

# Introdução ao Makefile

Por **Lincoln Uehara** - 11/09/2017



## ÍNDICE DE CONTEÚDO [OCULTAR]

- Compilação nativa X Compilação cruzada
- GCC e como o programa é compilado
- Escrevendo as primeiras linhas do makefile
- Macros e variáveis automáticas
- Deixando organizado e elegante
- Considerações finais
- Referências e links interessantes

A resposta para a pergunta da capa, em suma, é: os arquivos-fonte são transformados em arquivos-objetos pelo compilador, e os arquivos-objetos são ligados pelo *linker* para se transformar num binário. Porém esse processo é muitas vezes mascarado pois no final das contas “o IDE acaba fazendo tudo”. Certamente temos no mercado IDE’s poderosos, *user-friendly*, e que prometem aumento na produtividade, mas que em algum momento o desenvolvedor poderá se deparar em alguma incompatibilidade de ferramentas e principalmente com a falta de documentação apropriada para o seu próprio projeto.

Gostou? [Junte-se à comunidade Embarcados](#)

A intenção deste artigo é apresentar e ensinar a escrever um makefile, um arquivo que consta as instruções de como gerar um binário. Dentre as vantagens de se escrever um makefile estão:

- É uma forma de documentação. Ao invés de guardar na cabeça ou pesquisar no Google como você configura o IDE para aquele microcontrolador, você pode escrever um arquivo que estará junto aos arquivos-fonte, e que dá para fazer um git e trabalhar em outros computadores ou com outros colegas sem ter diferença entre os binários gerados;
  - É um script, então dá para fazer outras coisas além de gerar o binário, como procurar ou organizar os arquivos, e até integrar com ferramentas de documentação como doxygen;
  - Os IDE's podem gerar o binário baseando no makefile que você escreveu. Portanto a única configuração que você precisaria saber sobre o IDE é ligar ao makefile, aumentando a sua in
  - Aprofunda os conhe
- processo de produção  
programação.

## Pesquisa

### Pesquisa sobre o Mercado Brasileiro de Desenvolvimento de Sistemas Embarcados 2017

São aproximadamente 30 questões e você levará  
menos de 8 minutos para responder. Concorra ao  
sorteio de mais de 60 itens.

## Compilação nat



## RESPONDER PESQUISA

A compilação nativa é usada para criar aplicações para a mesma arquitetura da sua máquina de desenvolvimento. Por exemplo, um programa de caixa de supermercado desenvolvido em linguagem Java em sua máquina x86, que vai rodar em uma outra máquina x86.

Já na compilação cruzada (*cross-compiling*), você vai gerar um binário para uma arquitetura diferente da sua máquina de desenvolvimento, como é o caso de microcontroladores, Linux embarcado e aplicativos de celular.

Gostou? [Junte-se à comunidade Embarcados](#)

A ferramenta open source mais famosa é a [GNU Compiler Collection](#) (GCC). Ela trabalha tanto com compilação nativa quanto a cruzada.

## GCC e como o programa é compilado

GCC é a componente chave de um [GNU toolchain](#), que, como o próprio nome diz, é uma “cadeia de ferramentas” para desenvolvimento e depuração. De uma forma bem resumida, o GCC vai transformar o seu código-fonte em um arquivo-objeto e vai ligar esses arquivos-objetos para transformá-los em um binário.

