Bruno Bernardo de Moura

Laboratório de Sistemas Computacionais: Redes de Computadores - Relatório Final

São José dos Campos - Brasil

Julho de 2019

Bruno Bernardo de Moura

Laboratório de Sistemas Computacionais: Redes de Computadores - Relatório Final

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratórios de Sistemas Computacionais: Redes de Computadores.

Docente: Prof. Dr. Lauro Paulo da Silva Neto

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Julho de 2019

Resumo

O relatório em questão apresenta detalhes à respeito implementação de um sistema de comunicação UART para o sistema operacional desenvolvido nos laboratórios anteriores (BM32OS). Tal comunicação buscará ser feita de maneira *half-duplex* entre as plataformas FPGA e Arduino.

Palavras-chaves: Sistema, Comunicação, *half-duplex*, FPGA, Arduino.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 Motivação

Cada vez mais presentes no dia-a-dia das pessoas, independente da área em que atuem ou trabalhem, computadores são de extrema importância para o funcionamento da sociedade atual como um todo, seja no âmbito de cadastro de serviços, no movimento de transações bancárias ou simplesmente como ferramenta de planejamento.

Seu funcionamento se dá com base em diversos fatores envolvendo elementos de ambos *hardware* e *software* para a execução de um determinado conjunto de instruções sobre um certo dado a fim de se obter um resultado específico. Em outras palavras, tais ações são tomadas a fim da execução de algoritmos, os quais envolvem ações como armazenamento, envio, recebimento e operação de dados, para os quais é necessária a criação de uma unidade de processamento, eventuais memórias, um módulo de entrada e saída, dentre outros.

Uma vez criadas tais unidades de *hardware*, há um programa específico responsável por garantir a comunicação entre elas e o usuário, o Sistema Operacional. Tal programa é gerado por meio de um compilador capaz de traduzir programas de uma linguagem de alto nível para uma de baixo nível, no caso, as instruções do processador.

Ainda, uma vez executando seus respectivos Sistemas Operacionais, o que torna o uso de computadores tão importante é o fato de poderem realizar a rápida troca de informações com outras máquinas, de forma que para tal estes precisam estar conectados em rede.

Assim, este relatório visa relatar os detalhes à respeito da comunicação por via serial do Sistema Operacional desenvolvido nos laboratórios anteriores com um microcontrolador Arduino. Para isso serão necessárias adaptações em suas arquiteturas de *hardware* e *software* para que assim possa prover suporte ao protocolo de comunicação a ser desenvolvido dadas as limitações dos equipamentos escolhidos para a implementação do projeto.

1.2 Objetivos

1.2.1 Geral

Planejar e desenvolver componentes em *hardware* e *software* que ofereçam suporte básico à uma comunicação serial entre o sistema operacional previamente desenvolvido e um microcrontrolador Arduino.

1.2.2 Específico

Adaptar as partes do projeto e implementar estruturas que suportem chamadas bloqueantes no quesito de envio e recebimento de dados entre as partes do projeto, de forma que para isso sejam implementados um protoclo de comunicação que permita tal envio e recbimento de dados juntamente com uma interface que permita ao usuário enviar e receber dados após seus processamento por algoritmos do sistema operacional.

2 Fundamentação Teórica

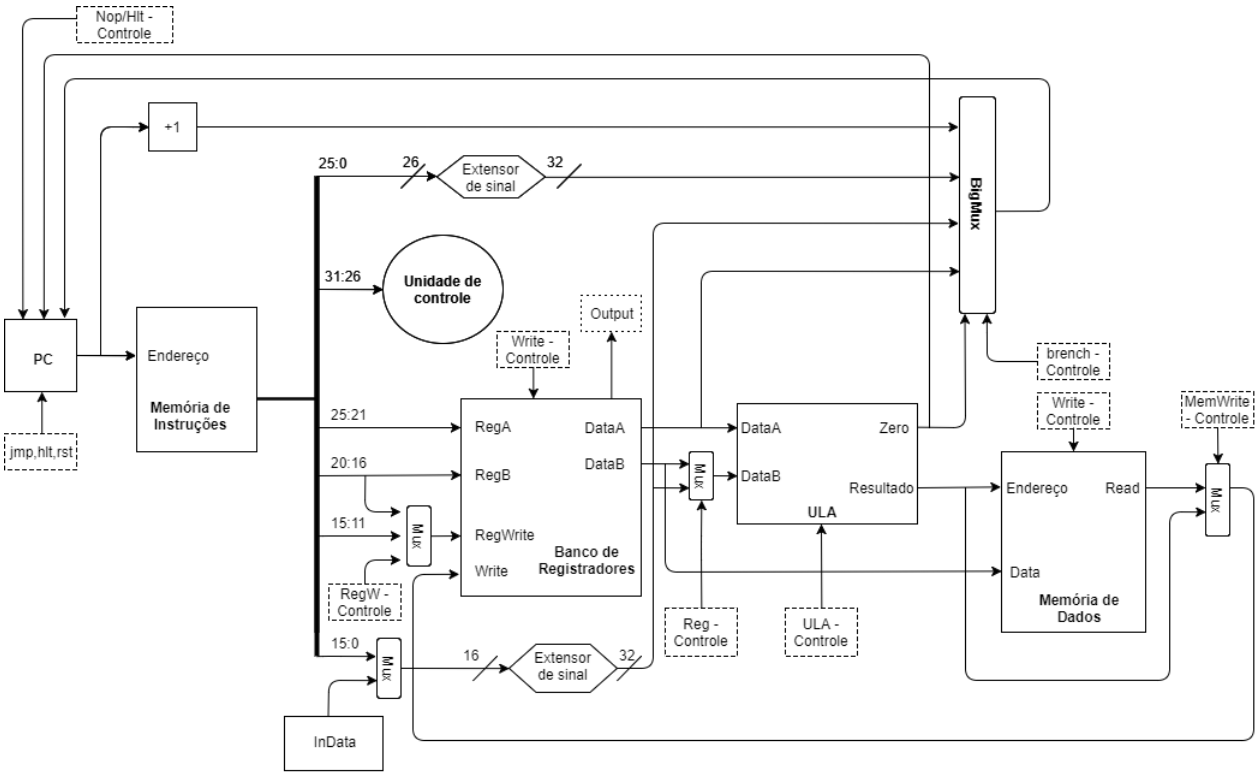
2.1 Arquitetura do Processador

O processador utilizado neste projeto (1), o qual deu suporte ao sistema operacional, possui arquitetura inspirada na arquitetura MIPS (2), logo um conjunto de instruções RISC (*Reduced Instruction Set Computer*), uma vez que menos instruções tornam a organização de seus sinais de controle e módulos mais simplificada. Assim, segue uma explicação à respeito das especificidades de tal arquitetura.

2.1.1 Detalhes da Arquitetura

Como pode ser visto na Figura 1, o processador em questão possui em sua Unidade de Processamento um Contador de Programa (*Program Counter*), duas memórias, uma de instrução (*Instruction Memory*) e uma de dados (*Data Memory*), um Banco de Registradores (*Register Bank*) e uma Unidade Lógica e Aritmética (*Arithmetical Logic Unity*), fora eventuais extensores de sinal e multiplexadores.

Figura 1 – Caminho de Dados



Fonte: Autor

2.1.1.1 *Program Counter*

O contador de programa é o módulo responsável pelo armazenamento do índice da próxima instrução a ser executada, de maneira que é o módulo que garante o funcionamento de saltos normais (*jumps*) e condicionais.

2.1.1.2 *Instruction Memory*

A memória de instruções armazena todas as instruções do algoritmo que está sendo executado pelo processador. Nela há uma série de registros de 32 *bits*, cada um responsável pelo armazenamento de uma instrução, de forma que sua entrada é o índice desses registros a ser acessado (saída do *Program Counter*) e a saída é o registro armazenado em tal índice.

2.1.1.3 *Register Bank*

O banco de registradores da arquitetura em questão é utilizado em sua maior parte para o armazenamento de valores temporários pertinentes às operações em questão. Possuindo um total de 32 registradores de propósito geral, alguns deles são utilizados para fins específicos simplesmente por questão de uma maior facilidade com relação à organização dos dados quanto à utilização do compilador. Tais registradores são referentes ao endereço de retorno (*register addres*), ao ponteiro da pilha (*stack pointer*) e ao registrador de retorno de chamadas de funções.

2.1.1.4 *Arithmetic Logic Unity*

Responsável por tratar todas as possíveis operações lógicas e aritméticas do processador, a Unidade Lógica e Aritmética (ULA), atua de forma a receber dois dados (podendo eles estarem contidos no Banco de Registradores ou não) e realizar uma determinada operação com eles com base em um sinal de controle. Assim, sua saída pode ser utilizada como endereço de acesso da Memória de Dados ou simplesmente como um novo dado a ser armazenado no Banco de Registradores.

2.1.1.5 *Data Memory*

Por fim, o último dos componentes principais do caminho de dados da arquiteutra utilizada é a Memória de Dados. Nela são armazenados dados vindos do banco de registradores com base no cálculo de um endereço vindo da ULA. No âmbito da utilização do compilador, é nela que ficam armazenadas informações a respeito de todas as variáveis declaradas no decorrer do código. Vale ainda ressaltar que nesse módulo se encontra uma pilha implícita utilizada em contextos de parâmetros de funções, consecutivas chamadas de funções para o armazenamento dos endereços de retorno e recursão.

2.1.2 Tipos de Instruções

Por ser baseado na arquitetura MIPS, o processador utilizado possui três tipos de instrução: R, I e J.

- **Instruções do Tipo R:** tipo no qual se encontram todas as instruções que realizam operaçoes lógicas ou aritméticas com os dados dos registradores, tais como somas, subtraçoes, deslocamentos e afins (3) (Tabela 1). Devido a isso são divididas em *opcode*, RS, RT, RD, *shamt* e *funct*. O *opcode* é o responsável por diferenciar cada uma das instruções das outras, de forma que cada uma possua o seu *opcode* específico; RS e RT representam os endereços do banco de registradores dos quais serão retirados dados; RD possui o valor do endereço no qual serão escritos os dados resultantes da operação; *shamt* representa a quantidade de *bits* a serem deslocados caso seja necessário e por fim *funct* é utilizado para a diferenciação de instruções na ULA.

