Esse é o nono artigo da série escrita pelo engenheiro Ismael Lopes da Silva, exclusivamente para o site "www.embarcados.com.br". Nessa série focarei no Microcontrolador da STMicroelectronics, o MCU STM32F103C8T6, que é um ARM Cortex-M3. Os pré-requisitos para uma boa compreensão dos artigos é ter o domínio da Linguagem C Embedded e conceitos de eletrônica.

Diferentes seções de Memória

Seções da memória de programa/código (FLASH ou ROM):

- ✔ Tabela de vetores;
- ✔ Seção .text;
- ✔ Seção .rodata;
- ✔ Seção .data.

Seções da memória de dados (SRAM):

- ✔ Seção .data;
- ✔ Seção .bbs;
- ✔ Heap;
- ✔ Stack (pilha).

Seção ".text" - é a seção de código, também conhecido como segmento de texto ou simplesmente texto, é onde uma parte de um arquivo de objeto ou a seção correspondente do espaço de endereço do programa que contém instruções executáveis é armazenada e geralmente é somente de leitura e de tamanho fixo.

Seção ".data" - é a seção de dados que contém quaisquer variáveis globais ou estáticas que possuem um valor predefinido e podem ser modificadas. Ou seja, quaisquer variáveis que não são definidas em uma função, portanto, podem ser acessadas de qualquer lugar, ou são definidas em uma função, mas são definidas como estáticas para que retenham seu endereço nas chamadas subsequentes. Os valores para essas variáveis são inicialmente armazenados na memória FLASH e são copiados para o segmento ".data", na memória SRAM, durante a rotina de inicialização do programa.

Seção ".bss" - significa "**b**lock **s**tarted by **s**ymbol" - Toda variável global não inicializada, variável estática não inicializada e variável inicializada com zero são armazenadas na seção ".bss".

Seção ".rodata" - é a seção com todas as variáveis globais constantes.

Diferentes Dados de Programa

Existem diferentes tipos de dados, que são:

- ✔ Global não inicializado;
- ✔ Global inicializado;
- ✔ Global estático não inicializado;
- ✔ Global estático inicializado;
- ✔ Local não inicializado;
- ✔ Local inicializado;
- ✔ Local estático não inicializado;
- ✔ Local estático inicializado;
- ✔ Global constante;
- ✔ Local constante.

Dado global não inicializado é armazenado na seção ".bss" da memória de dados (SRAM) e inicializado com o conteúdo zero. Esse dado não carrega informação importante. Dado global não inicializado não é armazenado na FLASH para não consumir espaço da memória de programa com informação não relevante.

Dado global inicializado é armazenado na seção ".data" da memória de programa (FLASH), e também é copiado para a seção ".data" da memória de dados (SRAM), durante o processo de inicialização (startup code). Esse dado carrega informação importante.

Dado global estático não inicializado é um dado privado, e é armazenado na seção ".bss" da memória de dados (SRAM) e inicializado com o conteúdo zero. Esse dado não carrega informação importante.

Dado global estático inicializado é armazenado na seção ".data" da memória de programa (FLASH), e também é copiado para a seção ".data" da memória de dados (SRAM), durante o processo de inicialização (startup code). Esse dado carrega informação importante.

Dado local não inicializado e inicializado é armazenado no "stack" da memória de dados (SRAM). Esse dado está relacionado com o escopo da função, então, é um dado transiente que é criado e destruído dinamicamente. Quando o programa entra na função o dado é criado e quando retorna da função o dado é destruído.

Dado local estático não inicializado é um dado global que é privado para o escopo da função, e é armazenado na seção ".bss" da memória de dados (SRAM), e inicializado com o conteúdo zero.

Dado local estático inicializado é um dado global que é privado para o escopo da função, e é armazenado na seção ".data" da memória de programa (FLASH), que também é copiado para a seção ".data" da memória de dados (SRAM), durante o processo de inicialização (startup code).

Dado global constante é armazenado na seção ".rodata" da memória de programa/código (FLASH).

Dado local constante é armazenado no "stack" da memória de dados (SRAM).

