Engenharia Reversa em Sistemas Embeded

Mestrado em Cibersegurança

Gonçalo Almeida, Maria Cunha, Sofia Vaz

Engenharia Reversa em Sistemas Embeded

Engenharia Reversa

Mestrado em Cibersegurança

Gonçalo Almeida, Maria Cunha, Sofia Vaz
(79994) goncalo.almeida@ua.pt, (93089) mariastrecht@ua.pt,
(92968) sofiateixeiravaz@ua.pt

30/06/2022

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Excertos

Capítulo 1

Introdução

Este documento visa a explicitar o processo seguido para analisar uma câmara térmica ligada a um sistema *embedded* baseado num micro-controlador PIC32MC745F512H. O objeto de análise são os elementos relacionados com o funcionamento do controlador.

O documento está organizado em preâmbulo (Capítulo 2), processo de análise e conclusão. No final há ainda anexos (Capítulo 10) onde consta todo o código fonte que foi alvo de *reverse engineering* e ainda a representação visual dos resultados observados na análise de sinais.

Capítulo 2

Preâmbulo

Este capítulo visa a clarificar dúvidas que os leitores possam ter sobre o processo. Assim, detalha as ferramentas utilizadas (Seção 2.1) e termos técnicos(Seção 2.3)

2.1 Ferramentas

Este capítulo apresenta a lista de ferramentas usadas ao longo de todo o processo, com uma breve explicação do contexto no qual foram usadas.

PulseView Esta ferramenta foi usada sempre que foi necessário analisar os sinais emitidos pelos canais da da placa utilizada.

Ghidra Esta ferramenta foi utilizada para análise estática do código fornecido como estando a correr na placa. A análise é detalhada no Capítulo 8.

man pages O comando `man` foi útil no que toca a analisar código decompilado, uma vez que informou o que instruções faziam.

2.2 Termos técnicos

***Reverse Engineering*, Engenharia Reversa** *(processo)*
O processo de *reverse engineering* é a análise de sistemas de modo a identificar componentes destes e inferir como estes comunicam entre si ([1]). Assim, *reverse engineering* passa por perceber como um sistema funciona internamente sem ter acesso a detalhes de implementação.
Baud-rate *(medida)*
Taxa máxima de transferência de bits por segundo num canal de comunicação.

Visual Studio Code[2] Esta ferramenta foi usada sempre que foi necessário abrir ficheiros e ler o seu conteúdo "em branco", isto é, sem pré processamento necessário.

Ghidra[3] Esta ferramenta foi utilizada para análise estática do ficheiro `main`.

man pages O comando `man` foi extremamente útil no que toca a analisar código decompilado, uma vez que informou o que instruções faziam.

2.3 Termos técnicos

***Reverse Engineering*, Engenharia Reversa** *(processo)*

O processo de *reverse engineering* é a análise de sistemas de modo a identificar componentes destes e inferir como estes comunicam entre si ([1]). Assim, *reverse engineering* passa por perceber como um sistema funciona internamente sem ter acesso a detalhes de implementação.

Capítulo 3

RS232C

Um dos sinais do protocolo `RS232C` é utilizado para *debug*, deste modo torna-se bastante interessante a engenharia inversa deste sinal.

Para a análise das mensagens enviadas será necessário a identificação do sinal e das especificações do protocolo, como *baudrate*, *parity*, *number of stop bits* e *data bits*.

3.1 Identificação de Sinais

Para a análise do protocolo `RS232C` foi necessário primeiramente identificar quais dos pinos mencionados como *output* do `RS232C`, `OC4` e `OC3`, são o *Tx* e o *Rx*.

Para este propósito foram conectados esses dois pinos a dois cabos e ligamos a um analisador ligado ao computador. Assim, com a ajuda do *Pulseview*, foi possível visualizar os sinais enviados.

A identificação dos sinais foi simples visto que apenas um dos pinos tinha variações de sinal, o `OC4`. Sendo assim, é possível concluir que o `OC4` é o *Tx* e o `OC3` é o *Rx*.

3.2 Estrutura de Dados

Visualizando o sinal no com a configuração default do *PulseView* não foi possível interpretar o sinal. Foi necessário obter as especificações do sinal primeiro.

Para o cálculo do *baudrate* do `RS232C` foi calculado o menor intervalo em que um valor foi 1 ou 0. E depois foi calculado o *baudrate*, que resultou em 38400 hz (f = 1/26.00 micros = 38.46 kHz = 38 460 Hz → baudrate = 38 460 bps +- 38 400 bps).

Este cálculo pode ser confirmado com a ajuda do *PulseView* que, como visível no Figura 3.1, calcula o intervalo e a frequência.

Tendo agora o *baudrate* foi apenas necessário testar as várias possibilidades até conseguir um sinal válido, ou seja, que não tenha nem *frame errors* nem *parity error*.

As especificações obtidas foram as seguintes:

- *baudrate* : 38400 hz
- *data bits* : 8 bits
- *parity* : *even*, par

Figura 3.1: Cálculo do *baudrate* do RS232C

- *stop bits* : 1 bit

Com as especificações do RS232C corretas já foi possível interpretar o sinal que, como visível na Figura 3.2, já é *human readable*.

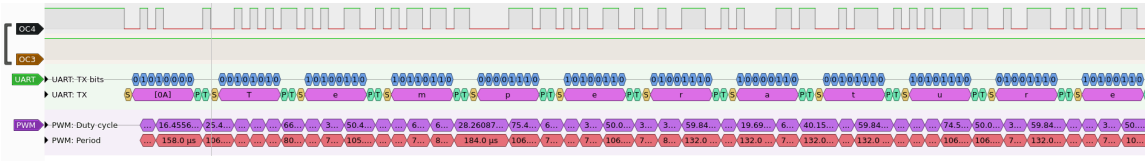

Figura 3.2: Captura do barramento RS232C com as especificações corretas

3.3 Estrutura das Mensagens DEBUG

Após a correta identificação do sinal *Tx* e das especificações corretas do RS232C, é possível analisar a estruturas das mensagens enviadas.

Uma das possibilidades de análise das mensagens seria com a utilização de um conversor de RS232C para USB, mas devido a incompatibilidades com esta hipótese foi descartada.

Sendo assim, decorremos a segunda opção que foi pela análise do output através do *PulseView*.

Inicialmente, foi possível concluir que eram impressos valores de temperatura e valores dos *set points*, mas não especificamente que valores e por que ordem eram estes impressos. Variando os valores de temperatura foi possível concluir que as mensagens enviadas têm a estrutura mostrada no Listing 3.1.

Listing 3.1: Estrutura das mensagens enviadas pelo RS232C

```
Temperatures:
{{ os ultimos 25 valores do sensor de temperatura 1 do mais antigo
para o mais recente, separados por espa os }}
{{ os ultimos 25 valores do sensor de temperatura 2 do mais antigo
para o mais recente, separados por espa os }}
Set Points:
```

`{{ set point 1 }} {{ set point 2 }} {{ set point da average temperature }}`

Esta mensagem é enviada em períodos de 12,4 segundos, como visível na Figura 3.3.

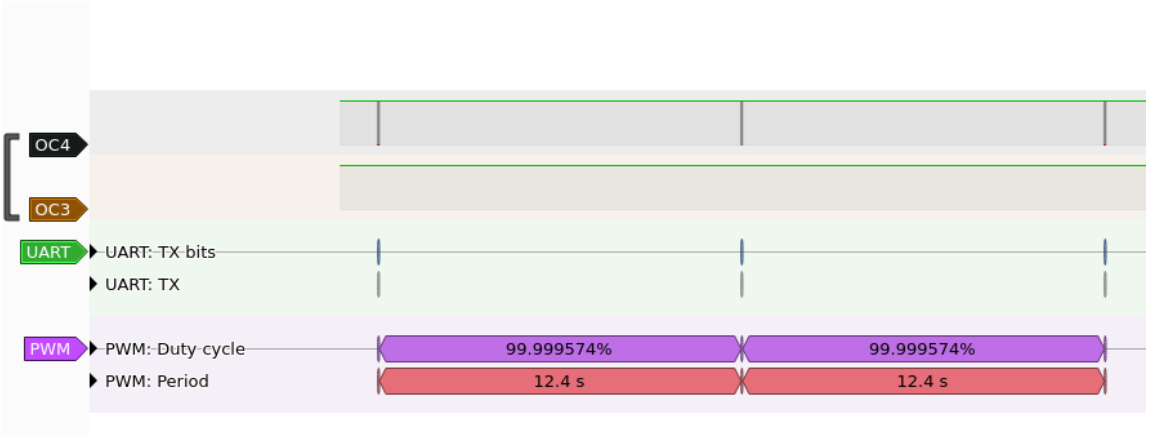

Figura 3.3: Calculo do período das mensagens do `RS232C`

Capítulo 4

SPI

4.1 Identificação dos Pinos

Para ser possível a análise do sinal do `SPI`, foi necessário a identificação dos pinos com os sinais do *standard* `SPI`.