Tabela 1 – Formato R de Instrução

Tamanho (bits)	6	5	5	5	5	6
Campo	opcode	RS	RT	RD	shamt	funct
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 -0

Fonte: Autor

- **Intruções do Tipo I:** nesta categoria se encontram todas as instruções que realizam operações lógicas e aritméticas com dados de *offset*, vindos do campo imediato, executam saltos condicionais e realizam acesso à memória (Tabela 2). Neste caso a instrução é dividida, além do *opcode*, nos campos RS, RT e *offset*. O campo RS é o responsável por mostrar o valor do endereço do banco de registradores do qual será lido o dado a ser utilizado; RT representa o endereço do mesmo banco no qual será escrito o possível resultado da instrução e por fim o campo *offset*, ou imediato, carrega consigo um valor codificado que será usado diretamente na instrução, seja para possíveis saltos condicionais ou para operações aritméticas.

Tabela 2 – Formato I de Instrução

Tamanho (bits)	6	5	5	11
Campo	opcode	RS	RT	offset
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: Autor

- **Instrução do Tipo J:** tipo de instrução responsável pelas instruções de saltos (Tabela 3). Devido a isso sua divisão é a mais simples das instruções: *opcode* representa a identificação da instrução em si e todos os outros 26 *bits* são destinados ao endereço para o qual se quer saltar, o que garante uma distância muito maior para o salto do que se para o seu cálculo fosse utilizada a mesma quantidade de *bits* que em outros tipos de instrução.

Tabela 3 – Formato J de Instrução

Tamanho (bits)	6	26
Campo	opcode	address
Bits	31 - 26	25 - 0

Fonte: Autor

2.1.3 Conjunto de Instruções

Dadas as devidas explicações, o conjunto de instruções contido na arquitetura do processador utilizado pode ser visto em Tabela 4. Na tabela, os campos RS, RT e RD são referentes aos campos das instruções propriamente ditos, M representa a memória e STACK a pilha implícita citada.

Tabela 4 – Conjunto de Instruções

Instrução	**Tipo**	**Mnemônico**	***Opcode***	**Operação**
adição	R	add	000000	R[RD] ← R[RS] + R[RT]
subtração	R	sub	000001	R[RD] ← R[RS] - R[RT]
set on less than	R	slt	000110	if(R[RS] < R[RT]) R[RD] = 1
set on less or equal than	R	slt	011101	if(R[RS] <= R[RT]) R[RD] = 1
set on equal	R	slt	011110	if(R[RS] == R[RT]) R[RD] = 1
set on not equal	R	slt	011111	if(R[RS] != R[RT]) R[RD] = 1
deslocamento à esquerda	R	shfl	000111	R[RD] ← sl(*shamt*)R[RS]
deslocamento à direita	R	shfr	001000	R[RD] ← sr(*shamt*)R[RS]
NOT	R	not	001001	R[RT] ← !R[RS]
AND	R	and	001010	R[RD] ← R[RS] & R[RT]
OR	R	or	001011	R[RD] ← R[RS] | R[RT]
XOR	R	or	001100	R[RD] ← R[RS] ∧ R[RT]
multiplicação	R	mult	001101	R[RD] ← R[RS] x R[RT]
divisão	R	div	001110	R[RD] ← R[RS] / R[RT]
NOP	R	nop	010110	não realiza operação
HLT	R	hlt	010111	para a contagem do PC
adição com imediato	I	addi	000010	R[RT] ← R[RS] + IM
subtração com imediato	I	subi	000011	R[RT] ← R[RS] - IM
incremento	I	inc	000100	R[RT] ← R[RS] + 1
decremento	I	dec	000101	R[RT] ← R[RS] - 1
load	I	ld	001111	R[RT] ← M[IM + R[RS]]
load imediato	I	ldi	010000	R[RD] ← IM
store	I	str	010001	M[R[RS] + IM] ← R[RT]
branch on equal	I	beq	010010	if(R[RS] == R[RT]) PC ← IM
branch on not equal	I	bneq	010011	if(R[RS] != R[RT]) PC ← IM
jump to register	I	jmpr	010101	PC ← R[RS]
input	I	in	011000	R[RS] ← IN
output	I	out	011001	OUT ← R[RS]
push	I	push	011011	STACK[SP] ← R[RT]
pop	I	pop	011100	R[RT] ← STACK[SP]
jump	J	jmp	010100	PC ← IM
jump and link	J	jal	011010	R[31] ← PC + 1, PC ← IM

Fonte: Autor

2.2 Compilador para *Cminus*

Para este projeto foi também utilizado um compilador de linguagem *Cminus* (4), criado utilizando *Antlr4*, a partir do qual foi possível realizar em tal linguagem a implementação do sistema operacional BM32OS em si. Essa linguagem se assemelha muito com a linguagem C já conhecida, com exceção apenas de alguns elementos nela não presentes, como por exemplo vetores bidimensionais e de alocação de memória, a falta de laços de repetição do tipo *for*, inexistência de ponteiros e do tipo de variável de ponto flutuante (*float*), dentre outros. Vale ainda ressaltar que este último se dá devido à limitações da arquitetura do processador desenvolvido, a qual não suporta operações de ponto flutuante.

Segue então alguns dos erros semânticos reconhecidos pelo compilador, todos, como já descrito, semelhantes à erros semânticos da linguagem de programação.

- Variável não declarada;

- Atribuição de tipos de dados inválidos (*int* para *void*);

- Declaração com tipo inválido de variável;

- Variável já declarada;

- Função não declarada;

- Chamada de função com número errado de parâmetros;

- Função "*main*"não declarada;

- Declarações inválidas (variáveis com nomes de funções ou funções com nomes de variáveis);

2.3 BM32OS

O sistema operacional (BM32OS) utilizado neste projeto foi implementado de forma a dar suporte à todos os requerimentos básicos de um sistema operacional real, sendo limitado apenas pela plataforma de *hardware* utilizada pra executá-lo.

Assim, ele possui um gerenciamento de processos por meio do algoritmo de escalonamento *Round Robin*, endereçamento na memória relativo à cada número em *hardware* de cada um de seus programas e um sistema de gerenciamento de arquivos que permite sua criação, renomeação e deleção à vontade do usuário.

Sua interface com o *hardware* BMCORE1 utilizado foi feita de maneira dita bipartida, ou seja, cada região de memória que armazenasse dados referentes à processos seria subdividida em duas partes; uma exclusivamente para o armazenamento dos dados do sistema operacional em si e a outra para o armazenamento dos dados referentes aos

processos sendo escalonados e executados.

A seguir podem ser vistas as instruções adicionadas à arquitetura para o suporte ao sistema operacional.

Tabela 5 – Adição ao Conjunto de Instruções

Instrução	**Tipo**	**Mnemônico**	***Opcode***	**Operação**
BIOS to Memory	X	btm	100000	Muda o fluxo de instruções para a memória
load os	X	ldos	100010	MEM[OS][RD] ← HD[PROC=RS][RT]
move HD to Memory	X	mhdm	100011	MEM[PROC=RS][RD] ← HD[PROC=RS][RT]
move HD to reg	X	mdhr	100100	RD ← HD[PROC=RS][RT]
store HD	X	strhd	100101	HD[PROC=RS][RT] ← RD
swap process	X	sprc	100110	processo atual ← RS
concatenação	R	conc	100111	R[RD] ← {RS[15:0], RT[15:0]}
change message	X	cmsg	101000	display lcd ← IM
change write shift	X	cwsfh	101001	troca *shift* de escrita
change read shift	X	crsfh	101010	troca *shift* de leitura
get pc	X	getpc	101011	R[28] ← PROC PC
set pc	X	setpc	101100	PROC PC ← R[RS]
syscall	X	syscall	111110	R[28] ← INTERRUPTION
end program	X	endp	111111	Sinaliza o fim do programa em execução

Fonte: Autor

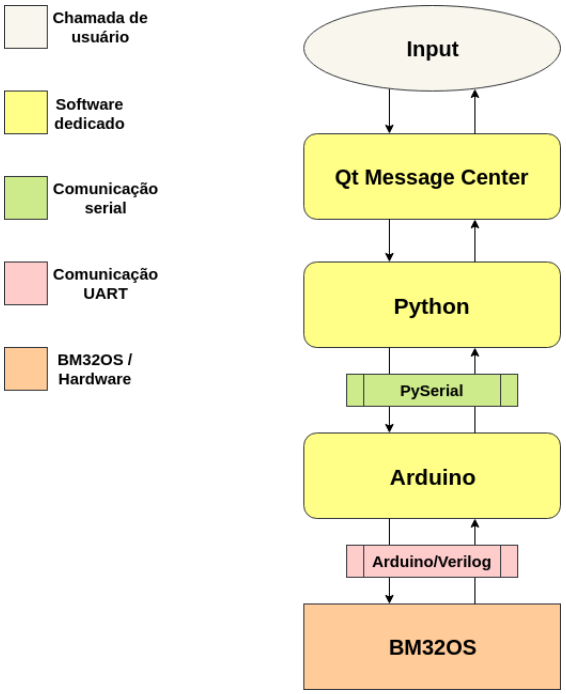
3 Desenvolvimento

3.1 Panorama geral

Para este projeto, inicialmente foi planejado a incorporação de comunicação *blueto-oth* entre o sistema operacional BM32OS e um microcontrolador Arduino. Contudo, após a avaliação de tal proposta dadas as limitações do *hardware* utilizado para o projeto, esta foi descartada.

Assim, foi criado um novo plano geral de funcionamento da comunicação a ser de fato criada, que por sua vez seria realizada entre o sistema operacional BM32OS e uma interface gráfica de utilização do usuário por comunicação serial UART (5). Levando em consideração as limitações de *hardware* do próprio sistema operacional e dos componentes disponíveis para utilização no laboratório, seu funcionamento se deu como pode ser visto na figura 2.