GNU Coleção do Compilador (GNU Compiler Collection)

Escrevemos nossa aplicação em linguagem "C", então, nosso arquivo-fonte é escrito numa linguagem de alto nível. Uma linguagem de alto nível é uma linguagem de programação com alto grau de abstração dos detalhes da linguagem de máquina.

Na programação embarcada, denominamos "host" o computador onde escrevemos nossa aplicação e compilamos, e "target" o MCU que transferimos nossa aplicação convertida em código de máquina, onde a mesma rodará.

O processo de compilação é converter um programa de linguagem de alto nível para um arquivo executável que contém código de máquina. Depois transferimos esse arquivo executável para o "target". Nossa IDE, STM32CubeIDE, tem incorporado um compilador, conhecido como GCC. Esse compilador é gratuito de código aberto, denominado GNU Tools (GCC) for ARM Embedded Processors – GCC significa GNU Compiler Collection.

Mesmo que o GCC vem incorporador no STM32CubeIDE, vamos compilar, criar executável, converter arquivo e analisar executável na linha de comando para entender o que passa por trás dos bastidores, então, faça um download do GNU Cross Toolchain, através do seguinte link https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads, e instale o GCC em seu "host". Minha plataforma é Windows.

Aplicação (Linguagem de Alto Nível)	>	**Cross Compilation**	>	**Executável** (código de máquina)

Figura 1 – Fluxo de compilação

O que é "GCC"? É um processo na qual um conjunto de ferramentas roda no "host" e cria executável que rodará em diferentes "target". Em meu contexto o "host" é meu desktop e o "target" é o MCU STM32F103C8T6, baseado na arquitetura ARM Cortex-M3. O GCC, é uma coleção de arquivos binários que permite compilar, converter para linguagem Assembly e criar arquivo executável (aplicação). Também contém arquivo binário para depurar (debug) a aplicação no "target". Essas ferramentas veem com outros arquivos binários que ajudam analisar o executável, então, podemos:

- ✔ Dissecar as diferentes seções do executável;
- ✔ Ver o código em linguagem de baixo nível (Assembly) – Disassemble;
- ✔ Extrair símbolos e informações de tamanho;
- ✔ Converter executável em outros formatos como ".bin" e "ihex";
- ✔ Prover as bibliotecas padrões da linguagem "C".

A tabela a seguir mostra os arquivos binários importantes das ferramentas do GCC.

Arquivo	Função
arm-nome-eabi-gcc	Compilar, Converter para Assembly e Criar o Executável (*.elf - Linker)
arm-nome-eabi-as	Converter para Assembly
arm-nome-eabi-ld	Criar o Executável (*.elf - Linker)
arm-nome-eabi-objcopy	Converte (*.elf) para outros formatos (*.bin, *.ihex)
arm-nome-eabi-objdump	Analisar arquivo executável (*.elf)
arm-nome-eabi-readelf	Analisar arquivo executável (*.elf)
arm-nome-eabi-nm	Analisar arquivo executável (*.elf)
arm-nome-eabi-gdb	Depurar a aplicação (Debugging)

Tabela 1 – Arquivos binários importantes do GCC

A figura 2 ilustra onde estão instalados os arquivos binários importantes do GNU Cross Toolchain. Eles são localizados na subpasta [bin] no path de instalação.

