UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA - ENGENHARIA DE COMPUTAÇÃO
PCS3732 - LABORATÓRIO DE PROCESSADORES (2024)

Henrique Murakami de Paula - NUSP 12551154
Italo Roberto Lui - NUSP 12553991
Rafael Menas Tamasi - NUSP 12553775

PiCLIs
Documentação Final

SÃO PAULO
2024

- **Contexto**

O projeto PiCLIs é uma proposta de CLI (*command line interface*) para interação direta com a placa Raspberry Pi, fazendo uso de comunicação serial pela UART para o uso das diferentes funcionalidades e módulos do mini computador.

- **Motivação**

A motivação para a criação desse projeto foi o desejo de ter disponível um shell ou um CLI com a possibilidade de criação de comandos personalizados. Após ver as possibilidades de interação com o *gdbstub* em conjunto com o gdb, principalmente no que diz respeito ao controle e visualização de posições de memória, tivemos a ideia de criar uma espécie de CLI baseado no *gdbstub*, mas menos orientado à depuração.

- **Objetivos e Modelo do Problema**

Dada a motivação anterior, o objetivo principal do projeto foi a criação de um programa que aja como uma extensão do programa *gdbstub* usado durante as aulas, mas não orientado à depuração, e sim à manipulação e controle dos diferentes componentes do *Raspberry Pi*, possibilitando a conexão com comunicação serial sem o uso do *gdb* (usando somente um terminal serial usual) e o uso de novos comandos definidos pelo grupo. Os comandos a serem adicionados devem fazer uso de formas diversas dos módulos da placa, como a comunicação serial, a memória e as GPIOs (como o led verde presente nativamente na placa).

A primeira modelagem do problema, logo, depende da divisão do problema em passos:

- O programa *gdbstub* terá configurações específicas para conexão serial entre o computador e a placa, feitas especificamente para o uso conjunto com o *gdb*;
- O programa base é orientado à depuração, e logo, a preparação inicial com envio de estados e determinação de *breakpoints* deve ser revisada;
- Apesar de implementar uma biblioteca para padronizar a comunicação pela UART, o programa base não implementa o uso de GPIOs;
- Comandos presentes no programa base podem ser mantidos, mas extensões devem ser realizadas com a criação de comandos específicos para o PiCLIs.

- **Modelo de Solução**

O programa foi construído tomando como base o programa *gdbstub* fornecido em aula, mas sua função principal é a de permitir a manipulação e a visualização rápida de informações referentes ao Raspberry Pi. Dada a natureza flexível do projeto, comandos variados podem ser incorporados, como comandos para uso dos

LEDs, transmissão serial, leitura e edição de memória e até mesmo pesquisa na memória da placa.

A estruturação geral do PiCLIs é, essencialmente, uma fusão do *gdbstub* com funções auxiliares para o circuito integrado BCM2836, para a comunicação com UART, e para o uso das GPIOs. Ele é carregado diretamente na memória da placa por meio de um cartão SD, e como seu uso primário não é para depuração, ele não precisa acompanhar o uso do módulo *gdb-multiarch* no computador. A implementação de uso das GPIOs foi baseada na biblioteca *GPIO* construída em sala, e integrada com o projeto de forma modular.

Alguns artifícios, como os envios de status, são menos usados no PiCLIs, pois tornaram-se desnecessários para a sincronização de mensagens. As funções originais disponíveis no *gdbstub* não deveriam ser alteradas, mas as extensões criadas podem ser diferenciadas pelo uso de um prefixo nos comandos.

Após a inicialização, comandos podem ser recebidos através da leitura constante de caracteres de forma serial (considerando os casos nos quais um comando não existe) e seus parâmetros são recebidos posteriormente. Para os comandos do PiCLIS, convencionamos que seu formato é "**$p<comando em letras maiúsculas> <parâmetro>**". Após a leitura de um comando reconhecido, o programa realiza um salto para a rotina de tratamento apropriada, e sua execução só é iniciada após a tecla ENTER ser apertada (na realidade, após o *carriage return* ser detectado).

A execução das instruções em si pode variar significativamente, dependendo de seu propósito. A escolha dos comandos implementados e o fluxo de execução de cada uma delas serão explicados mais detalhadamente a seguir.

