o operador de redirecionamento de saída para o mesmo arquivo, o comando irá apagar e reescrever com a nova mensagem. Para adicionar uma mensagem ao final do arquivo pode-se utilizar o operador **>>**.

Além desses operadores existe um outro para redirecionamento de entrada, o **<**, que pode ser utilizado para mudar a entrada padrão de um determinado comando. Um outro operador bastante útil é o de concatenação de comandos, o **|**, com ele pode-se executar mais de um comando de uma única vez. O algoritmo 3 exemplifica o uso desses operadores.

```
$ ls
Desktop bin
$ echo -e "Estou escrevendo em um arquivo com o comando echo" >
arquivo.txt
$ ls
Desktop arquivo.txt bin
$ more arquivo.txt
Estou escrevendo em um arquivo com o comando echo
$ echo -e "Estou escrevendo de novo" > arquivo.txt
$ more arquivo.txt
Estou escrevendo de novo
$ echo -e "Crie uma nova linha" >> arquivo.txt
$ more arquivo.txt
Crie uma nova linha
$ echo "apague tudo" > arquivo.txt | more arquivo.txt
apague tudo
```

Algoritmo 3: Utilizando alguns operadores para escrever em arquivos com o comando ***echo*** (Próprio autor).

2.5.4 Arquivos no Linux

Ao contrário do senso comum, no Linux, diretórios e aplicativos são apenas um tipo de arquivo. Em outras palavras, tudo Linux são arquivos, incluindo aí diretórios, aplicativos, partições, pinos de entrada e saída do BeagleBone (GPIO), periféricos como PWM, UART e outros.

Para exemplificar o que isso significa, o mostra como acender o *User LED3* (Item 7 na figura 5) através do terminal. Perceba que tudo que foi feito foi acessar a pasta onde está localizado o LED, entrar neste componente e rescrever a opção ***brightness*** para 1 com o comando ***echo***, neste caso é necessário entrar com o pré-fixo ***sudo***, pois o arquivo em questão estava fora da pasta ***home***. O comando ***ls***, neste caso, listou as opções disponíveis para esse LED. É possível alterar as configurações de outros componentes de maneira semelhante.

```
$ cd /sys/class/leds
$ ls
beaglebone:green:usr0 beaglebone:green:usr2
beaglebone:green:usr1 beaglebone:green:usr3
$ cd beaglebone\:\green\:\usr3 | ls
brightness device max_brightness power subsystem trigger uevent
$ sudo su
# echo 1 > brightness
```

Algoritmo 4: Acendendo o *User LED3* pelo terminal (Próprio autor).

2.6 Programando no BeagleBone Black

O Linux permite que o BeagleBone Black seja programado em diversas linguagens, dentre elas, quatro serão abordadas nesse capítulo C++, Python, *JavaScript* e *Bash*.

2.6.1 Linguagens de *script*, interpretadas e compiladas

As linguagens de programação de alto nível podem ser divididas em três categorias. A primeira delas são as linguagens compiladas. Compilar em ciência da computação significa transformar o código diretamente em linguagem de máquina, ou seja, em um arquivo binários contendo zeros e uns. O programa compilado, ao ser executado, passa a conversar diretamente com a máquina, sem intermédios. Isso faz com que este tipo de linguagem seja a mais rápida e

eficiente, porém, pelo mesmo motivo, acaba perdendo em outros quesitos como portabilidade e facilidade de programação.

Um aplicativo compilado feito para Linux, por exemplo, não pode rodar no Windows. O mesmo vale para arquiteturas diferentes. Neste caso para rodar em outras plataformas o mesmo código deve ser adaptado. Um outro fator negativo desse tipo de linguagem é a dificuldade de se programar. A necessidade de gerar um binário que conversa diretamente com a máquina cria algumas limitações no sentido de abstração, assim o programador deve se preocupar com mecânicas como tipo de variável, endereçamentos e alocação de memória. São linguagens compiladas C++, Pascal e Fortran.

O segundo tipo se refere as linguagens interpretadas. Neste grupo o código também é compilado. Mas, ao contrário do primeiro, o código roda em cima de um programa que serve como intermediário entre a máquina e o arquivo. O programa interpretador, neste caso, é uma máquina virtual criada especificamente para rodar aplicativos criados em uma determinada linguagem interpretada. São exemplos de linguagem interpretada o *Java* e o *JavaScript*.

As linguagens interpretadas estão no meio termo quando se refere a performance. A grande vantagem delas é a portabilidade, pois elas são compiladas para rodar em cima de um programa interpretador e se esse programa existir nas mais diversas plataformas o código passa a ser praticamente universal.

O último tipo são as linguagens de *script*. Neste tipo o código é interpretado linha a linha por um outro programa. No momento de execução do *script*, o programa interpretador ler uma linha, executa a tarefa, passa para próxima linha e assim sucessivamente. O lado negativo dessa forma de execução é que, geralmente, tem baixa performance comparado a outros métodos.

A tabela 6 compara a execução de 5000 iterações um algoritmo de performance computacional, *n-Body*. O programa foi escrito em diferentes linguagens de programação e colocado para rodar em BBBs com diferentes frequências de *clock* da CPU. Pela tabela, percebe-se que a linguagem interpretada não tem tanta perda de performance em relação a compilada. O mesmo não ocorre com as linguagens de *script*, o tempo de execução pode chegar a ser 32,2 vezes mais lento que o código compilado em C++. Ainda na mesma tabela, observa-se que a frequência de *clock* é diretamente proporcional

ao tempo de execução, portanto, para aumento de performance, é recomendado configurar para o *clock* máximo.

Linguagem	Tipo	BBB@1Ghz	BBB@800Mhz	BBB@300Mhz
C++	Compilada	1,00x (33s)	1,00x (42s)	1,00x (114s)
Java	Interpretada	1,18x (39s)	1,16x (49s)	1,19x (134s)
JavaScript	Interpretada	2,36x (78s)	2,21x (93s)	2,19x (248s)
Perl	<i>Script</i>	27,8x (917s)	27,8x (1170s)	28,7x (3244s)
Python	<i>Script</i>	32,2x (1063s)	30,8s (1294s)	31,7x (3586s)

Tabela 6: Tempo de processamento para a execução de 5000 iterações do algoritmo *n-Body* em um BBB com Debian, em diferentes frequências (9).

Se por um lado as linguagens de *scripts* são lentas computacionalmente, por outro lado, são conhecidas por alta versatilidade, facilidade de programação e portabilidade. O fato de não ser compilada permite altas abstrações em relação a máquina, e ainda, a execução do programa ao mesmo tempo em que está sendo escrito. Isso permite que os códigos sejam desenvolvidos e testados rapidamente, podendo ainda, rodar em diferentes arquiteturas sem qualquer modificação no código, tornando-a bastante atrativa para desenvolvimento de projetos. São exemplos de linguagens de script: MATLAB, Python, Lua, Bash e Perl.

2.6.2 Linguagens orientadas a objeto

A maioria das linguagens de programação da atualidade são orientadas a objeto, mas não é trivial o ensino desses conceitos de programação nos cursos de engenharia elétrica, por isso, este tópico irá falar brevemente sobre essa nova forma de programar.

Os conceitos de programação orientada a objeto tentam trazer elementos do mundo real para a lógica algoritmos. Os principais conceitos desse novo paradigma são as classes e objetos.

Classe é o local onde se declara todas as características de um objeto, se fizermos a comparação com a biologia, a classe está relacionada com a espécie dos indivíduos. A classe *homo sapiens* deve indicar que todo indivíduo dessa espécie deve ter dois olhos, mãos, pés e assim por diante. Já o objeto é a pessoa que pertence à classe *homo sapiens*.

De uma maneira geral a classe é o escopo onde será definida todas as características e funções de um conjunto de objetos. Os objetos só existem quando são inicializados e estes devem seguir todas as mecânicas descritas nas

classes. Os objetos podem ser chamados de instâncias de uma determinada classe.

As variáveis de uma classe ou objeto são chamadas de atributos e as funções de métodos. Para exemplificar, imagine que exista uma classe que defina os lápis. Os atributos dessa classe são: marca, tamanho, ponta, cor, dentre outros, e os métodos são: escrever, quebrar ponta e assim por diante. Ao criar uma instância, ou objeto, dessa classe, este objeto passa ser um lápis que tem os métodos e atributos definidos na classe.

Métodos especiais de uma determinada classe são os métodos construtores que inicializam assim que um objeto é criado e o métodos destrutores, inicializado assim que o objeto é destruído. Em métodos construtores todos os parâmetros passados a um objeto é parâmetro desses métodos, assim podemos inicializar uma instância de uma classe com determinadas características predefinidas.

Um outro conceito importante é o de herança. Voltando ao exemplo da biologia, podemos dizer que os *homo sapiens* herdam características da classe mamíferos e por isso tem todas as suas características. Neste caso, a classe mãe são os mamíferos e a classe filha são os *homo sapiens*. O algoritmo 29 mostra um exemplo de um código orientado a objeto.

2.6.3 C++

C++ é uma linguagem de médio/alto nível desenvolvida pela Bell Labs durante a década de 80. Ela é uma evolução da linguagem C, desenvolvida pela AT&T na década de 70, sendo que foram incorporados vários conceitos, principalmente aqueles relacionados a programação orientada a objeto.

Diferente de outras linguagens, como Java, C++ não pertence a uma única empresa. Durante anos houve contribuições de várias empresas e, em 1998, a ISO passou a padronizar a linguagem C++ para remover as inconsistências entre os vários compiladores de C++ das diferentes organizações. O padrão ISO deu força a linguagem, e ainda hoje, é uma das mais utilizadas no mundo.

Para compilar um código em C++ pode-se utilizar o comando **`g++`** como mostra no exemplo do algoritmo 5. Neste comando o primeiro parâmetro é o código fonte. Por padrão o nome de arquivo de saída é **`a.out`**, para poder escolher o nome do arquivo de saída, pode-se utilizar a opção **`-o`** seguido do nome do arquivo de saída, neste caso **`olamundo`**.

Para executar o arquivo compilado basta escrever `./<nome_do_arquivo>`. Observe que, diferente do Windows, não é necessário que o arquivo binário tenha a extensão `.exe` para ser executado.

```
$ g++ olamundo.cpp -o olamundo
$ ./olamundo
Ola mundo!
```

Algoritmo 5: Compilando em C++ (Próprio autor).

A linguagem C++ é conhecida por criar programas com alta eficiência, entretanto, ela tem algumas desvantagens. A principal delas está relacionada ao suporte da linguagem no BeagleBone Black. Programar em C++ é uma tarefa um tanto complicada para usuários menos experientes, por isso a própria comunidade propõe o uso de linguagens mais simples e diretas de se programar como JavaScript, e principalmente, o Python. Isso implica que a maior parte do material para leitura, códigos e bibliotecas estão disponível nessas linguagens.

Além disso, por padrão o C++ não suporta interface gráfica, protocolos de comunicação e outras funções relacionadas aos periféricos do BeagleBone Black. Portanto, caso não exista alguma biblioteca desenvolvida para uma determinada função, o caminho se torna muito mais longo. Aliado a isso, existe a necessidade que estas bibliotecas sejam adaptadas para o BeagleBone, visto que o código criado em C++ não é portável, ou seja, boa parte das bibliotecas criadas para desktop não podem ser usada diretamente no BBB. Como a comunidade de C++ é menor é normal que se tenha menos coisas prontas para esta linguagem.

2.6.4 Bash

A linguagem Shell é bastante genérica. Na verdade, a linguagem foi evoluindo e várias pessoas criaram seus próprios programas interpretadores que incluía novos comandos e mecânicas. Isso fez com que existissem várias versões da linguagem Shell, semelhante ao que ocorria com o C++ antes da padronização pela ISO.

Para resolver o problema de compatibilidade o usuário deve digitar qual interpretador será utilizado (Algoritmo 6), caso essa linha não seja escrita o sistema operacional irá escolher o interpretador padrão. No Linux, o interpretador de comandos mais utilizado é o Bash e também o padrão em muitas distribuições, inclusive do Debian, por isso, muitas vezes o programa funciona mesmo com a

ausência da linha do algoritmo 6. Todavia, é recomendado a inclusão dessa linha, pois nem sempre o interpretador padrão é o Bash.

```
#!/bin/bash
LED3_PATH=/sys/class/leds/beaglebone:green:usr3
echo 1 > $LED3_PATH/brightness
```

Algoritmo 6: Código Bash para acender o *User LED3* (Próprio autor).

Depois de escolher o interpretador, a programação se torna parecida com a execução de comandos do terminal, com a vantagem de poder utilizar *ifs*, *fors* e variáveis. Muitas vezes, os comandos do terminal são aplicações criadas em Shell, muitas vezes utilizando o interpretador Bash. Essas aplicações, por sua vez, interagem com os arquivos da mesma forma que poderia ser feito através da linha de comando.

Shell não é uma linguagem muito poderosa e nem tem muitos recursos além daqueles relacionados ao sistema operacional. Por isso, sua utilização se resume a operações relacionadas a manipulação e execução de arquivos (Lembre-se que arquivos é qualquer coisa no Linux).

2.6.5 JavaScript

JavaScript é uma linguagem originalmente desenvolvida para a *web* e, por isso, tem uma comunidade forte e bastante desenvolvida, incluindo grandes empresas como a Google. Embora o JavaScript seja uma linguagem interpretada, ela se assemelha muito com as linguagens de *script*, pois não é necessário compilar a linguagem antes de sua execução, neste caso, a compilação ocorre momentos antes da execução, de forma transparente para o usuário.

JavaScript é uma linguagem de excelente desempenho, pois existe um esforço muito grande das empresas de tecnologia para que ela se torne tão eficiente quanto um código compilado. Pode-se afirmar que o desempenho da linguagem aumenta a cada ano.

No BeagleBone Black esta linguagem é implementada através do *node.js*. Uma plataforma de desenvolvimento de aplicações, cujo o programa interpretador é o *V8 JavaScript Engine*, a mesma componente utilizada pelo navegador Google Chrome para executar JavaScript, desenvolvida pela Google. Com o *node.js* o BeagleBone passa a funcionar como um servidor, ou seja, caso haja uma

conexão com o meio externo, por exemplo entre o computador e o BBB, é a placa que comanda os serviços fornecidos ao cliente, neste caso o computador.

A grande vantagem dessa linguagem em relação ao C++ é que existe um suporte muito forte no BBB. O *node.js* foi colocado no BeagleBone Black propositalmente, pela própria fabricante, inclusive os tutoriais iniciais da BeagleBone.org ensinam a programar nesta linguagem. Aliado a isso, a própria fabricante desenvolveu várias bibliotecas para acesso e configuração dos periféricos.

Por ser uma linguagem interpretada, as bibliotecas e *frameworks* desenvolvidas para JavaScript não precisam ser adaptadas como as C++. Além disso, a própria comunidade está continuamente desenvolvendo soluções para do BBB com esta linguagem.

2.6.6 Python

Python é uma linguagem de script, modular e fortemente orientada objeto, criada em 1991 por Guido van Rossum em 1991. Hoje possui desenvolvimento comunitário e *open source* gerenciado pela organização sem fins lucrativos *Python Software Foundation* (27).

Ela é uma linguagem de fácil leitura e entendimento, pois sua sintaxe obriga o programador a escrever o código de forma clara e concisa. É muito comum os programadores endentar blocos, como *ifs*, *fors* e funções, para facilitar a leitura. O Python usa a própria endentação como delimitador de bloco deixando o código mais legível. Em outras linguagens, a delimitação é feita por caracteres, no caso de C++ usa-se as chaves, { }, para essa função.

Uma outra vantagem do Python é que no Linux pode-se utilizar instruções Shell diretamente no código, em outras palavras, um código em Python é capaz substituir quase por completo um código feito em Bash. Isso torna a linguagem Python muito mais atrativa para o desenvolvimento de tarefas relacionadas ao sistema operacional do que o *node.js*.

Outro recurso do Python é o modo interativo, ou IDLE (*Integrated Development and Learning Environment*). Nesse modo o usuário pode escrever o código e executá-lo em tempo real, de modo é semelhante a janela de comandos do MATLAB e ao terminal do Linux. Para acessar o modo interativo basta escrever no terminal o comando ***python***, já para executar um *script* Python basta escrever o comando ***python <nome_do_arquivo>***.

O Python para Linux possui um gerenciador de bibliotecas semelhante ao gerenciador de pacotes ***apt-get***, o comando ***pip***. Para adicionar uma biblioteca basta escrever ***pip install <nome_da_biblioteca>*** e o próprio sistema se encarregará de baixar e instalá-la. Por padrão já existem muitas bibliotecas pré-instaladas no BeagleBone, incluindo bibliotecas de processamento digital de sinal, interface gráfica, comunicação serial, soquetes UDP e TCP e as bibliotecas para trabalhar com os periféricos do BeagleBone, semelhante àquela disponível para *node.js*. Todos esses conteúdos e facilidades torna o Python a linguagem, para o BBB, com mais recursos prontos e acessíveis.

Os principais tipos de variáveis compostas do Python são: Listas, Tuplas e Dicionários. As Listas são tipos de dados compostos que tem a mesma função dos *arrays* em C++, ou seja, são conjunto de dados em que o endereçamento de cada dado é feito por índices numéricos. Já as Tuplas não têm equivalentes em linguagens tradicionais, mas são, basicamente, listas imutáveis, ou seja, ao declarar a variável, esta não pode ser alterada posteriormente. Em C++, o mais próximo disso são as constantes. Por último, dicionários são listas em que o endereçamento não é feito por índices numéricos, mas por variáveis que podem ser *string* ou numérica. O algoritmo 7 tem-se um exemplo que explica as Listas, Tuplas e Dicionários

```
>>> lista = ['Gato', 'Cachorro', 'Coelho']
>>> tupla = ('Gato', 'Cachorro', 'Coelho')
>>> dionario = {'Neko' : 'Gato', 'Inu' : 'Cachorro', 'Usagi' : 'Coelho'}
>>> lista[1]
'Cachorro'
>>> tupla[0]
'Gato'
>>> lista[0] = tupla[1]
>>> lista[0]
'Cachorro'
>>> tupla[0] = lista[1]
TypeError: 'tuple' object does not support item assignment
>>> dionario['Neko']
'Gato'
```

Algoritmo 7: Exemplo usando uma Tupla, uma Lista e um Dicionário no IDLE (Próprio autor).

2.6.7 Cloud9 IDE

Cloud9 é uma IDE (*Integrated Development Environment*) que roda no navegador através do protocolo HTTP. Ela é uma ferramenta que aceita várias linguagens como C, C++, PHP, Ruby, Perl, Python, Bash e JavaScript. Além disso, permite acessar o terminal pelo próprio ambiente e tem um navegador de arquivos.

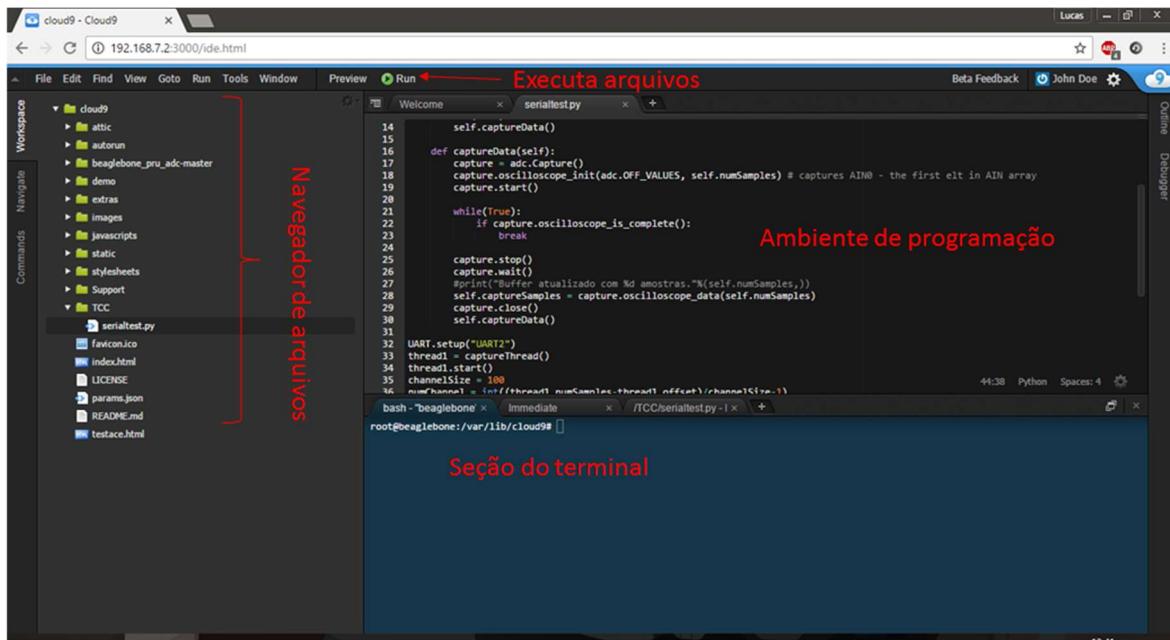


Figura 16: Ambiente de desenvolvimento Cloud9 (Próprio autor).

Essa ferramenta também tem a possibilidade de executar diretamente aplicativos criados em Python, *node.js*, Shell e Ruby através do botão *Run*, de tal forma que, para programar nestas linguagens não é necessário digitar nenhuma linha no terminal, tornando esta ferramenta uma mão na roda para usuários menos experientes e facilitando o trabalho dos mais avançados. O ambiente de desenvolvimento já vem integrado por padrão em todos os BeagleBone Black e para acessá-lo basta digitar o endereço do BeagleBone Black na porta 3000, neste caso *http://192.168.7.2:3000*. Observe que a porta 3000 foi utilizada, pois a porta 80 já está sendo utilizada para a página web *Getting Started* do BeagleBone Black, que pode ser acessada por *http://192.168.7.2*. Não é necessário digitar :80 no final, pois a porta 80 é a padrão para o protocolo HTTP.

2.7 Configurando o BeagleBone Black

Esta seção irá explicar como configurar o BeagleBone Black para adicionar novas funcionalidades e adequar a placa para um determinado propósito.

2.7.1 Adicionando *internet*

Na seção 2.4 foi visto que quando se conecta o BBB ao computador pelo cabo USB é criada uma rede IP entre o PC e a placa. Nessa rede o BBB pode se comunicar com o computador através da pilha de protocolos TCP/IP, entretanto, mesmo que o computador esteja conectado à *internet*, não existe conexão com a rede mundial de computadores no lado do BeagleBone, pois a rede entre a placa e o PC não é a mesma do computador e o roteador, portanto o BBB não enxerga os outros elementos desta outra rede. Contudo, existe uma maneira de compartilhar os dados de uma rede para outra.

No menu iniciar do Windows pesquise por Conexões de rede e clique em “Exibir conexões de rede”. Depois verifique qual é o nome da conexão do BBB, no meu caso é a Ethernet 4. Clique duas vezes na rede em que você deseja compartilhar a *internet*, depois clique em propriedades, uma nova janela se abrirá. Por fim, na aba compartilhamento, selecione “Permitir que outros usuários da rede se conectem pela conexão deste computador à *internet*”, escolha a rede do BeagleBone e pressione OK. Todo esse processo está explicado na figura 17.

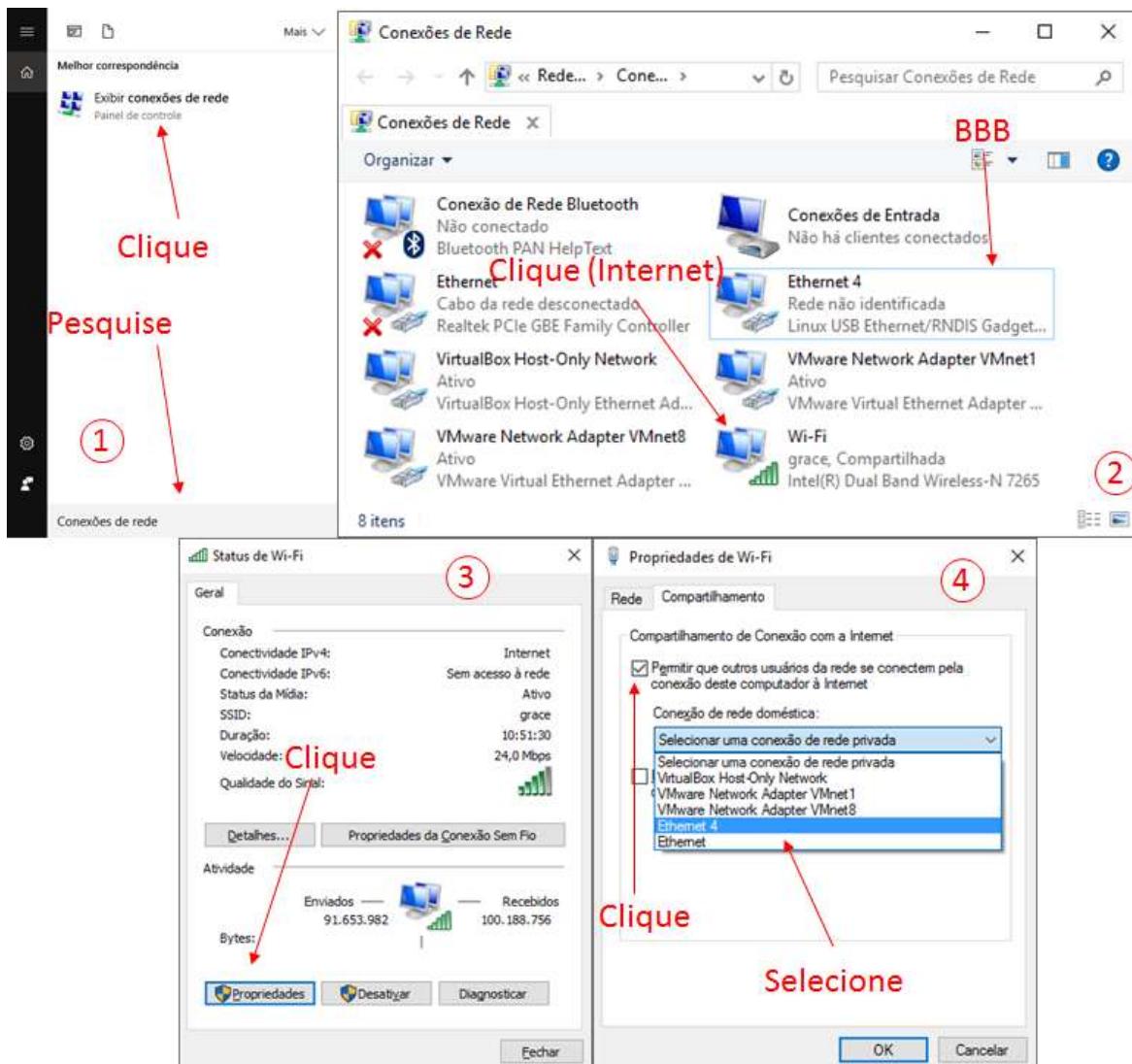


Figura 17: Compartilhando a rede de *internet* com a rede do BBB (Próprio autor).