Figura 2 – Arquitetura do projeto



Fonte: Autor

Para a criação da interface utilizada, chamada de *Message Center*, foi utilizada a linguagem *Python3* e suas bibliotecas *PyQt* e *pyserial*, para a utilização de janelas e elementos gráficos e de comunicação serial, respectivamente. Por sua vez, tais bibliotecas permitiram a comunicação da interface desenvolvida com um microcontrolador Arduino,

que por sua vez, utilizando de comunicação serial UART, se comunicou propriamente com o sistema operacional sendo executado na FPGA.

Assim, as próximas seções desse capítulo serão destinadas à cada um dos elementos do projeto que fizeram parte de alguma forma da comunicação desenvolvida, sendo eles a interface *Message Center*, o gerenciador de conexão serial implementado no Arduino e o módulo gerenciador UART implementado e incorporado ao sistema operacional BM32OS. Todos os códigos referentes à essas partes podem ser encontrados nos apêndices do documento.

3.2 *Message Center*

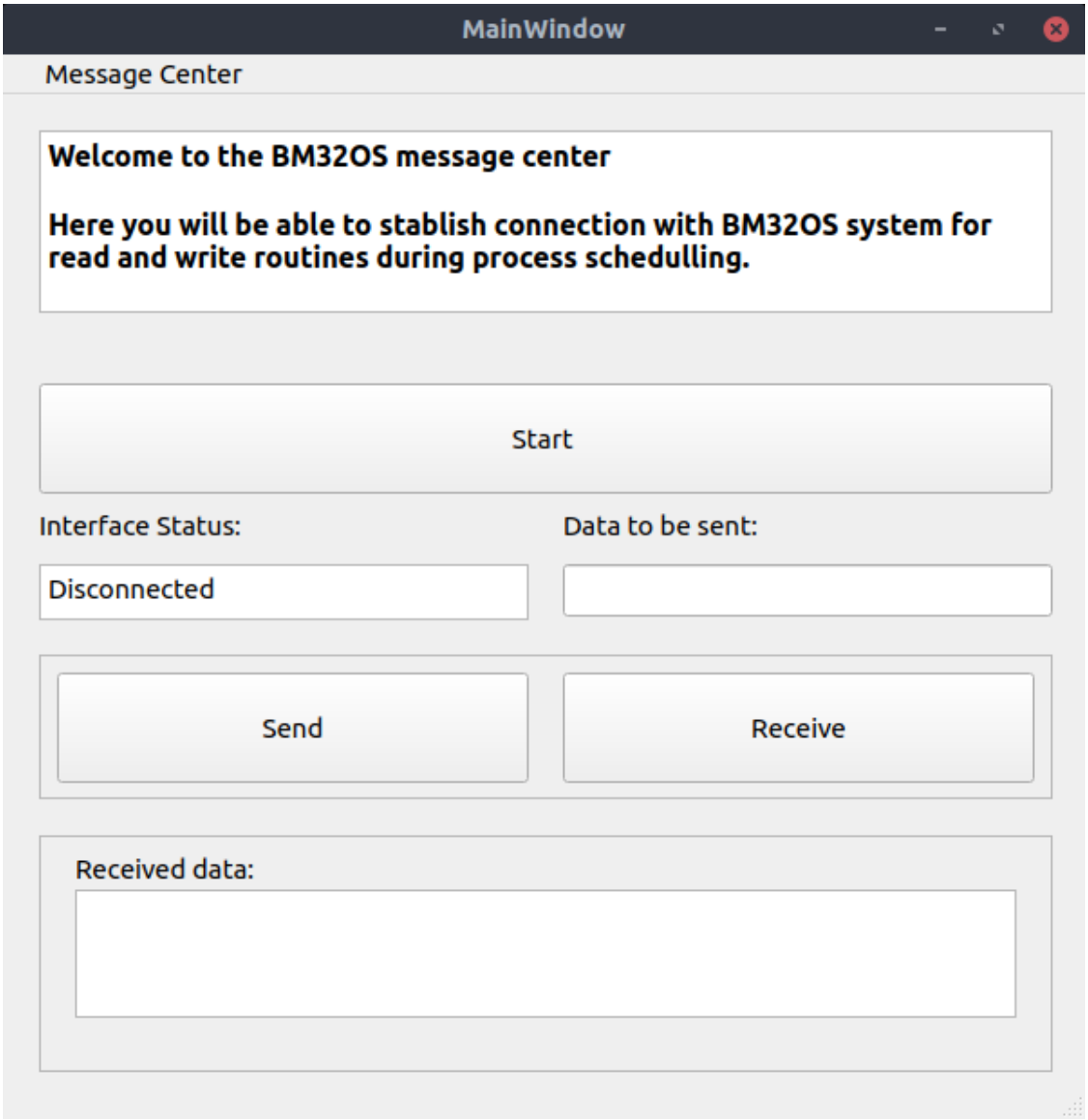
A interface gráfica desenvolvida à princípio serviria apenas como uma facilitadora de utilização do usuário; contudo, sua implementação se demonstrou mais desafiadora do que o esperado. Isso porque a maneira com que a biblioteca utilizada para comunicação serial, *pyserial*, lida com os *bytes* recebidos e enviados ocorre de maneira diferente da utilizada para a troca de tais dados por parte da API serial dos microcontroladores Arduino, a qual será tratada mais detalhadamente mais a frente.

A ideia principal do funcionamento da interface seria a seguinte: um dado selecionado pelo usuário seria enviado da interface para o programa gerenciador de comunicação serial no Arduino, que por sua vez enviaria o dado para o sistema operacional que estaria aguardando por ele. Então, para isso, o primeiro ponto a ser tratado foi o envio de dados do *Message Center* para o microcontrolador Arduino.

Para a realização dessa comunicação, foram realizados alguns tratamentos por parte da interface, de forma que foi estabelecido que um número maior que 255 não poderia ser enviado dadas limitações de *hardware* descritas mais a frente. Além disso, pelo dado, por parte da interface, ser tratado como uma *string*, seu tamanho foi limitado a 4 *bytes*, reservando assim um *byte* para cada caractere nele presente. Assim, um valor numérico enviado da interface para a arduino possuía o valor de 4 *bytes*, mesmo que alguns desses eventualmente fossem zeros à esquerda.

Relacionado à interface do usuário em si, seu *design* foi realizado da maneira mais simples possível, com dados referente ao seu *status* de conexão e com campos e botões nos quais se pudesse digitar, enviar e realizar a leitura dos dados. Assim, seu *displáy* pode ser visto na figura 3.

Figura 3 – *Message Center*



Fonte: Autor

Idealmente, o recebimento de dados por parte da interface deveria ser realizado automaticamente; contudo isso se tornou impossível dado o fato de que pela biblioteca utilizada, a detecção de dados recebidos no *buffer* serial deveria ser realizada por meio do disparo de *threads*, que por sua vez bloqueariam a utilização de outras funcionalidades do *Message Center*. Assim, tal opção foi descartada e foi então utilizado o botão *Receive*, o qual, caso interagido, desencadearia a checagem do buffer serial e detectaria dados enviados pelo Arduino.

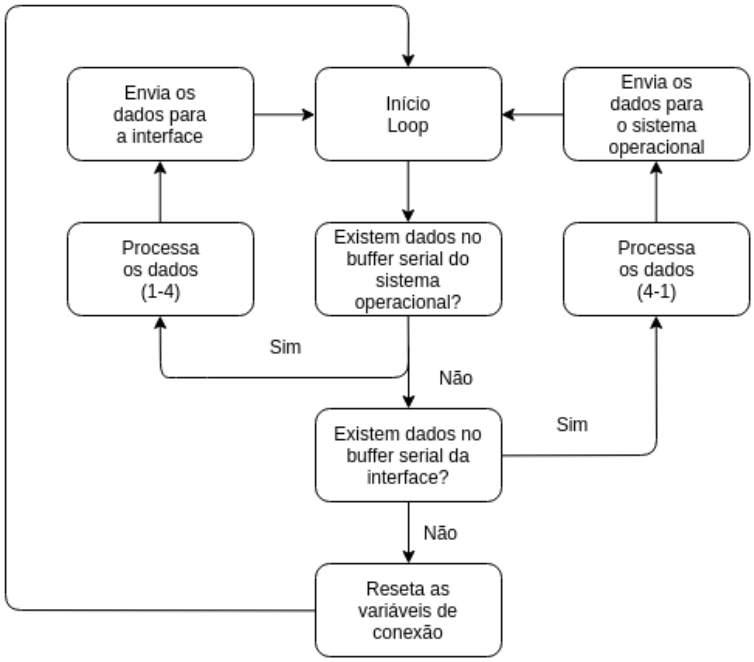
3.3 *Serial Manager*

Por parte da aplicação implementada no microcontrolador Arduino, seu funcionamento se deu de maneira um pouco mais direta do que o já descrito *Message Center*. Foi utilizada a biblioteca *Software Serial* para a criação de uma nova comunicação serial, dado que seriam necessárias duas, uma para a troca de dados com a interface criada e outra para a troca de dados com o sistema operacional; além do fato de que o modelo de Arduino utilizado no projeto foi o Uno, o qual apenas permite nativamente uma única API de comunicação serial com seus pinos 0 (RX) e 1(TX).

Basicamente, a cada nova chamada da função *loop* eram realizadas checagens relativas à presença de dados no *buffer* de cada uma das comunicações seriais abertas, e simplesmente, caso fossem detectados tais dados, eles eram processados e enviados bidirecionalmente, de forma que, por exemplo, caso fossem detectados dados vindos do *Message Center*, esses seriam então convertidos para um inteiro de 1 *byte* e enviados ao sistema operacional; por outro lado, caso fossem detectados dados enviados pelo sistema operacional, estes seriam então convertidos para um dado de 4 *bytes* concatenando-se zeros à sua esquerda e então enviados para leitura por parte da interface.

Assim, resumidamente, o funcionamento do gerenciador de comunicação serial implementado no microcontrolador Arduino pode ser visto na figura 4.