- **Protótipo**

- Pinagem e Conexões

Como os comandos são enviados serialmente, as únicas conexões necessárias são as empregadas na UART, por meio de um conversor USB-serial:

Sinal da GPIO	Pino
GND	6
GPIO15_UART0RXD (3.3V)	8
GPIO14_UART0TXD (3.3V)	10

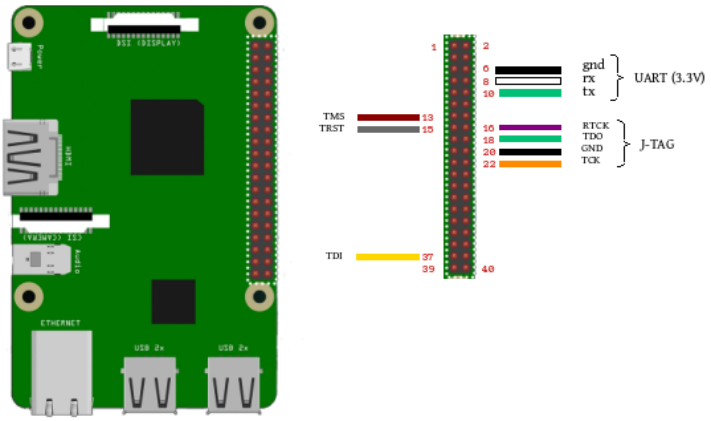


Imagem 1: Conexões de comunicação da porta de expansão do Raspberry Pi 2B.

É conveniente testar o conversor USB-serial através de um fio conectando seu TX e RX (ao enviar uma mensagem, ela deve ser visível no terminal serial). A comunicação serial é realizada com 115200 bauds, 8 bits de dados, 1 bit de início (nível lógico “0”) e 1 bit de parada (nível lógico “1”), e as funções que facilitam o uso da UART foram planejadas com este padrão em mente. O terminal mais usado durante o desenvolvimento desse projeto foi o Minicom, mas terminais como CuteCom e *screen* também podem ser utilizados.

Imagem 2: Conexões do conversor USB-serial para o funcionamento da transmissão pela UART.

- Comandos do PiCLIs

Além dos comandos já presentes no *gdbstub*, as instruções adicionais elaboradas são apresentadas abaixo com os respectivos comandos):

- **Morse ($pMORSE <palavra>)**: Usa o LED verde da placa (GPIO 47) para piscar em código morse uma palavra fornecida. A palavra fornecida pode ter até 99 caracteres alfanuméricos (cada um usa 2 caracteres hexadecimais, ou 8 bits). O fluxo normal de execução só retorna após o fim da sinalização morse.

A implementação desse comando foi baseada em um *switch case* com uma sequência de instruções gpio_put (determina se o pino é de entrada ou saída, ou neste caso, se o LED está ligado ou desligado), gpio_toggle (troca o estado do led) e delay (espera por certo tempo). Cada um dos caracteres é armazenado em um array assim que recebido, e a sinalização morse acontece pela iteração desse array com uma função que converte caracteres ASCII para sequências de mudanças de estado do LED.