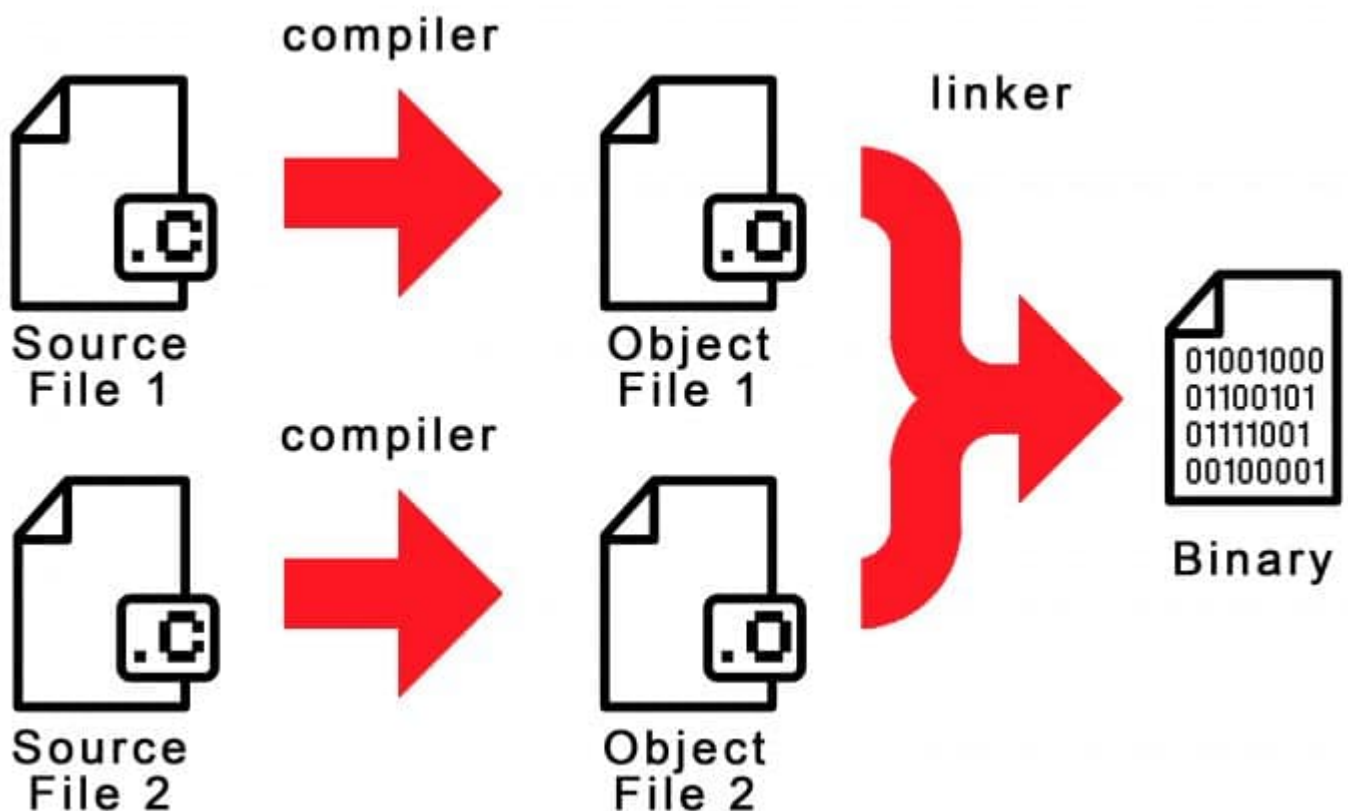


Figura 1 - Processo resumido de geração de um binário. Aqui omitimos a geração dos arquivos assembly.

Neste artigo vamos utilizar a ferramenta *make* do *GNU toolchain*, que já vem nas máquinas Linux. Já para o usuário do Windows é necessário instalar Cygwin ou MinGW e adicionar o seu caminho na variável de ambiente PATH.

## Escrevendo as primeiras linhas do makefile

O cerne do makefile consiste em regras definidas da seguinte maneira:

```
1 target : prerequisites
2 <TAB>recipe
```

O *target* é o nome da ação que você deseja executar ou usualmente o nome do arquivo que se queira produzir. Os pré-requisitos são arquivos que são usados como input para criar o *target*.

A receita é a ação que o comando *make* realiza. A receita pode ter mais de um comando, na mesma linha ou várias. ATENÇÃO: Você precisa usar o caractere <TAB> no começo de cada receita. O *make* interpreta como indicação de começo de comando a ser executado.