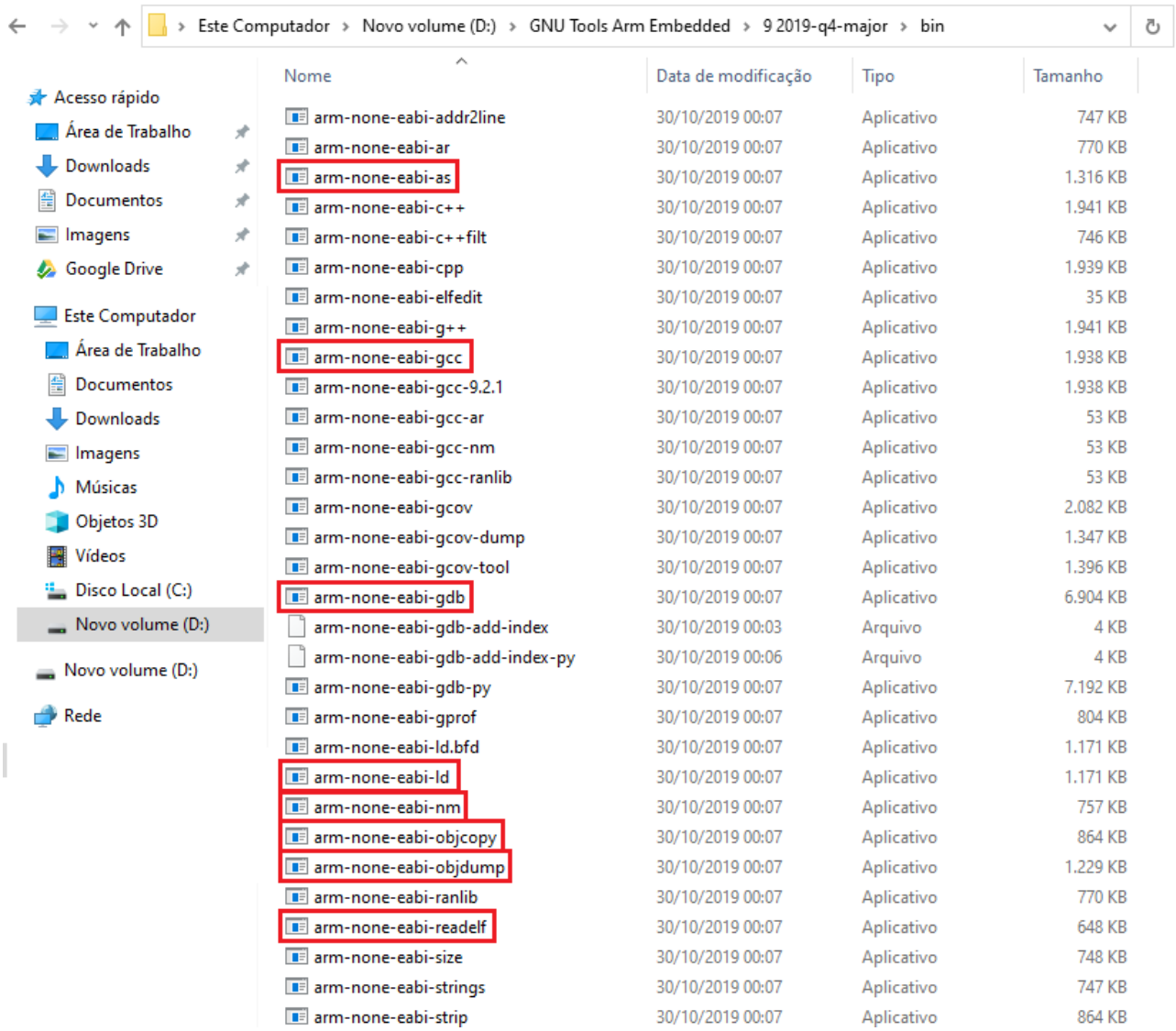

Figura 2 – Arquivos binários importantes instalados no “host”

Processo de Converter Aplicação de Linguagem C para Código de Máquina

O primeiro estágio é chamado de pré-processamento. Todas as diretivas do arquivo-fonte '*.c' será resolvida, e a saída é o arquivo '*.i'. Esse estágio não é uma compilação, é um pré-processamento da compilação. No pré-processamento diretivas como todos os '#includes', macros de 'C' e todas as macros de compilação condicional serão resolvidas e o arquivo de pré-processamento é criado, com a extensão '.i'.

O segundo estágio é o 'Parser', onde as sintaxes da linguagem 'C' será verificada. O terceiro estágio gera o código em linguagem Assembly, que é a tradução do arquivo-fonte para a linguagem Assembly. A entrada para esse estágio é o arquivo '*.i', e a saída é o arquivo '*.s'. Nesse estágio as instruções da linguagem 'C', de alto nível, será convertida para mnemônicos da arquitetura do processado, em linguagem Assembly.