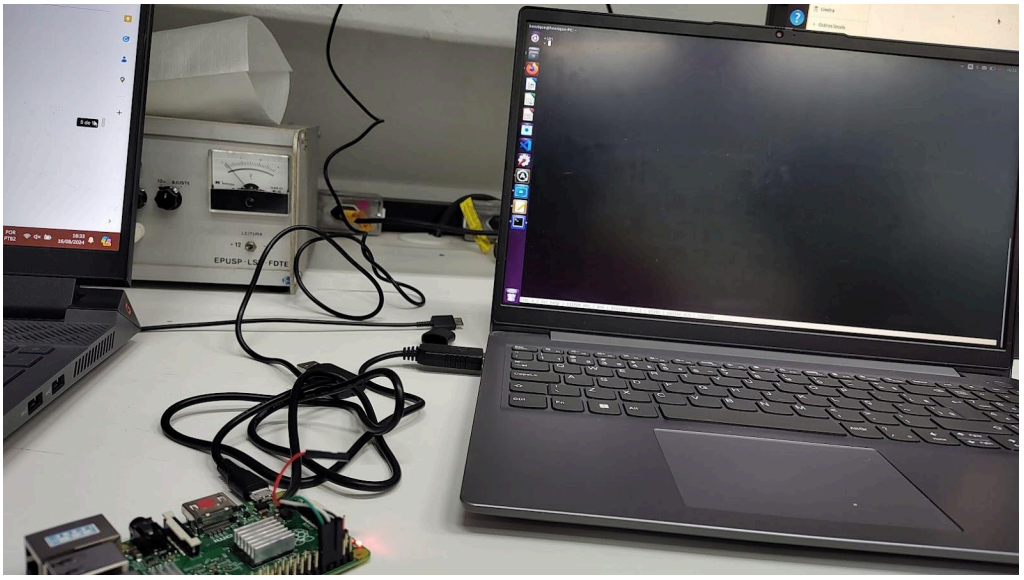


Imagem 3: Exemplos de execução do comando $pMORSE.

```
trata_morse:
    /*
    * Pisca uma palavra formada por caracteres alfanuméricos em código morse pelo LED verde da placa.
    * A palavra pode ter até 99 caracteres.
    * Formato do comando: $pMORSE <palavra>
    */
    skip(' ');
    char palavra[100];
    int letra = 0;
    uint8_t caractere = uart_getc();
    char caractere_char = (char) caractere;
    while (caractere_char != '\r' && letra < 99) {
        palavra[letra] = caractere_char;
        letra++;
        caractere = uart_getc();
        caractere_char = (char) caractere;
    }
    uart_puts("+");

    for (int i = 0; i < letra; i++) {
        char_to_morse(palavra[i]);
        uart_putc(palavra[i]);
        gpio_put(47, 0); // Desliga
        delay(3000000); // Espaço entre caracteres (3 unidades)
    }
    goto retry;
```

```
/**
 * Recebe um caractere (byte hexadecimal) e faz o LED verde (GPIO 47) piscá-lo em código morse.
 * @param c Valor a codificar (8 bits).
 */
void char_to_morse(char c) {
    if (((c >= '0') && (c <= '9')) || ((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z'))) {
        gpio_put(47, 1);
        switch (c) {
        case 'a':
        case 'A':
            delay(1000000); // Ponto (1 unidade)
            gpio_toggle(47);
            delay(1000000); // Espaço entre partes do mesmo caractere (1 unidade)
            gpio_toggle(47);
            delay(3000000); // Traço (3 unidades)
            break;
        case 'b':
        case 'B':
            delay(3000000); // Traço
            gpio_toggle(47);
            delay(1000000); // Espaço
            gpio_toggle(47);
            delay(1000000); // Ponto
            gpio_toggle(47);
            delay(1000000); // Espaço
            gpio_toggle(47);
            delay(1000000); // Ponto
            gpio_toggle(47);
            delay(1000000); // Espaço
            gpio_toggle(47);
            delay(1000000); // Ponto
            break;
        case 'c':
        case 'C':
```

Imagens 4 e 5: Código da função $pMORSE e trechos de seu tratamento interno.

- **Echo ($pECHO <palavra>)**: Recebe uma palavra de até 99 caracteres ASCII, e retransmite a mesma palavra da placa para o computador. O fluxo normal de execução só retorna após o fim da retransmissão.

A implementação dessa instrução é similar ao comando $pMORSE, formando um array de caracteres conforme eles são recebidos e iterando a transmissão sobre cada um deles.

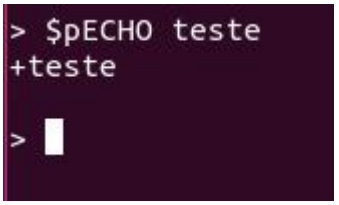


Imagem 6: Exemplos de execução do comando $pECHO.

```
trata_echo:
   /*
   * Envia uma mensagem para a placa, e retorna ela pela UART, para garantir seu funcionamento.
   * A mensagem pode ter até 99 caracteres.
   * Formato do comando $pECHO <palavra>
   */
   skip(' ');
   char palavra_echo[100];
   int letra_echo = 0;
   uint8_t caractere_echo = uart_getc();
   char caractere_char_echo = (char) caractere_echo;
   while (caractere_char_echo != '\r' && letra_echo < 99) {
      palavra_echo[letra_echo] = caractere_echo;
      letra_echo++;
      caractere_echo = uart_getc();
      caractere_char_echo = (char) caractere_echo;
   }
   uart_puts("+");

   for (int j = 0; j < letra_echo; j++) {
      uart_putc(palavra_echo[j]);
   }
   uart_puts("\r\n");
   goto retry;
```

Imagem 7: Tratamento interno da função $pECHO.