Praticaremos com um exemplo: vamos supor que temos uma pasta com os seguintes arquivos: *main.c*, *helloWorld.h*, *helloWorld.c* e o makefile que você vai escrever. O GNU reconhece os seguintes nomes para makefile: *GNUmakefile*, *makefile* e *Makefile*. A ferramenta *make* é sensível a maiúsculas e minúsculas. Você quer criar um binário chamado *printy*. Abaixo estão descritos os conteúdos dos arquivos respectivos, e iremos depois analisar o arquivo makefile.

### Arquivo *main.c*:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "helloWorld.h"
4
5 int main(){
6     helloWorld();
7     return (0);
8 }
```

### Arquivo *helloWorld.h*:

```
1 #ifndef _H_TESTE
2 #define _H_TESTE
3
4 void helloWorld(void);
5
6 #endif
```


### Arquivo *helloWorld.c*:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void helloWorld(void){
5     printf("Hello World!\n");
6 }
```

### Arquivo *makefile*:

```
1 # My first makefile
2
3 all: printy
4
5 printy: main.o helloWorld.o
6     gcc -o printy main.o helloWorld.o
7
8 main.o: main.c helloWorld.h
9     gcc -o main.o main.c -c -W -Wall -ansi -pedantic
10
11 helloWorld.o: helloWorld.c helloWorld.h
12     gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
13
14 clean:
15     rm -rf *.o *~ printy
```

No terminal, ao digitar “make all”, o utilitário make vai executar o *target all* que se encontra na linha 3 do arquivo makefile. O seu pré-requisito é o arquivo binário *printy*, porém sem receita para *all*. Ele vai para a linha 5 para construir *printy*, e acha os pré-requisitos *main.o* e *helloWorld.o*. Antes de partir para a linha 6, o make vai produzir os respectivos objetos.

Na linha 8, o make vê que o pré-requisito para *main.o* é a existência dos arquivos *main.c* e *helloWorld.h* no diretório atual. O make executa então a receita da linha 9. Primeiro invoca o compilador gcc. A parte *-o main.o* indica que o output, ou seja, o arquivo produzido será *main.o*. Para isso vai usar o arquivo *main.c*, que é o pré-requisito indicado na linha anterior. Os flags seguintes são configurações a serem passados ao compilador. O *-Wall* e o *-pedantic* são importantes para achar erro em seu código, mais detalhes você pode ler [nesta página](#) . Terminando de produzir o arquivo-objeto *main.o*, o make vai para as linhas 11 e 12 para produzir o arquivo-objeto *helloWorld.o*.

Tendo os dois arquivos-objetos em mãos, o make volta para a linha 6 para cumprir a receita do binário *printy*. O make invoca agora o linker do gcc, e avisa utilizando *-o printy* que o output será um binário no diretório atual chamado *printy*. E indica ao linker os dois arquivos objetos *main.o* e *helloWorld.o*. Uma vez produzido o binário *printy*, o make cumpre o pré-requisito do comando *all* da linha 3.

O comando *clean* da linha 14 somente será executado se você escrever no terminal *make clean*. Ele não possui pré-requisitos, e a receita manda excluir todos os arquivos com extensão *.o*, os de backup *\*~* e o arquivo binário *printy* no diretório atual.

Se você produziu o binário em seu computador, obviamente somente rodará em sua máquina. Se colocar o binário em um microcontrolador ou em uma raspberry não vai funcionar pois não foi um cross-compiling. Para executar o arquivo, basta digitar “./printy” em seu terminal.