Analisando o comportamento dos sinais obtidos no *PulseView* foi possível identificar a correlação dos pinos com cada sinal.

A identificação concluída está demonstrada na Figura 4.1.

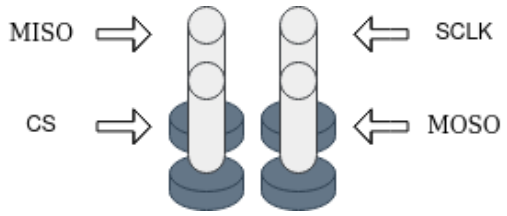

Figura 4.1: Identificação dos pinos do `SPI`

Após a correta identificação dos sinais foi obtido o seguinte, Figura 4.2.

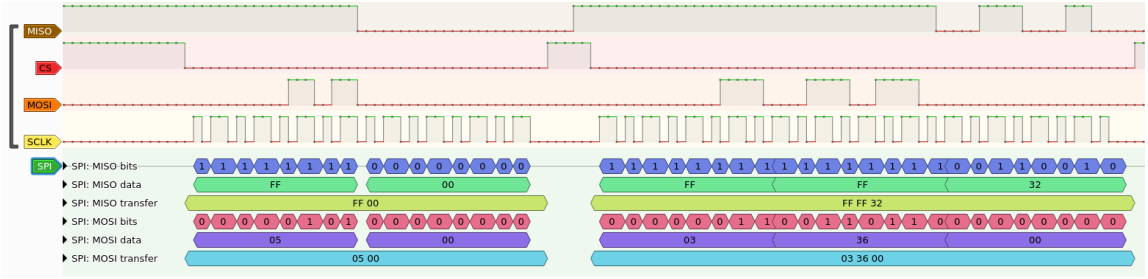

Figura 4.2: Snippet do `SPI`

4.2 Análise do Sinal

Agora, para obter o *baudrate* do sinal foi calculado o período do sinal. Com a ajuda do *PulseView* o valor obtido foi (Figura 4.3).

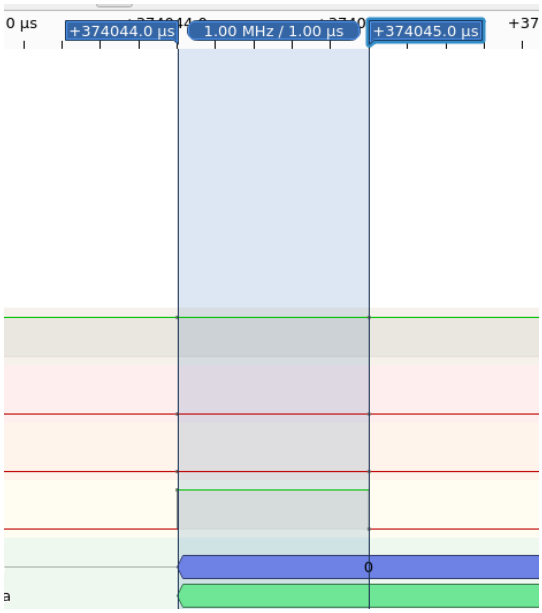

Figura 4.3: Cálculo do *baudrate* do `SPI`

4.3 EEPROM

Analisando o comportamento em geral do `SPI` é possível concluir que a EEPROM é utilizada para armazenar as temperaturas e comporta-se de forma similar a um *buffer* de memória circular.

No início de cada instrução é sempre obtido informação armazenada nos endereços `0x35` e `0x36`.

Após analise é possível concluir que no primeiro endereço acedido está guardado o tamanho do *buffer*, que é 50 (0x32), e a *head* do dito *buffer*, que vai alterando, visto que é circular.

A temperatura mais recente é sempre colocada na *head* do *buffer*, mas como a *head* é incrementada a cada escrita, a *head* contem a temperatura mais antiga.

Para além disso, também são utilizados os endereços `0x37`, `0x38` e `0x39` para guardar os valores do *set point* 1, do *set point* 2 e do *set point* da *average temperature* respetivamente.

4.4 Operações

4.4.1 Ler temperaturas

Inicialmente é obtido o tamanho do *buffer* e o endereço da sua *head*, como mencionado na secção anterior.

Depois, é verificado antes de cada leitura se a EEPROM esta a efetuar operações de escrita, através do *read status register*, e caso não esteja, é lido um dos valores do *buffer*.

Analisando a Figura 4.4, é possível concluir que o primeiro endereço a ser lido é o endereço da *head* mais 3, ou seja, o terceiro valor de temperatura armazenado mais antigo. Isto é visível, pois é obtido o valor da *head* que neste exemplo é 0x14 e o primeiro endereço lido é o 0x17.

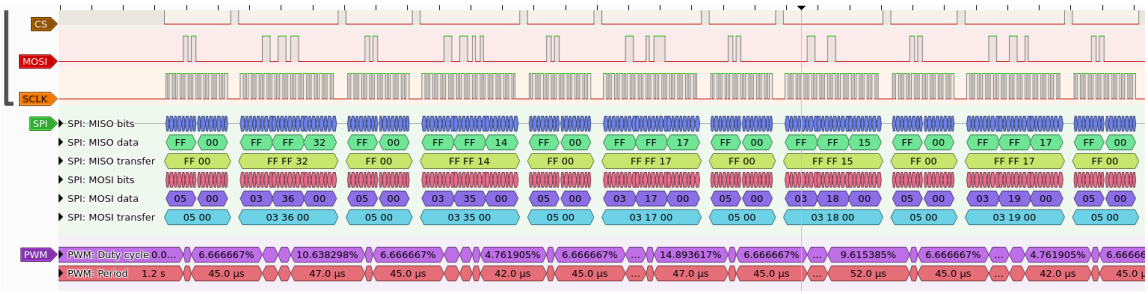

Figura 4.4: Inicio da operação de leitura das temperaturas do `SPI`

Com a ajuda da Figura 4.5 concluímos que são lidos todos os valores do *buffer*, que como mencionado anteriormente são 50.

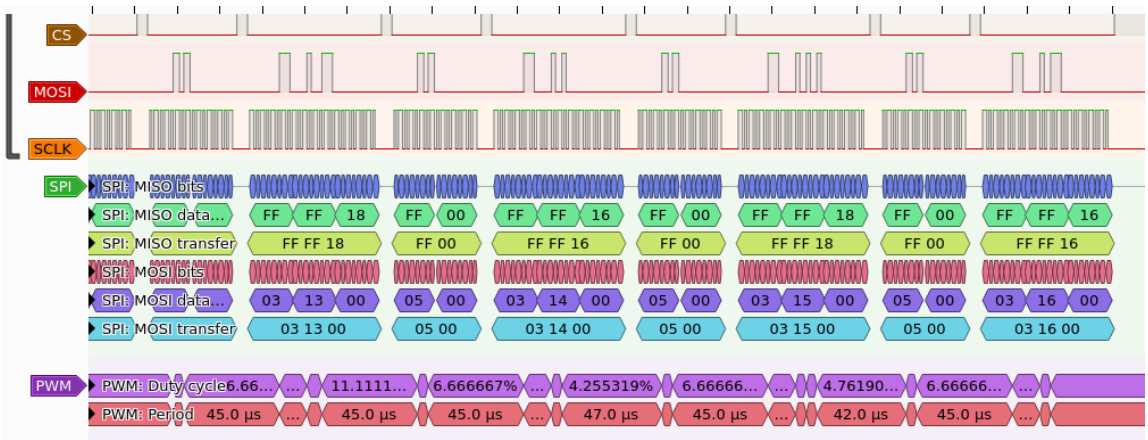

Figura 4.5: Fim da operação de leitura das temperaturas do `SPI`

4.4.2 Ler *set points*

A leitura dos *set points* é similar à leitura das temperaturas, só que lê um de cada vez.