- **Binary ($pBIN <número decimal>)**: Recebe um número decimal (de valor máximo 2147483647 e valor mínimo -2147483646, ou seja, limitado por 32 bits), e converte-o para binário em complemento de 2, transmitindo-o serialmente de volta para o computador.

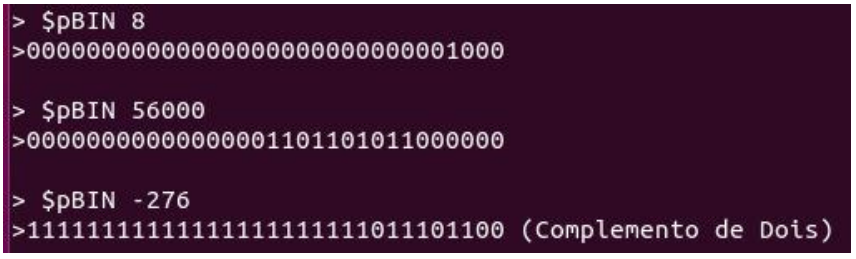


Imagem 8: Exemplos de execução do comando $pBIN.

```
trata_dectobin:
    /*
    * Converte um número fornecido de decimal para binário, tratando casos negativos com complemento de dois, e envia-o serialmente.
    * O limite do valor decimal é de 2147483647
    * Formato do comando $pBIN <número decimal>
    */
        skip(' ');

        int numero_decimal = 0;
        int is_negative = 0;
        uint8_t char_decimal = uart_getc();

        if (char_decimal == '-') {
            is_negative = 1;
            char_decimal = uart_getc();
        }

        while (char_decimal != '\r') {
            int algarismo_decimal = char_to_hex((char)char_decimal);
            numero_decimal = numero_decimal * 10 + algarismo_decimal;
            char_decimal = uart_getc();
        }

        if (is_negative) {
            numero_decimal = -numero_decimal;
        }

        uart_puts(">");

        uint32_t numero_binario = (uint32_t)numero_decimal;
        int algarismos_binarios = 32;
        char bin_str[33];
        bin_str[32] = '\0';
```

Imagem 9: Tratamento interno da função $pBIN.

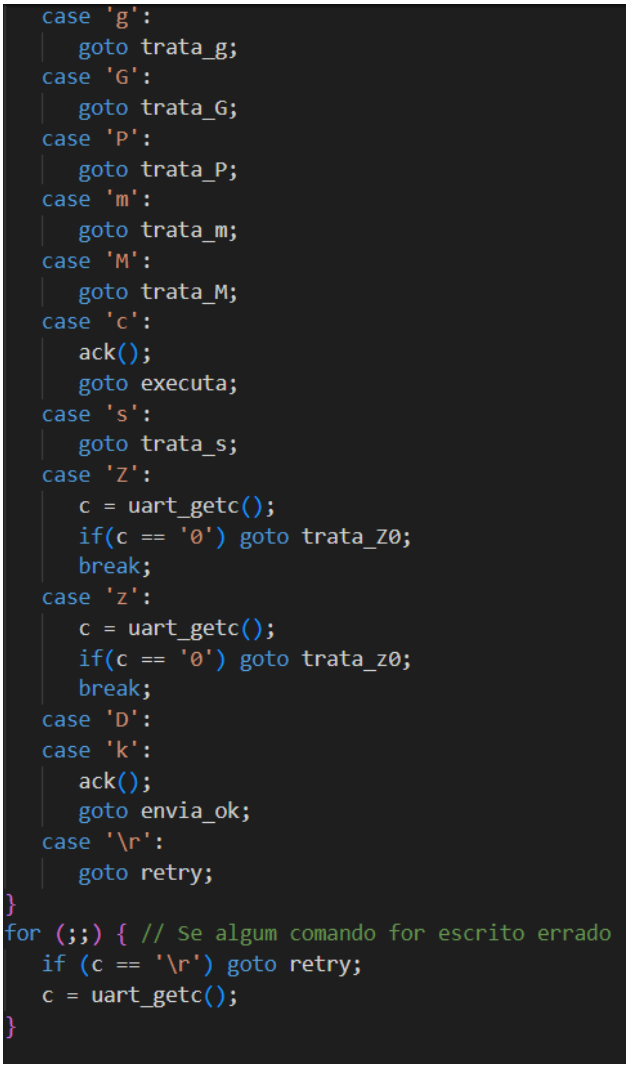
- **Search ($pSCH <palavra de 4 caracteres ASCII> <endereço inicial> <tamanho>)**: Recebe uma palavra de dados e procura por ela em uma região da memória RAM da placa, retornando seu número de ocorrências.

```
trata_search:
    /*
    * Conta o número de ocorrências de uma palavra de dados na memória (até 4 caracteres)
    */
    skip(' ');
    uint8_t first = char_to_hex(uart_getc());
    uint8_t second = char_to_hex(uart_getc());
    uint8_t third = char_to_hex(uart_getc());
    uint32_t search_word = (first << 12) | (second << 8) | (third << 4) | char_to_hex(uart_getc());     // palavra de dados
    skip(' ');
    uint32_t start_address_search = readword(' ');                  // endereço inicial
    uint32_t search_size = readword('\r');                  // tamanho da área de dados

    uint8_t times_search = compbytes(search_word, (uint8_t *) start_address_search, search_size);

    uart_puts("A palavra ");
    uart_puts(search_word);
    uart_puts(" aparece ");
    uart_putc(hex_to_char(times_search));
    uart_puts(" vezes na area procurada.");
    goto retry;
```