Se você editar o arquivo *helloWorld.c* para imprimir outra mensagem e depois digitar o comando “make all”, somente este arquivo será recompilado e religado para formar o binário final.

```
1 $ make all
2 gcc -o main.o main.c -c -W -Wall -ansi -pedantic
3 gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
4 gcc -o printy main.o helloWorld.o
5 $ ./printy
6 Hello World!
```

Após editar o arquivo *helloWorld.c*:

```
1 $ make all
2 gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
3 gcc -o printy main.o helloWorld.o
4 $ ./printy
5 Embarcados' Article!
```

## Macros e variáveis automáticas

Você deve ter percebido que seria enfadonho cada vez que você tivesse que adicionar um novo arquivo. E somando a isso, não daria para reaproveitar o makefile para outros projetos, já que dificilmente teriam os mesmos nomes de arquivo.

Para resolver isso, utiliza-se de variáveis para facilitar as alterações e deixar o arquivo mais claro.

```
1 # My second makefile
2
```


Gostou? [Junte-se à comunidade Embarcados](#)

```

7 C_SOURCE=$(wildcard *.c)
8
9 # .h files
10 H_SOURCE=$(wildcard *.h)
11
12 # Object files
13 OBJ=$(C_SOURCE:.c=.o)
14
15 # Compiler
16 CC=gcc
17
18 # Flags for compiler
19 CC_FLAGS=-c \
20         -W \
21         -Wall \
22         -ansi \
23         -pedantic
24
25 #
26 # Compilation and linking
27 #
28 all: $(PROJ_NAME)
29
30 $(PROJ_NAME): $(OBJ)
31     $(CC) -o $@ $^
32
33 %.o: %.c %.h
34
35     $(CC) -o $@ $< $(CC_FLAGS)
36
37 main.o: main.c $(H_SOURCE)
38     $(CC) -o $@ $< $(CC_FLAGS)
39
40 clean:
41     rm -rf *.o $(PROJ_NAME) *~

```

Vemos, da linha 3 a 23, as variáveis para facilitar a customização do arquivo. Algumas vezes as variáveis são chamadas de macros. O uso das variáveis é igual a do shell script, e escreve-se `$(VARIÁVEL)` ou `${VARIÁVEL}`, ambas formas estão corretas.

Nas linhas 7 e 10 utilizamos a função *wildcard* para pegar o nome de todos os arquivos com extensões `.c` e `.h` no diretório em que o makefile se encontra. Na variável `C_SOURCE` o nome de cada arquivo estará separado um do outro por um espaço. Na linha 10 copiamos todos os nomes da variável `C_SOURCE` para `OBJ`, mas com a substituição da extensão `.c` para `.o`. As funções *subst* e *patsubst* fazem funções parecidas, confira [a documentação oficial](#) . Fazemos isso pois inicialmente os arquivos-objeto não existem, então a função *wildcard* seria ineficiente nesse caso.

Gostou? [Junte-se à comunidade Embarcados](#)



Das linhas 19 a 22 utilizamos backlash ‘\’ para quebrar uma linha em várias, para tornar a leitura mais fácil. E atenção, para indentar aqui, não utilize a tecla <TAB>, utilize espaços para que o make não confunda com comando de receita.

Das linhas 28 a 40, num primeiro momento, pode causar uma sensação de “Cruz credo!”, mas dê uma pausa, vá beber água, sente-se na cadeira, respire fundo e vamos analisar cada linha com calma.

O resultado das linhas 28 e 30 deste segundo exemplo é igual às linhas 3 e 5 do primeiro exemplo. Vamos pular por ora para a linha 33. Nessa linha interpreta-se que “o target com extensão .o terá um pré-requisito com extensões .c e .h com mesmo nome”. O símbolo ‘%’ pega o *stem* (tronco) do nome, que é utilizado de referência no pré-requisito. Por exemplo, se o *target* é *helloWorld.o*, o *stem* será *helloWorld*, consequentemente o pré-requisito será *helloWorld.c* e *helloWorld.h*. Perceba aqui que não usamos \$(OBJ) como target, pois na expansão da macros surgiria os objetos *main.o* e *helloWorld.o* antes dos dois pontos, o que não seria correto.