Através da e da e analisando os endereços acedidos e os valores obtidos foi possível concluir que, este primeiramente lê o valor do *set point* 1, depois o *set point* 2 e finalmente o *set point* da *average temperature*.

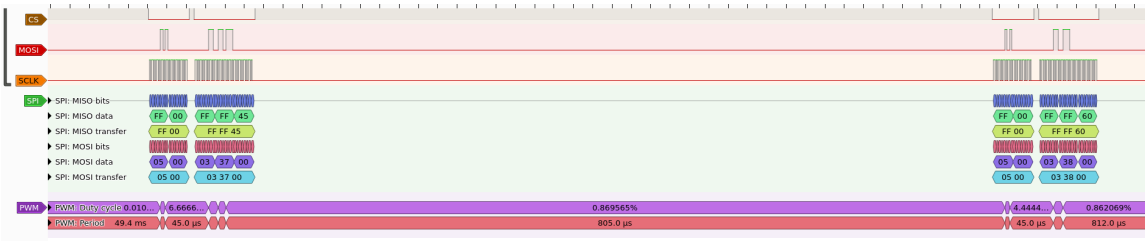

Figura 4.6: Fim da operação de leitura das temperaturas do `SPI`

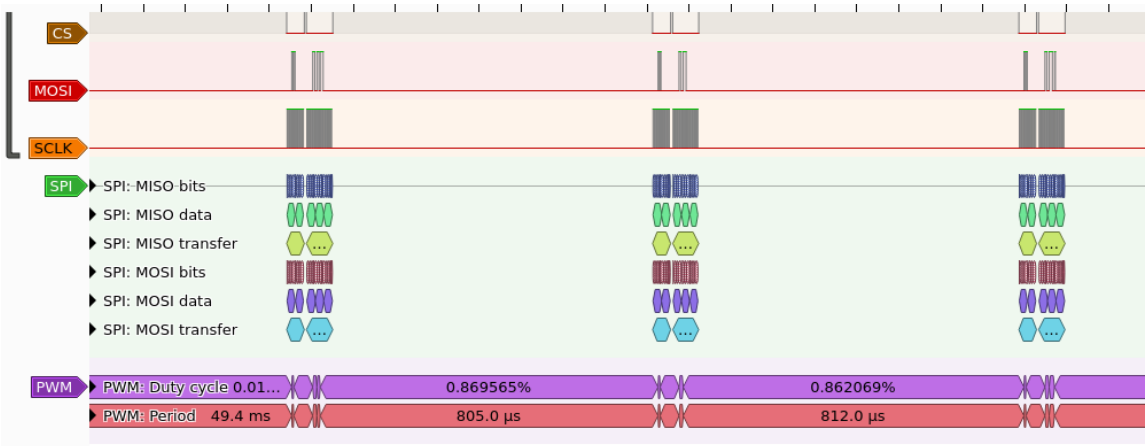

Figura 4.7: Fim da operação de leitura das temperaturas do `SPI`

4.4.3 Escrever temperaturas

A escrita das temperaturas é iniciada com a leitura dos endereços de modo a obter o tamanho do *buffer* e a sua *head*.

A seguir, é verificado se a EEPROM esta a efetuar operações de escrita e, caso não esteja, é iniciada uma sequência de *write enable* e é escrita para o endereço da *head* mais 3 o valor da temperatura do sensor de temperatura 1 (Figura 4.8).

Depois, este vai verificando continuadamente se a EEPROM esta a efetuar a escrita e, só quando esta responde que não é que é escrito o valor de temperatura do sensor de temperatura 2, para o endereço seguinte ao escrito anteriormente (Figura 4.9).

Finalmente, este espera que a operação fique concluída e atualiza a *head* do *buffer*, incrementando duas vezes, visto que foram escritas duas temperaturas (Figura 4.10).

4.4.4 Atualizar *set points*

A operação de atualização dos *set points* é bastante simples comparada com a anterior.

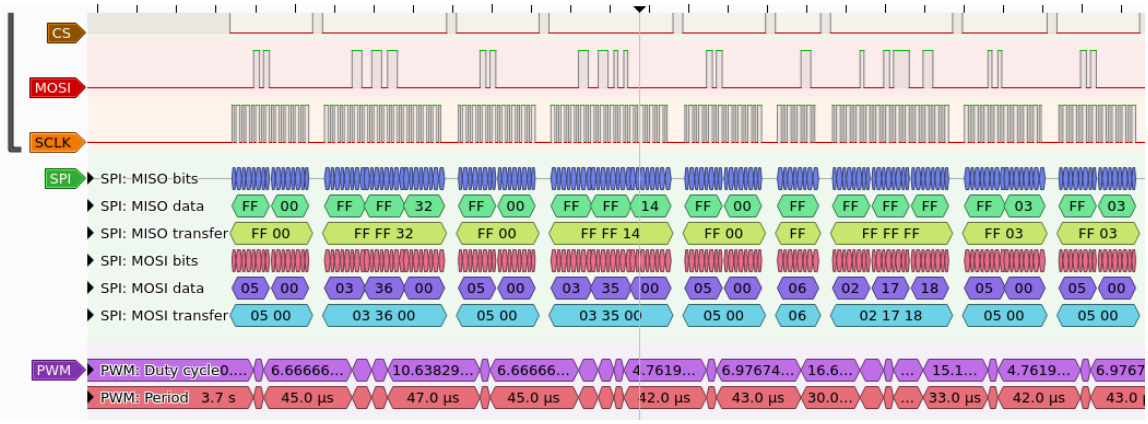

Figura 4.8: Inicio da operação de escrita das temperaturas do `SPI`

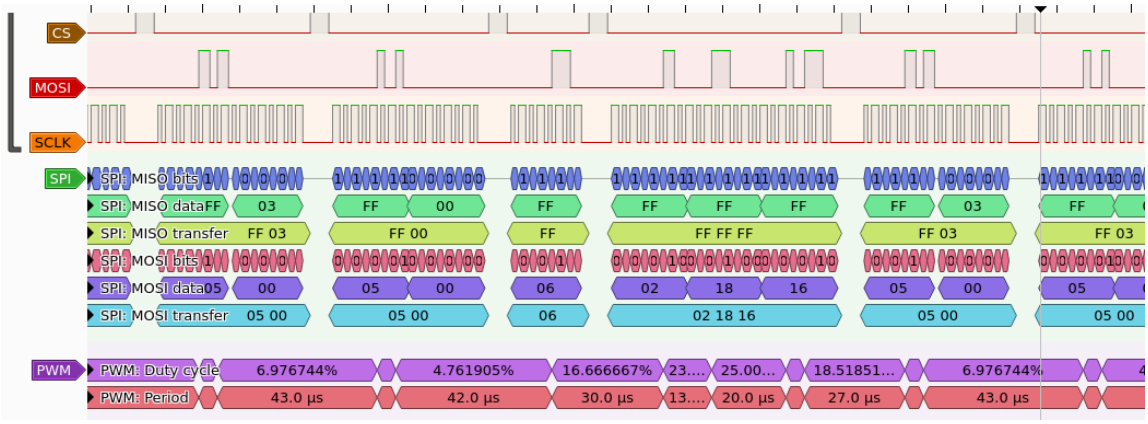

Figura 4.9: Escrita da temperatura do sensor de temperatura 2

Apenas é verificado se +e possível efetuar operações de leitura e, caso sim, o endereço do *set point* a ser atualizado é acedido e o seu novo valor é escrito.

Na Figura 4.11 é visivel a atualização do *set point* 1 para o valor 55.