Ao fazer todas as etapas da figura 17, por algum motivo o Windows irá fornecer um IP diferente do estabelecido para o BeagleBone. Para consertar isso na pasta Conexões de Rede clique na rede do BeagleBone Black (Ethernet 4 no meu caso) e depois clique em Propriedades. Em “Esta conexão usará os seguintes itens”, procure por “Protocolo IP Versão 4”, depois selecione “Obter um endereço de IP automaticamente” e “Obter o endereço dos servidores DNS automaticamente”, assim o BBB se encarregará de definir o seu IP automaticamente. Por fim, clique em OK e feche as janelas. A figura 18 explica todo processo.

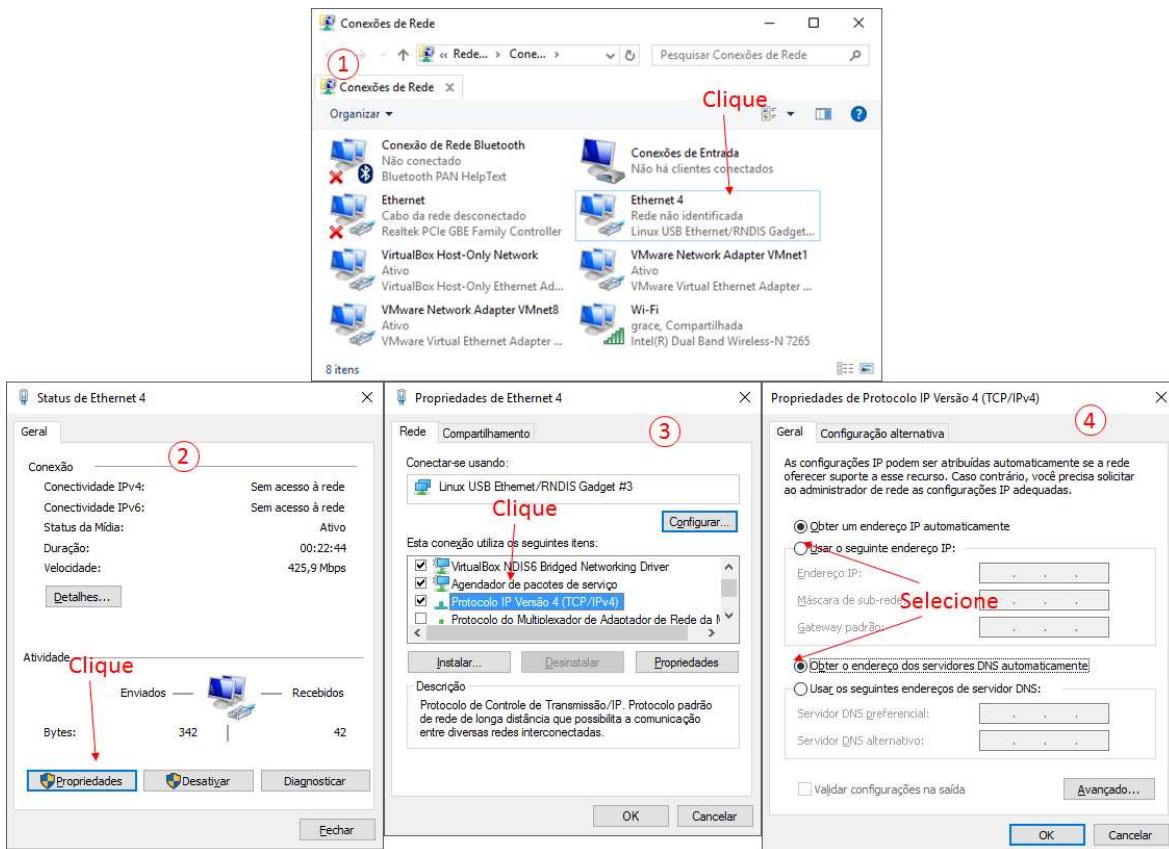


Figura 18: Permitindo que o BBB se conecte através do seu IP padrão (Próprio autor).

Depois disso, entre no terminal do BBB, através do PuTTY e entre como **root** para fazer as alterações fora da pasta **/home**. Assim, adicione o endereço de IP do *gateway* do BBB, que por padrão é 192.168.7.1. *Gateway* é uma máquina intermediária que faz a ligação do dispositivo com a rede, no caso do BeagleBone Black essa máquina é criada via software e seu endereço de IP é o citado anteriormente.

Agora a plaquinha tem acesso à internet, entretanto, ela só consegue acessar a *web* através dos endereços de IP, por exemplo, podemos acessar o site do Google através do seu IP que é 8.8.8.8, mas não conseguimos acessar através do seu endereço web ou domínio, que é, www.google.com. Para isso, a internet criou servidores especiais que armazena todos esses endereços *www*, os DNS (*Domain Name System*). Portanto, devemos nos conectar a esses servidores especiais e eles, de acordo com o domínio, nos redireciona para o endereço de IP equivalente. A própria Google tem um servidor DNS que é o 8.8.8.8.

No fim, testamos a conexão com o comando ping, que basicamente envia pacotes TCP para determinado endereço e retorna a resposta com a quantidade de *bytes* enviados e recebidos.

```
$ sudo su
#/sbin/route add default gw 192.168.7.1
echo nameserver 8.8.8.8 >> /etc/resolv.conf
more /etc/resolv.conf
nameserver 8.8.8.8
# ping www.google.com
PING www.google.com (200.216.56.24) 56(84) bytes of data.
```

Algoritmo 8: Permitindo o BBB ter acesso à *internet* (Próprio autor).

2.7.2 Controlando a frequência do processador

Os processadores atuais não têm uma frequência fixa, isso foi uma estratégia encontrada para economizar energia, aumentar a eficiência e reduzir o consumo. Na prática isso é bastante útil quando estamos falando em dispositivos móveis alimentados por bateria, pois nessa situação queremos que o aparelho funcione o maior tempo possível.

A CPU do BeagleBone é um ARM Cortex A8 com frequência máxima de 1Ghz. Por padrão esse tipo de processador é configurado para economizar energia. Contudo, na nossa aplicação isso pode atrapalhar, pois há perda de desempenho. Para evitar esses imprevistos é possível configurar o processador para uma frequência fixa, que pode ser no máximo 1Ghz, ou mesmo, configurar para que o processador opere com o máximo desempenho possível. Vamos escolher por trabalhar em uma frequência fixa, para evitar oscilações nas funções relacionadas ao tempo, principalmente se estas forem implementadas via *software*.

Como a parte de dissipação de calor foi projetada pensando em um processador com frequência variável não é recomendável fixar a frequência em 1Ghz. Para evitar problemas de sobreaquecimento e degradação prematura do *single board computer* vamos configurar a CPU para trabalhar a uma frequência fixa de 800Mhz através do comando ***cpu-freq-set*** (Algoritmo 9).

```
$ sudo su
# cpufreq-set -g performance
# cpufreq-set -f 800MHz
current CPU frequency is 800 MHz (asserted by call to hardware)
```

Algoritmo 9: Alterando a frequência do processador do BBB (Próprio autor).

2.7.3 Alterando o endereço de IP

Por padrão o BeagleBone Black vem com endereços de IP padrão, que são 192.168.7.2 e 192.168.8.2. De uma certa forma isso é uma praticidade que a fabricante trouxe aos usuários, pois, com isso, a fabricante nos permite acessar a plaquinha assim que esta for conectada a um computador, pela USB. Entretanto, isso pode ser um problema caso seja necessário conectar mais de um BBB no mesmo computador ao mesmo tempo.

Ao fazer isso, pode haver um conflito, pois os IPs não são mais únicos. Assim, como proteção, a rede rejeita os endereços repetidos, ou seja, não é possível acessar a segunda placa a menos que esta tenha um endereço distinto da primeira. Para mudar o endereço de IP padrão do BBB é necessário alterar alguns arquivos do Linux. São eles:

- /etc/network/interfaces
- /opt/scripts/boot/am335x_evm.sh
- /etc/udhcpd.conf

Dessa vez vamos alterar os arquivos de uma forma um pouco diferente, ao invés de utilizar o comando **echo**, vamos modificar os arquivos pelo editor de texto **nano**, que pode ser acessado através do comando de mesmo nome seguido do endereço do arquivo.

```
$ sudo su
# nano /etc/network/interfaces
# nano /opt/scripts/boot/am335x_evm.sh
# nano /etc/udhcpd.conf
```

Algoritmo 10: Modificando arquivos com o *nano* (Próprio autor).

O editor funciona como uma interface gráfica, portanto não tem como mostrar todo processo no algoritmo 10. Porém, é bastante simples, basta procurar nesses arquivos os endereços de IP 192.168.7.2 e 192.168.7.1 e substituí-los para algo como 192.168.8.2 e 192.168.8.1, respectivamente (28). Para salvar e

sair do editor basta apertar *ctrl + X* e selecionar *yes*. A figura 19 mostra como é a interface do nano ao editar o arquivo **/etc/network/interfaces**.

```

# wpa-ssid "essid"
# wpa-psk "password"

# Ethernet/RNDIS gadget (g_ether)
# ... or on host side, ushnet and random hwaddr
# Note on some boards, usb0 is automatically setup with an init script
iface usb0 inet static
    address 192.168.7.2
    netmask 255.255.255.0
    network 192.168.7.0
    gateway 192.168.7.1

```

Altere essas linhas

Figura 19: Editando o arquivo “/etc/network/interfaces” pelo nano (Próprio autor).

2.8 Sistema de boot do BeagleBone Black e a Linux Device Tree

BIOS (*Basic Input/Output System*) é um pequeno binário utilizado para inicializar os componentes principais da máquina para, posteriormente, dar o controle da máquina para o sistema operacional. A BIOS sabe a localização exata do sistema operacional porque geralmente ele fica salvo no primeiro endereço da memória não volátil, geralmente um HD ou SSD, também conhecida como. A UEFI (*Unified Extensible Firmware Interface*) é uma evolução da BIOS, que é um programa muito antigo, datado da década de 80, mas o princípio funcionamento é praticamente o mesmo.

Tanto a BIOS quanto a UEFI foram feitas para trabalhar em um conjunto de hardware padrão, por isso os celulares, palmtops e todos os outros dispositivos diferentes dos PCs tracionais não vêm com esses dois programas por padrão. Ao invés disso eles vêm com um programa conhecido como *bootloader*. Não existe um padrão para *bootloaders*, portanto cada fabricante, comunidade ou organização pode fazer o *bootloader* da melhor maneira que lhes convir. Para se ter uma ideia, até os Arduinos mais simples usam *bootloaders*.

Essa liberdade permite a criação de hardwares mais variados possíveis, mas tem o seu preço. A primeira delas é que não existe um padrão, portanto, algumas vezes pode ser uma tarefa complicada trocar de sistema operacional ou até mesmo fazer customizações no mesmo. O uso de *bootloaders* permitem que fabricantes bloqueiem seus dispositivos para evitar modificações por usuários não autorizados. Talvez um dos casos mais famosos desse tipo de prática são os

bootloaders para telefone Android, pois eles vêm com um *bootloader* bloqueado que impede a instalação de outras distribuições. A solução é trocar o *bootloader* da fabricante por um feito pela comunidade, para assim, instalar *roms* alternativas. Felizmente o *bootloader* do BeagleBone Black não bloqueia a instalação de outras distribuições, mas ainda assim os sistemas operacionais devem ser portados para a pequena placa. Na verdade, devido à falta de um padrão, praticamente todo dispositivo com *bootloader* deve ter o sistema operacional adaptado exclusivamente para aquele aparelho.

2.8.1 *Flattened Device Tree* (FDT)

O crescimento dos dispositivos usando Linux embarcado fez com que houvesse a proliferação de *kernel*s customizados. Essas modificações no código eram feitas em C++ e posteriormente compiladas no núcleo da *mainline* do Linux, aquele famoso processo de recompilar o *kernel*. Linus Torvalds, o criador do Linux, não gostou muito da ideia de inflar o código fonte principal do seu SO com as variações para cada dispositivo embarcado e exigiu uma solução para isso (9). A partir daí a comunidade criou o FTD (*Flattened Device Tree*), que são basicamente estruturas de dados que descrevem determinado componente do sistema operacional.

Os FDTs são linhas de código separadas do *kernel* do Linux e isso facilita muito a adição e remoção de novos hardwares, especialmente útil quando estamos trabalhando com as *capes*, que são nada mais, nada menos módulos de hardware adicionados a plaquinha. Durante o processo de boot os FDTs são os responsáveis por configurar todos os parâmetros do sistema do sistema operacional, incluindo aqueles relacionados aos próprios periféricos do BBB ou periféricos externos, como àqueles de uma determinada *capes*. Além disso, o usuário pode facilmente fazer alterações nos periféricos internos do próprio BBB, dentre eles, desabilitar a saída da HDMI que será explicada futuramente.

Um exemplo de um *Flattened Device Tree* pode ser visto no algoritmo 13. Este algoritmo descreve o modelo de SOC do BeagleBone Black, a memória RAM, os *user LEDs* e os pinos relacionados ao I2C.

```

/* Copyright (C) 2012 Texas Instruments Incorporated ... */
/include/ "am33xx.dtci"
{
model = "TI AM335x BeagleBone";
compatible = "ti,am335x-bone", "ti,am33xx";
cpus {
cpu@0 {
cpu0-supply = <&dcdc2_reg>;
};
};

memory {
device_type = "memory";
reg = <0x80000000 0x10000000>; /* 256 MB */
};

am33xx_pinmux: pinmux@44e10800 {
pinctrl-names = "default";
pinctrl-0 = <&userled_pins>;
userled_pins: pinmux_userled_pins {
pinctrl-single,pins = <
0x54 0x7 /* gpmc_a5.gpio1_21, OUTPUT | MODE7 */
0x58 0x17 /* gpmc_a6.gpio1_22, OUTPUT_PULLUP | MODE7 */
0x5c 0x7 /* gpmc_a7.gpio1_23, OUTPUT | MODE7 */
0x60 0x17 /* gpmc_a8.gpio1_24, OUTPUT_PULLUP | MODE7 */
>;
};

i2c0_pins: pinmux_i2c0_pins {
pinctrl-single,pins = <
0x188 0x70 /* i2c0_sda, SLEWCTRL_SLOW | INPUT_PULLUP | MODE0 */
0x18c 0x70 /* i2c0_scl, SLEWCTRL_SLOW | INPUT_PULLUP | MODE0 */
>;
};

...

```

Algoritmo 11: Um trecho da FTD do BBB (9).

2.8.2 Device Tree Overlay (DTO)

Infelizmente os FDTs não permite que o *kernel* do Linux seja modificado após o processo de *boot*. Em outras palavras, para que uma FDT tenha efeito é necessário reiniciar a máquina. Isso pode ser um pequeno incomodo, pois assim não poderíamos remover adicionar ou trocar de *capes* sem ter que desligar a máquina.

Para resolver esse problema foram criadas as extensões das *Flattened Device Tree*, as DTOs (*Device Tree Overlay*), que são nada mais nada menos que fragmentos de FDTs. Enquanto as FDTs são arquivos enormes que descrevem todo o hardware do BeagleBone Black as DTOs são pequenos e apenas alteram ou complementam determinadas características de uma FTD.

As DTOs têm outra vantagem, permitem que sejam feitas alterações sem a necessidade de reiniciar a placa, pois, como o próprio nome indica, elas foram feitas para sobreescriver (*overlay*) as tradicionais FDTs assim que carregadas. Ao adicionar modificações nos parâmetros do *kernel* é recomendado escrevê-las em uma *Device Tree Overlay*, pois caso isso seja feito em *Flattened Device Tree* e por algum erro de programação quebre o sistema operacional ao ponto de não poder inicializá-lo não há como reverter o problema a menos que seja a reinstalação da distribuição.

```
* OUTPUT GPIO(mode7) 0x07 pulldown, 0x17 pullup, 0x?f no pullup/down */
/* INPUT GPIO(mode7) 0x27 pulldown, 0x37 pullup, 0x?f no pullup/down */
/dts-v1/;
/plugin/;
/{
compatible = "ti,beaglebone", "ti,beaglebone-black";
part-number = "EBB-GPIO-Example";
version = "00A0";
fragment@0 {
target = <&am33xx_pinmux>;
__overlay__ {
ebb_example: EBB_GPIO_Example {
pinctrl-single,pins = <
0x070 0x07 // P9_11 $28 GPIO0_30=30 Output Mode7 pulldown
0x074 0x37 // P9_13 $29 GPIO0_31=31 Input Mode7 pullup
>;
};
};
};
};
fragment@1 {
target = <&ocp>;
__overlay__ {
gpio_helper {
compatible = "gpio-of-helper";
status = "okay";
pinctrl-names = "default";
pinctrl-0 = <&ebb_example>;
};
};
};
};
};
```

Algoritmo 12: Exemplo de DTO retirada de (9).

A estrutura de uma DTO se inicia com um cabeçalho que deve ter o hardware de compatibilidade (*compatible*), o nome de identificação (*part-number*) e a versão do arquivo (*version*). As linhas de código são escritas dentro dos blocos *fragment*. Observe que no algoritmo 12 tivemos dois blocos desse tipo,

fragment@0 e *fragment@1*, ou seja, houve duas alterações na descrição dos componentes do BeagleBone.

Para compilar um código em *Device Tree Overlay*, cujo o formato é **.dts** (Device Tree Source) podemos usar o comando **dtc**. O arquivo de saída é um **.dtbo** (Device Tree Blob Object) Por fim esse arquivo compilado deve ser copiado ou movido para a pasta **/lib/firmware** (Algoritmo 13), onde são salvas todas as **.dtbo**.

```
$ dtc -O dtb -o teste.dtbo -b 0 -@ teste.dts
```

```
$ sudo cp teste.dtbo /lib/firmware
```

Algoritmo 13: Compilando uma DTO e salvando na pasta do capes manager (Próprio autor).

2.8.3 Modos de Pinos do BeagleBone Black

Pino	Mode0	Mode1	Mode2	Mode3
1,2	-	-	-	GND
3,4	-	-	-	DC_3.3V
5,6	-	-	-	VDD_5V
7,8	-	-	-	SYS_5V
9	-	-	-	PWR_BUT
10	RESET_OUT	-	-	-
11	gpmc_wait0	mii2_crs	gpmc_csn4	-
12	gpmc_be1	mii2_col	gpmc_csn6	-
13	gpmc_wpn	mii2_txd3	gpmc_csn5	-
14	gpmc_a2	gmii2_txen	rgmii2_td3	-
15	gpmc_a0	mmc2_sdwp	I2C1_SCL	-
16	gpmc_a3	timer5	I2C1_SDA	-
17	spi0_cs0	timer6	dcan1_rx	-
18	Spi0_d1	uart2_txd	I2C2_SCL	-
19	uart1_rtsn	uart2_rxd	I2C2_SDA	-
20	uart1_ctsn	timer6	ehpr	
33	-	-	-	ANI4
42	Mcasp0_aclk	eQEP0A_in	Mcasp0_ax2	Mcasp0_axr2

Tabela 7: Tabela de modos dos pinos de extensão do BBB – Parte 1 (29).

Embora o BeagleBone Black tenha vários pinos externos, muitos deles têm funções compartilhadas e não podem ser utilizadas simultaneamente. Para

identificar a função de determinado pino o sistema operacional divide cada um deles em 8 modos de funcionamento (*mode0* a *mode7*). Nas tabelas 7 e 8 têm a função de cada modo para alguns pinos do barramento de extensão P9. Observando a tabela dá para perceber que nem todas as funções do BeagleBone estão disponíveis simultaneamente. Por exemplo, se quisermos usar o I²C2 devemos desativar a UART2, neste caso, os pinos 18 e 19 devem ser configurados para trabalhar no modo 2, caso queiramos voltar a utilizar o I²C configuraremos novamente esses dois pinos no modo 1, e se ainda quisermos utilizar esses pinos como pinos comuns de entrada e saída podemos configurá-los no modo 7.

Na prática podemos utilizar as DTOs para estas configurações de cada pino ou componente de *hardware* do BeagleBone Black. Algoritmo 12, por exemplo, configura os pinos P9_11 e P9_13 como pinos de entrada e saída, colocando-os no modo 7. Muitas outras configurações como habilitar as entradas UART, SPI, o suporte HDMI são basicamente alterações nos modos de pinos.

Pino	<i>Mode4</i>	<i>Mode5</i>	<i>Mode6</i>	<i>Mode7</i>
1 a 10	-	-	-	-
11	mmc1_sdcd	-	uart4_rxd_mux2	gpio0[30]
12	gpmc_dir	-	mcasp0_aclk_mux3	gpio1[28]
13	mmc2_sdcd	-	uart4_txd_mux2	gpio0[31]
14	gpmc_a18	-	ehrpwm1A_mux1	gpio1[18]
15	gpmc_a16	-	ehrpwm1_tripzone_input	gpio1[16]
16	gpmc_a19	-	-	gpio1[19]
17	-	-	-	gpio0[5]
18	-	-	-	gpio0[4]
19	spi_cs1	-	-	gpio0[13]
33	-	-	-	-
42	spi_sclk	mmc0_sdwp	xdma_event_intr2	gpio3[18]

Tabela 8: Tabela de modos dos pinos de extensão do BBB – Parte 2 (29).

2.8.4 Capemgr

As DTOs foram criadas para funcionar em conjunto com as *capes*, isso significa que cada uma dessas placas de extensão tem em *Device Tree Overlay* associada a mesma. De uma certa forma isso facilita bastante a vida do usuário,

pois ao conectar uma determinada *cape*, o sistema operacional identifica a DTO correspondente, a compila e transfere para pasta **/lib/firmware**, aquele mesmo processo que foi feito manualmente no algoritmo 13.

Todo esse processo é feito por uma pequena parte do *kernel* do BBB conhecida como Capemgr (Cape Manager). Entretanto essa parte do núcleo não faz tudo sozinha, toda *cape* devem ter uma memória EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) conectada ao I²C2 (Pinos 19 e 20 das tabelas 7 e 8), que contém a informações sobre a mesma, incluindo aí, seu nome e o arquivo **.dtbo** associado.

De uma certa forma essa metodologia facilita a conexão dessas placas de extensão, visto que boa parte delas são praticamente *plug-in-play*. Ou seja, o usuário conecta a capes nos pinos de extensão e a Capemgr se carrega de ler a DTO associada a placa de extensão e modificar as configurações do *kernel* para o funcionamento da capes. Ao remover a placa as alterações da *Device Tree Overlay* são removidas e passa a vigorar as configurações da *Flattened Device Tree*, ou seja, os padrões do sistema operacional.

Se por um lado isso facilita para as pessoas que compram as *capes* prontas e conectam ao BeagleBone, por outro aumenta a complexidade e o custo da capes para as pessoas que querem desenvolver uma dessas placas de extensão, visto que deve ser acrescido ao projeto de hardware uma memória EEPROM com conexão I²C, além de todo o conhecimento necessário para criar DTOs.

2.8.5 Slots

Nem todas as DTOs são feitas para funcionar em conjunto com as capes. O próprio usuário, por exemplo, pode necessitar fazer modificações nos parâmetros do *kernel* em sua aplicação utilizando DTOs. Nesses casos o usuário deve escrever o código, fazer manualmente o processo de compilação e a transferência para pasta **/lib/firmware/** deve ser de forma manual (Algoritmo 13).

Entretanto só isso não basta. A Capemgr não carrega todos os DTOs da pasta **/lib/firmware/** simultaneamente, até porque se isso acontecesse além de sobrecarregar o *kernel* com centenas de configurações distintas poderia haver *Device Tree Overlays* conflitantes, causando sérios problemas ao sistema operacional. Por isso, na *Cape Manager* existe o conceito de *slot*. Assim as informações contidas na EEPROM são transferidas o *slot* disponível.

Os *slots* de 0 a 3 são reservados para as *capes*, enquanto os *slots* com números maiores podem ser utilizados por *Device Tree Overlays* (Capes virtuais). Neste último caso o usuário deve adicionar manualmente o *part-number* ao arquivo `/sys/devices/bone_capemgr.9`. Por sorte, a comunidade já criou muitas DTOs para vários tipos de aplicação e algumas delas já estão incluídas na pasta `/lib/firmware/`, para ver todas as disponíveis basta escrever o comando `ls` nessa pasta. O algoritmo 13 mostra um exemplo de como adicionar o suporte a UART1 e a UART2 através de umas das DTOs disponíveis em `/lib/firmware`. Observe que ao escrever o comando `more` no arquivo `slots` as três primeiras linhas estão vazias, isso ocorre porque não há nenhuma cape conectada ao BeagleBone Black. Os *slots* 4 e 5 são *Devices Tree Overlay* relacionadas a HDMI e o eMMC e já vem ativada por padrão. Ao adicionar as DTOs cujo o part-number são BB-UART1 e BB-UART2 são ativadas a UART1 e a UART2. Isso pode ser verificado usando o comando `more` novamente no arquivo `slot`.

```
$ sudo su
# more /sys/devices/bone_capemgr.9/slots
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
# echo BB-UART2 > /sys/devices/bone_capemgr.9/slots
# echo BB-UART1 > /sys/devices/bone_capemgr.9/slots
# more /sys/devices/bone_capemgr.9/slots
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
7: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-UART2
8: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-UART1
```

Algoritmo 14: Adicionando as DTOs UART1 e UART2 (Próprio autor).

2.8.6 uEnv.txt

No algoritmo 14 vimos como carregar uma DTO manualmente e assim alterar as configurações do *kernel* instantaneamente, sem a necessidade de recopilá-lo. Contudo ao desligar e/ou reiniciar o *board computer* as configurações feitas no arquivo slots são apagadas, pois antes de inicializar o sistema operacional o *bootloader* do BBB ler as configurações do arquivo uEnv.txt e reescreve o arquivo slot e todos outros parâmetros do núcleo do Linux (30).

