

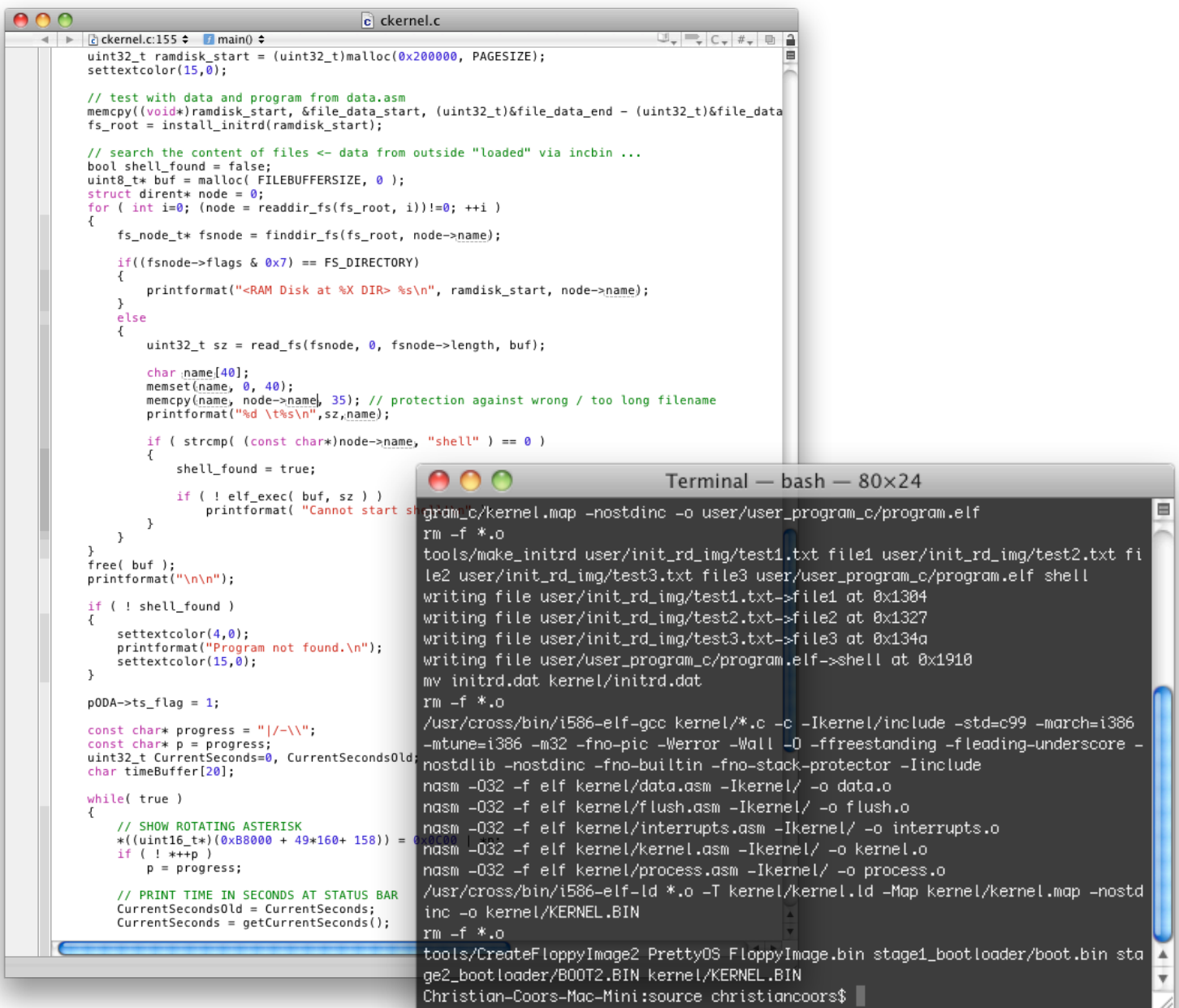
Erstellen von PrettyOS unter Mac OS X

Diese Anleitung befasst sich mit der Vorbereitung von Mac OS X auf das Erstellen von PrettyOS.