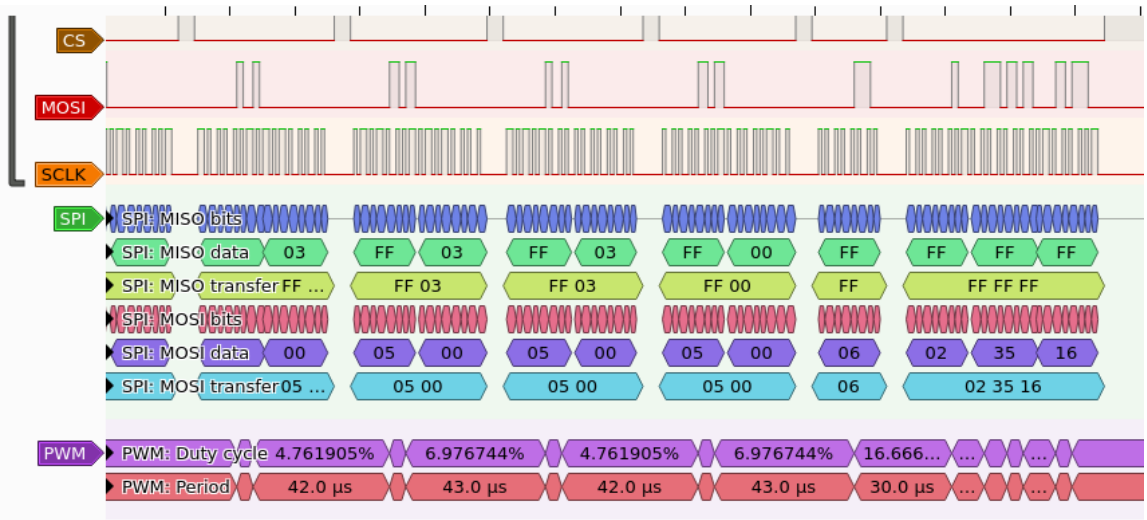

Figura 4.10: Inicio da operação de excrita das temperaturas do `SPI`

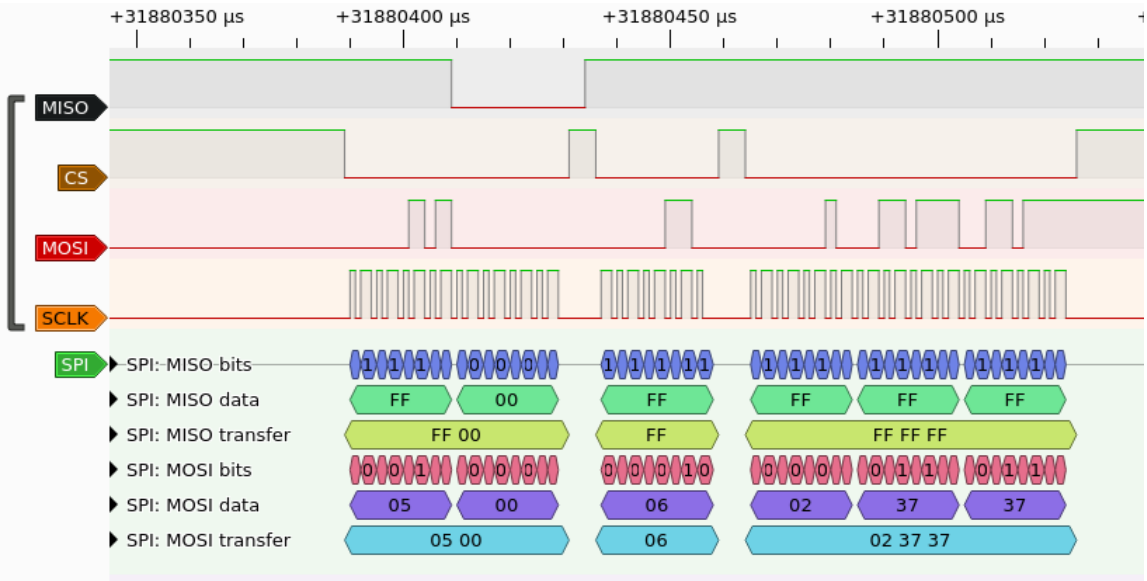

Figura 4.11: Operação de *set* de um *set point* no `SPI`

Capítulo 5

OCs

5.1 Identificação dos Pinos

Cada OC está associada a um sensor de temperatura. Logo para a sua identificação foi apenas necessário mexer nos *switches* 4 e 3 e verificar qual dos pinos sofria alteração de sinal.

Assim, foi possível concluir que a OC2 esta associada ao SW4, ou seja, está relacionado com o sensor do lado esquerdo, e a OC1 esta associada ao SW3, estando relacionado com o sensor do lado direito.

5.2 Análise do Sinal

Agora sabendo qual pino está relacionado com, qual sensor é possível analisar a relação entre a temperatura e o *set point* relativo.

Apôs alguma análise foi possível construir a tabela abaixo, .

Analisando esta tabela é possível concluir que o *Pulse Width Modulation* do sinal OC é igual à diferença entre a temperatura do *set point* e a temperatura real, do sensor correspondente, a multiplicar por 3,5, a arredondar para baixo.

Um exemplo é visível na Figura 5.1, neste exemplo a temperatura do sensor do lado esquerdo está a 22, o *set point* 1 está igual a 48 e o PWM é igual a 91%.

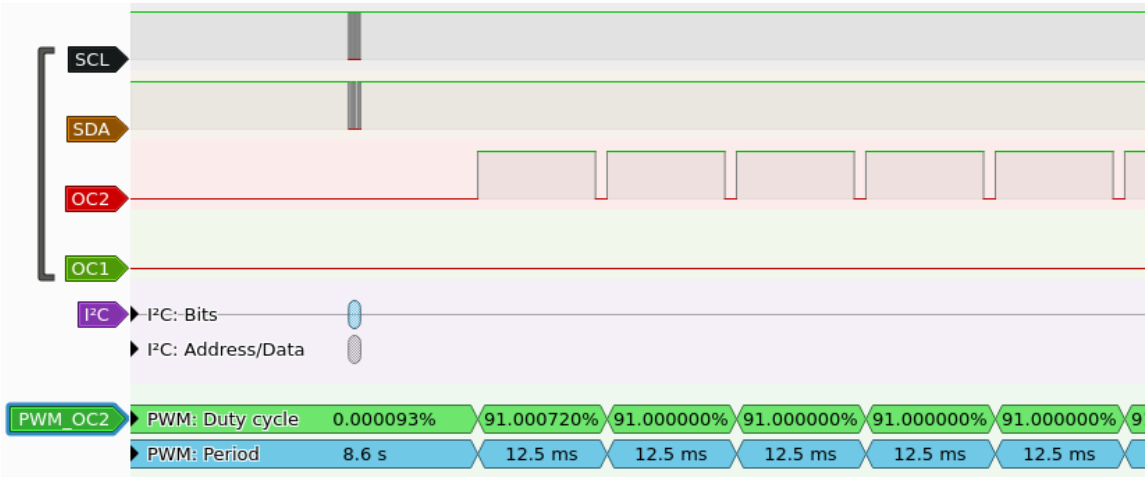

Figura 5.1: Exemplo do sinal de um `OC`

Capítulo 6

I2C

O I2C, também conhecido como *Inter-Integrated Circuit*, é um barramento série, que utiliza transferência série bidirecional, *half-duplex*, orientada ao *byte*, envolvendo sempre uma relação *master/slave*.

O barramento de comunicação apenas necessita de dois fios: o **sdl!** (**sdl!**) e o **scl!** (**scl!**). Estes dois fios têm dois pinos na placa, sendo que foi nestes que foram medidos os valores.

Para melhor especificar o sistema, será de notar que o pino da esquerda expõe **scl!** e o da direita o **sdl!**.

6.1 Identificação dos Sensores

Para identificar qual pino era, qual sinal ambos foram analisados com ajuda do *PulseView* e devido ao aspeto particular do *clock* foi possível logo o identificar.

O INT4 corresponde ao SCL e o OC5 ao SDA.

6.2 Caracterização do Sinal

O I2C em similariedade com o SPI, o seu *baudrate* também é calculado através do seu período.

Sendo assim foi concluído que o seu *baudrate* é igual a 71,4 khz.

6.3 Operações

No modo de operação de mostrar a temperatura dos dois sensores, foi visto que eram escritos dados nulos num endereço (48) e, passado algum tempo (15 microsegundos), esse mesmo endereço era lido com os dados 1E. Este mesmo ciclo de escrita e leitura repetiu-se, também, para o endereço 4D. Este comportamento é visível na Figura 6.2.

Tendo em conta que 1E é 30 em decimal, e tendo em conta a operação, será de assumir que este barramento obtém os dados dos leitores de temperatura.