O arquivo uEnv.txt está disponível na partição FAT montada no PC ao conectar o BeagleBone Black, por isso pode ser facilmente acessado pelo computador e editado com um editor como o bloco de notas. Nesse arquivo colocamos todas as DTOs que devem ser carregadas durante o processo de *boot*, assim como os parâmetros do *kernel*. Assim, podemos ativar permanentemente as DTOs adicionadas algoritmo 14 adicionando uma linha o final do arquivo Env.txt. Esse processo pode ser feito também pelo BeagleBone, neste caso o arquivo está localizado na pasta **/boot/uboot/** (Algoritmo 15).

```
# echo cape_enable=capemgr.enable_partno= BB-UART2, BB-UART1 >>
/boot/uboot/uEnv.txt
```

Algoritmo 15: Adicionando suporte à UART1 e UART2 no BBB (Próprio autor).

A porta HDMI consome cerca de 20 pinos de expansão do BeagleBone Black, além disso, ela utiliza os pinos da UART3 e UART5, como futuramente iremos trabalhar com as UART é interessante desativá-la. Acesse o arquivo uEnv.txt através do nano, procure a linha **#optargs=quiet capemgr.disable_partno=BB-BONELT-HDMI,BB-BONELT-HDMIN** e remova o # para descomentá-la.

2.9 BeagleBone Black e tarefas em tempo real

O Linux é um sistema operacional que não lida muito bem com operações críticas relacionadas ao tempo. Seu núcleo foi projetado para executar funções em tempo real mais leves, como tocar arquivos de áudio e vídeo. Contudo, em tarefas onde existe a necessidade de precisões na faixa de microssegundos ou menor não há garantias que todas as instruções serão executadas em tempo hábil, comprometendo-as. Um grande exemplo desse tipo de aplicação, são os próprios dispositivos de aquisição de dados, quando trabalham em frequências mais elevadas e em maior quantidade de canais.

Essa limitação vem da maneira de como o Linux lhe dá com os múltiplos processos do sistema operacional. O Linux é um SO (Sistema operacional) não *preemptive*. *Preemption*, em ciência da computação, é o ato de interromper determinadas tarefas para executar outras com mais prioridade, ou seja, àquele mesmo conceito de interrupções dos microcontroladores. Na prática, o ARM Cortex A8 tem suporte a interrupções, mas quando estamos programando no BeagleBone Black, estamos criando uma aplicação para Linux, e por isso, estamos sujeitos ao gerenciador de processos do sistema operacional. Como o Linux é não *preemptive*, não há como ele interromper determinado processo para executar outro com mais prioridade. Por exemplo, ao criar uma rotina para fazer uma leitura do ADC e configurá-la para ré executa-la cada $10\mu s$, através de um laço de repetição. O SO irá executar pela primeira vez a rotina, depois executar as rotinas dos outros processos, para finalmente voltar a rotina de leitura do ADC. Pode acontecer que o tempo para rodar uma ou mais rotina dos outros processos seja mais que $10\mu s$, assim não havendo garantias que a execução será feita em tempo rápido.

Em processos, mais lentos, geralmente, isso não é tão danoso, ou ainda, passa de maneira imperceptível pelo usuário. Na verdade, o processador é bem rápido e consegue lhe dar com os múltiplos processos de maneira bem eficiente, por isso, para aplicações relacionada a computação não há muito problemas com um sistema operacional não *preemptive*, pois as pequenas oscilações entre o tempo de execução das rotinas não importa muito.

2.9.1 Kernels em tempo real

Segundo (9) existem algumas soluções para contornar este problema. A primeira delas está em escrever o código diretamente para a CPU do BeagleBone, esta, por sua vez é uma tarefa bem complicada e muito pouco utilizada na prática, pois exige criar uma aplicação que está acima do sistema operacional, e ainda, envolve customizar o próprio núcleo do SO.

Por sorte, existem que já alteraram as linhas de código do *kernel* do Linux para lhe dar com tarefas em tempo real. Essa modificação consiste em adicionar um núcleo secundário com maior prioridade que o do SO, o Kobalt. Este núcleo permite a execução de rotinas em tempos pré-definidos. Basicamente o Kobalt, executa o Linux normalmente, mas assim que uma *flag* de interrupção é ativada, ele para todos os processos, executa a tarefa de prioridade e depois volta as

tarefas antigas da onde parou. Uma das distribuições que já vem com esse *kernel* customizado é o Machinekit, que é um Debian modificado. Essa distribuição foi feita, inicialmente, para CNCs e impressoras 3D, mas podem ser utilizadas tranquilamente em outras distribuições.

Uma outra possibilidade seria partir diretamente para sistemas operacionais *preemptive* não baseados em Linux. Existem algumas empresas que fornecem algumas soluções proprietárias desses tipos de SOs. Dentre elas tem o QNX Neutrino, bastante utilizado na eletrônica carros, instrumentos digitais e *home systems*; e o StarterWare para processadores ARM TI Sitara, uma plataforma de programação baseada em C que permite uma camada de abstração para os protocolos de comunicação e periféricos do BeagleBone Black (UART, SPI, I²C, LDC, Ethernet), semelhante ao Arduino. O lado negativo dessas últimas opções é que são pagas, proprietárias e não tem muitos aplicativos e tecnologias criadas para as mesmas como no Linux.

2.9.2 Beaglebone Black e microcontroladores

Outra alternativa é utilizar máquinas preparadas para soluções em tempo real em conjunto com o BeagleBone. Um exemplo desse tipo de aplicação é a utilização de microcontroladores em conjunto com o BeagleBone Black. A comunicação entre o BBB e os microcontroladores pode ser feita de diversas maneiras. Dentre elas, SPI e UART são as mais recomendadas, pois permite a troca de informação mais rapidamente.

Nessa topologia os microcontroladores seriam responsáveis por fazer a leitura do ADC, salvar os dados em um buffer e enviar para o BeagleBone através dos protocolos de comunicação citados nos parágrafos anteriores. O lado negativo dessa solução é o tamanho da solução final e o custo adicional da aquisição dos microcontroladores.

2.9.3 PRU-ICSS

O BeagleBone Black já vem com uma solução de hardware para tarefas em tempo real, o coprocessador PRU-ICSS (*Programable Real-Time Unit and Industrial Communication Subsystem*). Esse componente é uma solução proprietária da Texas Instruments que faz parte do SOC AM335x.

O fato de ser implementado no mesmo encapsulamento da CPU permite uma troca de informação entre ambas as partes muito mais rápido do que a

solução externa, como aquela com microcontroladores e o BBB. Além disso, o coprocessador tem acesso direto à alguns pinos de extensão das portas P8 e P9. Por ser um assunto mais extenso falaremos dele com mais detalhes na próxima seção.

2.10 O coprocessador PRU-ICSS

O coprocessador do BeagleBone Black é programado em linguagem de máquina, por isso, é muito importante conhecer a arquitetura deste componente. Nesta seção falaremos da arquitetura e do processo de criação e gravação de programas para o PRU-ICSS.

2.10.1 Arquitetura do PRU-ICSS

O PRU-ICSS é a segunda versão do PRUSS (*Programmable Real-time Unit Sub System*), ou PRUSSv2. Dentro do PRU-ICSS existem dois núcleos RISC (*Reduced Instruction Set Computer*) de 32 bits com *clock* de 200 Mhz, os PRU0 e PRU1. Cada núcleo tem uma memória interna de 8 KB que utilizada para armazenar os programas criados para as mesmas (*program memory*) e mais 8 KB para armazenar dados quaisquer (*data RAM memory*), em resumo, as *program memory* armazenam as instruções que serão executadas por cada PRU e as *data RAM memory* tipicamente são usadas para armazenar os valores ou *arrays* que serão manipulados pelas instruções da *program memory*. Essas memórias exclusivas de cada PRU permitem que cada núcleo seja programado separadamente e, portanto, utilizados para fazer tarefas distintas simultaneamente. Por isso, em teoria, o BBB dispõe de dois microcontroladores auxiliares integrados ao SOC.

Além das memórias exclusivas para cada núcleo, cada PRU pode acessar os dados do outro núcleo através de uma memória separada, compartilhada e de uso geral com capacidade de 12 KB. Em termo de velocidade de acesso, escrita e leitura não há diferenças entre a memória compartilhada e as RAMs de cada PRU, segundo a documentação da AM335x (31), considerando os melhores casos, as latências de leitura de ambos tipos de memória são de 3 ciclos de *clock*. Na prática, esse valor é um pouco maior, pois todas elas compartilham através do SCR (*Switched Central Resource*), um barramento de 32 bits que interconecta as PRUs com todos os outros componentes do PRUSSv2. Assim usar uma memória ou outra vai depender da aplicação, em casos em que os dois núcleos trabalham

em conjunto para manipular os mesmos dados é interessante salvá-los na memória compartilhada, já em casos em que cada PRU trabalha de forma independente para finalidade distintas pode-se utilizar a sua RAM exclusiva, mas nada impede que essa regra seja obrigatória.

A arquitetura de cada unidade de processamento de uma PRU é muito simples, pois não existe *pipeline* (32). A ideia de utilizar *pipeline* é a mesma das linhas de montagem industriais, as instruções que demoram mais tempo para serem executadas podem ser divididas em instruções menores. Portanto, temos instruções mais homogêneas em relação ao tempo de execução. Como a PRU não tem esse recurso, com exceção das instruções de acesso a memória, todas as instruções utilizam um ciclo de *clock* (5 ns), mesmo aquelas menores, que poderiam ser executadas em um menor tempo. Se por um lado deixa o processo como um todo mais lento, por outro, fica muito mais fácil determinar o tempo de execução de uma determinada rotina. A figura 20 mostra a diferença entre a PRU e um processador hipotético com *pipeline*.

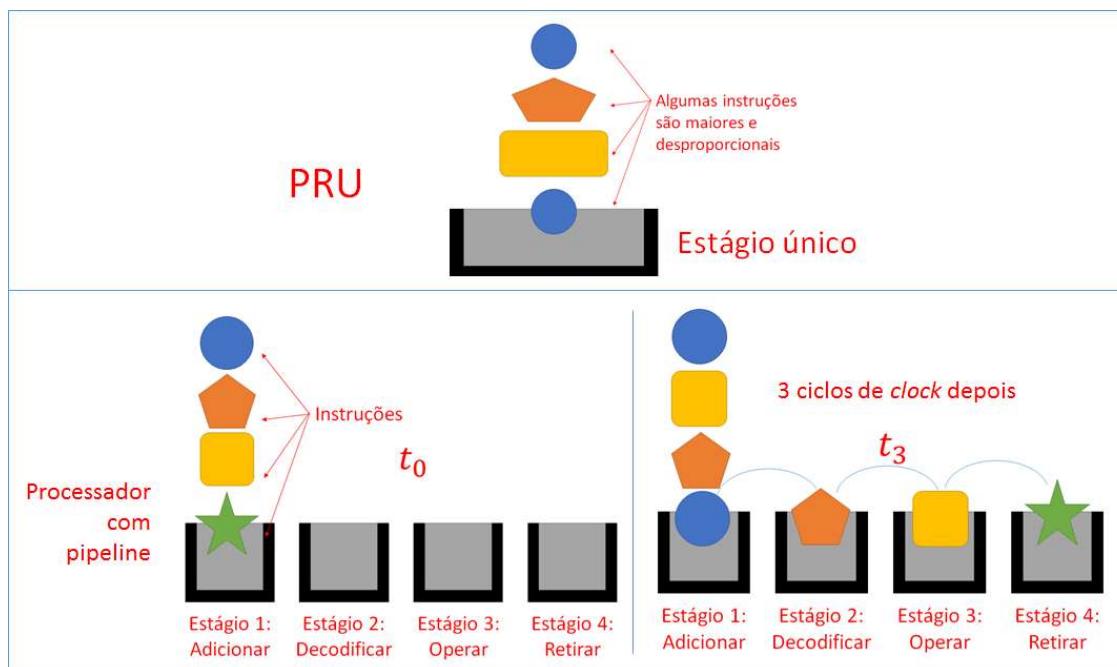


Figura 20: Comparaçao entre o PRU e um processador hipotético com *pipeline* (Próprio autor).

Um outro fator que diminui o IPC (*Instructions per cycle*) é a ausência de memória *cache*, uma memória de acesso rápido interna ao processador. Como o PRU não tem esse recurso ele depende do barramento SCR toda vez que for buscar dados na memória. Por isso, embora o PRU tenha uma frequência de *clock* elevada comparado a outros microcontroladores, seu IPC é bem abaixo em

relação a outros do gênero. Entretanto, isso torna-o muito barato de se produzir, visto que a própria Texas Instruments detém a tecnologia e, devido a sua simplicidade, ocupa pouquíssimo espaço no *wafer* de silício, tornando o BeagleBone Black relativamente barato.

Outro componente importante é o *Scratch pad* (SPAD) que são 3 bancos, cada um com 30 registradores de 32 bits. Esses bancos podem ser acessados por ambos PRUs em apenas um ciclo de *clock*. E ainda, para cada núcleo existe um *interrupt controller* (INTC), que serve para notificar cada núcleo que ocorreu um evento de interrupção e um multiplicador com acumulador opcional, capaz de multiplicar 2 operandos de 32 bits e retornar um resultado de 64 bits. Por fim, existe um módulo UART dedicado de 192 Mhz que pode ser acessado pelos pinos de extensão, desde que seja configurada uma DTO para isto. Um esquemático da arquitetura do PRUSS pode ser visto na figura 21.

Além dos registradores externos (SPAD) dentro de cada núcleo PRU existem 32 registradores, sendo que, dos registradores 1 ao 29 são de propósito geral, o registrador 0 pode ser usado como propósito geral ou como índice (*index*) e os registradores 30 e 31 são de propósito específico. Nota-se que 30 registradores de propósito geral é algo impressionante em um microcontrolador, lembre-se que cada registrador pode ser utilizado e reutilizado várias vezes.

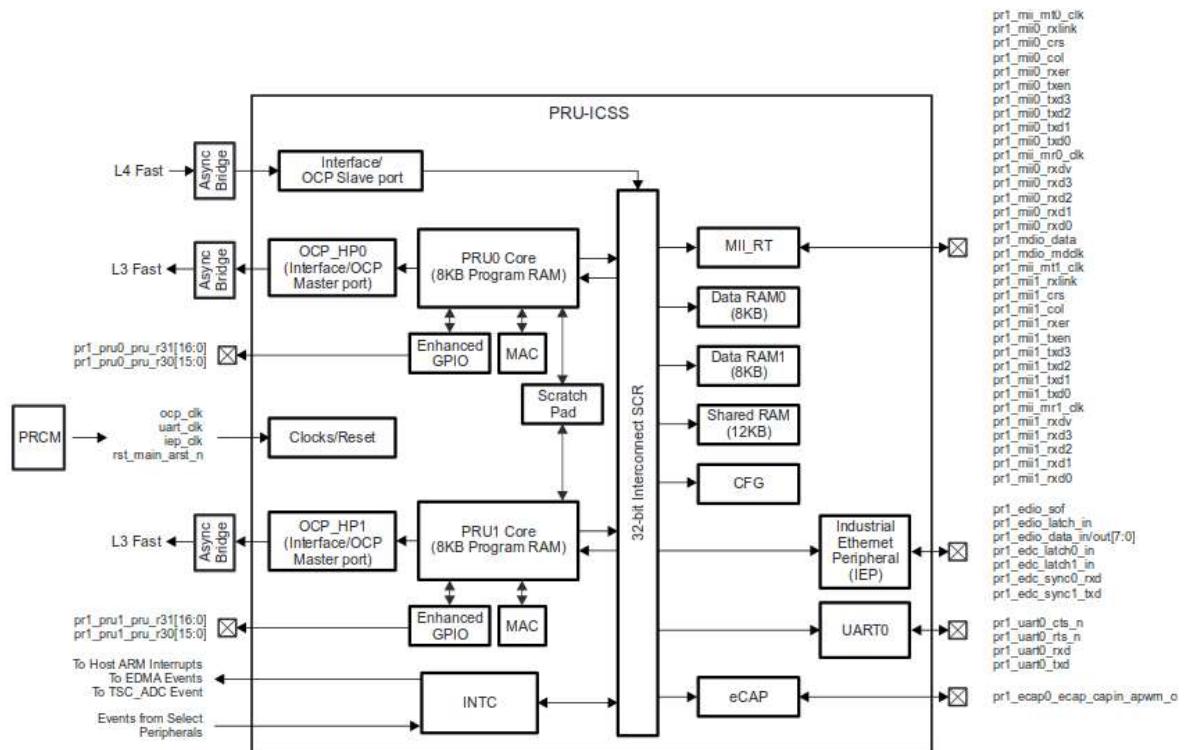


Figura 21: Arquitetura do coprocessador PRU (33).

2.10.2 PRU-ICSS e a comunicação com mundo externo

Além dos blocos e componentes citados na seção anterior, o coprocessador do BeagleBone Black tem acesso direto a determinadas portas que podem ser acessadas através dos pinos de extensão P8 e P9, desde que estes sejam configurados no modo 5 ou 6 usando *device tree overlays* (9). Mais detalhes podem ser encontrados em (34) e (35).

P9			P8		
DGND	1	2	DGND		
VDD_3V3	3	4	VDD_3V3		
VDD_5V	5	6	VDD_5V		
SYS_5V	7	8	SYS_5V		
PWR_BUT	9	10	SYS_RESETN		
GPIO_30	11	12	GPIO_60		
GPIO_31	13	14	GPIO_50		
GPIO_48	15	16	GPIO_51		
GPIO_5	17	18	GPIO_4		
I2C2_SCL	19	20	I2C2_SDA		
GPIO_3	21	22	GPIO_2		
GPIO_49	23	24	GPIO_15		
PRUO_7	25	26	PRU1_16 IN		
PRUO_5	27	28	PRUO_3		
PRUO_1	29	30	PRUO_2		
PRUO_0	31	32	VDD_ADC		
AIN4	33	34	GNDA_ADC		
AIN6	35	36	AIN5		
AIN2	37	38	AIN3		
AIN0	39	40	AIN1		
PRUO_6	41	42	PRUO_4		
DGND	43	44	DGND		
DGND	45	46	DGND		

DGND	1	2	DGND		
GPIO_38	3	4	GPIO_39		
GPIO_34	5	6	GPIO_35		
GPIO_66	7	8	GPIO_67		
GPIO_69	9	10	GPIO_68		
PRUO_15 OUT	11	12	PRUO_14 OUT		
GPIO_23	13	14	GPIO_26		
GPIO_47	15	16	GPIO_46		
GPIO_27	17	18	GPIO_65		
GPIO_22	19	20	PRU1_13		
PRU1_12	21	22	GPIO_37		
GPIO_36	23	24	GPIO_33		
GPIO_32	25	26	GPIO_61		
PRU1_8	27	28	PRU1_10		
PRU1_9	29	30	PRU1_11		
GPIO_10	31	32	GPIO_11		
GPIO_9	33	34	GPIO_81		
GPIO_8	35	36	GPIO_80		
GPIO_78	37	38	GPIO_79		
PRU1_6	39	40	PRU1_7		
PRU1_4	41	42	PRU1_5		
PRU1_2	43	44	PRU1_3		
PRU1_0	45	46	PRU1_1		

Figura 22: Pinos de entrada e saída que podem ser acessados diretamente pelos PRUs (35).

A figura 22 (Em branco) mostra quais são as saídas que estão disponíveis para acesso direto, no qual, cada um deles estão associados diretamente aos registradores dos PRUs. Observe que aqueles que começam com PRU0 são I/O exclusivas do primeiro núcleo e os que começam com PRU1 exclusivas do segundo núcleo. Alguns pinos só podem serem utilizados como entrada ou saída, estes são acompanhados com as palavras IN ou OUT, respectivamente.

O acesso aos componentes do SOC é feito através interface OCP (*Open Core Protocol*), um protocolo de comunicação mestre/escravo¹⁰ ponto a ponto comumente utilizado em subsistemas *on chip* (36). Cada PRU tem acesso direto a uma porta mestre, OCP_HP0 e OCP_HP1, respectivamente. Que por sua vez, são conectados ao barramento L3 do Sitara AM3358A (Figuras 4 e 21). Existe uma porta escrava conectada ao barramento SCR, para o caso algum componente necessite fazer solicitações de comunicação com o PRUSS.

¹⁰ Em um sistema mestre/escravo somente a estação chamada de mestre detém o direito de comunicação e as estações escravas seguem as regras e pedidos da estação mestre (76).

Assim, o acesso ao ambiente Linux, incluindo a gravação de programas em cada PRU e a troca de informações entre ambas as partes é feito por esse protocolo de comunicação. E ainda, o acesso aos outros periféricos do BBB deve ser feito, diretamente ou indiretamente, através do OCP. Por exemplo, o acesso aos pinos não exclusivos do BBB só pode ser feito através do OCP_HP0 ou OCP_HP1, o mesmo é válido para outros subsistemas do SOC, incluindo o próprio ADC do BeagleBone Black. Assim a latência de comunicação entre os periféricos e o PRU-ICSS é maior que a do mesmo e seus pinos dedicados (Figura 22), mas ainda assim é rápida o suficiente para a nossa aplicação.

2.10.3 Memória local e memória global

Para facilitar o acesso entre a troca de informações entre o PRUSSv2 e o ambiente Linux, toda memória local de uso geral desse subsistema é mapeada no Debian em um endereço global. Em outras palavras, existe uma correspondência entre os endereços de memória do PRU-ICSS (Memória local) e os endereços de memória do Linux (Memória global). Isso permite que os aplicativos Linux possam acessar o conteúdo do PRU.

Por exemplo, cada PRU acessa o endereço de memória através do endereço 0x0000 0000. Para um núcleo acessar o endereço do outro núcleo ele deve utilizar o endereço 0x0000 2000. Para acessar o endereço da memória compartilhada utiliza-se o endereço 0x0001 0000 e assim por diante. Essa é a memória local na qual o PRU tem acesso (Figura 23).

No ambiente Linux existe um *driver uio_pruss* aloca uma parte da memória RAM para o PRUSS. A posição inicial do endereço de memória depende do gerenciador de memória do sistema operacional, observe que na figura 23 há um offset de 0x4a30 0000, mas se o driver for recarregado é muito provável que ele inicie em um endereço diferente, semelhante a um processo qualquer. A partir desses endereços que os aplicativos para Linux têm acesso as memórias e registradores do PRUSS.

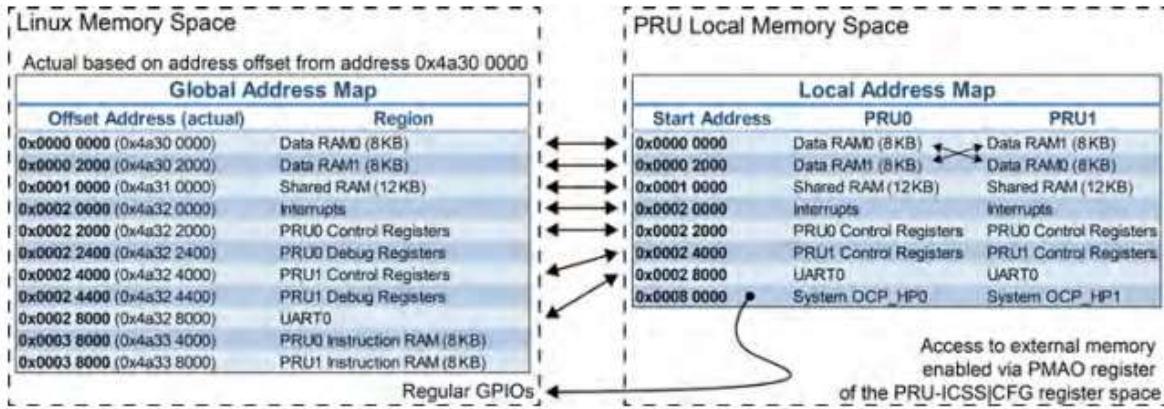


Figura 23: Correspondência entre a memória local do PRU e a memória do Linux (9).

É válido lembrar que para haver a correspondência o driver deve ser ativado. E isso pode ser feito através de um fragmento de *Device Tree Overlay*, como mostra no exemplo do algoritmo 16. Ao carregar um DTO com este fragmento, o *uio_pru* aloca uma parte da SRAM (Por padrão 16kb) e uma parte da DRAM (Por padrão 256kb), além de permitir a comunicação entre ambos subsistemas.

```
...
fragment@1 {           // Ativa o PPRUSS
    target = <&pruss>;
    __overlay__ {
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&pru_pru_pins>;
    };
}
...
```

Algoritmo 16:Parte do código da DTO utilizada por (37) para a sua aplicação de captura de dados de um ADC externo em tempo real.

Observe que a quantidade de DRAM alocada é muito maior do que a necessária para mapear todo o PRUSS, que não passa de 64kb. O PRU pode acessar toda a memória global alocada, mas a latência é um pouco maior, pois utiliza o OCP. Os primeiros endereços dessa memória alocada são apenas cópia da memória do PRU, mas os endereços subsequentes estão livres para ser utilizados para outras aplicações. Existe a possibilidade de alocar mais de 256kb da DRAM, mas para isso, o *driver* deve ser recarregado utilizando como parâmetro a quantidade de memória especificada.

É possível ativar, desativar e passar parâmetros dos módulos através do comando ***modprobe***. Além disso, o comando ***lsmod*** lista quais são os *drivers* em funcionamento e o ***rmod*** remove determinado módulo. Como falamos, ao carregar uma DTO com o fragmento do algoritmo 16 o módulo ***uio_pruss*** é carregado automaticamente, então para recarregá-lo é necessário executar o comando ***rmod*** para posteriormente carrega-lo com o ***modprobe***, especificando a quantidade de memória utilizada. Para verificar se a memória foi alocada pode-se verificar o conteúdo do arquivo ***/sys/class/uio/uio0/maps/map1/size***. No algoritmo 14 mostra um exemplo alocando 2 milhões de bytes DRAM, ou 0x1E8480 bytes em hexadecimal. ***extram_pool_sz*** (*External RAM pool size*) é a variável que indica a quantidade de memória externa alocada.

```
# lsmod
Module      Size Used by
uio_pruss    4066  0
g_multi      50407  2
libcomposite  15028  1 g_multi
mt7601Usta   641118  0
# rmmod uio_pruss
# lsmod
Module      Size Used by
g_multi      50407  2
libcomposite  15028  1 g_multi
mt7601Usta   641118  0
# modprobe uio_pruss extram_pool_sz=0x1E8480
# lsmod
Module      Size Used by
uio_pruss    4066  0
g_multi      50407  2
libcomposite  15028  1 g_multi
mt7601Usta   641118  0
# more /sys/class/uio/uio0/maps/map1/size
0x1e8480
```

Algoritmo 17: Alterando a quantidade de memória alocada para a PRUSS (Próprio autor).