WICHTIGE HINWEISE:

Mit den hier aufgeführten Änderungen werden nicht ungefährliche Bestandteile im System geändert. Bitte lesen Sie sorgfältig alle Hinweise und brechen Sie lieber ab, wenn Sie einen Fehler machen oder ein unvorhergesehenes Problem auftritt. Weiterhin wird eine vollständige Sicherung des Systems (z.B. mit TimeMachine, auf jedem neuen Mac mitgeliefert) dringend empfohlen. UNIX-Kenntnisse sind auch dringend empfohlen.

Wollen Sie PrettyOS unter Mac OS X erstellen? Wollen Sie Xcode als Editor für .c, .cpp und .h Dateien nutzen? Dann befolgen Sie diese Anleitung!



Inhalt:

1. Xcode
2. Anpassen von PrettyOS
3. Binutils und GNU C Compiler für i586-elf

1. Xcode:

Essentiell für die Softwareentwicklung unter Mac OS X sind die Developer Tools (Xcode). Zunächst sollten Sie aber sicherstellen, dass alle aktuellen Updates sowohl für Mac OS X als auch alle Programme installiert sind und dass kein Neustart ansteht.

Obwohl auf allen aktuellen Mac OS X CDs bzw. DVDs Xcode enthalten ist, empfehle ich, immer die neueste Version herunterzuladen.

Öffnen Sie auf Ihrem Mac die folgende Website: <http://developer.apple.com/mac/>. Dort finden Sie nach einer kostenlosen Registrierung Xcode als Download. Laden Sie sich diese Datei herunter (Achtung! Die Datei ist sehr groß!). Installieren Sie Xcode wie jedes andere Programm auch mit dem mitgelieferten Installationsprogramm. Nach einem Neustart sollten Sie wiederum über das Apfel-Menü aktuelle Updates suchen und installieren. Nun ist Xcode installiert und bereit. Die Installation von Xcode ist damit abgeschlossen.

2. Anpassen von PrettyOS:

Im PrettyOS-Makefile müssen alle „**rm *.o -f**“ durch „**rm -f *.o**“ ersetzt werden! Des Weiteren müssen einige Variablen angepasst werden. Mein aktuelles Makefile sieht so aus:

```
STAGE1DIR= stage1_bootloader
STAGE2DIR= stage2_bootloader
KERNELDIR= kernel
USERRDIR= user/init_rd_img
USERDIR= user/user_program_c

ifeq ($(OS),WINDOWS)
    NASM= nasmw
    CC= i586-elf-gcc
    LD= i586-elf-ld
    #NASM= tools/nasmw
    #CC= tools/i586-elf/bin/i586-elf-gcc
    #LD= tools/i586-elf/bin/i586-elf-ld
else
    NASM=nasm
    CC=/usr/cross/bin/i586-elf-gcc
    LD=/usr/cross/bin/i586-elf-ld
endif

all: boot1 boot2 ckernel

boot1: $(wildcard $(STAGE1DIR)/*.asm $(STAGE1DIR)/*.inc)
    $(NASM) -f bin $(STAGE1DIR)/boot.asm -I$(STAGE1DIR)/ -o $(STAGE1DIR)/
boot.bin

boot2: $(wildcard $(STAGE2DIR)/*.asm $(STAGE2DIR)/*.inc)
    $(NASM) -f bin $(STAGE2DIR)/boot2.asm -I$(STAGE2DIR)/ -o $(STAGE2DIR)/
BOOT2.BIN

ckernel: $(wildcard $(KERNELDIR)/* $(KERNELDIR)/include/*) initrd
    rm -f *.o
```