O quarto estágio, denominado 'Assembler', os mnemônicos da linguagem Assembly são convertidos em opcodes. Opcodes nada mais são do que código de máquina para várias instruções, porque o 'target', ou o processador, entende números ou código de máquina. Nesse estágio um arquivo objeto realocável (sem endereços absolutos) é criado. A entrada para esse estágio é o arquivo '*.s', e a saída é o arquivo '*.o'.

O último estágio do processo de compilação é a criação do arquivo executável '*.elf', que significa executável e formato linkavel (executable and linkable format). Nesse estágio todos os símbolos e outras informações são resolvidas, e também faz um merge de todas as diferentes seções do arquivo objeto realocável '*.o', criando um único arquivo executável.

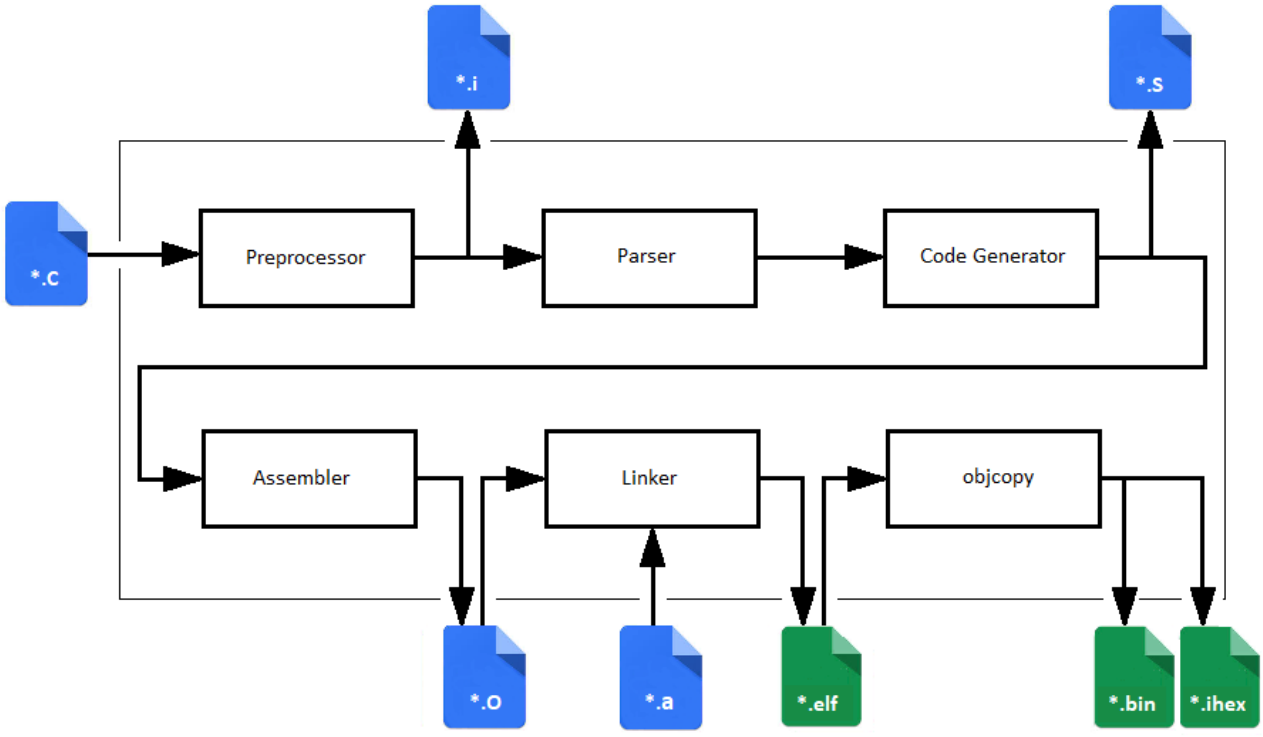

Figura 3 – Diagrama do GNU Cross Toolchain

A figura 3 ilustra todo o processo de compilação realizada pelo GCC. Nesse diagrama foi acrescentado um bloco utilitário, chamado 'objcopy', que converter o arquivo executável '*.elf' em outros formatos. Como já vimos a coleção de ferramentas do GCC tem outros utilitários, que podem ser revistos na tabela 1.