A linha 34 desse segundo exemplo é relativo às linhas 9 e 12 do primeiro exemplo. Aqui a variável automática \$@ pega o nome do *target* e \$< pega o nome do primeiro pré-requisito. Então, \$@ está para *helloWorld.o* e \$< está para *helloWorld.c*. Perceba que as linhas 33 e 34 vão ser executadas para tantos outros arquivos .c e .h que existirem. Já as linhas 36 e 37 foi escrita para o caso especial do *main.o*, que não possui arquivo *main.h* e que depende de outros arquivos headers.

Voltando a linha 31, esta é referente à linha 6 do primeiro exemplo, e aqui utilizamos a variável automática \$^ para listar todos os pré-requisitos do *target*. Neste caso, será expandido para todos os nomes que a variável OBJ contém.

## Deixando organizado e elegante

Saber organizar, além de aumentar a produtividade, gera sincronia com a equipe com a qual você está trabalhando. Se você usa GitHub como seu cartão de visita, não seria interessante destacar a sua organização?

No seguinte exemplo, quis organizar da seguinte forma: na pasta do projeto, quero colocar um makefile e duas pastas, um para os códigos-fonte e outra para os objetos criados, cuja pasta vai ser criada pelo makefile. O binário deve ser criado na pasta do projeto. Entenda como fica a organização da pasta antes e depois da compilação:

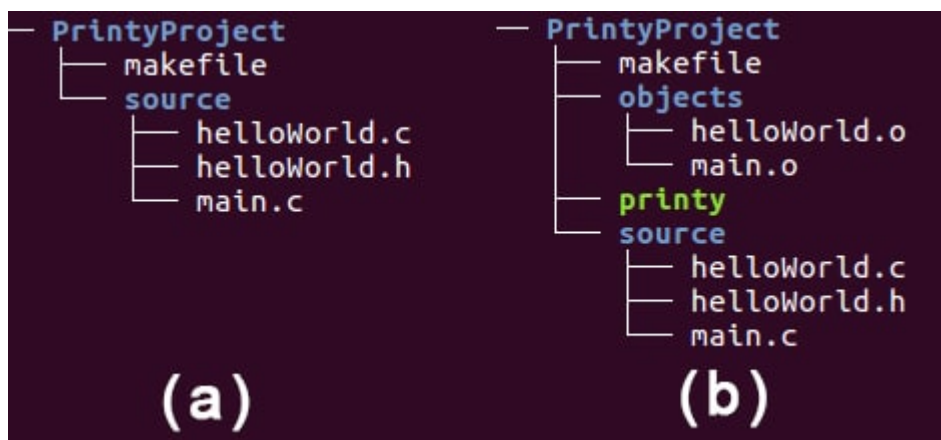


Figura 2 - Organização do projeto PrintyProject antes (a) e depois (b).

O makefile seria escrito da seguinte maneira:

```
1 # My third makefile
2
3 # Name of the project
4 PROJ_NAME=printy
5
6 # .c files
7 C_SOURCE=$(wildcard ./source/*.c)
8
```

Gostou? [Junte-se à comunidade Embarcados](#)

```

12 # Object files
13 OBJ=$(subst .c,.o,$(subst source,objects,$(C_SOURCE)))
14
15 # Compiler and linker
16 CC=gcc
17
18 # Flags for compiler
19 CC_FLAGS=-c \
20         -W \
21         -Wall \
22         -ansi \
23         -pedantic
24
25 # Command used at clean target
26 RM = rm -rf
27
28 #
29 # Compilation and linking
30 #
31 all: objFolder $(PROJ_NAME)
32
33 $(PROJ_NAME): $(OBJ)
34 @ echo 'Building binary using GCC linker: $@'
35 $(CC) $^ -o $@
36 @ echo 'Finished building binary: $@'
37 @ echo ' '
38
39 ./objects/%.o: ./source/%.c ./source/%.h
40 @ echo 'Building target using GCC compiler: $<'
41 $(CC) $< $(CC_FLAGS) -o $@
42 @ echo ' '
43
44 ./objects/main.o: ./source/main.c $(H_SOURCE)
45 @ echo 'Building target using GCC compiler: $<'
46 $(CC) $< $(CC_FLAGS) -o $@
47 @ echo ' '
48
49 objFolder:
50 @ mkdir -p objects
51
52 clean:
53 @ $(RM) ./objects/*.o $(PROJ_NAME) *~
54 @ rmdir objects
55
56 .PHONY: all clean