Após alguma análise foi possível concluir que, é primeiramente lido o valor de temperatura do sensor do lado esquerdo e depois o valor de temperatura do lado direito.

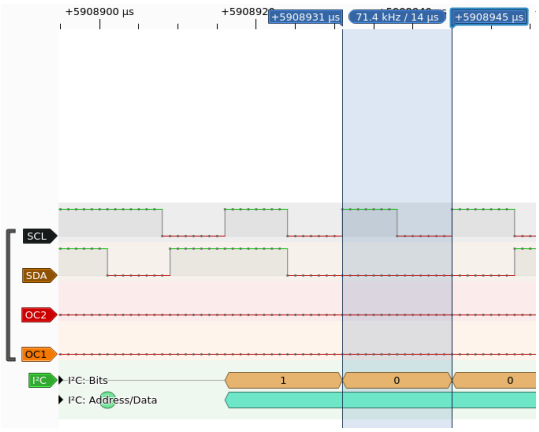

Figura 6.1: Calculo do período do `I2C`

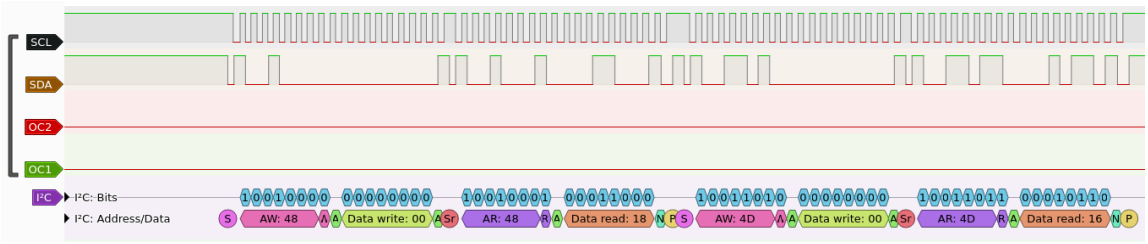

Figura 6.2: Snippet do sinal do `I2C`

Sendo assim, também se conclui que o endereço do sensor do lado esquerdo é 0x48 e do lado direito é 0x4D.

Capítulo 7

LEDs

Inicialmente, pela simples utilização dos switches foi possivel concluir que os LEDs tinham o seguinte comportamento.

- Com nenhum switch ligado : acendiam o LED2 e LED5
- Com o SW1 ligado: acendia o LED0
- Com o SW2 ligado: acendia o LED7
- Com o SW3 ligado: acendia o LED3
- Com o SW4 ligado: acendia o LED4

A seguir, o comportamento dos LEDs foi analisado variando as temperaturas e os *set points* e verificando quais acendiam.

Após, uma análise exaustiva foi possível concluir os comportamentos apresentados na Tabela 7.1. Como visível é concluído posteriormente o LED4 está sempre ligado quando uma operação de *set* de temperatura está em funcionamento (SW3 ou SW4 estão ligados).

Na as linhas em branco foram propositadas de maneira a ser mais visível os grupos em que as temperaturas necessárias para 'acionar' cada LED são as mesmas.

Analisando a tabela foi possível compreender uma expressão relacionada com o comportamento dos *switches*.

O Incremento da temperatura necessária para 'acionar' um LEDx está relacionada com a diferença entre o valor do *set point*, o seu valor mínimo (35) e a temperatura necessária para 'acionar' o LEDx-1.

A diferença entre 35 e o *set point* é calculado e é depois dividida por 4, criando assim um género de um *offset*. A divisão por 4 foi concluída pela forma como os grupos das temperaturas são criadas de 4 em 4 incrementos do *set point*, visível na .

Este *offstet* é depois então usado para calcular a temperatura necessária para acender cada LED para cada *set point* e temperatura. Um pouco do raciocínio é visível na Listing 7.1.

Tendo então o *offset*, as temperaturas necessárias são calculadas da seguinte maneira:

- Temp LED 5 = 35 (valor mínimo do *set point*) + offset (o *floor* deste valor) cima)
- Temp LED 6 = (valor do Temp LED 5) + offset (o *floor* deste valor)

- Temp LED 7 = (valor do Temp LED 6) + offset (o *floor* deste valor)

Listing 7.1: Snippets das notas escritas para chegar a expressão

```
dif=80-35 = 45  | offset=45/4 = 11.25
35 + 11.25 = 46
46 + 11.25 = 57
57 + 11.25 = 68

dif=81-35 = 46  | offset=46/4 = 11.5
35 + 11.5 = 47
47 + 11.5 = 59
59 + 11.5 = 71

dif=82-35 = 47  | offset=47/4 = 11.75
35 + 11.75 = 47
47 + 11.75 = 59
59 + 11.75 = 71
```

Tabela 7.1: Tabela com os comportamentos dos LEDs face a alteração da temperatura e dos set points

Temperature	**SetPoint**	**LEDS**
21	80	4
46	80	4,5
57	80	4,5,6
68	80	4,5,6,7
21	81	4
47	81	4,5
59	81	4,5,6
71	81	4,5,6,7
21	82	4
47	82	4,5
59	82	4,5,6
71	82	4,5,6,7
21	83	4
47	83	4,5
59	83	4,5,6
71	83	4,5,6,7
21	84	4
47	84	4,5
59	84	4,5,6
71	84	4,5,6,7
21	85	4
48	85	4,5
61	85	4,5,6
74	85	4,5,6,7
21	86	4
48	86	4,5
61	86	4,5,6
74	86	4,5,6,7
21	87	4
48	87	4,5
61	87	4,5,6
74	87	4,5,6,7
21	88	4
48	88	4,5
61	88	4,5,6
74	88	4,5,6,7
21	89	4
49	89	4,5
63	89	4,5,6
77	89	4,5,6,7

Capítulo 8

Análise de código decompilado

Muitas operações sobre DATs (DAT_ops_01 a DAT_ops_21, calls_DAT_*)

Muitas operações numéricas e bitwise (number_ops_1 e number_ops_2)

Funções mais relevantes: main e assigns_command

No fundo há muitas operações escondidas por DATs, que provocam alterações em variáveis adjacentes em memória, que por sua vez produzir um resultado quando essas variáveis adjacentes são chamadas nalgum momento.

Capítulo 9

Conclusão

Este trabalho mostrou-se vital para colocar em prática os conhecimentos adquiridos na unidade curricular.

Foi também possível atestar a importância de uma análise de baixo nível, nomeadamente ao nível da análise de sinais emitidos por periféricos de um microcontrolador. Desta forma foi possível descodificar informação impercetível aos sentidos humanos e ganhar conhecimento sobre o funcionamento de um sistema embedded, como foi o caso do microcontrolador analisado.

Bibliografia

[1] E. J. Chikofsky e J. H. Cross, «Reverse engineering and design recovery: A taxonomy», *IEEE software*, vol. 7, n.º 1, pp. 13–17, 1990.

[2] *Visual Studio Code*, `https://code.visualstudio.com/`, Acedido: 17-05-2022.

[3] *Ghidra*, `https://ghidra-sre.org/`, Acedido: 17-05-2022.