Resumindo o processo de construção de um arquivo executável, conhecido como 'build', tem as seguintes etapas principais, executadas pelo arquivo binário 'arm-nome-eabi-gcc', que são:

- ✔ Pré-processamento (Preprocessor);
- ✔ Compilação (Parser, Code Generator e Assemble);
- ✔ Linkagem (Linker).

Compilação na Linha de Comando do DOS

Depois de baixado e instalado o GNU Tools (GCC) for ARM Embedded Processors em seu 'host', e também garantindo que a subpasta [bin] esteja no PATH do seu 'host', vamos aplicar o que vimos até aqui.

Segue o link da documentação do GCC https://gcc.gnu.org/onlinedocs/gcc/index.html, para obter mais detalhes. Na tabela de conteúdo desse site podemos encontrar a opção 'GCC Command Options – Options Controlling the Kind of output e Machine-Dependent Options – ARM Options', onde tem as opções que usaremos aqui.

Abra uma janela de comando do DOS e crie uma simples aplicação 'main.c', como por exemplo:

```
D:\GNU Tools Arm Embedded\Test>type main.c

void vFunction(void);
int vVar1 = 10;
static int vVar2;
char vVar3 = 'A';
const int vConst1 = 5;

int main(void)
{
	vFunction();
}

void vFunction()
{
	int vVar4 = 20;
	vVar1 -= 5;
	vVar2 = 7;
	vVar3 = 'B';
	vVar4 += 10;
}
```

- As ferramentas de compilação e analise nos oferecem muitas opções, então, veremos algumas. Para compilar e criar o arquivo objeto realocável 'main.o', digite o seguinte comando na janela do DOS.

```
>arm-none-eabi-gcc -c -mcpu=cortex-m3 -mthumb main.c -o0 -o main.o
```

O seguinte comando permite vermos que o arquivo 'main.o' foi criado

```
D:\GNU Tools Arm Embedded\Test>dir
```

```
 O volume na unidade D é Novo volume
 O Número de Série do Volume é 1067-F974

 Pasta de D:\GNU Tools Arm Embedded\Test

04/09/2020  07:42    <DIR>          .
04/09/2020  07:42    <DIR>          ..
04/09/2020  07:32               222 main.c
04/09/2020  07:42             1.224 main.o
               2 arquivo(s)        1.446 bytes
```

Na linha de comando o argumento '-c' determina que o GCC compile e crie o arquivo objeto realocável (main.o), mas, não processa no Linker, então, não cria o arquivo executável 'main.elf'. O argumento '-mcpu=cortex-m3' informa a arquitetura do processador 'target'. O argumento '-mThumb' informa que as instruções do código são 'Thumb'. O processador ARM Cortex-M3 não suporta instruções de código no padrão 'ARM', somente 'Thumb'. O argumento '-o0' determina que o compilador não otimize o processo de compilação. O argumento '-o' determina que será criado um arquivo de saída, que nesse caso 'main.o', nosso arquivo objeto realocável.

Na documentação do GCC https://gcc.gnu.org/onlinedocs/gcc/index.html, na tabela de conteúdo podemos encontrar a opção 'GCC Command Options – Options Controlling the Kind of output'. Sobre o argumento '-c' encontramos:

"-c

Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix '.c', '.i', '.s', etc., with '.o'.

Unrecognized input files, not requiring compilation or assembly, are ignored."