2.10.4 Programando o PRU

Como foi dito na seção anterior, a programação e compilação de rotinas para o PRU, assim como a transferência do código é feito através do ambiente Linux. Em outras palavras, é necessário criar um programa para o PRU, compilá-lo, depois criar e executar um aplicativo para Linux que transfira o programa compilado para a memória ROM de um dos núcleos do PRUSS.

```

/* Carregando as bibliotecas */
#include <prussdrv.h>
#include <pruss_intc_mapping.h>
#define PRU_NUM 0 /*Define que será gravado no 1o núcleo*/
void main (void){
    /* Cria a estrutura de dados que representa os dados do controlador de interrupção*/
    /* Essa estrutura de dados serão utilizadas nas funções como prussdrv_init */
    /* PRUSS_INTC_INITDATA é uma constante de pruss_intc_mapping.h */
    tpruss_intc_initdata pruss_intc_initdata = PRUSS_INTC_INITDATA;
    /* Inicializa o PRU e aloca a memória do PRU no ambiente Linux */
    prussdrv_init 0;
    prussdrv_open (PRU_EVTOUT_0);
    /* Identifica a interrupção que é ocorrida após a PRU ser inicializada */
    prussdrv_pruinitc_init(&pruss_intc_initdata);
    /* Carrega e executa o programa no PRU */
    prussdrv_exec_program (PRU_NUM, "./PRU_ADC.bin");
    /* Espera a rotina ser executada, ou seja, até a PRU mandar uma interrupção */
    prussdrv_pru_wait_event (PRU_EVTOUT_0); // Isso assume que houve uma interrupção
    /* Desativa a PRU e fecha o mapa de memória */
    prussdrv_pru_disable(PRU_NUM);
    prussdrv_exit 0;
}

```

Algoritmo 18: Programa para executar o binário PRU_ADC.bin (Próprio autor).

Por padrão o Debian do BeagleBone Black inclui duas bibliotecas em C (***prussdrv.h*** e ***pruss_intc_mapping.h***) para este fim, a *PRU Linux Application Loader API*¹¹ (*Application Programming Interface*). Ela tem todas as funções para gravar, executar binários para PRU, fazer *debug*, ativar, desativar e fazer *reset* de cada núcleo desse coprocessador, além de outras funções auxiliares. Mais

¹¹ Conjunto de funções, rotinas e padrões para acesso a uma determinada aplicação de software ou hardware.

detalhes podem ser vistos em (38). No algoritmo 18 mostra um exemplo de código C para executar o binário ***PRU_ADC.bin*** no primeiro núcleo do PRUSSv2.

O binário para PRU deve ser feito em *assembly* e pode ser compilado pelo compilador ***pasm***, que pode ser executado com o comando de mesmo nome. A extensão desse código deve ser **.p** e pode ser feito em um editor de texto qualquer como o Nano, ou até mesmo na Cloud IDE. O algoritmo 19 mostra um exemplo de um código *assembly* escrito no arquivo ***PRU_ADC.p*** sendo compilado pelo ***pasm*** e gerando o binário ***PRU_ADC.p***, que, posteriormente, foi gravado e executado no pelo programa gerado a partir do código do algoritmo 18. A opção **-b** é utilizada para verificar erros de sintaxe.

```
$ pasm -b PRU_ADC.p
PRU Assembler Version 0.84
Copyright (C) 2005-2013 by Texas Instruments Inc.
Pass 2 : 0 Error(s), 0 Warning(s)
Writing Code Image of 13 word(s)
```

Algoritmo 19: Compilando um código *assembly* para o PRU (Próprio autor).

O PRU-ICSS possui uma quantidade de instruções relativamente pequena. São 45 instruções que podem ser categorizadas em aritméticas, lógicas, gravação e carregamento de registradores e controle de fluxo. Programação em *assembly* consiste em executar cada uma dessas instruções sequencialmente. Uma instrução consiste em um código de operação, ou *operation code*, (opcode) seguido de parâmetros, que podem ser uma variável, um registrador ou endereço de memória. No algoritmo 20 mostra como é feito um código em *assembly*. Observe que cada linha representa uma instrução. Códigos como *MOV*, são opcode responsáveis por mover uma variável, registrador ou valor alocado em um endereço de memória em outro. Já *ADD* são opcode para somar dois valores. A descrição e funcionamento de todas as instruções, incluindo exemplos, está disponível em (39).

Existe a possibilidade de acessar determinado *bit* ou palavra (8 *bits*) de um registrador, utilizando a notação, *.bY* e *.wX*, respectivamente. Onde *Y* indica o número do bit e *X* o número da palavra. A figura 24 mostra como isso funciona e na penúltima linha do algoritmo 20 exemplifica como é utilizada a notação em

uma instrução, no exemplo, é somado o valor do registrador 2 com a primeira palavra do registrador 1 e o resultado é salvo no registrador 1.

```

MOV r1, 0x25 // set r1 = 0x25 = 37 (dec)
MOV r2, 0b100 // set r2 = 100 (binary) = 4 (dec)
ADD r1, r1, 5 // set r1 = r1 + 5 = 42 (dec)
ADD r2, r2, 1<<4 // set r2 = r2 + 10000 (bin) = 20 (dec)
ADD r1, r2, r1.w0 // set r1 = r2 + r1.w0 = 20 + 42 = 62 (dec)
MOV r0, 0x00002000 // place PRU1 data RAM1 base address in r0

```

Algoritmo 20: Exemplo de código *assembly* para PRU (9).

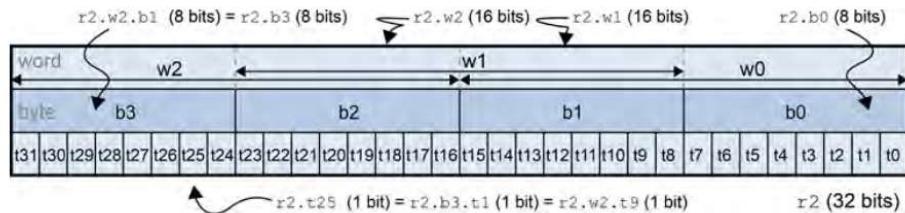


Figura 24: Notação utilizada para acessar palavras ou bits do registrador 2, r2 (9).

Em resumo, todo o processo de programação no PRU é mostrado na figura 25. Observe que antes de tudo foi carregada uma DTO para configurar os modos dos pinos de extensões do BBB e fazer outras configurações de *hardware* necessárias. Feito isso, pode-se partir para o código hospedeiro é aquele do algoritmo 18, que deve ser compilado pelo compilador **gcc** ou **g++** e gerar o programa hospedeiro. No outro lado devemos criar um código em *assembly* compilar com o **pasm** e gerar o arquivo **.bin** que vai ser carregado e executado através do programa hospedeiro, que também é um binário.

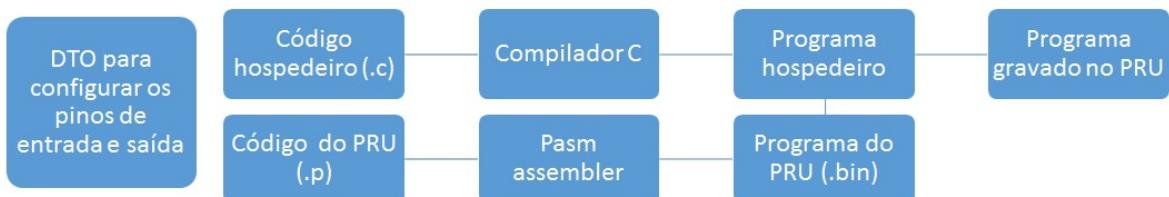


Figura 25: Processo de programação na PRU (9).

1.1.1. Outras formas de programar a PRU

Como vimos na seção anterior a forma clássica de programação na PRU é bastante complicada, pois envolve o conhecimento de um hardware proprietário não muito comum, o PRU, e ainda, toda a programação deve ser feita em baixo nível. A comunidade tem se esforçado para trazer soluções mais atraentes, sejam em APIs para fazer as rotinas mais comuns na PRUSS ou aplicações prontas

para ser utilizadas em conjunto com outras linguagens de alto nível, como Python ou JavaScript.

É possível utilizar funções da PRU em conjunto com o *node.js* através do pacote ***pru***, que pode ser instalado pelo terminal através do comando ***npm install pru***. ***npm install*** é um comando para instalar pacotes para *node.js*, semelhante ao comando ***pip install*** para o Python. Antes de utilizar o pacote é necessário seguir uma sequência de passos disponível em (40), que são basicamente aqueles da figura 25. Com a diferença que os códigos das *device tree overlays*, códigos *assembly* e código C já estão prontos.

Um outro exemplo é PyPRUSS que é basicamente um *port* da *PRU Linux Application Loader API* para Python. Neste caso, o script Python funciona como programa hospedeiro, havendo a necessidade de criar o código em *assembly* e gerar o arquivo *.bin*. Todo processo de instalação e exemplos estão disponíveis em (41) e (42).

Além desses, temos o *driver libpruio-0.2* que permite utilizar diretamente o PRU para trabalhar com aplicações relacionadas ao controle das entradas e saídas alta velocidade, PWM, ADC e controle de interface *Touch-Screen*, este último um subsistema do SOC que pode ser visto na figura 4. O processo de instalação pode ser encontrado em (43). A documentação da API, incluindo exemplos, podem ser encontrados em (44).

E ainda, existe um *port* não oficial do ***gcc***, o compilador C do Linux, para o PRUSS. O compilador está disponível nos repositórios oficiais do Debian Jessie e pode ser instalado através do comando ***apt-get install gcc-pru***. Além disso, é necessário instalar outras dependências que estão disponíveis na página do GitHub do compilador (45). Este compilador não substitui completamente o ***pasm***, pois não tem todos os recursos e existe alguns *bugs*, mas é uma boa alternativa para quem não quer escrever o código em *assembly*.

3 O DAQ

O dispositivo de aquisição de dados requerido pelo Centro Tecnológico de Automação Industrial deve ser um dispositivo modular com a capacidade de conectar até 8 módulos. Esses módulos serão ADCs ou DACs que poderão ser conectados em um dos 8 slots do DAQ. A figura 26 mostra como seria esse DAQ em duas situações distintas: a esquerda, com os 8 módulos conectados e a direita com apenas 5 módulos conectados. A camada física da conexão entre o DAQ e o computador deve ser USB, por motivos já explicado no capítulo 1. As outras camadas serão padrões da pilha de protocolos TCP/IP ou UART, ambos explicados no capítulo 2.

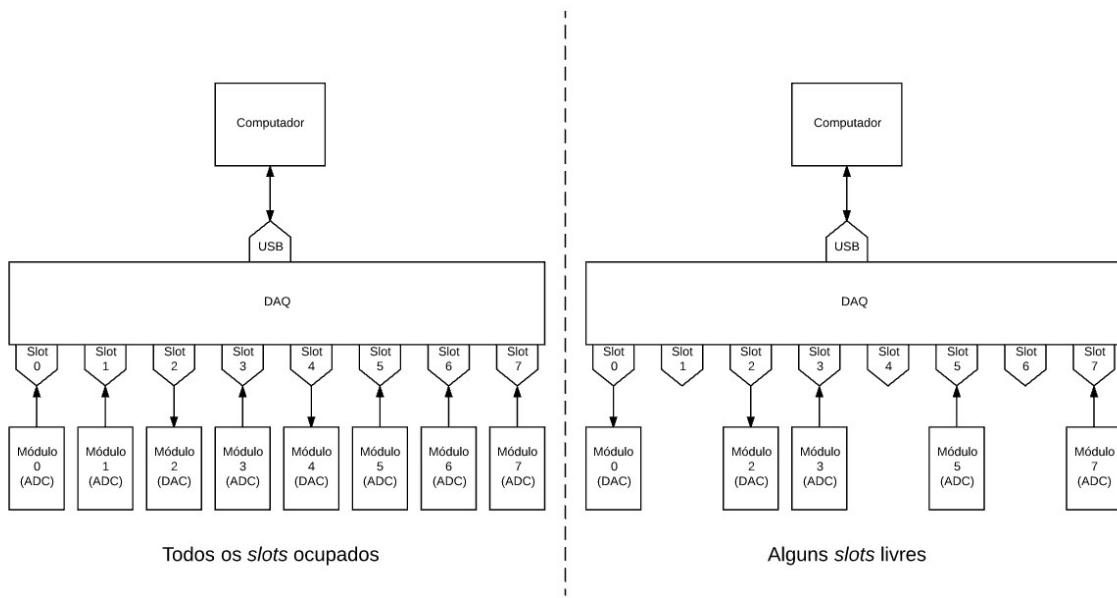


Figura 26: Concepção do DAQ para o CTAI (Próprio autor).

Como este trabalho é apenas uma prova de conceito, para verificar a viabilidade do BeagleBone Black como um dispositivo de aquisição de dados. Faremos uma versão simplificada daquele mostrado na figura 26. Neste trabalho não desenvolveremos os módulos. Serão utilizados o próprio ADC do BBB ou algum *shield* Arduino para esta finalidade. Além disso, utilizaremos apenas dois módulos para testar a capacidade de processamento simultâneo do BeagleBone Black. Se isso for possível, este conceito pode ser facilmente estendido para os 8 slots.

Lembrando que cada módulo deve trabalhar de maneira independente. Portanto, mesmo que o BeagleBone tenha um ADC de 8 canais não poderíamos utilizar todos eles simultaneamente, pois existe apenas um conversor analógico

digital SAR¹² (*Successive Approximation Register*) de 12 bits, que é conectado em um multiplexador (46). Esse multiplexador fica alternando entre os canais ciclicamente e durante esse processo o ADC faz a leitura do canal corrente (Figura 27). A vantagem da multiplexação é o barateamento do produto final e o espaço ocupado, a desvantagem é que ao utilizar mais de um canal há perda de performance.

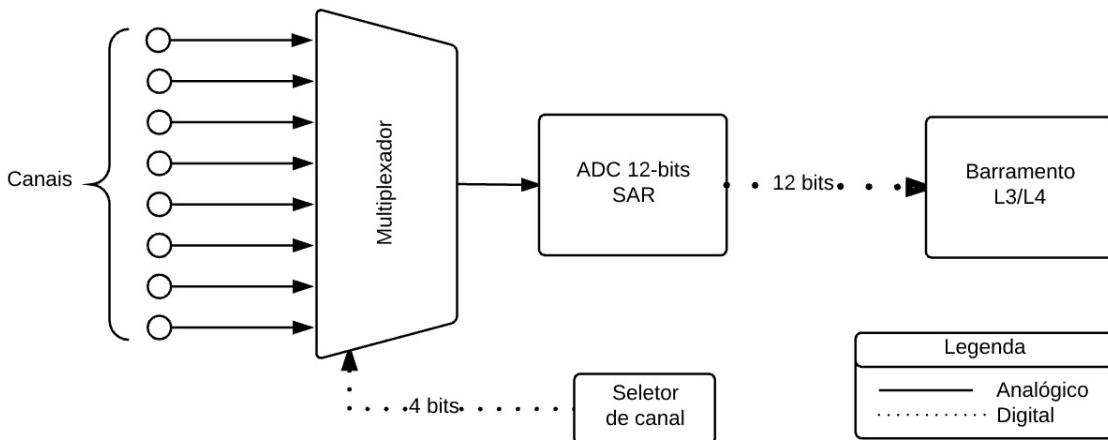


Figura 27: Esquemático da multiplexação de canais no ADC do BBB (Próprio autor).

Além disso, o BeagleBone deve ser capaz de receber a informação de cada módulo simultaneamente ao mesmo tempo que trabalha para enviar os dados para o computador. Por isso, será necessário trabalhar com o conceito de *multithread*, ou processamento simultâneo. Existem várias maneiras de fazer isso, veremos isso mais tarde.

Assim, existe duas possibilidades. A primeira seria utilizar o BeagleBone Black como o DAQ da figura 26. Um dos módulos seriam o próprio ADC interno ao BBB e o outro seria o *shield* para Arduino ADS1015 (Figura 28). Esse componente tem um ADC prover uma precisão de 12 bits a 3300 amostras por segundo (47). Além disso, ele suporta 4 canais, mas só utilizaremos um deles por motivos já citados. Embora a frequência de amostragem seja bem inferior ao do BeagleBone Black, esse *shield* será o suficiente para testar a capacidade multitarefas do *single board computer*.

¹² Uma topologia de conversor analógico digital, que tem como vantagem o pequeno espaço ocupado no *wafer* de silício e o baixo consumo, mais detalhes podem ser vistos em (77), incluindo o seu funcionamento e a comparação com outras topologias.

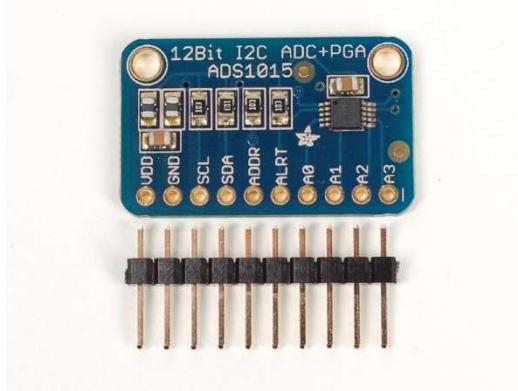


Figura 28: *Shield* para Arduino ADS1015 (47).

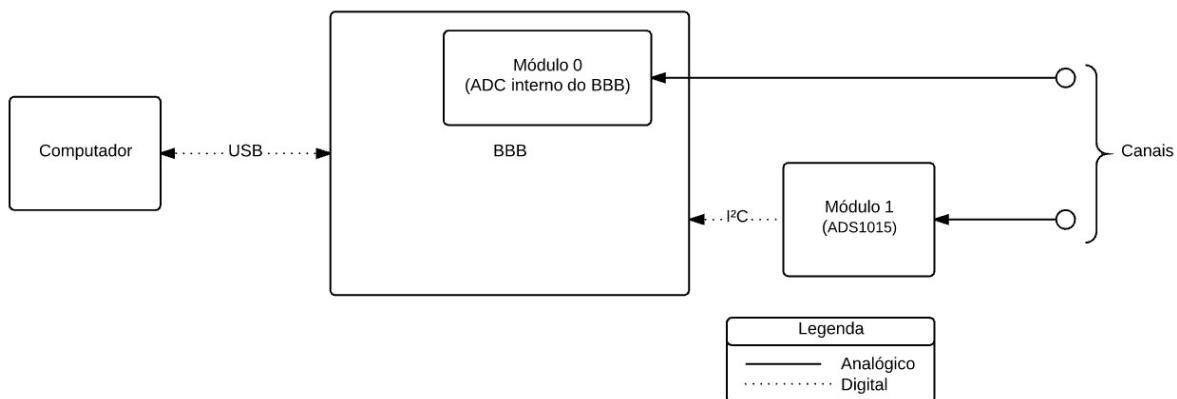


Figura 29: Esquemático do BBB funcionando como o DAQ (Próprio Autor).

A segunda possibilidade seria utilizar cada BeagleBone Black como um módulo e conectá-los através de um *hub* USB (Figura 31), que é um dispositivo utilizado para multiplicar uma porta USB em várias (48). Neste caso o *hub* faria o papel do DAQ da figura 26. Existem *hubs* de vários tamanhos, inclusive com 8 entradas, ou seja, não seria problema reproduzir o dispositivo da figura 26. A figura 30 mostra o exemplo de um *hub* com 3 entradas.



Figura 30: Exemplo de *hub* USB de 3 entradas (48).

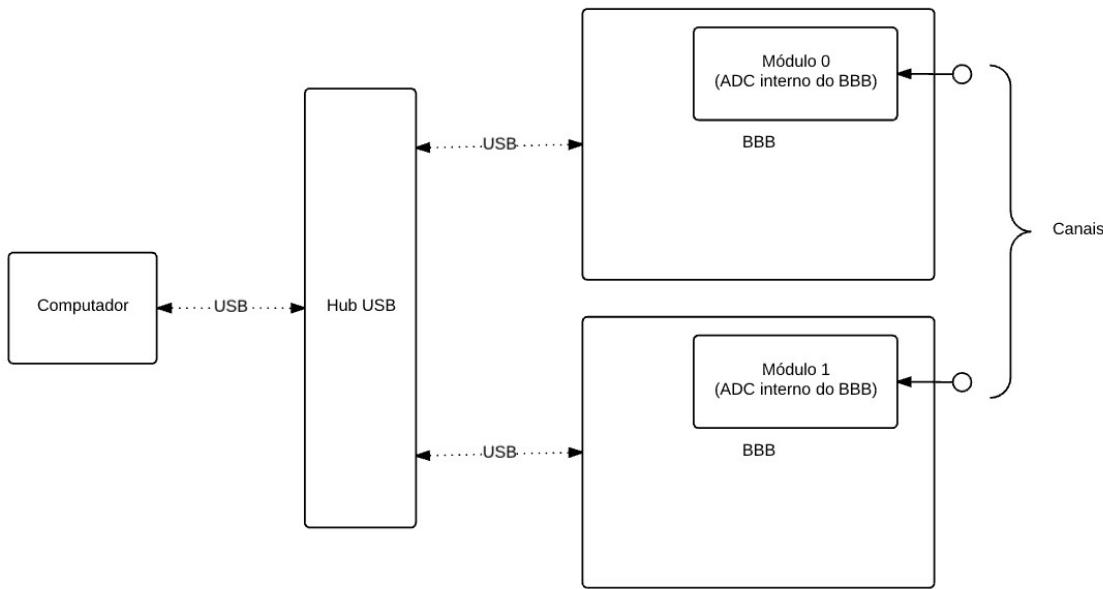


Figura 31: Esquemático do *hub* USB funcionando como DAQ (Próprio Autor).

Nas próximas seções veremos como capturar os dados do ADC em alta velocidade e como enviar esses dados capturados para o computador pela interface USB. Vamos tentar explorar as principais soluções existentes na literatura, mesmo que estas não sejam realmente utilizadas nesta monografia, visto que, um dos objetivos deste trabalho é adquirir informações para serem utilizadas em trabalhos futuros.

3.1 Métodos de captura de dados de um ADC em alta velocidade

Vimos no capítulo 2 algumas soluções para trabalhar em tempo real, dentre elas, a utilização do subsistema PRUSS. Vimos, também, que não é uma tarefa trivial programá-lo, pois envolve a utilização de linguagem de baixo nível, que nem todos os desenvolvedores estão acostumados. A comunidade criou soluções prontas para as rotinas mais frequentes desse coprocessador. No capítulo 2 vimos algumas delas, de uma maneira geral. Nesta seção iremos focar em soluções específicas para a captura de dados em alta velocidade utilizando o ADC.

3.1.1. Utilizando as I/O dedicadas do PRU como comunicação paralela

Esta solução consiste em utilizar um ADC externo com saída paralela e conectar estas saídas aos pinos dedicados do PRUSS (Figura 22). O usuário Shabaz publicou um post no fórum da Element14 um exemplo desse tipo de aplicação (49). Na solução dele foi utilizado um módulo com um ADC externo de 8 bits, o ADC08200 (Figura 33). Este módulo já vem com um amplificador e com um

clock interno. Shabaz preferiu desativar o *clock* interno do módulo e utilizar um *clock* externo de 32 Mhz. Além disso, ele incluiu um *buffer* para que as impedâncias das entradas do BBB não interfiram no ADC. A figura 34 mostra como ele fez a montagem do circuito.

Os pinos dedicados da PRU utilizados são compartilhados com os pinos da entrada HDMI, portanto deve-se desativar essa entrada como foi explicado no capítulo 2. Além disso, é necessário carregar a DTO responsável por reconfigurar os pinos corretamente. A relação de pinos utilizados pode ser vista na figura 32. Observe que ele utilizou 10 entradas, as duas últimas da figura 32 são relativas ao *clock* e *enable* do *buffer*. Lembre-se que a comunicação é feita entre o *buffer* e o BBB.

BBB Schematic	BBB port	Assign	Bit
LCD_DATA0	P8.45	D0	PRU1_R31_0
LCD_DATA1	P8.46	D1	PRU1_R31_1
LCD_DATA2	P8.43	D2	PRU1_R31_2
LCD_DATA3	P8.44	D3	PRU1_R31_3
LCD_DATA4	P8.41	D4	PRU1_R31_4
LCD_DATA5	P8.42	D5	PRU1_R31_5
LCD_DATA6	P8.39	D6	PRU1_R31_6
LCD_DATA7	P8.40	D7	PRU1_R31_7
LCD_PCLK	P8.28	CLK	PRU1_R31_10
LCD_DE	P8.30	*EN	PRU1_R30_11

Figura 32: Pinos dedicados à PRU utilizado no experimento de Shabaz.

Na parte de software ele criou um programa hospedeiro chamado, **adc_app** que carregava o programa no PRU1 e salva os dados na memória compartilhada do PRU. Este mesmo programa hospedeiro envia os dados para um servidor com o *node.js*. O servidor utiliza o protocolo HTTP na porta 8081 para enviar os dados para o navegador do lado do cliente, que no caso é o computador conectado ao BeagleBone Black pela porta USB. O esquemático completo do sistema está na figura 35.



Figura 33: Diagrama de pinos do ADC08200 (50).

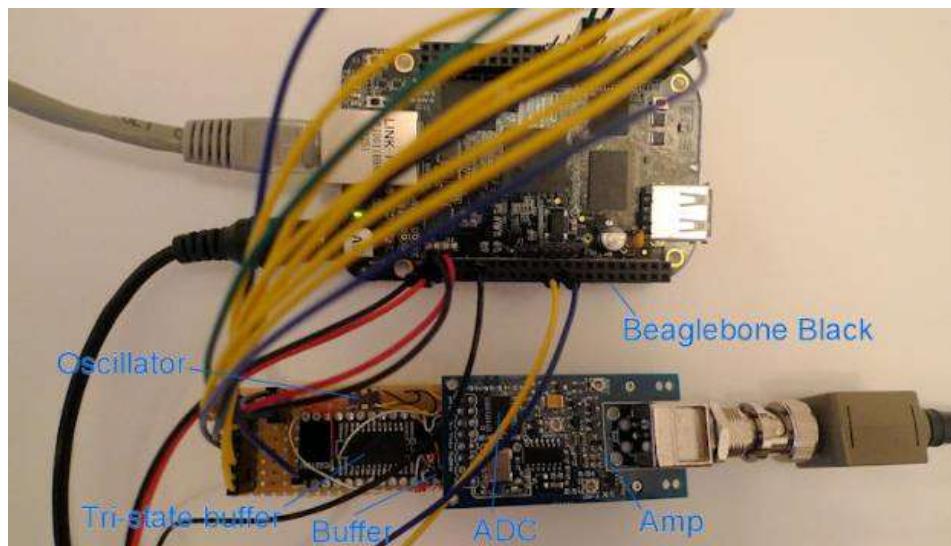


Figura 34: Fotografia de todo o sistema criado por Shabaz (49).