Figura 4 – Funcionamento *Serial Manager*



Fonte: Autor

3.4 *UART Manager*

Por fim, o sistema operacional BM32OS foi acrescido de um novo módulo, o *UART Manager*, responsável por receber e eviar os dados do Arduino.

Adaptado do exemplo disponível pelo professor da disciplina para utilização do módulo *Bluetooth*, o novo módulo implementado possui um comportamento próximo, mas ainda assim diferente dele, descrito a seguir.

Como comportamento padrão do módulo, sempre que nele é recebido um dado, no caso, enviado pelo Arduino, ele reenvia tal dado como resposta de volta ao microcontrolador. Dessa forma, por parte do *Message Center*, sempre que se interage com o botão de envio de dados, também é desencadeada uma rotina responsável por realizar a leitura do dado enviado como resposta, simplesmente para um *flush* no *buffer* serial da comunicação. Foi então adicionado ao módulo uma saída de 16 *bits*, referente por armazenar o valor do dado recebido do Arduino, caso existente.

Por fim, para que fosse desencadeado o envio dos dados agora por parte do sistema operacional para o Arduino, foi utilizada uma lógica de *debounce* responsável por durante um único pulso de *clock* permitir o envio do dado de entrada do módulo, como pode ser visto na figura 5.

Figura 5 – Sistema de *Debounce*

```
always@(posedge clk_auto)
begin
   // occurs only on send instruction
   if(send)
   begin
      // send enablers works holding send_reg value as 1 during only one pulse
      if( send_enabler == 1'b0 )
      begin
         send_enabler = 1'b1;
         send_reg = 1'b1;
      end
      // else statement doesn't let send_reg be 1 again until there is another send instruction
      else
      begin
         send_reg = 1'b0;
      end
   end
   else
   begin
      send_enabler = 1'b0;
      send_reg = 1'b0;
   end
end
```

Fonte: Autor

Assim, utilizando como tal dado de entrada o valor desejado para se realizar *output*, foi possível de se enviar dados genéricos do sistema operacional para o Arduino.

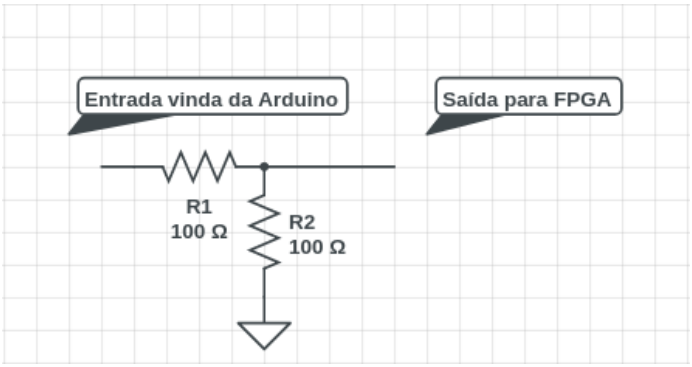
3.5 Alterações no compilador

Sucintamente, vale ainda comentar que foram adicionadas duas chamada de sistema ao sistema operacional BM32OS e suas referentes incorporações ao compilador. As chamadas de sistema em questão foram uma *send*, utilizada para enviar dados do sistema operacional por meio da comunicação uart pela função *UART_output*, e uma *receive*, utilizada para receber dados do Arduino por meio da função *UART_input*. Ambas podem ser vistas no código do sistema operacional em questão.

4 Funcionamento

O funcionamento se deu de maneira que a FPGA foi conectada ao Arduino por meio de seus pinos *GPIO*. No caso, para que fosse realizada essa conexão, pelo fato de ambas as plataformas funcionarem com alimentações diferentes ($5V$ no caso do Arduino, e $3.3V$ no caso da FPGA), foram utilizados dois divisores de tensão como pode ser visto na figura 6 (um para cada conexão entre o Arduino e a FPGA, pinos RX e TX), apenas responsáveis por cortar pela metade a tensão dos *jumpers* conectados ao Arduino (utilizando-se resistores de mesma resistência). Assim estes puderam ser conectados à FPGA sem danificar seus componentes e prejudicar seu funcionamento.

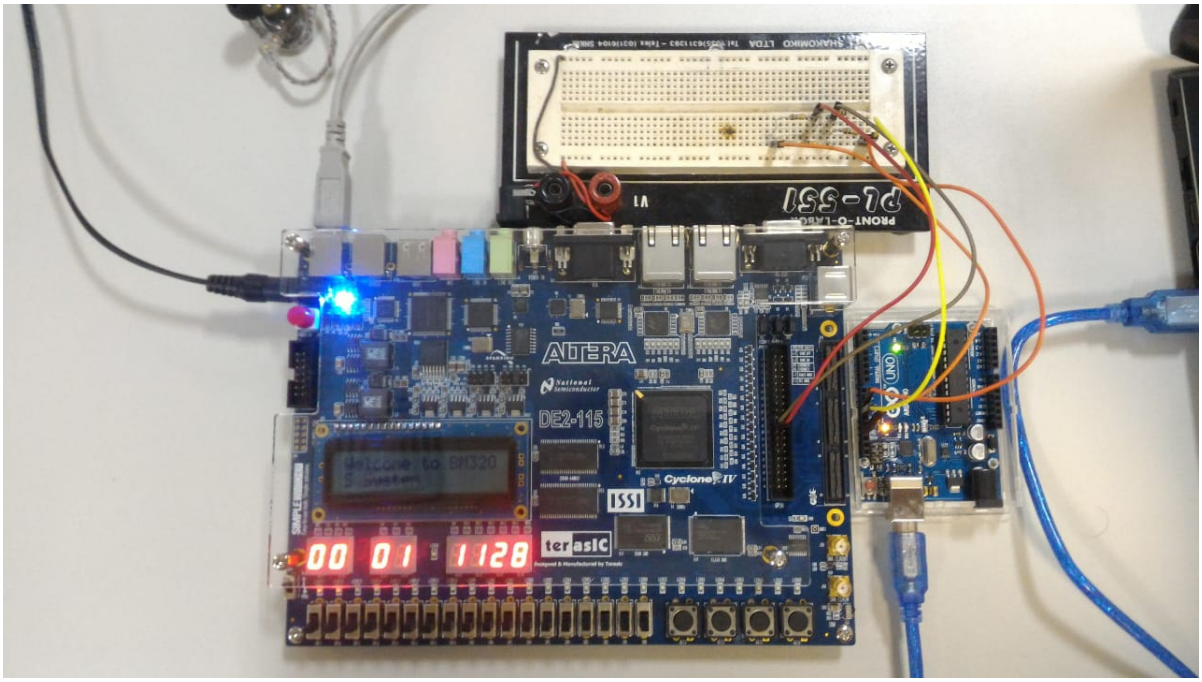
Figura 6 – Divisor de tensão



Fonte: Autor

Assim, para se iniciar a conexão, o programa referente ao *Serial Manager* do Arduino deveria ser carregado na placa, e seus respectivos pinos RX e TX (*setados* no próprio código) deveriam ser conectados à *protoboard* e passados pelos divisores de tensão por meio de *jumpers* macho-macho, que por sua vez deveriam ter suas saídas conectadas por meio de *jumpers* macho-fêmea aos pinos $AE22$ e $AF21$ da FPGA, pertencentes à sua região de pinos GPIO. Uma imagem da montagem de todo o sistema pode ser vista na figura 9.

Figura 7 – Sistema de comunicação



Fonte: Autor

Conectados os componentes do projeto e carregado o código do Arduino, o processador com o sistema operacional BM32OS e todos os processos e dados de seu HD deveriam ter seu arquivo compilado e passado para a placa FPGA. Por fim, a interface *Message Center* deveria ser aberta, simplesmente executando-a como um arquivo em *Python3* pelo próprio *bash*; e assim que aberta, sua conexão deveria ser estabelecida ao se clicar inicialmente no botão *start*.

5 Resultados Obtidos

A comunicação funcionou de maneira como esperado, sendo requisitada diretamente pelo sistema operacional ou por *syscalls* de processos em meio ao seu escalonamento de modo *half-duplex*. Assim foi possível enviar e receber dados em tempo de execução do sistema operacional pelo *Message Center*.

Para exemplificação, 3 processos previamente implementados foram adaptados para que passassem a possuir também requisição de dados da conexão implementada; o primeiro foi um cálculo de potenciação, o segundo de cálculo de membros da sequência de Fibonacci e por fim o terceiro foi um seletor capaz de encontrar o maior valor dentro de um vetor.

Como dito anteriormente, a reimplementação de tais códigos se deu referente apenas às funções utilizadas e responsáveis pelo desencadeamento das chamadas de sistema de comunicação em rede. Assim, apenas para visualização, o algoritmo 5.1 conta com o código do primeiro processo citado.

