Cross Development using Makefile

by Burglar/SCS*TRC

Introduction

GNU Make is a tool used to build executables, libraries and whole applications. You can use it to script all commands you are using to compile your code. Once properly set up, it will automatically resolve dependencies and only rebuild what has changed.

This will drastically speed up your development process for any project, small or large.

Once automated you will never have to do conversions of graphics and sids by hand anymore.

It will also help you think structured about your code and data.

How Makefile works

A basic Makefile is a plaintext file in your working directory consisting of one or more rules to satisfy all dependencies required to build the requested target. A dependency can be another file or another target.

```
target: dependencies
[tab]command
```

WARNING: The single tab is important, make sure your editor doesn't change them.

The target is what we want, the dependencies what we need and the command is to build the required target.

Let's say we're coding an intro in kickass and to compile it we type:

```
java -jar /usr/share/java/KickAss.jar intro.s -o intro.prg
```

To do that with a Makefile:

```
intro.prg: intro.s
    java -jar /usr/share/java/KickAss.jar intro.s -o intro.prg
```

Now you can run “make” or “make intro.prg” to compile.

Implicit Rules

The command to build your sources is probably always the same, so you can use wildcards to make

implicit rules.

The automatic variables $< and $@ are used as dependency and target respectively.

```
# % is a wildcard
# $< is the first dependency
# $@ is the target
# $^ is all dependencies

%.prg: %.s
    java -jar /usr/share/java/KickAss.jar $< -o $@
```

This enables you to compile any ".s" source file with "make anything.prg".

Setting up dependencies

We are still working on the intro, and now we're going to add music and a bitmap logo. We add the necessary imports and a LoadSid in our source-file intro.s. Time to add the dependencies in our Makefile:

```
# our implicit build rule
%.prg: %.s
    java -jar /usr/share/java/KickAss.jar $< -o $@

# default build target
all: intro.prg

# dependencies for our intro, which files do we need to compile intro.s into
intro.prg
# the implicit rule will take care of depending on intro.s itself
intro.prg: tune.sid logo.prg
```

Now type "make" twice, it builds, but the second time it won't waste your time like a .bat. If you modify any of the files "intro.s", "tune.sid" or "logo.prg", make will rebuild the intro.

Automatic Crunching

To add another rule to our build chain, we need to think of a new file extension for the new target. We'll use exomizer, so ".prg.exo" sounds about right.

```
exomizer sfx basic -n intro.prg -o intro.prg.exo
```

Let Makefile handle the thing:

```
%.prg: %.s
    java -jar /usr/share/java/KickAss.jar $< -o $@
```

```
%.prg.exo: %.prg
    exomizer sfx basic -n $< -o $@

all: intro.prg.exo

intro.prg: tune.sid logo.prg
```

This same technique can be used for many other things, see the full example below.

Variables/Macros

You can use variables to define some defaults and locations. It will also clean up your Makefile and make it more readable. It is best explained by example:

```
JAVA=java
KICKASS=$(JAVA) -jar /usr/share/java/KickAss.jar
EXOMIZER=exomizer
EXOMIZERFLAGS=sfx basic -n
SOURCEFILES=intro.s part1.s part2.s
OBJECTS=intro.prg part1.prg part2.prg
PACKEDOBJECTS=intro.prg.exo part1.prg.exo part2.prg.exo

%.prg: %.s
    $(KICKASS) $< -o $@

%.prg.exo: %.prg
    $(EXOMIZER) $(EXOMIZERFLAGS) $< -o $@

all: $(PACKEDOBJECTS)

$(OBJECTS): tune.sid logo.prg
```

Btw, this demo sucks, same tune and logo in every part ;)

Cleaning up your working directory

Sometimes you may want to get rid of every object to be able to explicately rebuild everything. Or you want to give your source to someone else, you'll only want to give the source, not pre-built objects.

The common way to do that is to add a “make clean” target:

```
clean:
    rm -f *.exo
    rm -f intro.prg
    rm -f part?.prg
```

Building in Parallel

Most people have a cpu with more than 1 core. Makefile can transparently utilize them to speed up your compiles even more. It is important you have all your dependencies correctly mapped out or strange things might happen.

```
# build using up to 4 cores in parallel
make -j4 all
```

Full Example

Combining the things we learned, we can now write a Makefile capable of:

- compiling kickass source code
- compiling 64tass source code
- packing with exomizer
- bitmap to sprite conversion
- building a d64 with all files
- autostarting vice with “make vice” (6581, press f7 at start to avoid cartridge).

We already have java, kickassembler, 64tass, exomizer, cc1541, bitmap2spr.py and vice installed and in our path.

Source Files

```
Makefile
intro.s             # uses LoadSid to include the .sid
lib.s               # contains macros and stuff
part1.s             # uses LoadSid and .import c64 "gfx/picture.prg"
part2.s             # uses LoadSid and .import c64 "gfx/spritelogo.spr"
part3.asm           # has no dependencies
gfx/picture.s       # calls a macro to convert the .png to .prg
gfx/picture.png
gfx/spritelogo.bmp
sid/introtune.sid
sid/part1.sid
sid/part2.sid
```

Makefile

```
JAVA=java
KICKASS=$(JAVA) -jar /usr/share/java/KickAss.jar
```

```
TASS64=64tass
EXOMIZER=exomizer
EXOMIZERFLAGS=sfx basic -n
CC1541=cc1541
BITMAP2SPR=bitmap2spr.py
VICE=x64
VICEFLAGS=-sidenginemodel 1803 -keybuf "\88"
SOURCEFILES=intro.s part1.s part2.s part3.asm gfx/picture.s
OBJECTS=intro.prg.exo part1.prg.exo part2.prg.exo part3.prg.exo

%.prg: %.s
    $(KICKASS) $< -o $@

%.prg: %.asm
    $(TASS64) $< -o $@

%.spr: %.bmp
    $(BITMAP2SPR) $< $@

%.prg.exo: %.prg
    $(EXOMIZER) $(EXOMIZERFLAGS) $< -o $@

all: demo.d64

vice: demo.d64
    $(X64) $(X64FLAGS) $<

intro.prg: lib.s sid/introtune.sid gfx/spritelogo.spr

part1.prg: lib.s sid/part1.sid gfx/picture.prg

gfx/picture.prg: lib.s gfx/picture.png

part2.prg: lib.s sid/part2.sid gfx/spritelogo.spr

demo.d64: $(OBJECTS)
    $(CC1541) -f "DEMO" -w intro.prg.exo \
    -f "PART1" -w part1.prg.exo \
    -f "PART2" -w part2.prg.exo \
    -f "PART3" -w part3.prg.exo \
    $@

clean:
    rm -f demo.d64
    rm -f *.prg
    rm -f *.exo
    rm -f gfx/*.spr
    rm -f gfx/*.prg
    rm -f gfx/*.exo
```

Questions and Comments

Please refer to the thread on CSDb.

You can also check the official GNU Make Manual.