O JavaScript permite utilizar os *web sockets*, que tem uma ideia parecida a comunicação TCP e os *socket* de rede. Isso facilita o *streaming* de dados em alta velocidade, segundo (49) por este método é possível alcançar uma taxa de 20Mbytes/s ou mais. Ao acessar o endereço do servidor, incluindo a porta, pelo navegador é carregado uma página web, que inclui um arquivo HTML, CSS e JavaScript. O código JavaScript do próprio navegador abre uma conexão por *web socket* entre o cliente (O navegador) e o servidor, a partir daí cria-se um *streaming* de dados entre ambas as partes. Mais detalhes de como funciona as páginas web será visto na [seção 3.2](#). Todo o esquemático criado por Shabaz está disponível na figura 35 e os resultados na figura 36. Nesta última figura foi feito um teste com uma senoide de 1GHz e de 100 kHz.

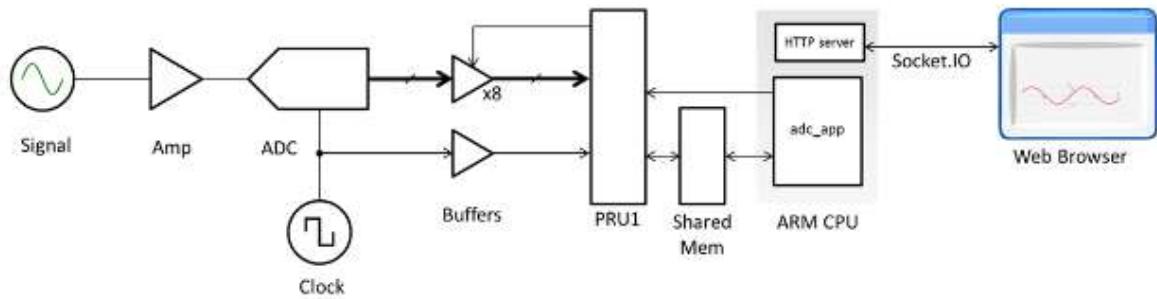


Figura 35: Esquemático completo do sistema de Shabaz (49).

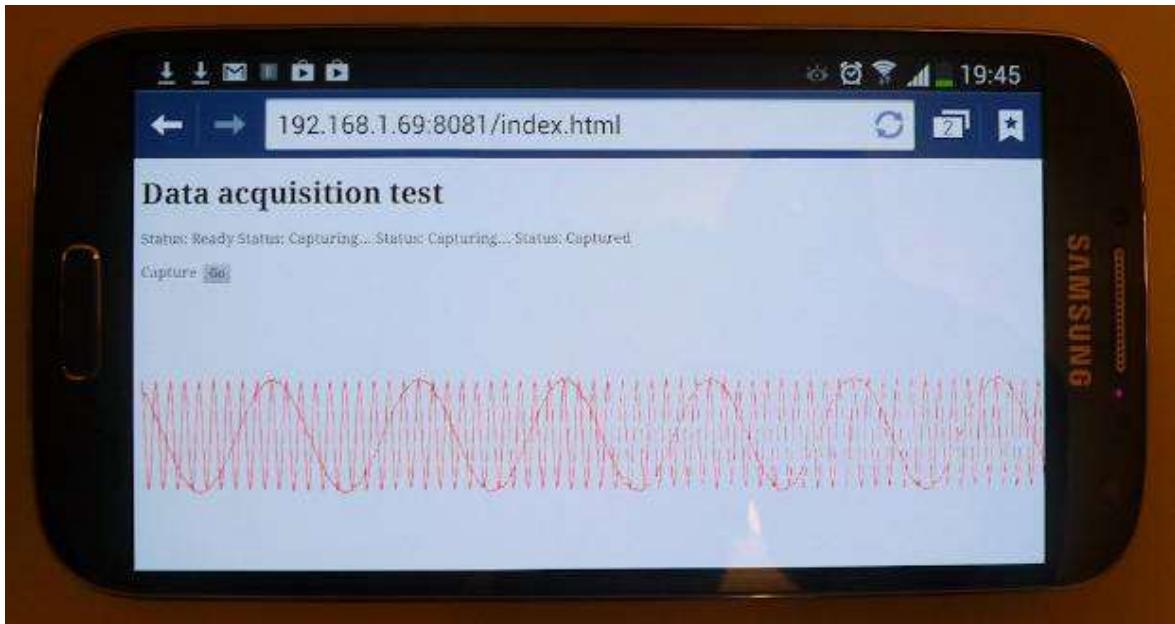


Figura 36: Resultado do teste feito por Shabaz (49).

3.1.2. Utilizando as I/O dedicadas do PRU como comunicação serial

Embora a comunicação paralela com as I/O dedicadas do PRU permita uma taxa de aquisição de dados muito alta, existe um inconveniente. O número de portas utilizadas é muito grande. E como o BeagleBone Black não tem tantas saídas dedicadas assim isso pode não ser uma boa ideia para criar um DAQ com vários canais.

Uma solução para este problema é o uso de ADCs externos com saída serial, que geralmente, são mais lentos que os de saída paralela, mas utilizam menos pinos para fazer a transferência de dados. Os ADCs com saída serial geralmente utilizam algum protocolo de comunicação consagrado como SPI ou I²C. A comunicação I²C é mais lenta, portanto, é mais utilizada em ADCs com baixa frequência de amostragem. Neste caso, é mais interessante adquirir um ADC externo com comunicação via SPI.

Derek Molloy fez um experimento em seu *blog* utilizando o MCP3008, um ADC de 8 canais do tipo SAR, 10-bits e 100kSps e o que se comunica através da SPI (Figura 37). Neste experimento ele optou por utilizar os dois PRU, no primeiro é gravado o código responsável por fazer a comunicação SPI e no segundo é utilizado exclusivamente para gerar o *clock* da comunicação SPI e do ADC, permitindo uma maior flexibilidade visto que o *clock* não é gerado por um cristal oscilador. Foi necessário utilizar um núcleo exclusivamente para isso, pois assim evita-se a geração de *clocks* impreciso, visto que a única rotina que está rodando neste núcleo é a geração de *clock*, ou seja, não há nada para atrapalhar.

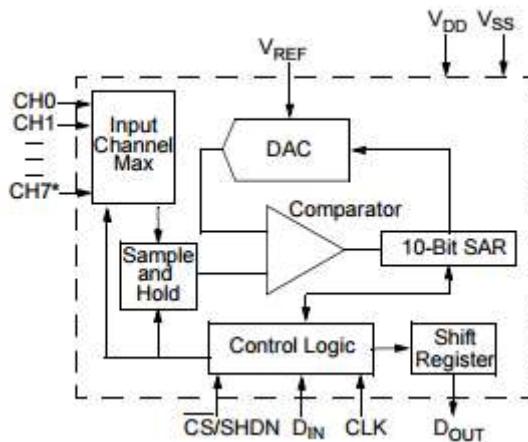


Figura 37: Diagrama de blocos do MCP3008 (51).

A. Comunicação SPI do MCP3008

A comunicação SPI, ao contrário da UART vista no capítulo 2, é sincronizada através de um sinal de *clock*. Isso permite que a mesma trabalhe em frequências mais elevadas, mas em compensação há uma necessidade de uma via a mais.

Basicamente existe um pino para enviar dados do mestre (BBB), MOSI (*Master Output Slave Input*), para o escravo (ADC), um outro para receber dados entre esses dispositivos, MISO (*Master Input Slave Output*), o sinal de *clock* (CLK) e o pino para seleção do escravo (*Chip Select*), ativo em nível baixo. A relação de pinos está disponível na tabela 9.

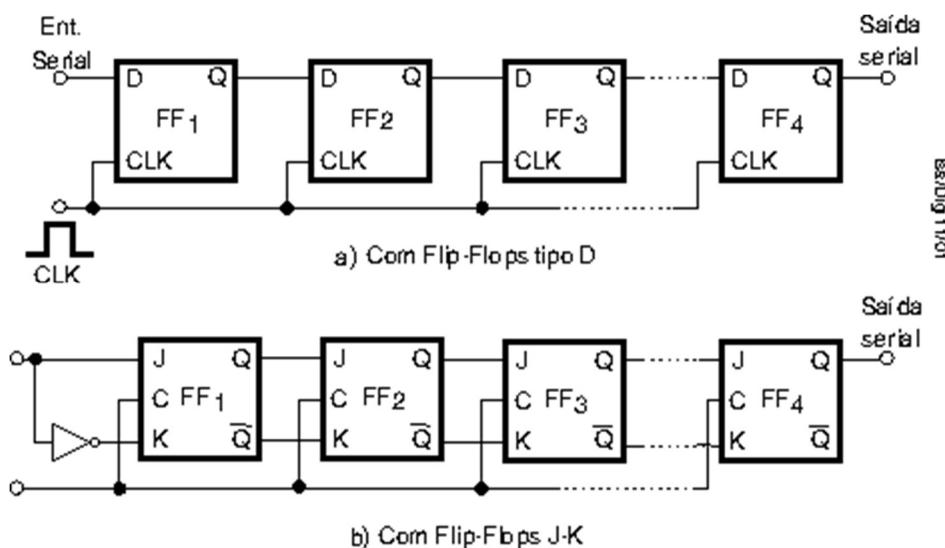
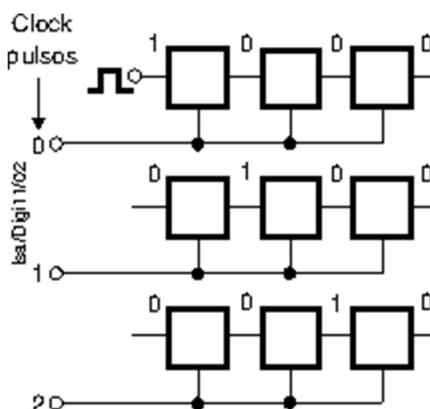
Nome	Função
MOSI	Transferência de dados do mestre para o escravo
MISO	Transferência de dados do escravo para o mestre
CLK	Serial <i>clock</i>

Chip Select

Seleção de escravo

Tabela 9: Pinos da comunicação SPI (Próprio autor).

Como o SPI é *full-duplex*, ou seja, existe uma via exclusiva para enviar dados (MISO) e uma via para receber dados (MOSI), o sinal de *clock* é utilizado também para indicar qual via deve ser lida. A forma de como essa regra é aplicada varia de dispositivo para dispositivo (52), mas no caso do MCP3008 a leitura do MISO é feita na borda de descida do sinal de *clock*, enquanto a leitura do MOSI é feita na borda de subida (Figura 40). Essa forma de operação obriga que para cada dado transferido do mestre para o escravo seja recebido um dado da outra ponta. Além disso, fisicamente o SPI é um registrador de deslocamento (*shift-register*), ou seja, um conjunto de *flip-flops*¹³ conectados sequencialmente (Figura 38). Seu princípio de funcionamento é simples, a cada pulso de *clock* há a transferência de um *bit* de um *flip-flop* para outro (Figura 39).

Figura 38: Registradores de deslocamento com *flip-flops* tipo D e *flip-flops* J-K (53).Figura 39: Princípio de funcionamento do *shift-register* (53).

¹³ Um circuito digital capaz de servir como memória de 1 *bit*.

Como estamos utilizando apenas um escravo, o pino *chip select* deve ficar em nível alto entre uma amostragem e outra. O processo inicia com o envio de dados do mestre (BBB) para o escravo (ADC). Assim, no pino MOSI, primeiro há o envio de um bit de início (Nível alto), seguido de 4 bits (Single/Diff, D2, D1 e D0) que indicam o modo de operação. O primeiro *bit* (Single/Diff) indica o modo de operação (1 para único, ou *single* e 0 para diferencial) e os outros *bits* indicam qual canal será lido, de 0 a 7 ou 000_b a 111_b , em binário. Os bits seguintes não importam (*don't care*), representados por *X*. Na aplicação de Derik foi utilizado a leitura da entrada 0 no modo único, ou seja, $11000XXXXX..._b$. Enquanto o ADC espera o modo de operação os bits do MISO não importam (*don't care*). Assim que o ADC recebe essa informação há um pequeno *delay* (*don't care*) seguido de um bit nulo (*null*) e, posteriormente, é enviado o resultado da conversão (10 *bits*). Todo o processo pode ser visto na figura 41.

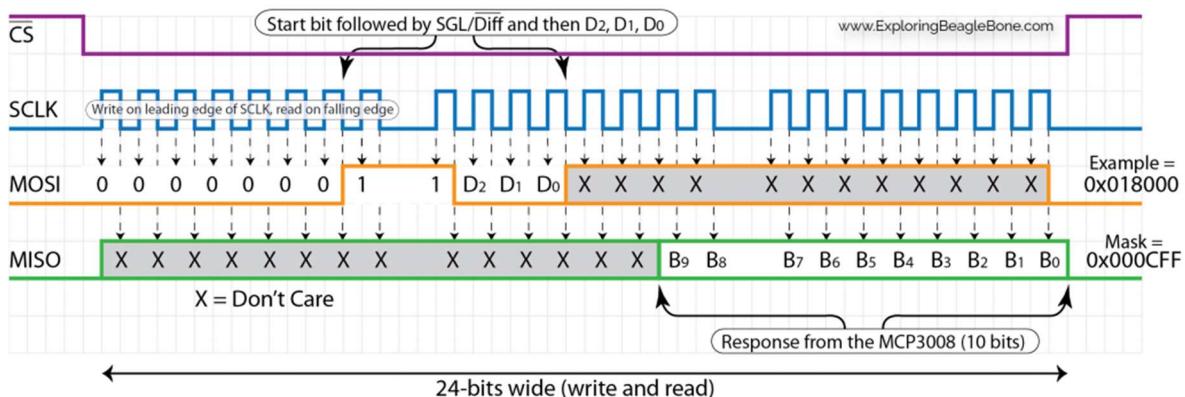


Figura 40: Sinais de comunicação do MCP3008 com o BBB (37).

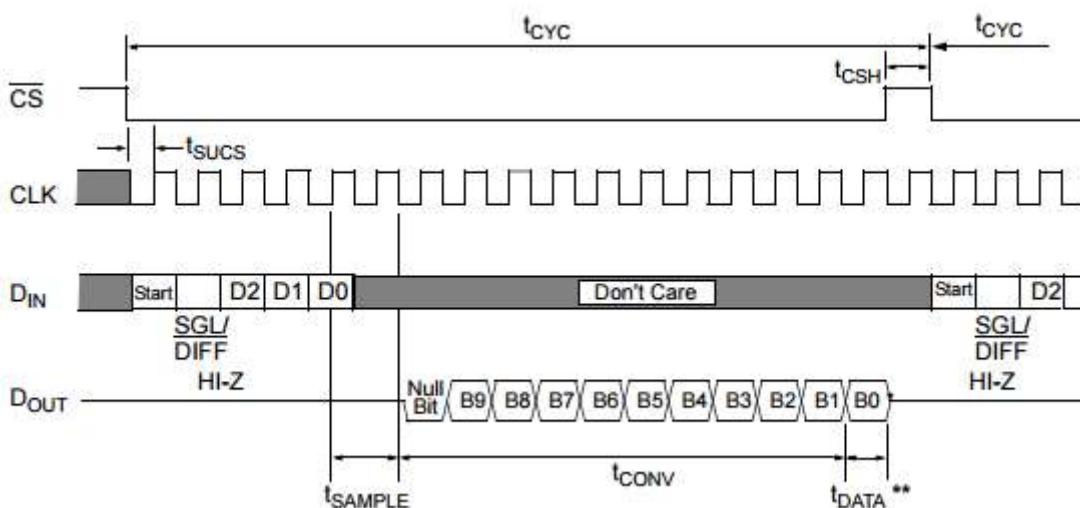


Figura 41: Regras para envio de amostras no M3004, através da SPI (51).

Em alguns microcontroladores o envio do *clock* só pode ser feito em grupos de 8 bits. Isso faz com que o processo de conversão fique um pouco mais lento que o normal, geralmente, 3 ciclos de 8 bits deste microcontrolador. Esse processo é ilustrado na figura 42. Observe a semelhança entre o *clock* e a forma de envio dessa figura e da figura 40.

Em resumo, a comunicação SPI utiliza pelo menos três pinos mais o número de escravos (52), como neste caso só existe um escravo, que no caso é o ADC, então o total de pinos é 4. Mas é possível aumentar a quantidade de escravos, ou ADCs externos, tranquilamente.

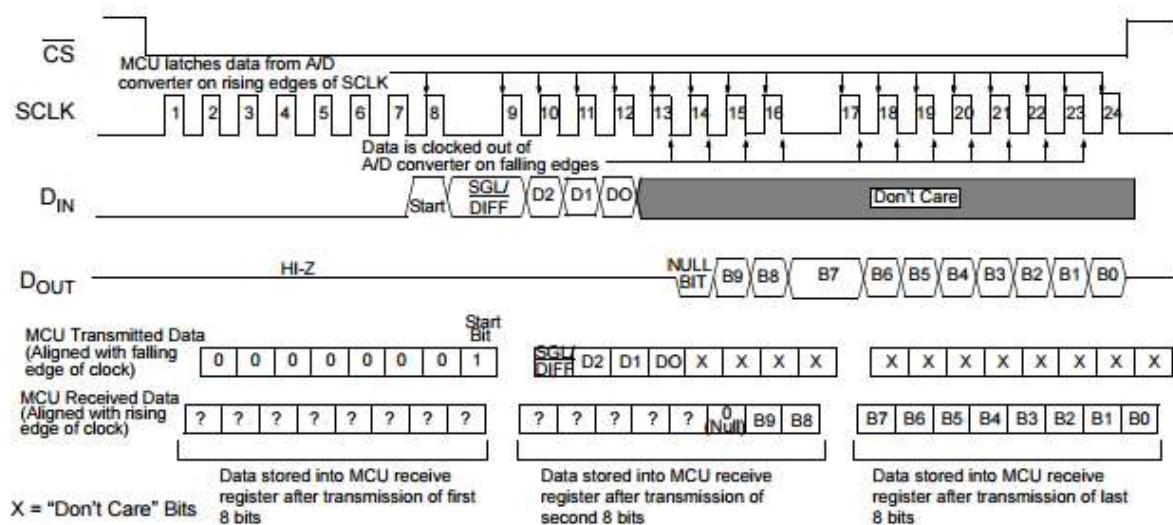


Figura 42: Envio de uma amostra através da SPI utilizando um *clock* em pacotes de 8 bits (51).

Se por acaso houvesse a necessidade de utilizar um segundo ADC, este estaria conectado no mesmo MOSI, MISO e CLK do primeiro e em um pino adicional de seleção de escravo (*Chip Select 2*). Assim caso o PRUSS quiser acessar um dos ADCs basta jogar o *Chip Select* correspondente para nível baixo, enquanto os outros pinos de seleção de escravo estão em nível alto. Alternando esses pinos é possível fazer a leitura de vários componentes sem a necessidade de utilizar muitos pinos a mais. A figura 43 mostra como é feita a conexão entre o BBB e o ADC externo.

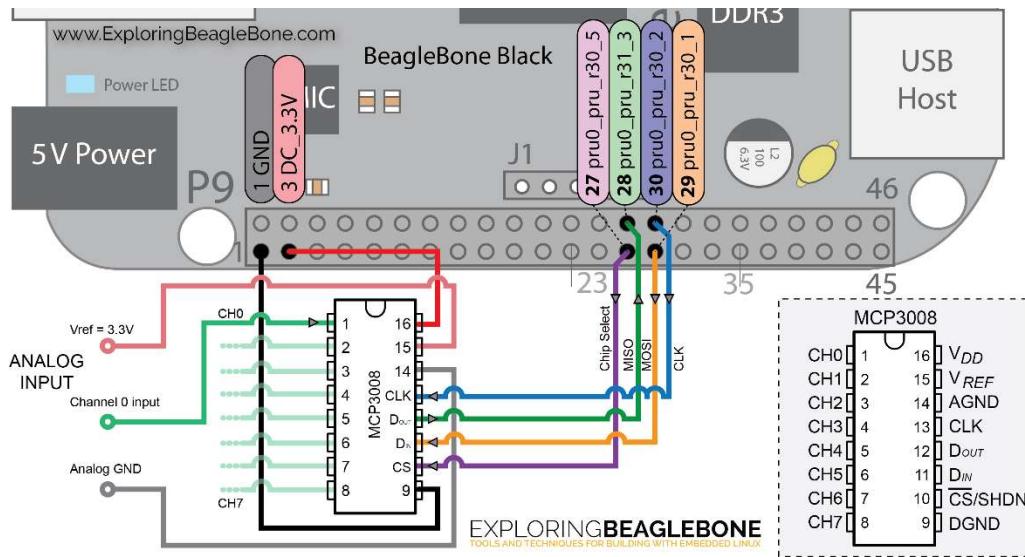


Figura 43: Experimento de Molloy com o MCP3008 (37).

B. Funcionamento do código

A relação de sequência de ações que o BBB deve tomar está descrito abaixo e na figura 44:

1. Carregar a *Device Tree Overlay* em um *slot* para *virtual cape* (Terminal);
2. Alocar mais memória para salvar grande quantidade de amostras (Terminal);
3. Executar o arquivo hospedeiro pruadc (.c);
4. Executar o programa do PRU0 responsável pela comunicação SPI (.p);
5. Executar o programa da PRU1 responsável pela geração do sinal de *clock* (.c);
6. Salva os dados num arquivo de texto (.sh).

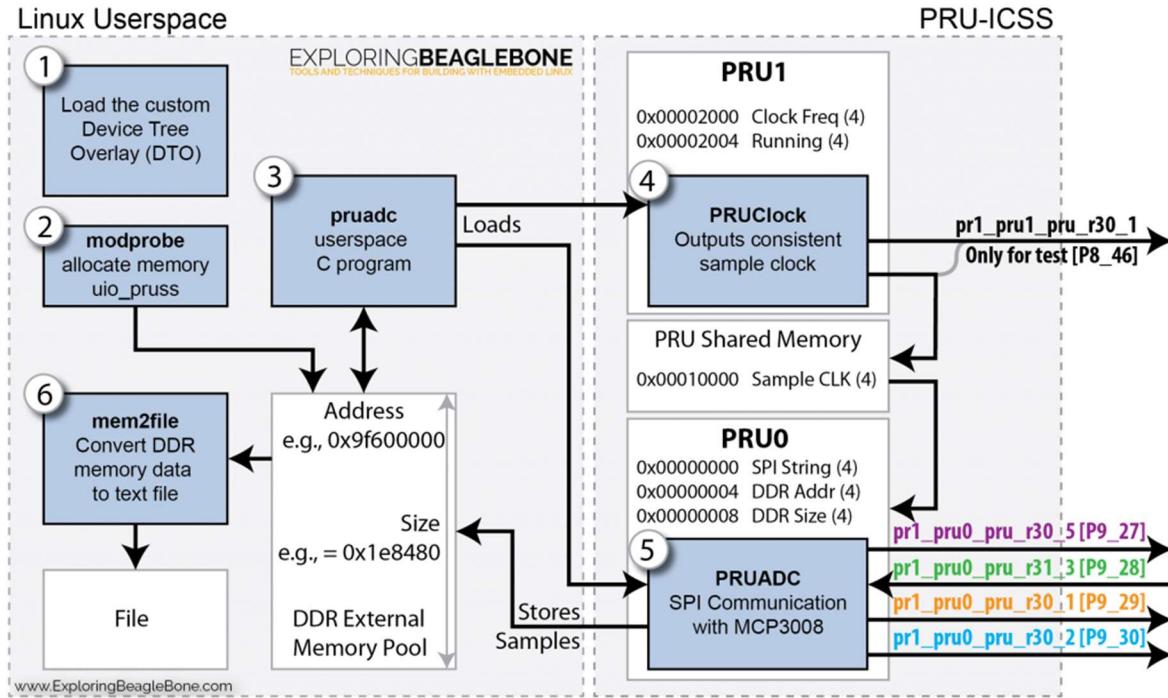


Figura 44: Estrutura e interação entre o ambiente Linux, o PRUSS e o ADC (37).

Todos esses passos já são conhecidos e bem parecidos com o do item anterior, com a exceção do passo 2. Molloy resolveu salvar todos os dados na RAM antes de salvá-los em um arquivo de texto. Como, dependendo da quantidade de amostras o espaço de RAM necessário pode ser maior que 256kB é necessário recarregar o módulo `ui0_pruss` com o comando `modprobe` (Algoritmo 17). O passo 6 foi através do script em `shell`, pois o autor quis salvar os resultados na forma de gráfico em um PDF (*Portable Document Format*).

```

#!/bin/bash

echo "Capturing the first 2000 samples from the memory and
dumping to capture.dat"
./mem2file 2000 > capture.dat

echo "Plotting the data to a PS file"
gnuplot << EOF_
set term postscript enhanced color
set output 'plot.ps'
set title 'EBB Plot'
plot 'capture.dat' with linespoints lc rgb 'blue'
_EOF_

echo "Converting the PS file to a PDF file"
ps2pdf plot.ps plot.pdf

```

Algoritmo 21: Script shell para capturar os dados da RAM e salvar em um PDF (37).

C. Resultados

Em sua página Molloy mostra o resultado de uma senoide 500Hz (Figura 45).

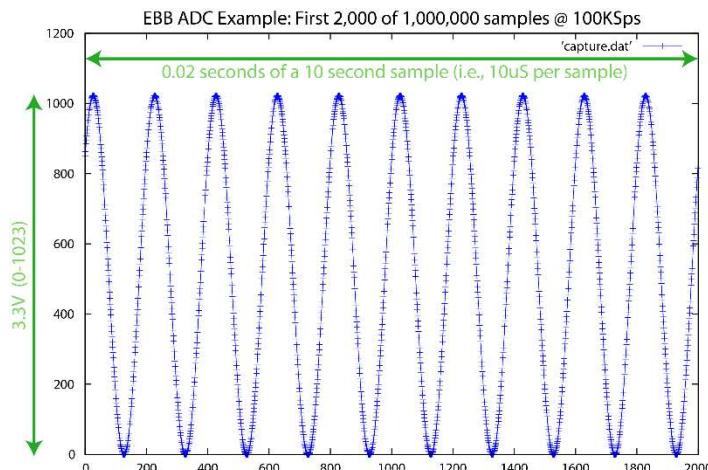


Figura 45: Resultado do experimento de (37).

Posteriormente Molloy adaptou o código para o ADS7883, um ADC externo de um canal, 12 bits e com capacidade de até 2Msps. Entretanto, devido a limitações de processamento do PRUSS a frequência de amostragem máxima foi limitada a 1Msp. O esquemático do circuito está na figura 46. Observe que ele adicionou um filtro capacitivo para eliminar ruído de alta frequência, o motivo pode

ser visto na figura 47. Veja também que o CI não tem pino para enviar dados, visto que não há a necessidade de configurar modos, tornando o código mais simples e direto.