Capítulo 10

Anexos

Função 10.1: `main` do ficheiro `proj_153.elf`

```
void main(void)

{
  int thisis1;
  uint uVar1;
  int ret2;
  int iVar2;
  byte *pbVar3;
  uint uVar4;
  int iVar5;
  uint uVar6;
  uint uVar7;
  byte local_c8 [56];
  char ret;
  char str3;
  char str2;
  char char_array1 [5];
  uint num1;
  uint local_84 [11];
  int num2;
  uint local_54 [11];

  str2 = '\0';
  char_array1[0] = '\0';
  returns3(0x23);
  returns7(0x99);
  num1 = 0;
  num2 = 0;
  DAT_ops_13(0x9600,8,0x45,1);
```

```
DAT_ops_07();
DAT_ops_06(400000);
DAT_ops_01(70000);
DAT_ops_15();
DAT_ops_16(1);
DAT_ops_20();
DAT_ops_17();
if (-1 < _DAT_bf8860d0 << 0x17) {
  calls_DAT_ops_12();
}
calls_DAT_ops_14("Reverse Engineering\n");
Status = Status | 1;
iGpffff8024 = DAT_ops_11();
iGpffff8020 = DAT_ops_11();
iGpffff801c = DAT_ops_11();
do {
  do {
  } while (iGpffff803c == 0);
  iGpffff803c = 0;
  if (uGpffff8048 != (_DAT_bf886050 << 0x1c) >> 0x1e) {
    cGpffff8029 = '\x01';
  }
  uGpffff8048 = (_DAT_bf886050 << 0x1c) >> 0x1e;
  if (uGpffff8048 == 0) {
    i2cError(0,0);
    if (uGpffff8044 != (_DAT_bf886050 & 3)) {
      cGpffff8029 = '\x01';
    }
    uGpffff8044 = _DAT_bf886050 & 3;
    if (uGpffff8044 == 0) {
      _DAT_bf886120 = _DAT_bf886120 & 0xff00 | 0x24;
    }
    else {
      ret2 = DAT_ops_18();
      uVar1 = (ret2 * 0x40 + 0x1ff) / 0x3ff + 0x23;
      uGpffff8040 = number_ops_2(uVar1 & 0xff);
      if (uGpffff8044 == 1) {
        _DAT_bf886120 = _DAT_bf886120 & 0xff00 | 1;
        uGpffff8018 = uVar1;
      }
      if (uGpffff8044 == 2) {
        _DAT_bf886120 = _DAT_bf886120 & 0xff00 | 0x80;
        uGpffff8014 = uVar1;
      }
      if (uGpffff8044 == 3) {
        _DAT_bf886120 = _DAT_bf886120 & 0xff00 | 0x81;
```

```
      uGpffff8010 = uVar1;
    }
  }
}
else {
  if ((iGpffff8034 != 0) || (cGpffff8029 == '\x01')) {
    cGpffff8029 = '\0';
    iGpffff8034 = 0;
    ret = calls_DAT_ops_02_04();
    ret2 = calls_DAT_ops_02_04(1);
    str3 = (char)ret2;
    if (ret == -1) {
      calls_DAT_ops_14("I2C Error (A0)");
    }
    else if (ret2 == -1) {
      calls_DAT_ops_14("I2C Error (A5)\n");
    }
    else {
      assigns_command(0,&ret,&str2,&str3,char_array1);
      uVar1 = num1;
      uVar6 = (int)ret + (int)str2;
      uVar7 = (int)str3 + (int)char_array1[0];
      num1 = num1 + 1;
      local_84[uVar1] = uVar6;
      local_54[num2] = uVar7;
      num2 = num2 + 1;
      uVar1 = (uint)(char)((int)(uVar6 + uVar7) / 2);
      if (uGpffff8048 == 1) {
        uGpffff8040 = number_ops_2(uVar6 & 0xff);
        i2cError((int)((iGpffff8024 - uVar6) * 0x23) / 10,0);
        uVar7 = _DAT_bf886120 & 0xfff0;
        uVar1 = number_ops_1(0x23,iGpffff8024,uVar6,4,0);
        _DAT_bf886120 = (uVar7 | uVar1) & 0xff0f;
      }
      else if (uGpffff8048 == 2) {
        uGpffff8040 = number_ops_2(uVar7 & 0xff);
        i2cError(0,(int)((iGpffff8020 - uVar7) * 0x23) / 10);
        uVar1 = _DAT_bf886120 & 0xff00;
        _DAT_bf886120 = _DAT_bf886120 & 0xfff0;
        ret2 = number_ops_1(0x23,iGpffff8020,uVar7,4,1);
        _DAT_bf886120 = uVar1 | ret2 << 4;
      }
      else if (uGpffff8048 == 3) {
        uGpffff8040 = number_ops_2(uVar1 & 0xff);
        i2cError((int)((iGpffff801c - uVar1) * 0x23) / 10,
                 (int)((iGpffff801c - uVar1) * 0x23) / 10);
```

```
          uVar4 = _DAT_bf886120 & 0xfff0;
          uVar1 = number_ops_1(0x23,iGpffff801c,uVar6,4,0);
          uVar4 = uVar4 | uVar1;
          _DAT_bf886120 = uVar4;
          ret2 = number_ops_1(0x23,iGpffff801c,uVar7,4,1);
          _DAT_bf886120 = uVar4 & 0xff0f | ret2 << 4;
        }
        ret2 = num2;
        iVar5 = 0;
        if ((num1 & 3) == 0) {
          iVar2 = 0;
          pbVar3 = local_c8;
          uVar1 = num1;
          if (0 < (int)num1) {
            do {
              uVar1 = uVar1 - 1;
              iVar2 = iVar2 + *(int *)(pbVar3 + 0x44);
              iVar5 = iVar5 + *(int *)(pbVar3 + 0x74);
              pbVar3 = pbVar3 + 4;
            } while (uVar1 != 0);
          }
          iVar2 = iVar2 / (int)num1;
          if (num1 == 0) {
            trap(7);
          }
          num1 = 0;
          num2 = 0;
          if (ret2 == 0) {
            trap(7);
          }
          calls_DAT_ops_11_12((int)(char)iVar2,(int)(char)(iVar5 / ret2));
        }
      }
    }
    thisis1 = returns1();
    assigns_command(thisis1,&ret,&str2,&str3,char_array1);
  }
  if (iGpffff802c == 1) {
    iGpffff802c = 0;
    iVar5 = 0;
    ret2 = calls_DAT_ops_11(local_c8);
    calls_DAT_ops_14("\nTemperatures:\n");
    pbVar3 = local_c8;
    if (0 < ret2) {
      do {
        iVar5 = iVar5 + 2;
```

```
          DAT_ops_19((uint)*pbVar3);
          DAT_ops_14(0x20);
          pbVar3 = pbVar3 + 2;
        } while (iVar5 < ret2);
      }
      iVar5 = 0;
      calls_DAT_ops_14("\n");
      pbVar3 = local_c8;
      if (0 < ret2) {
        do {
          iVar5 = iVar5 + 2;
          DAT_ops_19((uint)pbVar3[1]);
          DAT_ops_14(0x20);
          pbVar3 = pbVar3 + 2;
        } while (iVar5 < ret2);
      }
      ret2 = 0;
      calls_DAT_ops_14("\nSet Points:\n");
      do {
        uVar1 = DAT_ops_11();
        DAT_ops_19(uVar1 & 0xff);
        ret2 = ret2 + 1;
        DAT_ops_14(0x20);
      } while (ret2 < 3);
    }
  } while( true );
}
```

Função 10.2: `assigns_command` do ficheiro `proj_153.elf`

```
/* Code reversed by: Goncalo Almeida  */

void assigns_command(char c1,char *str1,char *str2,char *str3,char *str4)

{
  char tmp;
  int number_tmp;
  char tmp2;

  if (c1 == '1') {
    tmp = *str2 + '\x01';
LAB_9d001d4c:
    *str2 = tmp;
  }
  else {
    if (c1 == 'q' || c1 == 'Q') {
```

```
      tmp = *str2 + -1;
      goto LAB_9d001d4c;
    }
    if (c1 == 'a' || c1 == 'A') {
      *str2 = '\0';
    }
    else if (c1 == '2') {
      *str4 = *str4 + '\x01';
    }
    else if (c1 == 'w' || c1 == 'W') {
      *str4 = *str4 + -1;
    }
    else {
      if (c1 != 's' && c1 != 'S') {
        tmp = *str1;
        goto LAB_9d001d54;
      }
      *str4 = '\0';
    }
  }
  tmp = *str1;
LAB_9d001d54:
  if ((int)tmp + (int)*str2 < 100) {
    number_tmp = (int)tmp + (int)*str2;
  }
  else {
    tmp2 = 'c' - tmp;
    *str2 = tmp2;
    tmp = *str1;
    number_tmp = (int)tmp + (int)tmp2;
  }
  if (number_tmp < 0) {
    *str2 = '#' - tmp;
  }
  tmp = *str3;
  if ((int)tmp + (int)*str4 < 100) {
    number_tmp = (int)tmp + (int)*str4;
  }
  else {
    tmp2 = 'c' - tmp;
    *str4 = tmp2;
    tmp = *str3;
    number_tmp = (int)tmp + (int)tmp2;
  }
  if (-1 < number_tmp) {
    return;
```

```
  }
  *str4 = '#' - tmp;
  return;
}
```