- Outra opção é quando queremos obter apenas o arquivo 'main.s', que é a aplicação convertida em linguagem Assembly.

```
>arm-none-eabi-gcc -S -mcpu=cortex-m3 -mthumb main.c
```

O seguinte comando permite vermos que o arquivo 'main.o' foi criado

```
D:\GNU Tools Arm Embedded\Test>dir
 O volume na unidade D é Novo volume
 O Número de Série do Volume é 1067-F974

 Pasta de D:\GNU Tools Arm Embedded\Test

05/09/2020  08:45    <DIR>          .
05/09/2020  08:45    <DIR>          ..
04/09/2020  07:32               222 main.c
05/09/2020  08:44             1.224 main.o
05/09/2020  08:49             1.783 main.s
               3 arquivo(s)        3.229 bytes
```

```
               2 pasta(s)   2.983.266.111.488 bytes disponíveis
```

Para ver o conteúdo do arquivo criado ‘main.s’, digite o seguinte comando:

```
D:\GNU Tools Arm Embedded\Test>type main.s
        .cpu cortex-m3
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 1
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "main.c"
        .text
        .global vVar1
        .data
        .align  2
        .type   vVar1, %object
        .size   vVar1, 4
vVar1:
        .word   10
        .bss
        .align  2
vVar2:
        .space  4
        .size   vVar2, 4
        .global vVar3
        .data
        .type   vVar3, %object
        .size   vVar3, 1
vVar3:
        .byte   65
        .global vConst1
        .section        .rodata
        .align  2
        .type   vConst1, %object
        .size   vConst1, 4
vConst1:
        .word   5
        .text
        .align  1
        .global main
        .arch armv7-m
        .syntax unified
        .thumb
        .thumb_func
        .fpu softvfp
```

```
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
        push    {r7, lr}
        add     r7, sp, #0
        bl      vFunction
        movs    r3, #0
        mov     r0, r3
        pop     {r7, pc}
        .size   main, .-main
        .align  1
        .global vFunction
        .syntax unified
        .thumb
        .thumb_func
        .fpu softvfp
        .type   vFunction, %function
vFunction:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        movs    r3, #20
        str     r3, [r7, #4]
        ldr     r3, .L4
        ldr     r3, [r3]
        subs    r3, r3, #5
        ldr     r2, .L4
        str     r3, [r2]
        ldr     r3, .L4+4
        movs    r2, #7
        str     r2, [r3]
        ldr     r3, .L4+8
        movs    r2, #66
        strb    r2, [r3]
        ldr     r3, [r7, #4]
        adds    r3, r3, #10
        str     r3, [r7, #4]
        nop
        adds    r7, r7, #12
        mov     sp, r7
        @ sp needed
        pop     {r7}
        bx      lr
.L5:
        .align  2
```

```
.L4:
        .word   vVar1
        .word   vVar2
        .word   vVar3
        .size   vFunction, .-vFunction
        .ident  "GCC: (GNU Tools for Arm Embedded Processors 9-2019-q4-major) 9.2.1 20191025
(release) [ARM/arm-9-branch revision 277599]"
```

Na documentação do GCC https://gcc.gnu.org/onlinedocs/gcc/index.html, na tabela de conteúdo podemos encontrar a opção 'GCC Command Options – Options Controlling the Kind of output'. Sobre o argumento '-S' encontramos:

"-S

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix '.c', '.i', etc., with '.s'.

Input files that don't require compilation are ignored."

- Outra opção que veremos são algumas análises que podemos fazer, no arquivo objeto realocável 'main.o'.

```
>arm-none-eabi-objdump -h main.o
```

O seguinte comando permite analisarmos as principais seções do arquivo 'main.o', que são '.text', '.data', '.bss' e 'rodata'.

```
D:\GNU Tools Arm Embedded\Test>arm-none-eabi-objdump -h main.o

main.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000004c  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000005  00000000  00000000  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  00000088  2**2
                  ALLOC
  3 .rodata       00000004  00000000  00000000  00000088  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000007a  00000000  00000000  0000008c  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002d  00000000  00000000  00000106  2**0
                  CONTENTS, READONLY
```

O argumento '-h' mostra os conteúdos das seções do arquivo objeto realocável.

São muitas possibilidades, mas, já temos uma ideia de como funciona atrás dos bastidores.