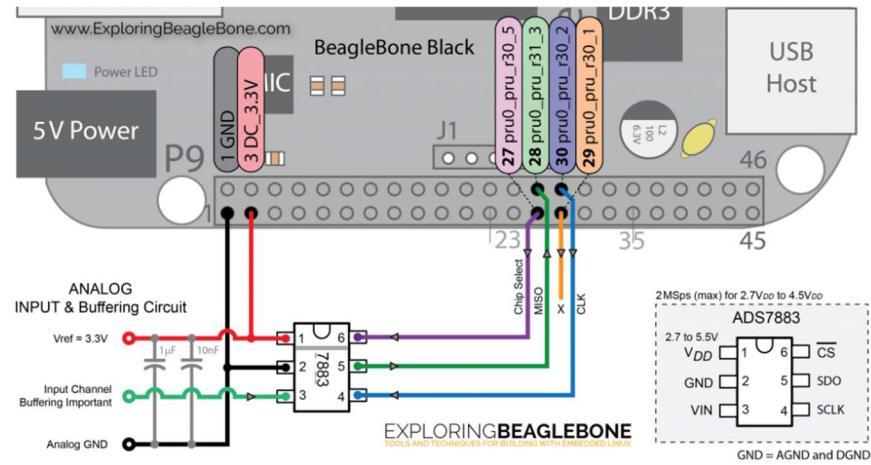


Figura 46: Circuito com o ADS7883 (37).

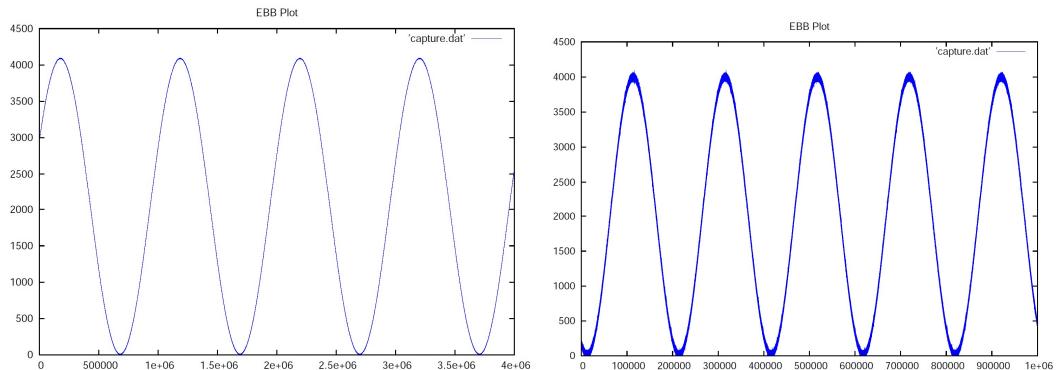


Figura 47: Resultado utilizando o ADS7883 com o filtro capacitivo (a esquerda) e sem o filtro capacitivo (a direita) (37).

3.1.3. Utilizando o PRUSS para salvar os dados do ADC interno

A terceira solução disponível envolve utilizar o próprio ADC interno do BBB e o PRUSS. Neste caso esta é a solução mais simples e barata, pois não envolve a utilização de um dispositivo externo. Contudo o usuário fica limitado às características do ADC do BBB.

Existem diversas soluções que permitem esse modo de conexão. Dentre elas destacam a, libpruio-0.2 já citada no capítulo 2, e uma solução já pronta a `beaglebone_pru_adc`, uma biblioteca em Python para este fim.

O autor desta biblioteca disponibilizou a API na sua página do GitHub¹⁴ (54). A documentação foi feita para o Ångström, por isso, o processo de

¹⁴ Plataforma online para compartilhamento de código aberto

instalação difere um pouco do disponível na sua página. Se o BBB tiver conectado à *internet*, basta executar o algoritmo 22.

A biblioteca tem alguns modos de execução, assim como alguns exemplos.

```
# apt-get update && apt-get install python-pip python-setuptools
python-smbus
# git config --global http.sslVerify false
# git clone https://github.com/pgmmpk/beaglebone_pru_adc.git
# cd beaglebone_pru_adc
# python setup.py install
```

Algoritmo 22: Processo de instalação da biblioteca beaglebone_pru_adc (54).

Desses modos o mais interessante para a nossa aplicação é o modo osciloscópio (*oscilloscope mode*), cujo exemplo está disponível em ***examples/oscilloscope.py***. Nesse modo todo dado capturado é transferido para a memória RAM do BBB semelhante a aplicação de Molloy, explicada na seção anterior. Por isso, a quantidade de amostras está limitada a quantidade de memória alocada pelo ***uiopru***, que por padrão permite salvar apenas um pouco mais de 260000 amostras, ou aproximadamente, um tempo de 0,5 s, segundo o autor. Como já vimos é possível alocar mais memória através do algoritmo 17, aumentando o número máximo de amostras para mais de 1 milhão, se necessário. Na seção 3.2 veremos uma forma mais eficiente de salvar muitas amostras sem consumir tanta RAM, através de técnicas de *multithread*. O algoritmo 23, mostra um exemplo do modo osciloscópio capturando 1 milhão de amostras e salvando-os na tupla ***save_data***. No algoritmo 24 mostra alguns atributos da classe ***capture*** não utilizados no algoritmo 23, mas que podem ser importantes.

Além da libpruio-0.2 e da beaglebone_pru_adc existem muitas outras soluções, muitas delas parecidas, como alguns pacotes Python disponível através do comando ***pip install*** e soluções para outras linguagens. Inclusive o próprio autor da beaglebone_pru_adc adaptou sua biblioteca para *node.js*, que pode ser encontrada em (55). Infelizmente esta solução não tem o modo osciloscópio, que é essencial para a nossa aplicação, por isso iremos focar na opção para Python. Contudo, em trabalhos futuros poderão ser testadas outras possibilidades.

```

import beaglebone_pru_adc as adc #Importa a biblioteca como o objeto adc
numsamples = 1000000 #
capture = adc.Capture() #cria o objeto Capture, filho de ADC
#capture.oscilloscope_init(No do registrador do ADC, No de amostras)
capture.oscilloscope_init(adc.OFF_VALUES, numsamples) #Método de capture para configurar o
modo osciloscópio
#adc.OFF_VALUES <- Atributo de adc que representa o endereço de memória do ADC0
capture.start() #Método de capture que inicia a captura
while True:
    if capture.oscilloscope_is_complete(): #capture.oscilloscope_is_complete() <- Método
        para verificar se o todas as amostras foram capturadas
        break
capture.stop() #Método para parar a captura de dados e finalizar o arquivo do PRU
capture.wait() #Método que espera o programa do PRU ser finalizado
save_data = capture.oscilloscope_data(numsamples) #Método que retorna uma tupla com as
amostras salvas na memória RAM
Capture.close() #Método para liberar os dados da RAM para o sistema operacional

```

Algoritmo 23: Exemplo em Python comentado executando o modo osciloscópio (Próprio autor).

```

#Outros atributos de capture
Capture.ema_pow #Se for diferente de 0, aplica um filtro EMA (Exponential moving average),
cuja fórmula é ema += (value - ema / 2^ema_pow), por padrão é desativado
Capture.cap_delay #Atributo que define a taxa de amostragem. Valor padrão é 0, ou seja. É
aplicado um loop no código assemble do PRU

```

Algoritmo 24: Outros métodos e atributos importantes da classe capture (Próprio autor).

3.2 Métodos de transmissão de dados

Na seção 3.1 vimos algumas maneiras de capturar os dados de um conversor analógico digital em alta velocidade. Destas maneiras, explicamos no início desse capítulo que utilizaremos, inicialmente, a última, por ser mais fácil de ser implementada, pois não precisaria de um ADC externo.

Agora é necessário retransmitir esses dados para o computador utilizando como camada física a USB. No capítulo 2 vimos que é possível fazer essa transmissão através de duas maneiras. A primeira delas é tirando proveito do *driver* USB-eth, que implementa a camada de rede e enlace na interface física da USB, e a outra é utilizando o protocolo UART em conjunto com TTL USB.

Nesta seção iremos focar nestas duas maneiras. Embora possam existir outras possibilidades, que infelizmente, não foram encontradas na literatura.

3.2.1 Utilizando somente TCP e UDP

Como falamos no segundo capítulo, nem todas as camadas da OSI são obrigatórias. A própria pilha de protocolos TCP/IP não implementa todas as 7 camadas deste padrão. A forma mais básica de enviar dados do BeagleBone Black para o computador é através dos protocolos TCP ou UDP, negligenciando a camada de aplicação e seus vários protocolos pré-definidos.

Na prática iremos implementar algum protocolo na camada mais elevada do padrão OSI, pois, ao mesmo tempo que estamos enviando dados através da camada de transporte estes dados devem existir uma regra para que o computador entenda o que o BeagleBone Black esteja enviando.

Por padrão, o Python oferece suporte aos protocolos TCP e UDP através da biblioteca *socket*. Antes de começar a falar de como implementar estes protocolos vamos falar um pouco de cada um deles separadamente.

A. Protocolo TCP

Como vimos no capítulo 2, o protocolo TCP surgiu primeiro e é mais complexo que o UDP. O protocolo TCP é famoso por ser orientado a conexão, ou seja, ambos os lados (O BeagleBone Black e o computador) precisam enviar pacotes utilizando o mesmo protocolo antes de iniciar a transferência de dados. Esta etapa é conhecida como estabelecimento da conexão (56).

Para ficar mais claro, imagine que o computador queira se comunicar com o BBB através do protocolo TCP. Então ele deve enviar um pacote (Ou segmento, como são conhecidos os pacotes TCP) dizendo que quer se comunicar com o *computer board*. Este segmento é identificado através de um *bit*, ou *flag*, SYN (Que vem do inglês *synchronizing*) junto com a especificação do endereço e porta de ambos os lados, PC e BBB, que podem ser vistos no esquemático do cabeçalho deste protocolo figura 48.

Ainda neste mesmo segmento é enviado o ISN (*Initial Sequence Number*) como número de sequência, que seria algo como o instante inicial em que este pacote foi enviado. A ideia de utilizar números de sequências é a mesma ideia de numerar os pacotes no tempo, de tal forma que podemos identificá-los quais deles foram enviados primeiro.

Quando o BBB recebe o segmento com a *flag* SYN, ele responde com um segmento contendo a *flag* SYN e ACK (do inglês *acknowledgement*) ativas, com o número de sequência igual, mas com o número de confirmação¹⁵ igual a ISN + 1. Novamente o computador responde o BBB, agora apenas com a *flag* ACK ativa, também, com um incremento no número de sequência e o mesmo número de confirmação, assim, a conexão é estabelecida. Este tipo de estabelecimento de conexão é conhecido como *handshake* de 3 vias e é esquematizado na figura 49 (57).

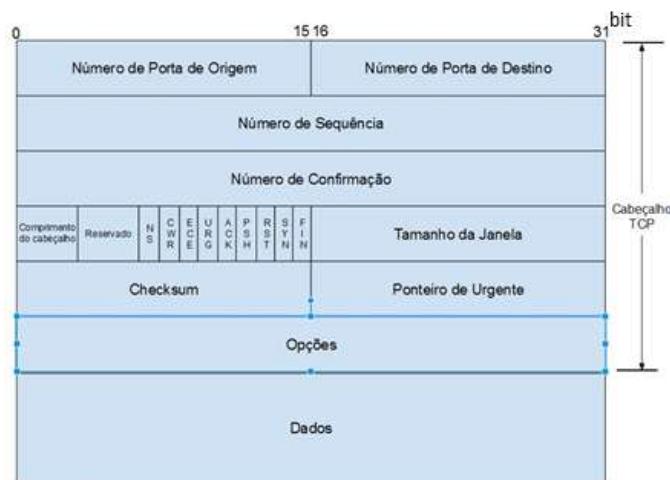


Figura 48: Descrição do cabeçalho do protocolo TCP (56).

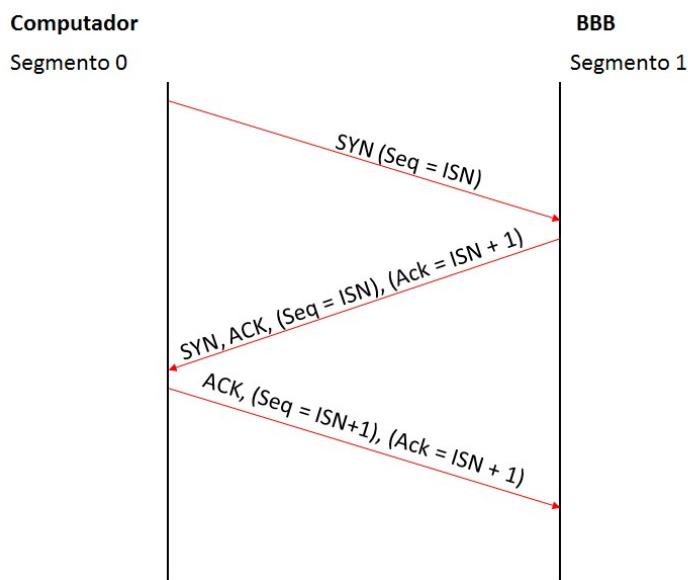


Figura 49: Descrição do cabeçalho do protocolo TCP (Próprio autor).

¹⁵ O número de confirmação segue o mesmo raciocínio do número de sequência, porém, refere-se ao lado do destinatário.

Se nada de errado ocorrer no processo de conexão os dois lados podem trocar dados livremente entre si. Caso contrário, o processo deve ser iniciado novamente. O protocolo TCP permite a transferência de pacotes de até 64kb, mas é comumente utilizado 1500 bytes, que é o tamanho máximo do *frame* Ethernet. Assim que o processo de transferência de dados termina, deve-se finalizar a conexão, que é feita de maneira semelhante ao processo de estabelecimento, porém, utilizando a *flag* FIN para isso.

Além do processo de estabelecimento e finalização de conexão o protocolo TCP tem mais outros campos nos seus segmentos. Dentre eles, o *checksum*, que permite verificar a integridade do cabeçalho e dos dados enviados, caso não haja coerência entre o *checksum* e o segmento, este é descartado e é requisitado um novo.

De uma maneira geral o protocolo TCP é confiável, pois a cada segmento enviado pelo transmissor, existe uma resposta do destinatário confirmando se o dado foi enviado corretamente. Além disso, existe a verificação de erros *checksum*, que permite verificar a integridade dos bits como um todo. E há uma numeração de segmentos, ou seja, se segmentos chegarem em ordem diferente podem ser reordenados de acordo com o período que foi enviado. Da mesma forma, com a base de tempo é possível eliminar pacotes repetidos. Por fim, como existe a necessidade de confirmação, o fluxo de dados é enviado de acordo com a capacidade de ambos os lados, ou seja, a próxima conexão só é feita assim que a anterior é finalizada.

B. Protocolo UDP



Figura 50: Descrição do cabeçalho do protocolo UDP (58).

No protocolo UDP não há a necessidade de estabelecer a conexão nem finalização. Além disso, os dados enviados não são verificados e nem há um ordenamento dos dados enviados. De uma certa forma, este protocolo é bem

menos confiável que o protocolo TCP, mas é muito mais rápido e mais simples. Ele é bastante utilizado em *streaming* de dados, como os vídeos do Youtube, jogos online, dentre outros. Na figura 50 podemos ver o quanto mais simples é o cabeçalho do protocolo UDP em relação ao TCP. O único recurso existente é o *checksum*, que serve apenas para descartar os pacotes inválidos.

C. Implementação e resultados

Vimos com detalhes como funcionam os protocolos TCP e UDP, suas vantagens e desvantagens. Para esta aplicação, como iremos fazer um *streaming* de dados capturados de um ADC, talvez seja melhor utilizar diretamente o protocolo UDP. Para fazer a conexão precisamos criar dois programas um localizado no BeagleBone Black (Servidor) e outro localizado no computador (Cliente). Inicialmente fizemos ambos os códigos em Python, tanto no Beaglebone quanto no computador. Como o computador tinha o Windows instalado, foi necessário instalar o interpretador Python primeiro, que está disponível em (59).

```
import socket
import time

HOST = '192.168.174.97' # Endereço IP do computador na rede
PORT = 5005 # Porta que o Servidor esta
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
dest = (HOST, PORT)
msg = 'Olá mundo'

while True:
    sock.sendto (msg, dest)
    time.sleep(2)
    sock.close()
```

Algoritmo 25: Código para comunicação UDP do lado do BBB (Próprio autor).

A partir daí, foi criado os algoritmos 25 e 26 e executados em ambas as máquinas. Os algoritmos são bem simples, um deles envia continuamente a mensagem “Olá mundo”, enquanto o outro recebe essa mensagem e imprime na tela do IDLE.

No primeiro momento não houve conexão entre ambas as partes. Para verificar, utilizou-se o comando **ping** no terminal do BBB para ver se era possível estabelecer uma conexão entre ambas as partes. O BBB não conseguia enviar dados para o computador. Foi feito o mesmo do lado do computador, desta vez,

utilizando o **cmd.exe** e houve resposta do *single board computer*. Assim foi descoberto que o problema estava no Windows, que recusava a conexão do BeagleBone. Para tentar solucionar este problema tentou-se desativar o Firewall do Windows 10 (60) e tentou reutilizar o comando **ping** no terminal do BBB. Desta vez deu certo, o computador recebia os dados da pequena placa. Tentou-se executar os algoritmos 25 e 26 novamente e obteve-se resultado (Figura 51).

```
import socket
HOST = "" #Quando vazio indica que pode receber de qualquer IP
PORT = 5005
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
orig = (HOST, PORT)
sock.bind(orig)
while True:
    data, addr = sock.recvfrom(1024) # Tamanho do buffer, ou seja, quantidade de dados
    #enviados em uma conexão (1024 bytes)
    print data
sock.close()
```

Algoritmo 26: Código para comunicação UDP do lado do PC (Próprio autor).

```
*Python 3.5.2 Shell*
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\lucas\PycharmProjects\untitled\teste.py ======
Ola Mundo
Ola Mundo
Ola Mundo
```

Figura 51: Resultado do teste utilizando o protocolo UDP (Próprio autor).

A partir daí, manteve-se o aplicativo Python no BeagleBone Black e tentou-se criar um código em MATLAB para substituir o algoritmo 26. Infelizmente, o MATLAB não conseguiu conversar com o código Python do BeagleBone Black, mesmo, posteriormente, configurando ambos os lados para enviar os dados em binário e em ASCII (*American Standard Code for Information Interchange*). Não se sabe dizer o que aconteceu, mas provavelmente o processo de codificação do MATLAB é diferente do Python, ou então, o Windows, por algum motivo, mesmo com o Firewall desligado, não permitiu a conexão entre ambas as partes. Como

não foi possível se comunicar com o MATLAB, e ainda, houve a necessidade de desativar o Firewall do Windows, tornando-o mais vulnerável, descartou-se a possibilidade de utilizar TCP ou UDP sem um protocolo padrão na camada de aplicação.

3.2.2 Utilizando o protocolo HTTP

Como falamos no item anterior, não foi possível se comunicar com o MATLAB através do UDP “puro”, e ainda, houve a necessidade de desativar o Firewall do Windows, deixando-o vulnerável. Para tentar contornar estes problemas, imaginou-se utilizar algum dos protocolos de aplicação, disponíveis na pilha de protocolos TCP/IP.

Um dos meios mais famosos e bastante utilizado na literatura é a comunicação através do HTTP. O próprio BBB cria uma conexão HTTP na porta 80, ao conectar-lo pela USB, que pode ser acessado pela página <http://192.168.7.2>. O próprio Cloud IDE (Figura 16) utiliza o protocolo HTTP para se comunicar com o BBB, na porta 3000. Observe que em ambos os casos se utiliza navegadores no lado do computador, na próxima seção veremos como isso funciona.

A. Como funciona os navegadores e a web

Vimos no capítulo 2 que uma conexão é estabelecida quando existe dois processos comunicantes, o servidor e o cliente. No nosso caso, o cliente é aquele processo que fica do lado do computador, podendo ser um navegador, o MATLAB ou qualquer outro que estabeleça uma conexão com o *single board computer*. O processo do outro lado é chamado de servidor. Existem diversos programas que funcionam como servidores, como Apache (61), um servidor que interpreta a PHP e o próprio *node.js*, que interpreta códigos em JavaScript.

Observe que os servidores comumente são interpretadores de determinadas linguagens. Esse formato se dá, pois no lado do servidor é aonde está o desenvolvedor e, por isso, o uso de linguagem de *script* ou interpretadas é bem comum, visto que, deste jeito é muito mais fácil e produtivo criar novos programas ou alterá-los. O PHP, por exemplo, é uma linguagem de *script* criada exclusivamente para rodar em servidores *web*. Além dessas linguagens existem muitas outras que podem rodar no lado do servidor como Java, ASP.NET, Python,

Ruby e muitas outras. Todas essas linguagens interpretadas são chamadas de *server-side language*, ou linguagens do lado do servidor em inglês (62).

No lado do cliente a coisa fica um pouco mais simples. O *Hypertext Transfer Protocol* é um protocolo feito, basicamente, para transferir as páginas *web* de um servidor para o cliente. Neste caso o cliente geralmente é um navegador web e todos eles seguem os padrões W3C (*World Wide Web Consortium*), que é uma das principais organizações que padroniza a *internet* e o padrão *Word Wide Web*, atualmente.

No padrão atual as páginas webs se dividem em 3 partes o HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) e o JavaScript. No início da *web* só existia os arquivos HTML que, por coincidência ou não, se você olhar para o acrônimo HTTP, verá que a função desse protocolo é transferir arquivos, ou páginas, HTML. Durante os anos o nome do protocolo continuou o mesmo, mas devido a necessidade de páginas mais ricas e interativas surgiram novas linguagens para auxiliar o HTML. Assim, o CSS ficou responsável pela parte de estilo e *design* e o JavaScript responsável pela parte lógica.

Observe HTML e CSS não são linguagens de programação. HTML é uma linguagem de marcação de texto, semelhante ao Latex e ao XML, ela só indica onde aparece os elementos, como botões, links, textos, gráficos dentre outros, através das *tags*. O CSS é uma linguagem de estilo, portanto, é nela que descrevemos o formato daquele botão, do *link*, do texto e os outros elementos das páginas. Por fim, o JavaScript, das três, é a única linguagem de programação propriamente dita. É ela quem vai dizer o que acontece se você aperta um botão ou como um determinado gráfico se comporta com o tempo.

Na verdade, o protocolo HTTP continua com a função principal de transferir arquivos HTML. O que acontece é que, CSS e JavaScript são extensões do HTML. Não é à toa que em é possível escrever código CSS com a tag `<style>` e JavaScript com a tag `<script>`. Com outras *tags* podemos carregar arquivos externos de *script* ou de estilo, semelhante as bibliotecas nas linguagens de programação. A divisão em arquivos separados é feita apenas para facilitar o desenvolvimento e o entendimento do código.

```

<!DOCTYPE html>
<html>
    <!-- Cabeçalho -->
    <head>
        <title>Interface de controle do ADC</title>
    ...
    <!-- Carrega arquivo externo contendo código css -->
    <link href="static/style.css" rel="stylesheet" type="text/css">
    <!-- Carrega arquivo contendo código javascript -->
    <script language="javascript" type="text/javascript" src="static/jquery.flot.min.js">
    </script>
    <!-- Código JavaScript escrito no mesmo arquivo HTML -->
    <script type="text/javascript">
        $(function() {
            var adcExternal = {{ adcExternal }};
            ...
            function constructData(){
                series = [];
                if(adcExternal == true){
                    seriesExternal = [];
                }
                for (i=0;i<=y_axis.length;i++){
                    series.push([x_axis,y_axis[i]]);
                    if(adcExternal == true){
                        seriesExternal.push([x_axis,y_axis[i]]);
                    }
                    x_axis++;
                }
            }
            ...
        });
    <body>
        <!-- Inicio da página propriamente dita -->
        <div id="header">
            <h2>Toolbox</h2>
        </div>
        ...
        <!-- Tag que indica tabela -->
        <table>

```

Algoritmo 27: Fragmentos de código do **toolbox.html** (Próprio autor).

Além dessas *tags*, o HTML5, a versão atual da linguagem suporta muitas outras *tags*, incluindo áudio e vídeo. Mais detalhes sobre essas três linguagens podem ser vistos em (63). Para ficar um pouco mais claro, o algoritmo 27 mostra partes do código utilizado para criar o aplicativo *web* para visualizar e controlar o ADC (Figura 55).

Analizando todo o contexto, no protocolo HTTP a função das linguagens *server-side* são gerar os arquivos HTML customizados. Por exemplo, quando você entra no seu perfil do Facebook, o servidor gera uma página única do seu perfil. Essa página é atualizada com o tempo através do JavaScript, mais precisamente do AJAX (*Asynchronous JavaScript And XML*), uma tecnologia capaz de atualizar parte de uma página HTML sem a necessidade de recarregar a página, assim a página é atualizada dinamicamente. Além do AJAX, temos os *web sockets*, também suportados por JavaScript e permitem fazer uma ponte entre servidor e cliente.

Graças as essas tecnologias os aplicativos *web* podem ser bastante dinâmicos e com capacidade de trocar de informações constantemente entre ambas as partes. Hoje em dia existem jogos online, aplicativos de *chat* e *streaming* de áudio e vídeo rodando em navegadores, apenas com as 3 linguagens citadas acima. Vamos utilizar de toda essa flexibilidade da web para criar uma interface homem-máquina entre o computador e o BBB que está conectado ao ADC.

B. Microframework Flask

Na seção 3.1.3 vimos o `beaglebone_pru_adc` como uma solução em Python para captura de dados do ADC em alta velocidade. Assim, seria bastante útil se existisse a possibilidade de transferir os dados armazenados nas tuplas do aplicativo Python para um aplicativo *web*.

Essas soluções existem e uma delas é o Flask é um *microframework* para Python que é bastante utilizada em projetos com Raspeberry Pi (64). *Framework* é o termo utilizado para o conjunto de ferramentas que facilitam o desenvolvimento de aplicativos. No caso, o Flask, é o conjunto de ferramentas para a criação de aplicativa web. O termo “micro” refere-se ao seu caráter minimalista, ou seja, que consome poucos recursos. Podemos instalar a *microframework* através do comando **`pip install flask`**.