Função 10.3: `entry` do ficheiro `proj_153.elf`

```
/* Code reversed by: Goncalo Almeida  */
void entry(undefined4 param_1,uint number)

{
  uint ret;

  do {
    ret = DAT_ops_08();
  } while ((ret & 1) != 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  _DAT_bf805a20 = number & 0xff;
  return;
}
```

Função 10.4: `i2cError` do ficheiro `proj_153.elf`

```
void i2cError(int n1,int n2)

{
  if (100 < n1) {
    n1 = 100;
  }
  if (n1 < 0) {
    n1 = 0;
  }
  if (100 < n2) {
    n2 = 100;
  }
  if (n2 < 0) {
    n2 = 0;
  }
  _DAT_bf803020 = (uint)(_DAT_bf800820 * n1 + n1) / 100;
  _DAT_bf803220 = (uint)(_DAT_bf800820 * n2 + n2) / 100;
  return;
}
```

Função 10.5: `number_ops_1` do ficheiro `proj_153.elf`

```
int number_ops_1(int n1,int n2,int n3,uint n4,int n5)

{
  int tmp1;
  int tmp2;
  uint tmp3;

  if (n3 < n1) {
    n3 = n1;
  }
  tmp1 = (1 << (n4 & 0x1f)) + -1;
  if (n3 <= n2) {
    tmp1 = (int)((n2 - n1) * 2 + n4) / (int)(n4 << 1);
    if (n4 << 1 == 0) {
      trap(7);
    }
    tmp2 = 1 << (n4 & 0x1f);
    if (tmp1 == 0) {
      tmp1 = 1;
    }
    if (tmp1 == 0) {
      trap(7);
    }
    tmp3 = (n3 - n1) / tmp1 + 1;
    if ((int)tmp3 <= (int)n4) {
      n4 = tmp3;
    }
    tmp1 = (1 << (n4 & 0x1f)) + -1;
    if (n5 != 1) {
      tmp1 = tmp2 - (tmp2 >> (n4 & 0x1f));
    }
  }
  return tmp1;
}
```

Função 10.6: `number_ops_2` do ficheiro `proj_153.elf`

```
uint number_ops_2(uint number)

{
  return (number & 0xff) + ((number & 0xff) / 10) * 6 & 0xff;
}
```

Função 10.7: DAT_ops_01 do ficheiro proj_153.elf

```
void DAT_ops_01(int number)

{
  if (number << 1 == 0) {
    trap(7);
  }
  DAT_bf805301 = DAT_bf805301 | 0x80;
  _DAT_bf805340 = (number + 20000000U) / (uint)(number << 1) - 1;
  return;
}
```

Função 10.8: DAT_ops_02 do ficheiro proj_153.elf

```
void DAT_ops_02(void)

{
  _DAT_bf805300 = _DAT_bf805300 | 1;
  do {
  } while ((int)(_DAT_bf805300 << 0x1f) < 0);
  return;
}
```

Função 10.9: DAT_ops_03 do ficheiro proj_153.elf

```
void DAT_ops_03(void)

{
  do {
  } while ((_DAT_bf805300 & 0x1f) != 0);
  _DAT_bf805300 = _DAT_bf805300 | 4;
  do {
  } while ((int)(_DAT_bf805300 << 0x1d) < 0);
  return;
}
```

Função 10.10: DAT_ops_04 do ficheiro proj_153.elf

```
uint DAT_ops_04(uint number)

{
  do {
  } while (_DAT_bf805310 << 0x11 < 0);
  _DAT_bf805350 = number & 0xff;
```

```
  return (uint)(_DAT_bf805310 << 0x10) >> 0x1f;
}
```

Função 10.11: DAT_ops_05 do ficheiro proj_153.elf

```
int DAT_ops_05(uint number)

{
  do {
  } while ((_DAT_bf805300 & 0x1f) != 0);
  do {
  } while (-1 < _DAT_bf805310 << 0x1e);
  _DAT_bf805300 = _DAT_bf805300 & 0xffffffdf | 8 | (number & 1) << 5 | 0x10;
  do {
  } while ((int)(_DAT_bf805300 << 0x1b) < 0);
  return (int)(char)_DAT_bf805360;
}
```

Função 10.12: DAT_ops_06 do ficheiro proj_153.elf

```
void DAT_ops_06(int number)

{
  if (number << 1 == 0) {
    trap(7);
  }
  _DAT_bf805a30 = (number + 20000000U) / (uint)(number << 1) - 1;
  return;
}
```

Função 10.13: DAT_ops_07 do ficheiro proj_153.elf

```
void DAT_ops_07(void)

{
  do {
  } while (-1 < (int)(_DAT_bf805a10 << 0x1a));
  _DAT_bf805a10 = _DAT_bf805a10 & 0xffffffbf;
  DAT_bf805a00 = DAT_bf805a00 & 0xbf | 0x20;
  DAT_bf805a01 = DAT_bf805a01 & 0x71 | 0x81;
  DAT_bf805a02 = DAT_bf805a02 | 1;
  DAT_bf805a03 = DAT_bf805a03 | 0x10;
  return;
}
```

Função 10.14: `DAT_ops_08` do ficheiro `proj_153.elf`

```
int DAT_ops_08(void)

{
  do {
  } while (-1 < (int)(_DAT_bf805a10 << 0x1a));
  _DAT_bf805a10 = _DAT_bf805a10 & 0xffffffbf;
  do {
  } while ((int)(_DAT_bf805a10 << 0x14) < 0);
  _DAT_bf805a20 = 0;
  return 0;
}
```

Função 10.15: `DAT_ops_09` do ficheiro `proj_153.elf`

```
void DAT_ops_09(int number)

{
  uint ret;

  do {
    ret = DAT_ops_08();
  } while ((ret & 1) != 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  _DAT_bf805a20 = number;
  return;
}
```

Função 10.16: `DAT_ops_10` do ficheiro `proj_153.elf`

```
void DAT_ops_10(char addr)

{
  uint ret;

  DAT_ops_09(6);
  do {
    ret = DAT_ops_08();
  } while ((ret & 1) != 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  _DAT_bf805a20 = (int)addr;
  return;
}
```

Função 10.17: DAT_ops_11 do ficheiro proj_153.elf

```
int DAT_ops_11(void)

{
  uint ret;

  do {
    ret = DAT_ops_08();
  } while ((ret & 1) != 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  _DAT_bf805a20 = 0;
  return 0;
}
```

Função 10.18: DAT_ops_12 do ficheiro proj_153.elf

```
void DAT_ops_12(undefined4 param_1,uint number_hex)

{
  uint ret;

  do {
    ret = DAT_ops_08();
  } while ((ret & 1) != 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  do {
  } while (_DAT_bf805a10 << 0x14 < 0);
  _DAT_bf805a20 = number_hex & 0xff;
  return;
}
```

Função 10.19: DAT_ops_13 do ficheiro proj_153.elf

```
void DAT_ops_13(int n1,int n2,char c,int param_4)

{
  int number;

  if (n1 << 4 == 0) {
    trap(7);
  }
  number = 3;
```

```
  if (((n2 != 9) && (number = 1, c != 'E')) && (number = 2, c != 'O')) {
    number = 0;
  }
  _DAT_bf806000 = _DAT_bf806000 & 0xfffffff0 | number << 1 | param_4 - 1U & 1
  | 0x8000;
  DAT_bf806011 = DAT_bf806011 | 0x14;
  _DAT_bf806040 = (n1 * 8 + 20000000) / (n1 << 4) + -1;
  return;
}
```

Função 10.20: DAT_ops_14 do ficheiro proj_153.elf

```
void DAT_ops_14(char addr)

{
  do {
  } while (_DAT_bf806010 << 0x16 < 0);
  _DAT_bf806020 = (int)addr;
  return;
}
```

Função 10.21: DAT_ops_15 do ficheiro proj_153.elf

```
void DAT_ops_15(void)

{
  _DAT_bf886040 = _DAT_bf886040 & 0x80ff | 0xf;
  _DAT_bf886060 = _DAT_bf886060 & 0x80ff;
  _DAT_bf8860c0 = _DAT_bf8860c0 & 0xff9f;
  _DAT_bf8860e0 = _DAT_bf8860e0 & 0xff9f | 0x20;
  _DAT_bf886100 = _DAT_bf886100 & 0xff00;
  _DAT_bf886120 = _DAT_bf886120 & 0xff00;
  return;
}
```