Algoritmo 5.1 – Código cminus - Cálculo de potência

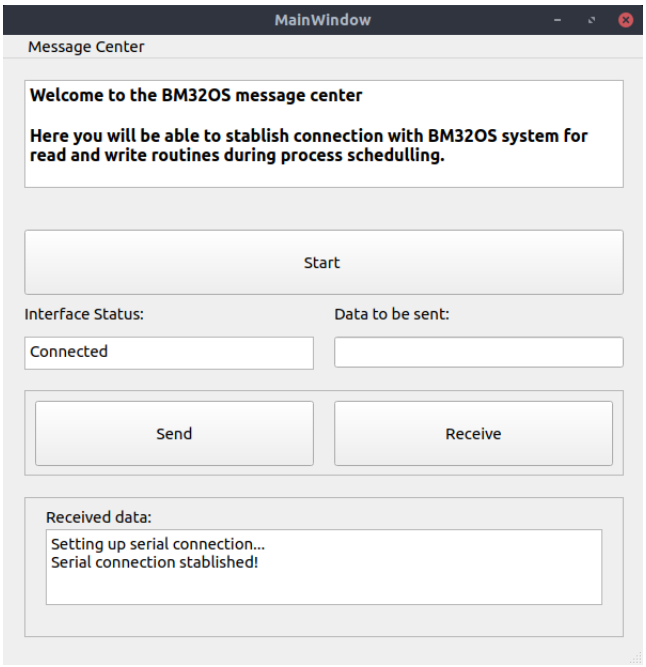
```
// prog 1 - pow

int pow( int a, int b ) {
   int aux; int val;
   val = 1;
   aux = 0;
   while( aux < b ) {
      val = val * a;
      aux = aux + 1;
   }
   return val;
}

void main(void){
   int a;
   int b;
   a = UART_input( );
   b = input( );
   a = pow( a, b );
   UART_output( a );
   return;
}
```

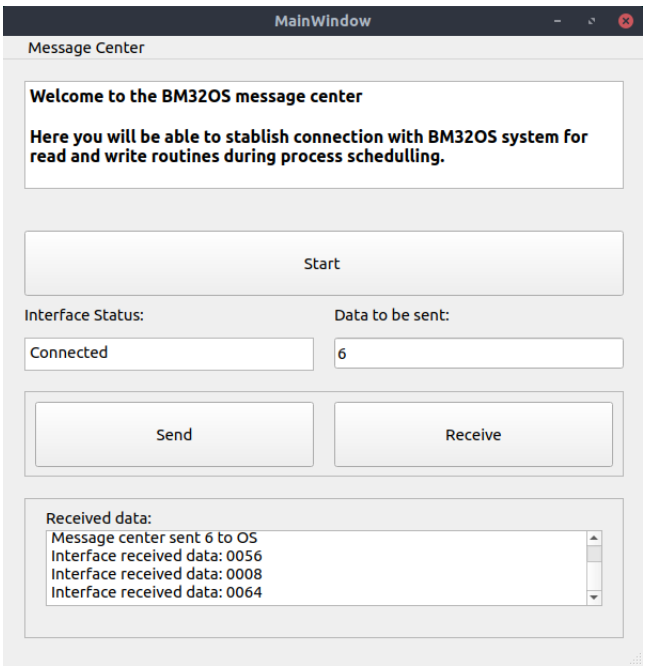
Por fim, podem ser vistas nas imagens alguns estados da tela do *Message Center* durante sua inicialização, envio e recebimento de dados em meio ao escalonamento de processos do sistema operacional.

Figura 8 – Inicialização *Message Center*



Fonte: Autor

Figura 9 – Utilização *Message Center*



Fonte: Autor

6 Conclusão

Pode-se concluir que o trabalho em questão foi de suma importância para o adquirimento de conhecimento por parte do aluno, uma vez que ele uniu questões relacionadas à interrupções, comunicação em rede, comunicação serial e protocolos de comunicação. Não bastante, o fez de forma a obrigar o aluno a revisitar e alterar projetos desenvolvidos em todos os laboratórios anteriores da Engenharia de Computação, permitindo assim a conclusão do ciclo de laboratórios de maneira eficaz e expondo assim o aluno ao funcionamento de um sistema computacional completo, desde sua arquitetura em *hardware*, gerenciamento de processo e de memória, abstrações de *software* com chamadas de sistema e comunicação com outros dispositivos.

Referências

1 MOURA, B. *BMCORE1*. 2017. <https://github.com/BrunoBMoura/BMCORE1>. [Online, acessado 5/05/2018]. Citado na página 9.

2 BRITTON, R. *Introduction to the MIPS Architecture*. 2013. <http://web.engr.oregonstate.edu/~walkiner/cs271-wi13/slides/02-MIPSArchitecture.pdf>. [Online, acessado 5/05/2017]. Citado na página 9.

3 STALLINGS, W. *Computer Organization and Architecture*. 9th edition. ed. Upper Saddle River/New Jersey, EUA: Pearson, 2013. Citado na página 11.

4 MOURA, B. *bmcorepiler*. 2018. <https://github.com/BrunoBMoura/bm_core_piler>. [Online, acessado 5/09/2018]. Citado na página 13.

5 TUTORIAL sobre Comunicação. <https://www.freebsd.org/doc/pt_BR/articles/serial-uart/>. Citado na página 15.

Apêndices

APÊNDICE A – Código *Qt Message Center*

Algoritmo A.1 – Código fonte

```
from PyQt5 import QtGui, QtCore, QtWidgets
import sys
import msg_center_display
import serial
import time
# 11 = laranja, 10 = cinza

# Calls the msg_center_display.py generated file as an object.
class Msg_center(QtWidgets.QMainWindow, msg_center_display.Ui_MainWindow):

    def __init__(self, parent = None):

        super(Msg_center, self).__init__(parent)
        self.setupUi(self)
        # serial connection object
        self.arduino = None
        # connection atribute
        self.connected = False
        self.wait_time = 1
        # size of received data
        self.byte_num = 4
        # received data
        self.received_data = None
        # counters
        self.sent_values = []
        self.received_values = []
        self.clicked_on_send = 0
        self.clicked_on_receive = 0
        # qt signals
        self.text_browser_status.append("Connected" if self.connected else
            "Disconnected")
        self.push_button_send.clicked.connect(self.send_data)
        self.push_button_receive.clicked.connect(self.receive_data)
        self.push_button_start.clicked.connect(self.begin_serial_connection)
```

```
    # Verbose.
    def wait(self, mult = 1):

        time.sleep(mult * self.wait_time)

    # Change message center's status.
    def set_interface_status(self, status, status_val):

        self.connected = status_val
        self.text_browser_status.clear()
        self.text_browser_status.append(status)

    # Returns text written on text box.
    def get_input_data(self):

        return str(self.line_edit_data.text())

    # Writes msg to the text browser.
    def write_text_browser(self, msg):

        self.text_browser_receive.append(msg)

    # Reads data sent from arduino, thread usage.
    def read_from_arduino(self):

        try:
            # read is only done if there are 4 bytes received
            self.custom_print(f"Trying to read from the arduino. Available
                bytes: {self.arduino.in_waiting}")
            if self.arduino.in_waiting >= self.byte_num:
                data = self.arduino.read(self.byte_num)
                if data:
                    self.custom_print(f"Data was read from the arduino. Still,
                        available bytes: {self.arduino.in_waiting}")
                    self.custom_print(f"raw data: {data}")
                    read = True
                    # string treatment
                    data = ''.join(list(str(data))[2:6])

                # if burst is received, fpga is sending no-response data
                if int(self.arduino.in_waiting) > 16:
                    trash = self.arduino.read(self.arduino.in_waiting)
```

```
                self.custom_print("Data from the fpga received, cleaning
                    trash bytes.")
                trash = 0

            self.received_data = data
            self.wait()
        else:
            self.custom_print(f"The correct number of bytes haven't reached
                the interface!")
            self.custom_print(f"Available bytes: {self.arduino.in_waiting},
                which are:", False)
            trash = self.  arduino.read(self.arduino.in_waiting)
            self.custom_print(trash)

    except Exception as err:
        self.custom_print("Failed to receive data!")
        self.custom_print(err)

# Sends data to the arduino, thread_usage.
def write_to_arduino(self, msg):

    try:
        msg = ''.join(['0' for missing in range(4 - len(msg))]) + msg if
            len(msg) < 4 else msg
        self.custom_print(f"msg: {msg}, type(msg): {type(msg)},
            msg.encode(): {msg.encode()}")
        self.arduino.write(msg.encode())
        self.wait()
        # uart module behaviour will send the value sent to the interface,
            so it's bytes must be read and not used
        clear_response = self.arduino.read(self.byte_num)
        self.custom_print(clear_response)
    except Exception as err:
        self.custom_print("Failed to send data!")
        self.custom_print(err)

# Reads from the serial port.
def receive_data(self):

    if self.connected:
        self.read_from_arduino()
        if self.received_data:
```

```
                self.clicked_on_receive += 1
                self.received_values.append(self.received_data)
                msg = f"Interface received data: {self.received_data}"
            else:
                msg = f"Interface received data: {self.received_data}"
        else:
            msg = "Message center not connected to OS"

        self.write_text_browser(msg)
        self.custom_print(f"Clicks on receive: {self.clicked_on_receive}")
        self.custom_print("Received values: ", False)
        self.custom_print(self.received_values, True, False)

    # Sends input data to the OS.
    def send_data(self):

        if self.connected:
            data = self.get_input_data()
            if data.isdigit():
                self.clicked_on_send += 1
                self.sent_values.append(str(data))
                self.write_to_arduino(str(data))
                msg = "Message center sent " + str(data) + " to OS"
            else:
                msg = "Only integer values can be sent, please insert a valid
                    value!"
        else:
            msg = "Message center not connected to OS!"

        self.write_text_browser(msg)
        self.custom_print(f"Clicks on send: {self.clicked_on_send}")
        self.custom_print("Sent values: ", False)
        self.custom_print(self.sent_values, True, False)

    # Sets up serial connection.
    def begin_serial_connection(self):

        msg = "Setting up serial connection..."
        try:
            self.write_text_browser(msg)
            self.arduino = serial.Serial('/dev/ttyACM0', 9600)
            self.wait(2)
```

```
        except Exception as err:
            self.custom_print("Failed to stablish first connection!")
            self.custom_print(err)

        # text message, could have been any value
        #hs_msg = "9191"
        hs_msg = "0001"
        self.custom_print("Preparing hs msg.")
        # first read for initial fpga handdata
        self.read_from_arduino()
        # write sends the handshaking data
        self.custom_print("Sending hs msg.")
        self.write_to_arduino(hs_msg)
        # read result should be the same value as the handshaking message
        self.custom_print("Reading hs.")
        self.read_from_arduino()
        self.custom_print(f"Hs response: {self.received_data}.")
        status = True
        if status:
            interface_status = "Connected"
            msg = "Serial connection stablished!"
        else:
            interface_status = "Disconnected"
            msg = "Coudn't stablish serial connection D:"

        self.set_interface_status(interface_status, status)
        self.write_text_browser(msg)

        # cleanup
        self.sent_values.clear()
        self.received_values.clear()
        self.received_data = None

    # auxiliar
    def custom_print(self, msg, new_line = True, is_list = False):

        if new_line:
            if is_list:
                print(msg)
            else:
                print(f"Log: {msg}")
        else:
```

```
            print(f"Log: {msg}", end = '')

def main():
    app = QtWidgets.QApplication(sys.argv)
    window = Msg_center()
    window.show()
    app.exec()

if __name__ == '__main__':
    main()
```