```

Normalmente a ferramenta *make* imprime na tela do terminal cada linha da receita a ser executada. Para não poluir visualmente o terminal, colocamos no começo da linha o caractere '@' para suprimir essas impressões. Em conjunto com o comando *echo* podemos deixar mensagens do que está sendo feito e colocar espaço entre uma compilação e outra na tela do terminal.

```

1 $ make all
2 Building target using GCC compiler: source/helloWorld.c
3 gcc source/helloWorld.c -c -W -Wall -ansi -pedantic -o objects/helloWorld.o

```

Gostou? [Junte-se à comunidade Embarcados](#)

```
7
8 Building binary using GCC linker: printy
9 gcc objects/helloWorld.o objects/main.o -o printy
10 Finished building binary: printy
```

Na linha 56 do arquivo *makefile* vemos o *target* *.PHONY* (“alvo falso”). Nela colocamos como pré-requisito os targets que não possuem arquivos de mesmo nome. Uma das razões para usar *phony target* é para não dar conflito com arquivos que sejam criados com mesmo nome (por exemplo, se existisse um arquivo chamado *all.c* ou *clean.c*).

## Considerações finais

Para escrever um *makefile* voltado para microcontroladores ARM, por exemplo, é necessário baixar o pacote *GNU ARM Embedded Toolchain* [↗](#). Dentro do pacote você irá encontrar as ferramentas *arm-none-eabi-gcc* (compilador) e *arm-none-eabi-ld* (*linker*) e você pode indicar o caminho das ferramentas no *makefile* ou editar as variáveis de ambiente da sua máquina.

Se você não sabe exatamente como configurar para o seu microcontrolador, você pode conferir o arquivo *makefile* ou *makedefs* que vem junto à biblioteca que o fabricante te fornece. Aliás, toda documentação que o fabricante te fornece deve ser lido o máximo possível.

Outra possibilidade inusitada é conferir o terminal de sua IDE na hora da compilação. O IDE Eclipse produz automaticamente o *makefile* (na verdade são vários *makefiles*) e você também pode aprender fuçando esses arquivos.

Com a prática você pode automatizar e personalizar vários processos que você executa no dia a dia, e ainda repassar o arquivo aos seus colegas de trabalho através de repositórios *git*, criando um fluxo de trabalho mais dinâmico.

Confira também os documentos oficiais para saber quais outros potenciais o makefile possui. Lembre-se que não há fonte melhor que a documentação oficial!

## Referências e links interessantes

[1] [GCC, the GNU Compiler Collection](#) ↗

[2] [GNU Make Manual](#) ↗

[3] [GNU ARM Embedded Toolchain](#) ↗

[4] [Options Controlling the Preprocessor](#) ↗

[5] ["GNU Cross-toolchain - Processo de build"](#) por Henrique Rossi

[6] ["Desmistificando toolchains em Linux Embarcado"](#) ↗ – Blog Sergio Prado



*Introdução ao Makefile* por [Lincoln Uehara](#) ↗. Esta obra está licenciado com uma Licença [Creative Commons Atribuição-Compartilhual 4.0 Internacional](#) ↗.

**Lincoln Uehara**

<http://linkedin.com/in/lincoln-uehara> ↗

Gostou? [Junte-se à comunidade Embarcados](#)