O Flask se divide em duas partes: o Werkzeug é um WSGI (*Web Server Gateway Interface*), que permite interligar o código Python e o aplicativo *web* (65) e o Jinja 2, uma linguagem de programação web que trabalha junto com o HTML (66). O algoritmo 29 mostra alguns fragmentos do código utilizado para criar o servidor que interliga os dados amostrados do ADC utilizando o beaglebone_pru_adc e a página web. Na primeira linha do algoritmo carregamos vários módulos da biblioteca *flask*. Logo na linha seguinte criamos o objeto *app*, filho de Flask, que representa o nosso aplicativo web. Depois roteamos alguns endereços para o nosso aplicativo utilizando o método *route*. E por fim, na última linha, executamos o servidor com o método *run*.

O método *route* permite executar uma função que deve retornar o resultado da página web. Se essa página for gerada utilizando o Jinja como linguagem de programação, ou ainda, uma página HTML externa deve-se criar uma instância do módulo *render_template*, o módulo do Jinja. Observe que é passado como parâmetro a página HTML e as variáveis que devem ser transportadas do Python para o HTML.

O módulo *request* permite a aquisição de dados da página HTML disponibilizada no cliente. Isso é importante pois, caso o usuário mude alguma coisa na interface de configuração. Essa mudança deve ser enviada ao servidor e transformada em uma variável Python através do método *request.form*.

```
<title>{% block title %}{% endblock %}</title>
<ul>
{% for user in users %}
<li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

Algoritmo 28: Exemplo de código usando Jinja 2 (66).

```

from flask import Flask, render_template, request, jsonify
...
app = Flask(__name__)
...
@app.route('/')
def toolbox():
    web_data = []
    for x in thread1.captureSamples[thread1.offset:len(thread1.captureSamples)]:
        web_data.append(int(x))
    if (adc_external == True):
        return render_template('toolbox.html', web_data = web_data, noSamples = thread1.
numSamples, delayADC = thread1.delayCapture, offsetADC = thread1.offset, speedADC =
thread1.delayADC, interface_refreshType = thread1.interface_refreshType,
interface_frameRate = thread1.interface_refreshTime, windowSamples = thread1.
captureWindow, ytype = thread1.ytype, ymin = thread1.ymin, ymax = thread1ymax,
adcExternal = True, web_data_external = thread2.captureSamples, noSamples_external =
thread2.numSamples, delay_capture_external = thread2.delayCapture, ymin_external =
thread2.ymin, ymax_external = thread2ymax, ytype_external = thread2.ytype)
    else:
        ...
@app.route('/senddata', methods=['POST'])
def send_data():
    web_data = []
    for x in thread1.captureSamples[thread1.offset:len(thread1.captureSamples)]:
        web_data.append(int(x))
    if (adc_external == False):
        return jsonify(web_data)
    ...
@app.route('/config', methods=['POST'])
def toolbox_config():
    thread1.numSamples = int(request.form['adc_captureSamples'])
    thread1.delayCapture = int(request.form['adc_captureDelay'])
    ...
if __name__ == '__main__':
    app.run('0.0.0.0')

```

Algoritmo 29: Fragmentos do código utilizado como servidor utilizando o Flask (Próprio autor).

```

function onGetData(){
    $.ajax({
        url: '/senddata',
        type: "POST",
        dataType: "json",
        success: onDataReceived,
        error: function(error){
            console.log(error);
            $('#submit-result').text('Erro ao atualizar os dados.');
            $('#submit-result-box').show(0);
        }
    });
}

...
$('#config-button').click(function{
    ...
    $.ajax({
        url: '/config',
        data: $('form').serialize(),
        type: 'POST',
        success: function(response){
            console.log(response);
            $('#submit-result').text('Configurado com sucesso.');
            $('#submit-result-box').show(0);
        },
        error: function(error){
            console.log(error);
            $('#submit-result').text('Erro ao configurar.');
            $('#submit-result-box').show(0);
        }
    });
});

```

Algoritmo 30: Fragmentos de código do **toolbox.html**, exemplificando o uso de AJAX (Próprio autor).

Já o módulo *jsonify* faz o caminho contrário, envia os dados do servidor, no caso a tupla de dados do método *oscilloscope_data* (Ver algoritmo 23) em objetos reconhecidos pelo JavaScript, esse objeto em questão é um JSON (*JavaScript Object Notation*). Assim cada instância de *jsonify* é convertida em objetos reconhecidos pelo JavaScript, desde que exista seu equivalente nesta linguagem.

Como as tuplas não existem em JavaScript é necessário copiar a tupla para um *array* e utilizar esse *array* como parâmetro da instância. Com este módulo é possível atualizar a página em tempo real, utilizando AJAX para acessar o endereço “/senddata” a cada alguns milissegundos.

Como já foi dito, o módulo `render_template` é responsável pelo Jinja 2. Código Jinja deve ser escrito no mesmo arquivo do HTML utilizando a notação mostrada no algoritmo 28. É possível utilizar diversos blocos como funções, laços de repetições e *ifs*, mas nesta aplicação não foi explorado esses blocos, apenas as variáveis, que podem ser declaradas utilizando a notação `{{ variavel_python }}`. A primeira linha do código JavaScript no algoritmo 27 mostra um exemplo, onde foi exportada a variável que contém a tupla com os dados capturados do ADC.

É importante salientar que o Jinja é utilizado apenas para gerar a página HTML, que é carregada uma única vez, no momento que o usuário acessa o endereço da página web. As outras transações de dados são feitas através dos módulos `request` e `jsonify`, que utilizam AJAX. As requisições AJAX são feitas em código JavaScript, como mostra no algoritmo 30, sendo a primeira a requisição JSON dos dados do ADC e a segunda relacionada as opções de configuração da interface.

C. JQuery

Como vimos nas seções anteriores, a *web* evoluiu tanto, que hoje temos páginas ricas e dinâmicas. Durante esta evolução surgiram diversas bibliotecas e *frameworks* para facilitar a programação nas linguagens suportadas pelos navegadores. Entre estas bibliotecas, a mais utilizada é a JQuery, criada para simplificar os códigos JavaScript com o lema “*write less, do more*” (67). Atualmente a biblioteca têm a capacidade de manipular elementos HTML e CSS, criar efeitos, suporte a AJAX e outras utilidades.

JQuery se tornou tão popular que se tornou um pré-requisito nas mais diversas *frameworks*. A forma de programar em JQuery é um pouco diferente do JavaScript puro. Enquanto este último se assemelha a linguagens de programação tradicionais, onde é necessário declarar primeiro as variáveis, classes, métodos e atributos. No JQuery cada elemento HTML ou CSS é um objeto de uma determinada classe. Então, a partir deste objeto, podemos utilizar os mais diversos métodos, eventos e atributos, ou ainda, criar novos relacionados a estes. Lembrando que podemos utilizar desta característica em cascata, e

ainda, o conceito de herança também se aplica também aos elementos HTML ou CSS, por exemplo, as linhas e colunas de uma tabela HTML são elementos filhos da tabela. O algoritmo 31 mostra a sintaxe da linguagem JQuery. Observe que os códigos JavaScript nos algoritmos 27 e 30 foram escritos em JQuery. É válido lembrar que os navegadores não suportam JQuery nativamente, portanto é necessário baixar a biblioteca e incluí-la no cabeçalho do arquivo HTML.

```
$(elementoHTMLouCSS).MetodoOuEvento(... código ...)
```

Algoritmo 31: Sintaxe do JQuery (Próprio autor).

D. Flot

Como falamos na seção anterior, existem várias *frameworks* e bibliotecas que auxiliam na criação de um aplicativo *web*. Na nossa aplicação iremos utilizar uma *framework* para plotar gráficos. Dentre as várias disponíveis em (68), foi escolhida a Flot, cuja a documentação está disponível em (69). O fragmento de código disponível no algoritmo 32 exemplifica o uso dessa biblioteca, que é executada através do método *plot*. Uma amostra do gráfico resultante com apenas um canal pode ser visto na figura 52.

```
...
$.plot("#placeholder", [
  {data: current_data, label: "ADC interno"},
  {data: current_dataExternal, label: "ADC externo",y_axis: 2}
], {yaxes: [
  {min: y_min, max: y_max},
  {position: "right"}
])
...

```

Algoritmo 32: Fragmento de código do ***toolbox.html***, exemplificando o uso da biblioteca Flot (Próprio autor).

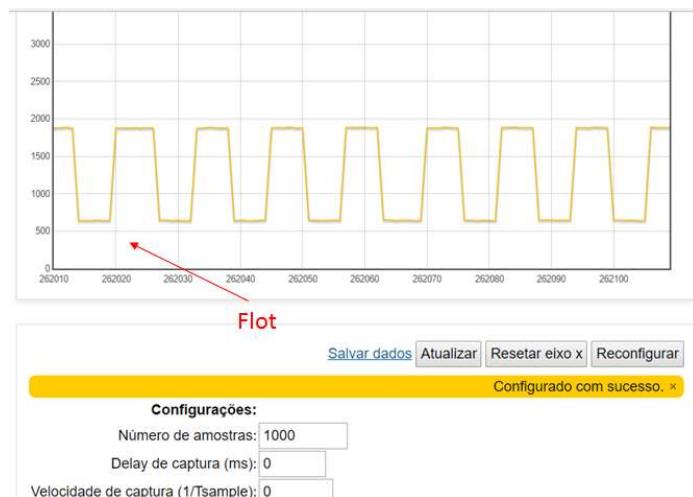


Figura 52: Programa adquirindo uma onda quadrada de 20kHz (Próprio autor).

E. Módulo Threading

Um dos requisitos dos servidores é a capacidade de trabalhar com múltiplas requisições ou processos em paralelo. No nosso caso não será diferente, como queremos capturar informações dos diferentes ADCs e depois enviá-las para o computador é interessante utilizar a programação paralela para esse fim.

Em ciência da computação *thread* é a menor sequência de instruções pré-programadas que podem ser gerenciadas de forma independente pelo sistema operacional. A existência de *threads* é o que permite o sistema operacional executar múltiplos processos simultaneamente, mesmo que o processador não mais de um núcleo físico. Threads é a mesma analogia que fazemos ao dizer que o Linux alterna entre as rotinas rapidamente de tal forma que, para nós humanos, pareça que todos os processos sejam simultâneos (Figura 53).

Em alguns sistemas a execução de *threads* simultâneas é real, pois desde a década passada os processadores passaram a ter mais de um núcleo físico. Atualmente existem processadores ARM com 10 núcleos físicos, como o Helio x20 (70). Nestes processadores os vários threads podem ser escalonados para serem executados em núcleos diferentes do processador, desde que sejam processos diferentes, ou ainda, o único processo seja programado para isso.

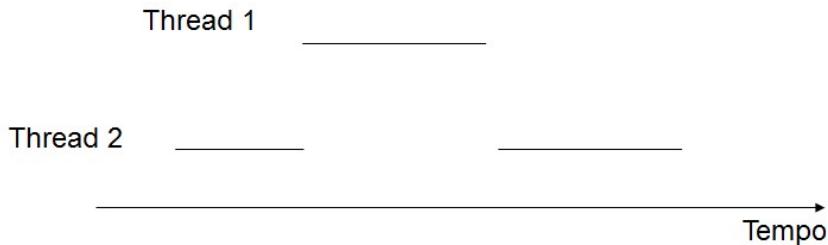


Figura 53: Sistema operacional executando várias *threads* ao longo do tempo (Próprio autor).

O módulo Threading é uma das bibliotecas Python que permitem que o mesmo processo rode rotinas paralelas em Threads separados. Para trabalhar com esse módulo, devemos definir uma classe filha do primeiro. Por padrão a classe pai do módulo Thread tem rotinas padrões para o método construtor e alguns métodos como *run*, *start*, *joining* e outros. Usualmente não queremos que a classe filha tenha o mesmo comportamento que a classe pai, definida pelo módulo. Para isso devemos criar métodos com o mesmo nome na classe filha. Assim aos métodos da classe pai são sobreescritos, de tal forma que, ao executar a instância da classe filha os métodos herdados da classe pai são ignorados, em favor dos métodos da classe filha.

Nas nossas rotinas precisamos sobreescrivemos o método construtor, `__init__`, e o método *run*, este último responsável por executar o código definido para a *thread*. Observe que cada instância da nova subclasse é considerada como uma nova *thread*. A subclasse é necessária apenas porque queremos redefinir como serão as rotinas desses novos objetos. O algoritmo 33 exemplifica o uso da subclasse para sobreescrivemos o método construtor e o método *run*. No método construtor definimos os parâmetros da nossa classe e no método *run* definimos as rotinas que a instância deverá executar ao iniciar a *thread*. Neste caso foi definido que a rotina irá executar o método *captureData* indefinidamente. Esse último está relacionado a captura de dados do ADC. No final do algoritmo criamos a instância da subclasse e utilizamos o método *start* para inicializar as threads. A partir daí as threads irão executar as suas rotinas de forma paralela ao código principal. Na figura 54 mostra o esquema de como as instâncias da classe *captureThread* se comportam em relação ao programa principal.

```

...
class captureThread(threading.Thread):
    def __init__(self, numSamples_ = 10000, delayCapture_ = 0, delayADC_ = 0, captureWindow_
 = 100, oneTime_ = False, offset_ = 10, interface_refreshTime_ = 100,
interface_refreshType_ = 0, ymin_ = 0, ymax_ = 4095, ytype_ = 1):
        threading.Thread.__init__(self)

...
def run(self):
    print("Iniciando captura do ADC interno...")
    while(self.oneTime == False):
        self.captureData()
    ...

thread1 = captureThread()
thread1.start()

...
thread2 = captureThread()
thread2.start()

```

Algoritmo 33: Fragmento de código de ***main.py***, exemplificando o uso de uma subclasse para sobrescrever métodos já predefinidos (Próprio autor).

Se compararmos essa solução com o programa de Molloy, na seção 3.1.2. Esta consome menos memória RAM, pois os dados podem ser fragmentados em pacotes menores, já que existe um thread paralela pegando partes desse pacote e transferindo para o computador, assim o pacote pode ser renovado com uma frequência maior, na faixa de alguns décimos de segundos. É importante enfatizar que todas as *threads* pertencem ao mesmo processo.

Na teoria, o esquema da figura 54 todas as *threads* seriam executadas simultaneamente. Mas na prática isso depende muito da arquitetura da máquina. Em máquinas com muitos núcleos e sistemas operacionais eficientes todos os Threading podem ser executados simultaneamente, porém o processador do BBB possui apenas um núcleo. Então, na prática, a execução de múltiplas *threads* é algo mais parecido com a figura 53, onde o sistema operacional simula a simultaneidade através da alternância entre as *threads* favorecido pelo alto IPC e frequência de *clock*.

Um problema comum ao utilizar threads é a sincronização e o acesso e a manipulação de dados simultaneamente. Para ficar mais simples o entendimento, imagine que a *thread* do programa principal necessita pegar os dados da thread1

para enviar para a página web e nesse momento a `thread1` está escrevendo na memória os resultados que ela adquiriu do ADC. Se essas duas rotinas acontecerem simultaneamente pode ocorrer que a *thread* do programa principal capture dados corrompidos ou ainda interfira na memória que os cálculos, portanto estes dados não podem ser manipulados separadamente.

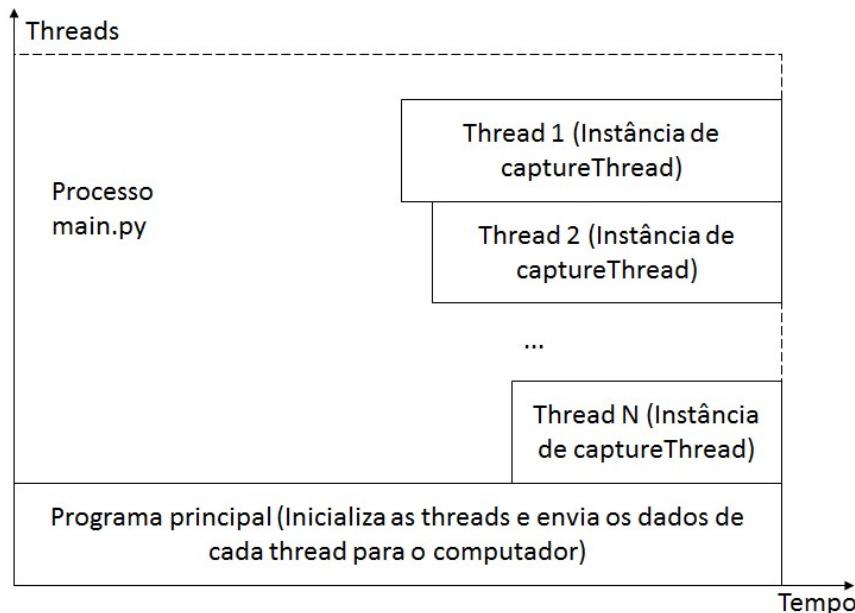


Figura 54: Esquema do módulo Threading executando várias *threads* em paralelo com o programa principal (Próprio autor).

A maneira correta para resolver este problema é a *thread* do programa principal mandar um comando para congelar a `thread1` para depois acessar os dados na memória e, depois do acesso ter concluído, liberar a `thread1`. Um outro exemplo em que há a necessidade de sincronização de *threads* é quando uma *thread* depende do resultado da outra *thread*, neste caso a *thread* que quer adquirir o resultado deve esperar a outra *thread* ser finalizada. Por causa das dependências entre as *threads* é raro encontrar métodos de execução de rotinas 100% simultâneo e isso pode ser um problema em situações em que é realmente necessário o uso de computação simultânea. Para tentar contornar estes problemas existem métodos de sincronização mais avançados como semáforo e prioridade de *threads* (71), infelizmente não houve tempo suficiente para testá-los.

F. Implementação e resultados

O aplicativo *web* consiste em um mostrador gráfico, gerado pelo Flot, um formulário com diversos campos e um conjunto de botões.

Botão	Função
Salvar	Salva os dados capturados em um arquivo de texto no formato .csv
Atualizar	Atualiza todos agora (Não necessário no modo contínuo)
Reset	Deleta os dados da memória
Reconfigurar	Reconfigura o sistema de acordo com os campos do formulário

Tabela 10: Relação de botões do aplicativo web (Próprio autor).

A intenção desse programa é testar a capacidade do BeagleBone Black em trabalhar com múltiplas *threads*, que representam os dados do ADC, utilizando o módulo Threading e ao mesmo que envia os dados de cada *thread* para a interface web, esta última representada pela *thread* do programa principal. Os dados foram capturados utilizando a biblioteca beaglebone_pru_adc.

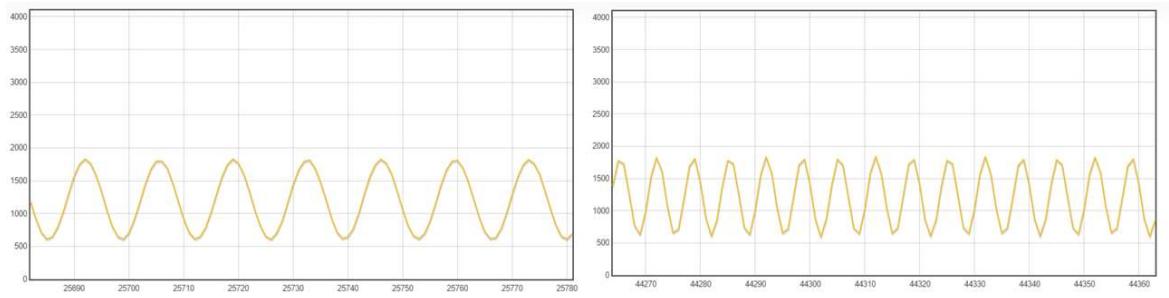


Figura 55: Resultado utilizando um aplicativo web com uma onda seinodal de 10kHz (Esquerda) e 20kHz (Direita) (Próprio autor).

Inicialmente foi feito o básico, a execução de uma única instância da classe captureThread em simultâneo com o programa principal responsável por rodar todos Flask e capturar os dados das *threads* em momentos predefinidos. Com essas duas *threads* rodando em paralelo não houve sinais de perda de performance ou perda de dados. Durante os testes foi possível enviar o buffer de 10 mil amostras a cada 0,1 segundos. Acima disso, houve gargalos na comunicação. É válido lembrar que os dados foram enviados utilizando codificação UTF-8 (*8-bit Unicode Transformation Format*), que utiliza 4 bytes por caractere, então existe espaço de sobra para otimização, visto que cada dígito ocupa até 4 caracteres. Se for utilizado JSON em binário, ou BSON (*Binary JSON*) podemos enviar uma amostra em 2 bytes, desconsiderando a parte do

cabeçalho. A figura 55 mostra o resultado de uma onda senoidal de 10kHz e 20kHz.

Campo	Função
Número de amostras	Tamanho do <i>buffer</i> de amostras (Padrão 10 mil)
<i>Delay</i> de captura (ms):	Intervalo de tempo entre um <i>buffer</i> e outro (Padrão 0)
Velocidade de captura (1/Tsample)	Período de amostragem em múltiplos da frequência original do ADC (Padrão 0 – desativado)
Offset de captura (Amostras)	Plota apenas os dados a partir de X amostras do <i>buffer</i> (Padrão 0)
Tamanho da janela (Amostras)	Plota apenas os dados a partir de X amostras do <i>buffer</i> (Padrão 10 mil)
Taxa de atualização (ms)	Permite o gráfico atualizar automaticamente a cada X milissegundos ou apenas quando o usuário apertar o botão atualizar amostras
Eixo Y (min/max)	Plota o gráfico considerando o valor mínimo e máximo do ADC (Padrão “Automático” – Gera o eixo a partir dos valores mínimos e máximos disponíveis)

Tabela 11: Relação de campos configuráveis do programa (Próprio autor).

Feito o teste básico. Tentou adicionar mais uma *thread* de captura de dados. Essa segunda *thread* capturava os dados de um ADC externo (Figura 28). Existe uma biblioteca Python para Raspberry Pi e BBB disponível em (72), que foi utilizada para capturar os dados.

É válido lembrar que esta biblioteca não é otimizada para trabalhar em tempo real, nem mesmo este ADC é considerado de alto desempenho. A ideia de adicionar um módulo a mais era somente para testar como sistema trabalharia com mais de um deles. Infelizmente, após a adição desse módulo o sistema ficou instável. O servidor do Flask foi interrompido várias vezes e muitas vezes as *threads* não enviava todos os dados para o servidor. Provavelmente, o BBB não conseguiu dá conta de rodar as três threads simultaneamente.

G. HTTP e MATLAB

O navegador não é o melhor lugar para manipular dados, para isso existem os programas dedicados a isso, como o MATLAB. O MATLAB, a partir da versão de 2016, oferece algumas opções para trabalhar com HTTP como mostra no exemplo retirado do site MathWorks (Algoritmo 34). Como essa opção foi introduzida ao programa apenas recentemente não houve como testá-la, visto que era necessário atualizar o *software*. Mas segue como sugestão para trabalhos futuros.

```
import matlab.net.*
import matlab.net.http.*
r = RequestMessage;
uri = URI('http://www.mathworks.com/support/contact_us');
resp = r.send(uri);
resp.StatusCode
```

Algoritmo 34: Código em MATLAB para enviar uma requisição e receber a resposta da aquisição (73).

3.2.3 Utilizando a conexão UART

Embora o protocolo HTTP tenha sido capaz de enviar os dados sem gargalos aparente, ele não muito popular fora dos navegadores para a *web*. Isso faz com que ele seja uma opção de transferência de dados pouco interessante para dispositivos que necessitam trabalhar com aplicativos de engenharia, como o MATLAB e Excel (74). Por isso, é interessante testar a viabilidade de se utilizar um meio de conexão com maior aceitação outros aplicativos, dentre eles a conexão UART, ou serial, é bastante utilizada, principalmente porque a USB consegue emular este tipo de comunicação através da classe CDC.

Como falamos no capítulo 2, o BeagleBone Black tem 5 portas seriais, que estão disponíveis para livre uso do usuário através dos pinos P8 e P9 (Figura 56). E ainda, podemos utilizar o mesmo cabo TLL para fazer a comunicação entre o BBB e o computador através do *serial debug*. Tínhamos em mãos dois conversores USB serial chineses baseados nos modelos CH340B e PL2302HX, respectivamente. Segundo os *datasheets*, disponíveis em (75) e (76), ambos os conectores suportam *baud rate* de até 1228800, que dá aproximadamente 1Mbps.

Em relação aos *drivers* para Windows não tivemos problemas com o modelo CH340B, entretanto com o modelo PL2302HX foi necessário utilizar um *driver* antigo, pois a partir da versão 3.3.2.150, o *driver* detecta se o conversor

TLL é clone ou não, e como conversor adquirido é chinês, não funcionou com o *driver* que já vinha no Windows 10 (77).

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
UART4_RXD	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
UART4_TXD	13	14	GPIO_40	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_4	17	18	GPIO_5	GPIO_27	17	18	GPIO_65
UART1_RTSN	19	20	UART1_CTSN	GPIO_22	19	20	GPIO_63
UART2_RXD	21	22	UART2_RXD	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	UART1_RXD	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	GPIO_123	GPIO_32	25	26	GPIO_61
GPIO_125	27	28	GPIO_122	GPIO_86	27	28	GPIO_88
GPIO_121	29	30	GPIO_120	GPIO_87	29	30	GPIO_89
GPIO_120	31	32	VDD_ADC	UART5_CTSN+	31	32	UART5_RTSN
AIN4	33	34	GNDA_ADC	UART4_RTSN	33	34	UART3_RTSN
AIN6	35	36	AIN5	UART4_CTSN	35	36	UART3_CTSN
AIN2	37	38	AIN3	UARRS_RXD+	37	38	UARTS_RXD+
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	UART3_RXD	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Figura 56: Pinos do P8 e P9 que estão disponíveis para a comunicação UART (34).

Essa taxa de transmissão talvez não seja suficiente para enviar dados de 8 módulos simultaneamente, considerando que todos eles tenham as mesmas especificações do ADC do BBB. Mas podemos mudar um pouco a abordagem e utilizar o BeagleBone como um módulo e um *hub* USB como o DAQ da figura 31. Como o *hub* divide a largura de banda da conexão USB cada ADC teria disponível uma largura de banda de 1Mbps, que é mais do que o suficiente para esta aplicação.

A. Código

Embora seja perfeitamente possível ativar os pinos UART através de DTOs, podemos fazer o mesmo com a biblioteca **Adafruit_BBIO**. No Python existe um módulo conhecido como **serial** que, geralmente, já vem instalado por padrão no BeagleBone Black. Se por algum motivo essa biblioteca não estiver instalada, podemos instalá-la utilizando o comando **pip install pyserial**.

O funcionamento do módulo é bastante simples. Basicamente, precisamos abrir uma conexão, enviar dados e fechar essa conexão. O algoritmo 35 mostra um fragmento de código utilizado para enviar aqueles mesmos dados da instância de captureThread (Figura 54) através da conexão serial, utilizando o formato de texto em ASCII.