Imagem 10: Tratamento interno da função $pSCH.

A identificação de cada uma das instruções ocorre em um loop do fluxo de execução após a inicialização do programa e cada término de comando, como mostrado abaixo:

```
retry:
   uart_puts("\r\n> ");

   /*
    * Identifica a mensagem
    */
   c = uart_getc();
   switch(c) {
      case '?':
            goto trata_status;
      case '$':
         c = uart_getc();
         if (c == 'p') {
            c = uart_getc();
            if (c == 'B') {
               c = uart_getc();
               if (c == 'I') {
                  c = uart_getc();
                  if (c == 'N') goto trata_dectobin;
               }
               break;
            }
            if (c == 'C') {
               c = uart_getc();
               if (c == 'H') {
                  c = uart_getc();
                  if (c == 'K') goto trata_checksum;
               }
               break;
            }
            if (c == 'S') {
               c = uart_getc();
               if (c == 'C') {
                  c = uart_getc();
```



```
    case 'g':
        goto trata_g;
    case 'G':
        goto trata_G;
    case 'P':
        goto trata_P;
    case 'm':
        goto trata_m;
    case 'M':
        goto trata_M;
    case 'c':
        ack();
        goto executa;
    case 's':
        goto trata_s;
    case 'Z':
        c = uart_getc();
        if(c == '0') goto trata_Z0;
        break;
    case 'z':
        c = uart_getc();
        if(c == '0') goto trata_z0;
        break;
    case 'D':
    case 'k':
        ack();
        goto envia_ok;
    case '\r':
        goto retry;
}
for (;;) { // Se algum comando for escrito errado
    if (c == '\r') goto retry;
    c = uart_getc();
}
```

Imagens 11 e 12: Identificação dos comandos referentes ao PiCLIs, caractere por caractere, e tratamento em caso de instrução não reconhecida ou comportamento inesperado.

Além disso, foram alterados os formatos dos seguintes comandos:

- **Ler memória (m <endereço inicial> <tamanho>)**;
- **Escrever memória (M <endereço inicial> <tamanho>**, seguido de **uma quantidade de caracteres ASCII igual ao tamanho)**;

Imagem 13: Execução dos comandos de leitura e edição de memória.

- **Considerações Finais**

Através da elaboração deste projeto, vimos algumas das capacidades da placa Raspberry Pi 2 B, e tivemos bastante experiência com a interface entre ela e o computador por meio de comunicação serial. Dessa forma, o projeto foi muito bom para termos novos contatos com a programação *bare metal*, e principalmente nos adequarmos à prática de uma metodologia adequada de desenvolvimento de projeto, dado que a modularidade é uma parte essencial desse programa.