Função 10.22: DAT_ops_16 do ficheiro proj_153.elf

```
void DAT_ops_16(int number)

{
  _DAT_bf809000 =
       _DAT_bf809000 & 0xffff0000 | (uint)((ushort)_DAT_bf809000 & 0xff1f | 0xf0)
       | 0x8000;
  _DAT_bf809010 = _DAT_bf809010 & 0xffffffc3 | (number - 1U & 0xf) << 2;
  _DAT_bf809020 = _DAT_bf809020 & 0xffffe0ff | 0x1000;
```

```
  _DAT_bf809040 = _DAT_bf809040 & 0xfff0ffff | 0x40000;
  DAT_bf809060 = DAT_bf809060 & 0xef;
  DAT_bf886040 = DAT_bf886040 | 0x10;
  return;
}
```

Função 10.23: DAT_ops_17 do ficheiro proj_153.elf

```
void DAT_ops_17(void)

{
  DAT_bf881030 = DAT_bf881030 & 0x7f;
  DAT_bf881060 = DAT_bf881060 | 0x80;
  _DAT_bf8810a0 = _DAT_bf8810a0 & 0xe3ffffff | 0x8000000;
  return;
}
```

Função 10.24: DAT_ops_18 do ficheiro proj_153.elf

```
undefined4 DAT_ops_18(void)

{
  do {
  } while (-1 < (int)(_DAT_bf881040 << 0x1e));
  _DAT_bf881040 = _DAT_bf881040 & 0xfffffffd;
  DAT_bf809000 = DAT_bf809000 | 4;
  return _DAT_bf809070;
}
```

Função 10.25: DAT_ops_19 do ficheiro proj_153.elf

```
void DAT_ops_19(int n1)

{
  uint number;

  number = number_ops_2(n1 & 0xff);
  DAT_ops_14((int)(char)((char)(number >> 4) + '0'));
  DAT_ops_14((number & 0xf) + 0x30);
  return;
}
```

Função 10.26: DAT_ops_20 do ficheiro proj_153.elf

```
void DAT_ops_20(void)
```

```
{
  int *pnumber;
  uint number3;
  int number2;
  int number;
  int array [4];
  int number_hex;
  int number_hex2;
  int number_hex3;

  number = 1;
  array[0] = 2;
  array[3] = 0x10;
  number_hex = 0x20;
  array[1] = 4;
  array[2] = 8;
  number_hex2 = 0x40;
  number_hex3 = 0x100;
  number3 = 0;
  pnumber = &number;
  number2 = 1;
  while( true ) {
    if (number2 == 0) {
      trap(7);
    }
    number2 = 250000 / number2 + -1;
    pnumber = pnumber + 1;
    if (number2 < 0x10000) break;
    number3 = number3 + 1;
    if (6 < (int)number3) {
      returns8();
      do {
                    /* WARNING: Do nothing block with infinite loop */
      } while( true );
    }
    number2 = *pnumber;
  }
  _DAT_bf800800 = _DAT_bf800800 & 0xffffff8f | (number3 & 7) << 4 | 0x8000;
  _DAT_bf803000 = _DAT_bf803000 & 0xfffffff0 | 0x8006;
  _DAT_bf803200 = _DAT_bf803200 & 0xfffffff0 | 0x8006;
  _DAT_bf800810 = 0;
  _DAT_bf800820 = number2;
  _DAT_bf803020 = 0;
  _DAT_bf803220 = 0;
  DAT_bf881031 = DAT_bf881031 & 0xfe;
```

```
  DAT_bf881061 = DAT_bf881061 | 1;
  _DAT_bf8810b0 = _DAT_bf8810b0 & 0xfffffffe3 | 4;
  return;
}
```

Função 10.27: DAT_ops_21 do ficheiro proj_153.elf

```
void DAT_ops_21(uint number,char c1)

{
  _DAT_bf8860e0 = _DAT_bf8860e0 ^ 0x60;
  if (c1 != 0) {
    _DAT_bf886060 = _DAT_bf886060 & 0x80ff | 0x100 << (c1 * 3 - 3U & 0x1f);
    return;
  }
  if ((_DAT_bf8860e0 & 0x20) != 0) {
    _DAT_bf886060 =
         _DAT_bf886060 & 0x80ff | (int)*(char *)((int)&DAT_a0000000 +
         (number & 0xf)) << 8;
    return;
  }
  _DAT_bf886060 =
       _DAT_bf886060 & 0x80ff | (int)*(char *)((int)&DAT_a0000000 +
       ((number & 0xff) >> 4)) << 8;
  return;
}
```

Função 10.28: calls_DAT_ops_02_04 do ficheiro proj_153.elf

```
int calls_DAT_ops_02_04(int number)

{
  int ret;
  uint n1;
  int n2;
  int ret2;

  ret2 = -1;
  n1 = 0x90;
  n2 = 0x91;
  if (number != 0) {
    n1 = 0x9a;
    n2 = 0x9b;
  }
  DAT_ops_02();
  ret = DAT_ops_04(n1);
```

```
  if ((ret == 0) && (ret = DAT_ops_04(0), ret == 0)) {
    DAT_ops_02();
    ret = DAT_ops_04(n2);
    if (ret == 0) {
      ret2 = DAT_ops_05(1);
    }
  }
  DAT_ops_03();
  return ret2;
}
```

Função 10.29: `calls_DAT_ops_11` do ficheiro `proj_153.elf`

```
int calls_DAT_ops_11(int number)

{
  int ret1;
  int iVar1;
  undefined *puVar2;
  int iVar3;

  ret1 = DAT_ops_11();
  DAT_ops_11();
  if (ret1 < 0x32) {
    iVar3 = 0;
    if (0 < ret1) {
      do {
        iVar1 = DAT_ops_11();
        puVar2 = (undefined *)(number + iVar3);
        *puVar2 = (char)iVar1;
        iVar3 = iVar3 + 2;
        iVar1 = DAT_ops_11();
        puVar2[1] = (char)iVar1;
      } while (iVar3 < ret1);
    }
    return ret1;
  }
  iVar3 = 0;
  do {
    iVar1 = DAT_ops_11();
    puVar2 = (undefined *)(number + iVar3);
    *puVar2 = (char)iVar1;
    iVar1 = DAT_ops_11();
    iVar3 = iVar3 + 2;
    puVar2[1] = (char)iVar1;
  } while (iVar3 < 0x32);
```

```
  return ret1;
}
```

Função 10.30: `calls_DAT_ops_11_12` do ficheiro `proj_153.elf`

```
void calls_DAT_ops_11_12(char c1,char c2)

{
  int ret1;
  int ret2;

  ret1 = DAT_ops_11();
  ret2 = DAT_ops_11();
  DAT_ops_12(ret2 + 3,(int)c1);
  DAT_ops_12(ret2 + 4,(int)c2);
  if (0x31 < ret1) {
    DAT_ops_12(0x35,(ret2 + 2) % 0x32);
    return;
  }
  DAT_ops_12(0x36,ret1 + 2);
  DAT_ops_12(0x35,(ret2 + 2) % 0x32);
  return;
}
```

Função 10.31: `calls_DAT_ops_12` do ficheiro `proj_153.elf`

```
void calls_DAT_ops_12(void)

{
  DAT_ops_12(0x35,0);
  DAT_ops_12(0x36,0);
  DAT_ops_12(0x37,0x32);
  DAT_ops_12(0x38,0x32);
  DAT_ops_12(0x39,0x32);
  return;
}
```

Função 10.32: `calls_DAT_ops_14` do ficheiro `proj_153.elf`

```
void calls_DAT_ops_14(char *n)

{
  for (; *n != '\0'; n = n + 1) {
    DAT_ops_14();
  }
```

```
  return;
}
```

Função 10.33: `returns1` do ficheiro `proj_153.elf`

```
int returns1(void)

{
  syscall(0);
  return 1;
}
```

Função 10.34: `returns3` do ficheiro `proj_153.elf`

```
int returns3(void)

{
  syscall(0);
  return 3;
}
```

Função 10.35: `returns7` do ficheiro `proj_153.elf`

```
int returns7(void)

{
  syscall(0);
  return 7;
}
```

Função 10.36: `returns8` do ficheiro `proj_153.elf`

```
int returns8(void)

{
  syscall(0);
  return 8;
}
```