```

...
numChannel = int((thread1.numSamples-thread1.offset)/channelSize-1)
ser = serial.Serial(port = "/dev/tty02", baudrate=921600)
ser.close()
print(ser.name)
while True:
    currentCapture = thread1.captureSamples
    for i in range(0,numChannel):
        currentOffset = thread1.offset+i*channelSize
        lastOffset = currentOffset+channelSize
        captureData = ''
        for x in currentCapture[currentOffset:lastOffset]:
            captureData += str(x) + ' '
            captureData += '\n'
        ser.open()
        ser.write(captureData)
        ser.close()
...

```

Algoritmo 35: Fragmento de código utilizado para a comunicação serial utilizando Python (Próprio autor).



Figura 57: Codificação proposta para enviar dados do ADC por serial (Próprio autor).

Byte	Função
10101010	Inicio
10100101	Fim do cabeçalho
10110101	Fim
00+6 bits	1º byte da amostra
01+6 bits	2º byte da amostra

Tabela 12: Codificação dos dados para enviar por serial (Próprio autor).

É possível tornar esse formato mais compacto se utilizarmos uma codificação em binário. Cada pacote dessa codificação é representado na figura 57, que indica também a quantidade de bytes utilizado em cada etapa. A codificação é mostrada na tabela 12. Como o método *write* não suporta binários devemos incluir um módulo auxiliar para transformar a codificação da figura 57, o

binascii, assim podemos converter o binário em arquivo de texto utilizando o método `b2a_uu`. Já para converter inteiros em binário utilizamos a função `bin` e, por fim, para operar binários utilizamos os operadores `<<`, `>>` e `&`.

Do lado do MATLAB utilizamos métodos parecidos para fazer a comunicação serial, se utilizarmos o formato de texto podemos utilizar a função `fscanf` e para ler em formato binário, utilizamos `fread`. Por fim, para atualizar os gráficos em tempo real podemos utilizar as funções `set` e `drawnow`. O algoritmo 36 mostra fragmentos do código para MATLAB utilizando o método de texto.

```

function serialhello
porta_com = 'COM13';
porta_com2 = 'COM3';
current_buffer1 = get_data(porta_com);
current_buffer2 = get_data(porta_com2);

...
while true
    current_buffer1 = get_data(porta_com);
...
    set(p1, 'YData', current_buffer1);
    set(p1, 'XData', t1);
    current_buffer2 = get_data(porta_com2);
...
    pause(0.001);
    set(p2, 'YData', current_buffer2);
    set(p2, 'XData', t2);
    pause(0.001);
    drawnow;
end
...
function buffer = get_data(porta_com)
s = serial(porta_com);
set(s, 'BaudRate', 921600);
fopen(s);
temp_str = fscanf(s);
...
fclose(s);
pause(0.1);
end

```

Algoritmo 36: Fragmento de código utilizado para a comunicação serial utilizando MATLAB (Próprio autor).

B. Resultados

Primeiramente utilizamos o método de texto por ser mais simples de ser implementado. Com este método conseguimos enviar os dados com uma *baud rate* de 921600, utilizando o *hub* e os dois conversores seriais. Posteriormente utilizamos o método binário e conseguimos enviar dados com uma *baud rate* de

230400. A figura 58 mostra o sinal gerado pelo mesmo sinal de tensão contínua em ambos os canais, observa-se um pequeno ruído.

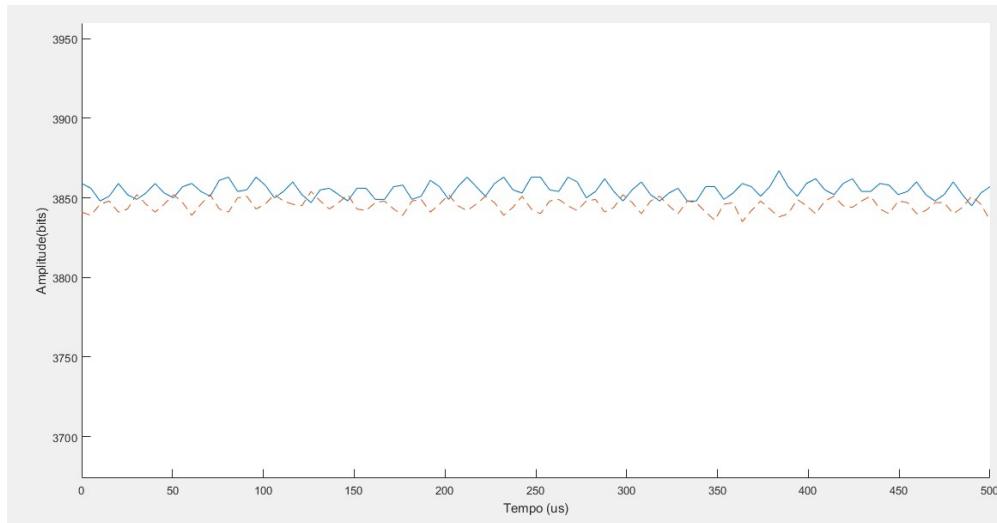


Figura 58: Resultado obtido utilizando a comunicação serial (Próprio autor).

Durante os testes o MATLAB não deu conta de plotar os gráficos em tempo real, então foi necessário adicionar um *delay* entre uma plotagem e outra. Além disso, houve muitas perdas de amostras. Neste caso poderíamos solucionar o problema enumerando cada amostra, ou um pequeno conjunto de amostra e utilizar uma comunicação bidirecional para enviar os dados semelhante ao protocolo TCP. Infelizmente, implementar o protocolo parecido com o TCP em serial não é algo tão simples e, por isso, não houve tempo para fazer os testes. Por isso, segue como sugestão para trabalhos futuros.

C. Utilizando HTTP

Uma outra solução para o problema de perda de amostras seria voltar novamente a utilizar o protocolo HTTP, com apenas um canal funcionando. Neste caso, precisamos mudar o IP de um dos BBB, como mostra no capítulo 2. Fizemos esse teste com os dois BeagleBone e ambos funcionaram de maneira semelhante aos testes anteriores.

É válido lembrar, que este teste por si só não soluciona o problema, pois não é nada prático ter cada canal em uma aba diferente do navegador. Se for utilizada essa solução deve-se trabalhar na comunicação com o MATLAB ou criar um aplicativo próprio em C++, ou então em linguagens da web.

4 Conclusão

Este trabalho se dividiu em duas partes: as formas de aquisição de dados do ADC e as formas de transferência destes dados para o computador. Na primeira parte a solução mais interessante foi a de Derek, que infelizmente não foi implementada devido à complexidade do código e por não ter em mãos o ADC utilizado por ele.

Na segunda parte não existiam soluções prontas, portanto houve a necessidade de implementá-las em código. Durante a implementação e testes houve diversos problemas, alguns deles foram propostas soluções e outros ficaram em aberto, principalmente em relação as técnicas de *multithreading*. Não se sabe se a limitação de utilizar muitos *threads* se dá pelo BeagleBone Black ter apenas um núcleo ou porque a linguagem Python e o método de sincronização de threads foi ineficiente.

Em resumo, se for resolvida a incógnita do *multithreading*, o BeagleBone Black pode ser utilizado como um DAQ. Porém, para dominá-lo é necessária muita experiência em e-Linux e ciência da computação de uma forma geral, pois o seu maior trunfo, o PRU, não é tão simples de se programar. Além disso, não existem tantas referências para este *board computer* quanto os seus concorrentes.

Talvez seja interessante partir para outra solução, mas para aqueles que querem utilizar o BeagleBone Black, esta monografia pode ser uma boa introdução, visto que, não foram encontrados trabalhos na língua portuguesa relativos a esta placa.

5 Trabalhos futuros

Dentre as várias soluções que vimos encontramos diversos problemas. Alguns deles foram propostas soluções que infelizmente não deu tempo de ser implementada. Assim podemos listar algumas tarefas que poderão ser feitas em trabalhos futuros:

- Utilizar o protocolo HTTP no MATLAB;
- Criar uma solução de correção de erros e enumeração de pacotes com a interface serial;
- Implementar o código utilizando linguagens de programação mais eficientes, como o C++;
- Implementar métodos mais avançados de sincronização de thread.

Além disso, é válido experimentar a solução de Derek para a captura de dados. Além de ser feita em C, que a torna bastante eficiente, é possível aumenta-la para uma quantidade maior de canais, ou seja, ADCs externos, sem consumir muitas portas do BBB.

Já em relação aos BBB e o *hub*, pode-se tentar uma solução mais barata, visto que os BeagleBone Black são muito caros para serem utilizados como módulos. Diversos microcontroladores com suporte a UART podem ser utilizados no lugar.

Por fim, uma outra solução seria partir para outro dispositivo semelhante ao BeagleBone Black. O BBB é bem interessante para esta aplicação, mas exige muita experiência e há pouca documentação em relação a outras soluções como Raspberry Pi e Arduino.

Bibliografia

- 1 MEASUREMENT COMPUTING CORPORATION. Data Aquisition Handbook, Norton, 2012. Disponivel em: <<http://www.mccdaq.com/pdfs/anpdf/Data-Acquisition-Handbook.pdf>>. Acesso em: 26 de setembro 2016.
- 2 PEREIRA, A. L. **O que é PCI-Express?**, 2012. Disponivel em: <<http://www.tecmundo.com.br/hardware/1130-o-que-e-pci-express-.htm>>. Acesso em: 26 de setembro 2016.
- 3 NI. **Aquisição de dados**, 2016. Disponivel em: <<http://www.ni.com/data-acquisition/pt/>>. Acesso em: 25 de Setembro 2016.
- 4 BANZ, M. **How Arduino is open-sourcing imagination**, 2012. Disponivel em: <https://www.ted.com/talks/massimo_banzi_how_arduino_is_open_sourcing_imagination>. Acesso em: 04 de outubro 2016.
- 5 BLAKEBEVIN. **TPower Laces- the Auto lacing shoe**, 2010. Disponivel em: <<http://www.instructables.com/id/Power-Laces-the-Auto-lacing-shoe/>>. Acesso em: 30 de setembro 2016.
- 6 MACHADO, J. **Raspberry Pi**: computador de 50 reais pode revolucionar a informática, 2012. Disponivel em: <<http://www.tecmundo.com.br/hardware/23175-raspberry-pi-como-um-computador-de-50-reais-pode-revolucionar-a-informatica.htm>>. Acesso em: 04 de outubro 2016.
- 7 DICOLA, T. **Embedded Linux Board Comparison**, 2014. Disponivel em: <<https://cdn-learn.adafruit.com/downloads/pdf/embedded-linux-board-comparison.pdf>>. Acesso em: 04 de outubro 2016.
- 8 COLEY, G. **BeagleBone Black System Reference Manual**. A5.2. ed. Dallas: Texas Instruments, 2013. Disponivel em: <https://cdn-shop.adafruit.com/datasheets/BBB_SRM.pdf>. Acesso em: 18 de outubro 2016.
- 9 MOLLOY, D. **Exploring BeagleBone**: Tools and Techniques for Building with Embedded Linux. [S.I.]: Wiley, 2014. Disponivel em: <<http://www.exploringbeaglebone.com/>>.
- 10 KOEN. **The Ångström Distribution**, 2015. Disponivel em: <<http://wp.angstrom-distribution.org/introduction/>>. Acesso em: 05 de outubro 2016.
- 11 E-LINUX.ORG. **Toolchains**, 2016. Disponivel em: <<http://elinux.org/Toolchains>>. Acesso em: 24 de outubro 2016.

- 12 PRACTICAL MICRO. **K9 CNC I/O Cape for the BeagleBone Black**, 2014. Disponivel em: <<http://www.pmdx.com/PMDX-432>>. Acesso em: 05 de outubro 2016.
- 13 BEAGLEBOARD.ORG. **Latest Firmware Images**, 2016. Disponivel em: <<https://beagleboard.org/latest-images>>. Acesso em: 2016 de outubro 2016.
- 14 DEBIAN, 2016. Disponivel em: <<https://debian.beagleboard.org/images/>>. Acesso em: 24 de outubro 2016.
- 15 E-LINUX.ORG. **Beagleboard**: BeagleBoneBlack Debian, 2016. Disponivel em: <http://elinux.org/Beagleboard:BeagleBoneBlack_Debian>. Acesso em: 24 de outubro 2016.
- 16 7-ZIP.ORG. **7-Zip**, 2016. Disponivel em: <<http://www.7-zip.org/>>. Acesso em: 24 de outubro 2016.
- 17 E-LINUX.ORG. **Beagleboard**: Updating The Software, 2016. Disponivel em: <http://elinux.org/Beagleboard:Updating_The_Software>. Acesso em: 24 de outubro 2016.
- 18 DAVIS, T.; DAVIS, J. **Win32 Disk Imager**, 2016. Disponivel em: <<https://sourceforge.net/projects/win32diskimager/>>. Acesso em: 2016 de outubro 2016.
- 19 FOROUZAN, B. A.; MOSHARRAF, F. **Redes de Computadores**: Uma Abordagem Top-Down. 5. ed. [S.I.]: AMGH Editora, 2013.
- 20 BEAGLEBOARD.ORG. **Getting Started**, 2016. Disponivel em: <<http://beagleboard.org/static/beaglebone/latest/README.htm>>. Acesso em: 24 de outubro 2016.
- 21 STACK OVERFLOW. **The best way to do TCP/IP over USB on Linux**, 2013. Disponivel em: <<http://superuser.com/questions/593757/the-best-way-to-do-tcp-ip-over-usb-on-linux>>. Acesso em: 13 de outubro 2016.
- 22 TATHAM, S. **Download PuTTY - a free SSH and telnet client for Windows**, 2016. Disponivel em: <<http://www.putty.org/>>. Acesso em: 24 de outubro 2016.
- 23 SPARKFUN. **Serial Communication**, 2015. Disponivel em: <<https://learn.sparkfun.com/tutorials/serial-communication>>. Acesso em: 15 de outubro 2016.
- 24 E-LINUX.ORG. **Beagleboard**: BeagleBone Black Serial, 2016. Disponivel em: <http://elinux.org/Beagleboard:BeagleBone_Black_Serial>. Acesso em: 15 de outubro 2016.

- 25 SANTOS, R.; PERESTRELO, L. **BeagleBone For Dummies**. 1. ed. New Jersey: Wiley, 2015.
- 26 PAULO, D. R. **Fundamentos do sistema Linux - comandos do Linux**, 2006. Disponivel em: <<https://www.vivaolinux.com.br/artigo/Fundamentos-do-sistema-Linux-comandos-do-Linux/>>. Acesso em: 25 de outubro 2016.
- 27 BORGES, L. E. **Python para desenvolvedores**. 2. ed. Rio de Janeiro: [s.n.], 2010.
- 28 WONG, E. **Changing usb0 IP address on the BeagleBone Black**, 2014. Disponivel em: <<http://ewong.me/changing-usb0-ip-address-on-the-beaglebone-black>>. Acesso em: 04 de julho 2016.
- 29 E-LINUX.ORG. **Beagleboard: Cape Expansion Headers**, 2013. Disponivel em: <http://elinux.org/Beagleboard:Cape_Expansion_Headers>. Acesso em: 10 de novembro 2016.
- 30 E-LINUX.ORG. **Capemgr**, 2016. Disponivel em: <<http://www.elinux.org/Capemgr>>. Acesso em: 10 de novembro 2016.
- 31 TEXAS INSTRUMENTS. **AM335x PRU Read Latencies**, 2016. Disponivel em: <http://processors.wiki.ti.com/index.php/AM335x_PRU_Read_Latencies>. Acesso em: 12 de novembro 2016.
- 32 E-LINUX.ORG. **Ti AM33XX PRUSSv2**, 2016. Disponivel em: <http://elinux.org/Ti_AM33XX_PRUSSv2>. Acesso em: 12 de novembro 2016.
- 33 E-LINUX.ORG. **Ti AM33XX PRUSSv2**, 2016. Disponivel em: <http://elinux.org/Ti_AM33XX_PRUSSv2>. Acesso em: 09 de novembro 2016.
- 34 E-LINUX.ORG. **Beagleboard: Cape Expansion Headers**, 2013. Disponivel em: <http://elinux.org/Beagleboard:Cape_Expansion_Headers>. Acesso em: 16 de novembro 2016.
- 35 BEAGLEBOARD.ORG. **BeagleBone open-hardware expandable computer**, 2016. Disponivel em: <<http://beagleboard.org/support/bone101>>. Acesso em: 16 de Novembro 2016.
- 36 ACCELLERA. **Open Core Protocol Specification 3.0**. Napa: [s.n.], v. 1.0, 2013.
- 37 MOLLOY, D. **Chapter 13: Real-Time BeagleBone Interfacing**, 2015. Disponivel em: <<http://exploringbeaglebone.com/chapter13/>>. Acesso em: 22 de Novembro 2016.

- 38 TEXAS INSTRUMENTS. **http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader_API_Guide**, 2016. Disponivel em: <http://processors.wiki.ti.com/index.php/PRU_Linux_Application_Loader_API_Guide>. Acesso em: 16 de Novembro 2016.
- 39 TEXAS INSTRUMENTS. **PRU Assembly Instructions**, 2016. Disponivel em: <http://processors.wiki.ti.com/index.php/PRU_Assembly_Instructions>. Acesso em: 16 de Novembro 2016.
- 40 MCAREE, O. **Access the Programmable Realtime Units (PRUs) of the BeagleBone from Node.js**, 2013. Disponivel em: <<https://www.npmjs.com/package/pru>>. Acesso em: 16 de Novembro 2016.
- 41 BAKKEN, E. **PyPRUSS on BeagleBone**, 2016. Disponivel em: <<https://bitbucket.org/intelligentagent/pypruss>>. Acesso em: 16 de Novembro 2016.
- 42 THING-PRINTER. **PyPRUSS on BeagleBone**, 2013. Disponivel em: <http://wiki.thing-printer.com/index.php?title=PyPRUSS_on_BeagleBone>. Acesso em: 16 de Novembro 2016.
- 43 RVEGA. **libpruio (BB D/A - I/O fast and easy)**, 2014. Disponivel em: <<http://www.freebasic.net/forum/viewtopic.php?f=14&t=22501>>. Acesso em: 16 de Novembro 2016.
- 44 TJF. **libpruio 0.2**, 2014. Disponivel em: <<http://users.freebasic-portal.de/tjf/Projekte/libpruio/doc/html/>>. Acesso em: 16 de Novembro 2016.
- 45 DINUXBG. **GCC and Binutils port for the TI PRU I/O processor**, 2016. Disponivel em: <<https://github.com/dinuxbg/gnupru>>. Acesso em: 17 de Novembro 2016.
- 46 TEXAS INSTRUMENTS. AM335x Sitara™ Processors, 2016. Disponivel em: <<http://www.ti.com/lit/ds/sprs717j/sprs717j.pdf>>. Acesso em: 17 de Novembro 2016.
- 47 ADAFRUIT. **ADS1015 12-Bit ADC – 4 Channel with Progrmmable Gain Amplifier**, 2012. Disponivel em: <<https://blog.adafruit.com/2012/10/26/new-product-ads1015-12-bit-adc-4-channel-with-progrmmable-gain-amplifier/>>. Acesso em: 17 de Novembro 2016.

48 ESCOLA DA TECNOLOGIA. **O que é Hub USB? Como funciona? Para que serve?**, 2015. Disponivel em: <<http://escoladatecnologia.com.br/o-que-e-hub-usb-como-funciona-para-que-serve/>>. Acesso em: 17 de Novembro 2016.

49 SHABAZ. **BBB - High speed data acquisition and web-based UI**, 2013. Disponivel em: <https://www.element14.com/community/community/designcenter/single-board-computers/next-gen_beaglebone/blog/2013/08/04/bbb--high-speed-data-acquisition-and-web-based-ui>. Acesso em: 17 de Novembro 2016.

50 TEXAS INSTRUMENTS. **ADC08200 8-Bit, 20 Msps to 200 Msps, Low Power A/D Converter with Internal Sampleand-Hold**, 2013. Disponivel em: <<http://www.ti.com/lit/ds/symlink/adc08200.pdf>>. Acesso em: 21 de Novembro 2016.

51 MICROCHIP. **MCP3004/3008**, 2008. Disponivel em: <<https://cdn-shop.adafruit.com/datasheets/MCP3008.pdf>>. Acesso em: 25 de Novembro 2016.

52 SACCO, F. **Comunicação SPI – Parte 1**, 2014. Disponivel em: <<https://www.embarcados.com.br/spi-parte-1/>>. Acesso em: 22 de Novembro 2016.

53 INSTITUTO NEWTON C. BRAGA. **Lição 11 - Como funcionam os registradores de deslocamento (Shift-registers)**, 2014. Disponivel em: <<http://www.newtoncbraga.com.br/index.php/eletronica-digital/102-licao-11-como-funcionam-os-registradores-de-deslocamento-shift-registers>>. Acesso em: 25 de Novembro 2016.

54 KROUTIKOV, M. **Fast analog sensor capture for Beaglebone Black**, 2014. Disponivel em: <https://github.com/pgmmpk/beaglebone_pru_adc>. Acesso em: 25 de Novembro 2016.

55 KROUTIKOV, M. **Fast analog sensor capture for Beaglebone Black**, 2014. Disponivel em: <<https://github.com/pgmmpk/node-pru-adc>>. Acesso em: 28 de Novembro 2016.

56 REIS, F. D. **Curso de Redes – Protocolo TCP (Transmission Control Protocol)**, 2015. Disponivel em: <<http://www.bosontreinamentos.com.br/redes-computadores/curso-de-redes-protocolo-tcp-transmission-control-protocol/>>. Acesso em: 28 de Novembro 2016.

57 REIS, F. D. **Curso de Redes – Protocolo TCP – Handshake de Três Vias**, 2015. Disponivel em: <<http://www.bosontreinamentos.com.br/redes-protocolo-tcp-handshake-de-tres-vias/>>.

computadores/curso-de-redes-protocolo-tcp-handshake-de-tres-vias/>. Acesso em: 28 de Novembro 2016.

58 REIS, F. D. **Curso de Redes – Protocolo UDP (User Datagram Protocol)**, 2016. Disponivel em: <<http://www.bosontreinamentos.com.br/redes-computadores/curso-de-redes-protocolo-udp-user-datatype-protocol/>>. Acesso em: 28 de Novembro 2016.

59 PYTHON SOFTWARE FOUNDATION. **Download the latest version for Windows**, 2016. Disponivel em: <<https://www.python.org/downloads/>>. Acesso em: 28 de Novembro 2016.

60 BIJORA, H. **Como ativar e desativar firewall do Windows 10**, 2016. Disponivel em: <<http://www.techtudo.com.br/dicas-e-tutoriais/noticia/2016/03/como-ativar-e-desativar-firewall-do-windows-10.html>>. Acesso em: 28 de Novembro 2016.

61 BAIXAKI. **Apache for Windows**, 2014. Disponivel em: <<http://www.baixaki.com.br/download/apache-for-windows.htm>>. Acesso em: 28 de Novembro 2016.

62 CLAVIJO, D. **Server-side programming language statistics**, 2013. Disponivel em: <<http://blog.websitesframeworks.com/2013/03/programming-language-statistics-in-server-side-161/>>. Acesso em: 28 de Novembro 2016.

63 W3SCHOOLS. **W3Schools Online Web Tutorials**, 2016. Disponivel em: <<http://www.w3schools.com/>>. Acesso em: 28 de Novembro 2016.

64 DANGERFIELD, B. **Building a wireless temperature sensor with a Raspberry Pi and Xbee Wireless modules**, 2014. Disponivel em: <http://www.brettdangerfield.com/post/raspberrypi_tempature_monitor_project/>. Acesso em: 29 de Novembro 2016.

65 RONACHER, A. **Werkzeug**, 2014. Disponivel em: <<http://werkzeug.pocoo.org/>>. Acesso em: 28 de Novembro 2016.

66 RONACHER, A. **Welcome to Jinja2**, 2008. Disponivel em: <<http://jinja.pocoo.org/docs/dev/>>. Acesso em: 28 de Novembro 2016.

67 THE JQUERY FOUNDATION. **What is jQuery?**, 2016. Disponivel em: <<https://jquery.com/>>. Acesso em: 29 de Novembro 2016.

68 SINGHAL, V. **11 Best jQuery Charting Libraries**, 2015. Disponivel em: <<https://www.sitepoint.com/11-best-jquery-charting-libraries/>>. Acesso em: 29 de Novembro 2016.

- 69 SCHNUR, D. **Attractive JavaScript plotting for jQuery**, 2014. Disponível em: <<http://www.flotcharts.org/>>. Acesso em: 29 de Novembro 2016.
- 70 HIGA, P. **Helio X20 é o novo processador deca-core da MediaTek para smartphones e tablets**, 2015. Disponível em: <<https://tecnoblog.net/178126 MEDIATEK-helio-x20-processador-deca-core/>>. Acesso em: 29 de Novembro 2016.
- 71 HELLMANN, D. **threading – Manage concurrent threads**, 2015. Disponível em: <<https://pymotw.com/2/threading/>>. Acesso em: 29 de Novembro 2016.
- 72 ADAFRUIT. **Adafruit Python ADS1x15**, 2016. Disponível em: <https://github.com/adafruit/Adafruit_Python_ADS1x15>. Acesso em: 29 de Novembro 2016.
- 73 MATHWORKS. **Send HTTP request message and receive response**, 2016. Disponível em: <<https://www.mathworks.com/help/matlab/ref/matlab.net.http.requestmessage.send.html>>. Acesso em: 29 de Novembro 2016.
- 74 NVIDIA. **O que é computação acelerada por placa de vídeo?**, 2016. Disponível em: <<http://www.nvidia.com.br/object/what-is-gpu-computing-br.html>>. Acesso em: 17 de outubro 2016.
- 75 PAANKIEWICZ, I. **O que é um port?**, 2009. Disponível em: <<http://www.tecmundo.com.br/video-game-e-jogos/1910-o-que-e-um-port-.htm>>. Acesso em: 21 de outubro 2016.
- 76 NOGUEIRA, T. A. **Redes de comunicação para sistemas de automação industrial**. Ouro Preto: Universidade Federal de Ouro Preto, 2009.
- 77 MAXIM INTEGRATED. **Understanding SAR ADCs: Their Architecture and Comparison with Other ADCs**, 2001. Disponível em: <<https://www.maximintegrated.com/en/app-notes/index.mvp/id/1080>>. Acesso em: 17 de Novembro 2016.
- 78 RONACHER, A. **Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions.**, 2016. Disponível em: <<http://flask.pocoo.org/>>. Acesso em: 28 de Novembro 2016.