APÊNDICE B – Código *Qt Message Center UI*

Algoritmo B.1 – Código gerado

```
# -*- coding: utf-8 -*-

# Form implementation generated from reading ui file 'msg_center_display.ui'
#
# Created by: PyQt5 UI code generator 5.12.1
#
# WARNING! All changes made in this file will be lost!

from PyQt5 import QtCore, QtGui, QtWidgets


class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(599, 589)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.send_frame = QtWidgets.QFrame(self.centralwidget)
        self.send_frame.setGeometry(QtCore.QRect(20, 310, 561, 80))
        self.send_frame.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.send_frame.setFrameShadow(QtWidgets.QFrame.Raised)
        self.send_frame.setObjectName("send_frame")
        self.push_button_send = QtWidgets.QPushButton(self.send_frame)
        self.push_button_send.setGeometry(QtCore.QRect(10, 10, 261, 61))
        self.push_button_send.setObjectName("push_button_send")
        self.push_button_receive = QtWidgets.QPushButton(self.send_frame)
        self.push_button_receive.setGeometry(QtCore.QRect(290, 10, 261, 61))
        self.push_button_receive.setObjectName("push_button_receive")
        self.textBrowser = QtWidgets.QTextBrowser(self.centralwidget)
        self.textBrowser.setGeometry(QtCore.QRect(20, 20, 561, 101))
        self.textBrowser.setObjectName("textBrowser")
        self.receive_frame = QtWidgets.QFrame(self.centralwidget)
        self.receive_frame.setGeometry(QtCore.QRect(20, 410, 561, 131))
        self.receive_frame.setFrameShape(QtWidgets.QFrame.StyledPanel)
```

```
self.receive_frame.setFrameShadow(QtWidgets.QFrame.Raised)
self.receive_frame.setObjectName("receive_frame")
self.label_2 = QtWidgets.QLabel(self.receive_frame)
self.label_2.setGeometry(QtCore.QRect(20, 10, 151, 17))
self.label_2.setObjectName("label_2")
self.text_browser_receive = QtWidgets.QTextBrowser(self.receive_frame)
self.text_browser_receive.setGeometry(QtCore.QRect(20, 30, 521, 71))
self.text_browser_receive.setObjectName("text_browser_receive")
self.label_3 = QtWidgets.QLabel(self.centralwidget)
self.label_3.setGeometry(QtCore.QRect(20, 230, 121, 17))
self.label_3.setObjectName("label_3")
self.text_browser_status = QtWidgets.QTextBrowser(self.centralwidget)
self.text_browser_status.setGeometry(QtCore.QRect(20, 260, 271, 31))
self.text_browser_status.setObjectName("text_browser_status")
self.push_button_start = QtWidgets.QPushButton(self.centralwidget)
self.push_button_start.setGeometry(QtCore.QRect(20, 160, 561, 61))
self.push_button_start.setObjectName("push_button_start")
self.label = QtWidgets.QLabel(self.centralwidget)
self.label.setGeometry(QtCore.QRect(310, 230, 151, 17))
self.label.setObjectName("label")
self.line_edit_data = QtWidgets.QLineEdit(self.centralwidget)
self.line_edit_data.setGeometry(QtCore.QRect(310, 260, 271, 29))
self.line_edit_data.setObjectName("line_edit_data")
self.receive_frame.raise_()
self.send_frame.raise_()
self.textBrowser.raise_()
self.label_3.raise_()
self.text_browser_status.raise_()
self.push_button_start.raise_()
self.label.raise_()
self.line_edit_data.raise_()
MainWindow.setCentralWidget(self.centralwidget)
self.menubar = QtWidgets.QMenuBar(MainWindow)
self.menubar.setGeometry(QtCore.QRect(0, 0, 599, 23))
self.menubar.setObjectName("menubar")
self.menuTESTE_PY = QtWidgets.QMenu(self.menubar)
self.menuTESTE_PY.setTitle("")
self.menuTESTE_PY.setObjectName("menuTESTE_PY")
self.menuMessage_Center = QtWidgets.QMenu(self.menubar)
self.menuMessage_Center.setObjectName("menuMessage_Center")
MainWindow.setMenuBar(self.menubar)
self.statusbar = QtWidgets.QStatusBar(MainWindow)
```

```
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)
        self.menubar.addAction(self.menuTESTE_PY.menuAction())
        self.menubar.addAction(self.menuMessage_Center.menuAction())

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.push_button_send.setText(_translate("MainWindow", "Send"))
        self.push_button_receive.setText(_translate("MainWindow", "Receive"))
        self.textBrowser.setHtml(_translate("MainWindow", "<!DOCTYPE HTML
            PUBLIC \"-//W3C//DTD HTML 4.0//EN\"
            \"http://www.w3.org/TR/REC-html40/strict.dtd\">\n"
"<html><head><meta name=\"qrichtext\" content=\"1\" /><style
    type=\"text/css\">\n"
"p, li { white-space: pre-wrap; }\n"
"</style></head><body style=\" font-family:\'Ubuntu\'; font-size:11pt;
    font-weight:400; font-style:normal;\">\n"
"<p style=\" margin-top:0px; margin-bottom:0px; margin-left:0px;
    margin-right:0px; -qt-block-indent:0; text-indent:0px;\"><span style=\"
    font-size:12pt; font-weight:600;\">Welcome to the BM32OS message
    center</span></p>\n"
"<p style=\"-qt-paragraph-type:empty; margin-top:0px; margin-bottom:0px;
    margin-left:0px; margin-right:0px; -qt-block-indent:0; text-indent:0px;
    font-size:12pt; font-weight:600;\"><br /></p>\n"
"<p style=\" margin-top:0px; margin-bottom:0px; margin-left:0px;
    margin-right:0px; -qt-block-indent:0; text-indent:0px;\"><span style=\"
    font-size:12pt; font-weight:600;\">Here you will be able to stablish
    connection with BM32OS system for read and write routines during process
    schedulling.</span></p></body></html>"))
        self.label_2.setText(_translate("MainWindow", "Received data:"))
        self.label_3.setText(_translate("MainWindow", "Interface Status:"))
        self.push_button_start.setText(_translate("MainWindow", "Start "))
        self.label.setText(_translate("MainWindow", "Data to be sent:"))
        self.menuMessage_Center.setTitle(_translate("MainWindow", "Message
            Center"))
```

APÊNDICE C – Código *Serial Manager*

Algoritmo C.1 – Código arduino

```
#include <SoftwareSerial.h>
#include <stdlib.h>

/* --- msgcenter communication data --- */

// buffer int value
int msgcenter_buff_int;
// number of bytes used in the msgcenter communication
int msgcenter_byte_num = 4;
// buffer for serial 4 byte reading/writing
char msgcenter_buff_char[] = {' ', ' ', ' ', ' '};

// waits until there is available data from the msgcenter
void wait_msgcenter_connection() {

  while (!Serial.available()) {}
}

// reads data sent by msgcenter
void read_msgcenter_serial() {

  wait_msgcenter_connection();
  Serial.readBytes(msgcenter_buff_char, msgcenter_byte_num);
}

// writes data to the msgcenter
void write_msgcenter_serial() {

  if (msgcenter_buff_int == 0) {
    fulfill_with_zeros(4);
  }
  else if (msgcenter_buff_int < 10) {
    fulfill_with_zeros(3);
  }
  else if (msgcenter_buff_int < 100) {
    fulfill_with_zeros(2);
```

```
  }
  else if (msgcenter_buff_int < 1000) {
    fulfill_with_zeros(1);
  }
  Serial.print(msgcenter_buff_int);
}

// fulfills serial connection with zero-bytes
void fulfill_with_zeros(int n_zeros) {

  for (int idx = 0; idx < n_zeros; idx++) {
    Serial.write('0');
  }
}

// clears the buffer manually
void clear_msgcenter_buffer() {

  msgcenter_buff_char[0] = (char)0;
}

// converts the buffer value to an int
void msgcenter_buffer_to_int() {

  msgcenter_buff_int = atoi(msgcenter_buff_char);
}

// converts the buffer value to char
void msgcenter_buffer_to_char() {

  clear_msgcenter_buffer();
  itoa(msgcenter_buff_int, msgcenter_buff_char, 10);
}

/* --- end of msgcenter communication data --- */

/* --- fpga communication data --- */

// rx = 10(red), tx = 11(brown)
SoftwareSerial fpga_serial(10, 11);
// buffer for serial reading/writing
int fpga_buff_int = 0;
```

```
// waits until there is available data from the fpga
void wait_fpga_connection() {

  while (!fpga_serial.available()) {}
}

// reads data sent by fpga
void read_fpga_serial() {

  wait_fpga_connection();
  //fpga_serial.readBytes(msgcenter_buff_char, msgcenter_byte_num);
  clear_fpga_buffer();
  fpga_buff_int = fpga_serial.read();
}

// writes data to the fpga
void write_fpga_serial() {

  fpga_serial.write(fpga_buff_int);
}

// clears the buffer manually
void clear_fpga_buffer() {

  fpga_buff_int = 0;
}

/* --- end of fpga communication data --- */

/* --- verbose functions --- */

// copy the value of the fpga buffer into the msgcenter buffer
void copy_buffer_fpga_to_msgcenter() {

  msgcenter_buff_int = fpga_buff_int;
}

// copy the value of the msgcenter buffer into the fpga buffer
void copy_buffer_msgcenter_to_fpga() {

  fpga_buff_int = msgcenter_buff_int;
```

```
}

/* --- end of verbose functions --- */

void setup() {

  // default serial, used for msgcenter communication
  Serial.begin(9600);
  // fpga serial, used for os communication
  fpga_serial.begin(115200);
}

void loop() {

  bool fpga = fpga_serial.available();
  bool msgcenter = Serial.available();
  // data is beeing sent from the msgcenter
  if (msgcenter) {
    read_msgcenter_serial();
    msgcenter_buffer_to_int();
    copy_buffer_msgcenter_to_fpga();
    write_fpga_serial();
  }
  // data is beeing sent by the fpga
  else if (fpga) {
    read_fpga_serial();
    copy_buffer_fpga_to_msgcenter();
    msgcenter_buffer_to_char();
    write_msgcenter_serial();
  }
  else { // zero conditions
    clear_fpga_buffer();
    clear_msgcenter_buffer();
  }
}
```

APÊNDICE D – Código *UART Manager*

Algoritmo D.1 – Código verilog

```
/*
send - control unit flag
clk_auto - 50MHz clock
output_data - data that will be sent to the interface
UART_RX - receiver, goes into the arduino tx pin
UART_TX - transmitter, goes into the arduino rx pin
data_receive - data received by UART communication
*/

module UART_Manager
#(parameter DATA_WIDTH = 16)
(
   input send,
   input clk_auto,
   input [(DATA_WIDTH/2)-1:0] output_data,
   input UART_RX,
   output UART_TX,
   output wire [(DATA_WIDTH/2)-1:0] data_receive
);

wire rx_data, tx_data, rdempty, write;
wire [(DATA_WIDTH/2)-1:0] uart_data_read, uart_data_write, data_send;
reg [DATA_WIDTH-1:0] UART_reg;
reg read, count, send_reg, send_enabler;

// uart data
assign UART_TX = tx_data;
assign rx_data = UART_RX;

// send/receive data directioning
assign data_send = output_data;
assign data_receive = UART_reg;

// send/receive treatment
assign uart_data_write = send_reg ? data_send : uart_data_read;
assign write = ( read & (~rdempty) ) || send_reg;
```

```
uart_control UART0(
   .clk(clk_auto),
   .reset_n(1'b1),
   // tx
   .write(write),
   .writedata(uart_data_write),
   // rx
   .read(read),
   .readdata(uart_data_read),
   .rdempty(rdempty),
   .uart_clk_25m(count),
   .uart_tx(tx_data),
   .uart_rx(rx_data)
);

always@(posedge clk_auto)
begin
  if (~rdempty)
      read <= 1'b1;
  else
      read <= 1'b0;
end

always@(posedge clk_auto)
begin
      if (send)
      begin
         UART_reg <= 16'b0;
      end
      if (write)
      begin
         UART_reg <= {8'b0 ,uart_data_read};
      end
      else
      begin
         UART_reg <= UART_reg;
      end
end

always@(posedge clk_auto)
begin
```

```
   count <= count + 1'b1;
end

always@(posedge clk_auto)
begin
   // occurs only on send instruction
   if(send)
   begin
      // send enablers works holding send_reg value as 1 during only one pulse
      if( send_enabler == 1'b0 )
      begin
         send_enabler = 1'b1;
         send_reg = 1'b1;
      end
      // else statement doesn't let send_reg be 1 again until there is another
          send instruction
      else
      begin
         send_reg = 1'b0;
      end
   end
   else
   begin
      send_enabler = 1'b0;
      send_reg = 1'b0;
   end
end

endmodule
```

APÊNDICE E – Código BM32OS

Algoritmo E.1 – Código cminus

```
// Bash messages:
// 0 = Welcome to BM32OS System
// 1 = Operations: 1-proc, 2-file
// 2 = How many files to execute?
// 3 = This file does not exist!
// 4 = Running processes...
// 5 = All processes finished running
// 6 = Choose your file operation
// 7 = 1 - create, 2 - rename, 3 - delete
// 8 = Too bad, hd is full!
// 9 = Enter file's instruction amount
// 10 = Instruction's first 16 bits
// 11 = Instruction's last 16 bits
// 12 = Insert a ID for the new file
// 13 = New ID can't already exist!
// 14 = File created succesfully
// 15 = Choose a file to be renamed
// 16 = Chose a existent file!
// 17 = Choose the new file ID
// 18 = Chosen ID must be unique!
// 19 = File was succesfully renamed
// 20 = Choose a file to be deleted
// 21 = File was succesfully deleted
// 29 = Insert the file's ID's
// 30 = Process requires input...
// 31 = Process requires output...
// 32 = A process finished running
// 33 = A process requires uart input
// 34 = A process requires uart output

/**** OS related values ****/

// maximum number of processes suported by the system
int MAX_PROC_NUM;

// maximum number of files suported by the system
```

```
int MAX_FILE_NUM;

// actual number of files stored on hd
int FILE_AMOUNT;

// special register interruption value
int INTERRUPTION;

// HD region for each file
int HD_REGION;

// data memory region size
int MEM_REGION;

// yolo variable for bash freezing
int STOP_BASH;

/**** File data structure ****/

// file ID's
int file_IDS[15];

// file initial HD position
int file_begin[15];

// file final HD position
int file_end[15];

/**** Processes data structure ****/

// process state related to it's execution | 1 - finished, 0 - ready to be
    executed
int proc_state[15];

// process queue for round robin
int proc_queue[15];

// processes respective program counters
int proc_PCS[15];

/**** Process related functions ****/
```

```
void reset_program_counters( void ) {
   int idx;

   idx = 1;
   while( idx < MAX_PROC_NUM ) {
      proc_PCS[ idx ] = 0;
      idx = idx + 1;
   }
   return;
}

// resets the process queue for future process executions
void reset_process_queue( void ) {
   int idx;

   idx = 0;
   while( idx < MAX_PROC_NUM ) {
      proc_queue[ idx ] = 0;
      idx = idx + 1;
   }
   return;
}

// checks the file ID, returns 1 case it's found and 0 otherwise
int check_ID( int ID ) {
   int idx;
   int ret_value;

   idx = 1;
   ret_value = 0;
   while( idx < MAX_PROC_NUM ) {
      if( file_IDS[ idx ] == ID ) {
         ret_value = 1;
         idx = MAX_PROC_NUM;
      }
      else {
         idx = idx + 1;
      }
   }
   return ret_value;
}
```

```
// given an process ID, returns it's respective index on the process (file)
    array
int get_proc_idx( int proc_ID ) {
   int idx;
   int proc_idx;

   idx = 1;
   while( idx < MAX_PROC_NUM ) {
      if( file_IDS[ idx ] == proc_ID ) {
         proc_idx = idx;
         idx = MAX_PROC_NUM;
      }
      else {
         idx = idx + 1;
      }
   }
   return proc_idx;
}

void load_proc_context( int proc_idx ) {
   int transfer_iterations;
   int transfer_idx;

   transfer_iterations = file_end[ proc_idx ] - file_begin[ proc_idx ];
   transfer_idx = 0;
   while( transfer_idx < transfer_iterations ) {
      // proc_num, track_line, mem_line
      move_HD_mem( proc_idx, transfer_idx, transfer_idx );
      transfer_idx = transfer_idx + 1;
   }
   transfer_idx = proc_idx * MEM_REGION;
   load_reg_context( 1, transfer_idx, 1 );
   load_reg_context( 2, transfer_idx, 2 );
   load_reg_context( 3, transfer_idx, 3 );
   load_reg_context( 4, transfer_idx, 4 );
   load_reg_context( 5, transfer_idx, 5 );
   load_reg_context( 6, transfer_idx, 6 );
   load_reg_context( 7, transfer_idx, 7 );
   load_reg_context( 8, transfer_idx, 8 );
   load_reg_context( 9, transfer_idx, 9 );
   load_reg_context( 10, transfer_idx, 10 );
   load_reg_context( 11, transfer_idx, 11 );
```

```
   load_reg_context( 12, transfer_idx, 12 );
   load_reg_context( 13, transfer_idx, 13 );
   load_reg_context( 14, transfer_idx, 14 );
   load_reg_context( 15, transfer_idx, 15 );
   load_reg_context( 16, transfer_idx, 16 );
   load_reg_context( 17, transfer_idx, 17 );
   load_reg_context( 18, transfer_idx, 18 );
   load_reg_context( 19, transfer_idx, 19 );
   load_reg_context( 20, transfer_idx, 20 );
   load_reg_context( 21, transfer_idx, 21 );
   load_reg_context( 22, transfer_idx, 22 );
   load_reg_context( 23, transfer_idx, 23 );
   load_reg_context( 24, transfer_idx, 24 );
   load_reg_context( 25, transfer_idx, 25 );
   load_reg_context( 26, transfer_idx, 26 );
   load_reg_context( 27, transfer_idx, 27 );
   load_reg_context( 28, transfer_idx, 28 );
   load_reg_context( 29, transfer_idx, 29 );
   load_reg_context( 30, transfer_idx, 30 );
   load_reg_context( 31, transfer_idx, 31 );
   return;
}

void store_proc_context( int proc_idx ) {
   int transfer_idx;

   transfer_idx = proc_idx * MEM_REGION;
   store_reg_context( 1, transfer_idx, 1 );
   store_reg_context( 2, transfer_idx, 2 );
   store_reg_context( 3, transfer_idx, 3 );
   store_reg_context( 4, transfer_idx, 4 );
   store_reg_context( 5, transfer_idx, 5 );
   store_reg_context( 6, transfer_idx, 6 );
   store_reg_context( 7, transfer_idx, 7 );
   store_reg_context( 8, transfer_idx, 8 );
   store_reg_context( 9, transfer_idx, 9 );
   store_reg_context( 10, transfer_idx, 10 );
   store_reg_context( 11, transfer_idx, 11 );
   store_reg_context( 12, transfer_idx, 12 );
   store_reg_context( 13, transfer_idx, 13 );
   store_reg_context( 14, transfer_idx, 14 );
   store_reg_context( 15, transfer_idx, 15 );
```

```
   store_reg_context( 16, transfer_idx, 16 );
   store_reg_context( 17, transfer_idx, 17 );
   store_reg_context( 18, transfer_idx, 18 );
   store_reg_context( 19, transfer_idx, 19 );
   store_reg_context( 20, transfer_idx, 20 );
   store_reg_context( 21, transfer_idx, 21 );
   store_reg_context( 22, transfer_idx, 22 );
   store_reg_context( 23, transfer_idx, 23 );
   store_reg_context( 24, transfer_idx, 24 );
   store_reg_context( 25, transfer_idx, 25 );
   store_reg_context( 26, transfer_idx, 26 );
   store_reg_context( 27, transfer_idx, 27 );
   store_reg_context( 28, transfer_idx, 28 );
   store_reg_context( 29, transfer_idx, 29 );
   store_reg_context( 30, transfer_idx, 30 );
   store_reg_context( 31, transfer_idx, 31 );
   return;
}

// round robin algorithm for process scheduling
void round_robin( int proc_amount ) {
   int proc_ID;
   int finished_procs;
   int queue_idx;
   int proc_idx;
   int io_var;
   int proc_pc;

   finished_procs = 0;
   queue_idx = 0;
   proc_pc = 0;
   write_lcd( 4 );
   STOP_BASH = input( );
   while( finished_procs < proc_amount ) {
      proc_ID = proc_queue[ queue_idx ];
      proc_idx = get_proc_idx( proc_ID );
      // checking if the selected process has already finished executing
      if( proc_state[ proc_idx ] == 0 ) {
         //output( proc_idx );
         proc_pc = proc_PCS[ proc_idx ];
         set_proc_pc( proc_pc );
         load_proc_context( proc_idx );
```

```
swap_process( proc_idx );
recover_OS( );
// get_interruption returns the value from register[28]
INTERRUPTION = get_interruption( );
proc_PCS[ proc_idx ] = get_proc_pc( );
// input interruption
if( INTERRUPTION == 2 ) {
   write_lcd( 30 );
   io_var = input( );
   output( io_var );
   // moves data from io_var, which belongs to OS, to process at
      register[27]
   move_reg_OS_proc( io_var, 27 );
}
// output interruption
if( INTERRUPTION == 3 ) {
   write_lcd( 31 );
   // moves data from register[27], which belongs to process, to
      io_var from OS
   move_reg_proc_OS( 27, io_var );
   output( io_var );
}
// end program interruption
if( INTERRUPTION == 4 ) {
   write_lcd( 32 );
   output( proc_idx );
   proc_state[ proc_idx ] = 1;
   finished_procs = finished_procs + 1;
}
// receive interruption
if( INTERRUPTION == 5 ) {
   write_lcd( 33 );
   io_var = UART_input( );
   output( io_var );
   // moves data from io_var, which belongs to OS, to process at
      register[27]
   move_reg_OS_proc( io_var, 27 );
}
// send interruption
if( INTERRUPTION == 6 ) {
   write_lcd( 34 );
```

```
            // moves data from register[27], which belongs to process, to
                io_var from OS
            move_reg_proc_OS( 27, io_var );
            UART_output( io_var );
         }
         store_proc_context( proc_idx );
      }
      // determinating next process and ensuring queue circularity
      if( queue_idx < proc_amount - 1 ) {
         queue_idx = queue_idx + 1;
      }
      else {
         queue_idx = 0;
      }
   }
   write_lcd( 5 );
   STOP_BASH = input( );
   return;
}

// process operation for schedulling setup
void process_operation( void ) {
   int proc_amount;
   int queue_idx;
   int proc_ID;

   write_lcd( 2 );
   proc_amount = input( );
   output( proc_amount );
   queue_idx = 0;
   write_lcd( 29 );
   while( queue_idx < proc_amount ) {
      proc_ID = input( );
      while( check_ID( proc_ID ) != 1 ) {
         write_lcd( 3 ) ;
         proc_ID = input( );
      }
      output( proc_ID );
      proc_queue[ queue_idx ] = proc_ID;
      proc_state[ queue_idx ] = 0;
      queue_idx = queue_idx + 1;
   }
```

```
   round_robin( proc_amount );
   reset_process_queue( );
   reset_program_counters( );
   return;
}

 /**** File related functions ****/

// get next free position on the file array
int get_free_file_position( void ) {
   int idx;
   int ret_idx;

   idx = 1;
   ret_idx = 0;
   while( idx < MAX_FILE_NUM ) {
      if( file_IDS[ idx ] == 99 ) {
            ret_idx = idx;
            idx = MAX_FILE_NUM;
      }
      else {
         idx = idx + 1;
      }
   }
   return ret_idx;
}

// create file
void create_file( int file_idx ) {
   int file_ID;
   int file_instructions;
   int idx;
   int first_bits;
   int last_bits;
   int instruction;

   write_lcd( 9 );
   file_instructions = input( );
   output( file_instructions );
   idx = 0;
   while( idx < file_instructions ) {
      write_lcd( 10 );
```

```
      first_bits = input( );
      write_lcd( 11 );
      last_bits = input( );
      instruction = concatenate( first_bits, last_bits );
      // file_num, track_line, data
      store_HD( file_idx, idx, instruction );
      idx = idx + 1;
   }
   // naming the file
   write_lcd( 12 );
   file_ID = input( );
   output( file_ID );
   while( check_ID( file_ID ) == 1 ) {
      write_lcd( 13 );
      file_ID = input();
   }
   file_IDS[ file_idx ] = file_ID;
   file_begin[ file_idx ] = 0;
   file_end[ file_idx ] = file_instructions;
   write_lcd( 14 );
   STOP_BASH = input( );
   return;
}

// file renaming
void rename_file( void ) {
   int idx;
   int file_ID;
   int new_ID;

   write_lcd( 15 );
   file_ID = input( );
   output( file_ID );
   while( check_ID( file_ID ) != 1 ) {
      write_lcd( 16 );
      file_ID = input( );
   }
   write_lcd( 17 );
   new_ID = input( );
   output( new_ID );
   while( check_ID( new_ID ) == 1 ) {
      write_lcd( 18 );
```

```
      new_ID = input( );
   }
   idx = 1;
   while( idx < MAX_PROC_NUM ) {
      if( file_IDS[ idx ] == file_ID ) {
         file_IDS[ idx ] = new_ID;
         idx = MAX_PROC_NUM;
      }
      else {
         idx = idx + 1;
      }
   }
   write_lcd( 19 );
   STOP_BASH = input( );
   return;
}

// file deletion
void delete_file( void ) {
   int idx;
   int file_ID;

   write_lcd( 20 );
   file_ID = input( );
   output( file_ID );
   while( check_ID( file_ID ) != 1 ) {
      write_lcd( 16 );
      file_ID = input( );
   }
   idx = 1;
   while( idx < MAX_PROC_NUM ) {
      if( file_IDS[ idx ] == file_ID ) {
         // erase HD references to file
         file_IDS[ idx ] = 99;
         file_begin[ idx ] = 0;
         file_end[ idx ] = 0;
         idx = MAX_PROC_NUM;
      }
      else {
         idx = idx + 1;
      }
   }
```

```
   write_lcd( 21 );
   STOP_BASH = input( );
   return;
}

// file operation for file creating/renaming/deleting
void file_operation( void ) {
   int file_idx;
   int file_ID;
   int file_position;
   int file_option;

   write_lcd( 6 );
   STOP_BASH = input( );
   write_lcd( 7 );
   file_option = input( );
   output( file_option );
   // create files
   if( file_option == 1 ) {
      if ( MAX_FILE_NUM == FILE_AMOUNT ) {
         write_lcd( 8 );
         STOP_BASH = input( );
      }
      else {
         file_idx = get_free_file_position( );
         create_file( file_idx );
         FILE_AMOUNT = FILE_AMOUNT + 1;
      }
   }
   // rename files
   if( file_option == 2 ){
      rename_file( );
   }
   // delete files
   if( file_option == 3 ){
      delete_file( );
      FILE_AMOUNT = FILE_AMOUNT - 1;
   }
   return;
}

 /**** Main functions ****/
```

```
void bash( void ) {
   int bash_option;

   write_lcd( 1 );
   bash_option = input( );
   output( bash_option );
   if( bash_option == 1 ) {
      process_operation( );
   }
   if( bash_option == 2 ) {
      file_operation( );
   }
   return;
}

void init( void ) {
   int idx;
   // setting up system variables
   MAX_PROC_NUM = 15;
   MAX_FILE_NUM = 15;
   FILE_AMOUNT = 10;
   HD_REGION = 1024;
   MEM_REGION = 252;

   idx = 1;
   // naming known files to it's respecctive index
   while( idx < MAX_PROC_NUM ) {
      if( idx < 10) {
         // if it is known, file_IDS[ idx ] = idx
         file_IDS[ idx ] = idx;
      }
      else {
         // if it isn't know, file_IDS[ idx ] = 99
         file_IDS[ idx ] = 99;
      }
      // reseting all of the processes program counters
      proc_PCS[ idx ] = 0;
      idx = idx + 1;
   }
   // programming good manners
   file_begin[ 0 ] = 0;
```

```
   file_end[ 0 ] = 0;
   // program 1 (pow) default information - net
   file_begin[ 1 ] = 0;
   file_end[ 1 ] = 51;
   // program 2 (fib) default information - net
   file_begin[ 2 ] = 0;
   file_end[ 2 ] = 68;
   // program 3 (max) default information - net
   file_begin[ 3 ] = 0;
   file_end[ 3 ] = 75;
   // program 4 (min) default information
   file_begin[ 4 ] = 0;
   file_end[ 4 ] = 76;
   // program 5 (selection_sort) default information
   file_begin[ 5 ] = 0;
   file_end[ 5 ] = 154;
   // program 6 (array average) default information
   file_begin[ 6 ] = 0;
   file_end[ 6 ] = 72;
   // program 7 () default information
   file_begin[ 7 ] = 0;
   file_end[ 7 ] = 0;
   // program 8 () default information
   file_begin[ 8 ] = 0;
   file_end[ 8 ] = 0;
   // program 9 () default information
   file_begin[ 9 ] = 0;
   file_end[ 9 ] = 0;
   return;
}

void main( void ) {
   init( );
   write_lcd( 0 );
   STOP_BASH = input( );
   while( 1 < MAX_PROC_NUM ) {
      bash( );
   }
   return;
}
```