Erstellen von PrettyOS unter Mac OS X - Getestet mit Snow Leopard 10.6.2 und den Xcode Tools in Version 3.1 und 3.2.2 BETA

```
$(CC) $(KERNELDIR)/*.c -c -I$(KERNELDIR)/include -std=c99 -march=i386 -
mtune=i386 -m32 -fno-pic -Werror -Wall -O -ffreestanding -fleading-underscore -
nostdlib -nostdinc -fno-builtin -fno-stack-protector -Iinclude
$(NASM) -O32 -f elf $(KERNELDIR)/data.asm -I$(KERNELDIR)/ -o data.o
$(NASM) -O32 -f elf $(KERNELDIR)/flush.asm -I$(KERNELDIR)/ -o flush.o
$(NASM) -O32 -f elf $(KERNELDIR)/interrupts.asm -I$(KERNELDIR)/ -o
interrupts.o
$(NASM) -O32 -f elf $(KERNELDIR)/kernel.asm -I$(KERNELDIR)/ -o kernel.o
$(NASM) -O32 -f elf $(KERNELDIR)/process.asm -I$(KERNELDIR)/ -o process.o
$(LD) *.o -T $(KERNELDIR)/kernel.ld -Map $(KERNELDIR)/kernel.map -nostdinc
-o $(KERNELDIR)/KERNEL.BIN
rm -f *.o
tools/CreateFloppyImage2 PrettyOS FloppyImage.bin $(STAGE1DIR)/boot.bin $
(STAGE2DIR)/BOOT2.BIN $(KERNELDIR)/KERNEL.BIN

initrd: $(wildcard $(USERDIR)/*)
rm -f *.o
$(NASM) -O32 -f elf $(USERDIR)/start.asm -I$(USERDIR)/ -o start.o
$(CC) $(USERDIR)/*.c -c -I$(USERDIR) -m32 -fno-pic -Werror -Wall -O -
ffreestanding -fleading-underscore -nostdlib -nostdinc -fno-builtin
$(NASM) -O32 -f elf $(USERDIR)/start.asm -o start.o
$(LD) *.o -T $(USERDIR)/user.ld -Map $(USERDIR)/kernel.map -nostdinc -o $
(USERDIR)/program.elf
rm -f *.o
tools/make_initrd $(USERDIR)/test1.txt file1 $(USERDIR)/test2.txt
file2 $(USERDIR)/test3.txt file3 $(USERDIR)/program.elf shell
mv initrd.dat $(KERNELDIR)/initrd.dat
```

3. Binutils und GCC als Crosscompiler für i586-elf

Da Mac OS X nur den „eigenen“ GCC mitbringt, der Mac OS X-Programme erstellen kann, muss man die Binutils und den GCC selbst compilieren. Ich rate dringend davon ab, da besonders viele Anfänger Probleme damit haben, ich selbst habe auch ein wenig Zeit gebraucht, um herauszufinden, wie das funktioniert. Aus diesen Grund habe ich den aktuellen GCC (4.4.3) und die aktuellen Binutils (2.20) für i586-elf compiliert und hochgeladen.

Da dies auch für die Windows-Versionen geschehen ist, sehe ich kein Problem darin. Die Dateien befinden sich unter

<http://www.fanofblitzbasic.de/prettyos/i586-elf-binutils-gcc-macos.zip> (Achtung, Dateigröße beträgt 33,1 MB. Darin befindet sich das gesamte Paket. Am besten nach /usr/cross entpacken!)

Sollte dennoch der Grund bestehen, beides neu zu erstellen (z.B. für aktuellere Versionen), so ist folgende Anleitung zu befolgen.

Sie stammt von

http://wiki.osdev.org/GCC_Cross-Compiler

und wurde für PrettyOS entsprechend angepasst.

Dummerweise muss man an einigen Stellen etwas „tricksen“, um die Binutils, bzw. den GCC zu erstellen. Man muss bei allen „configure“-Anweisungen einen „**--disable-werror**“ einfügen (was verhindert, dass Warnungen als Fehler gewertet werden).

Es treten bei beiden einige Warnungen auf (seit Snow Leopard, da einige Sachen als „alt“ markiert sind), allerdings beeinflussen diese den Vorgang nicht weiter. **Das Dokument wurde teilweise gekürzt/angepasst. Die Anleitung beginnt unter der roten Linie!**

GCC Cross-Compiler

From OSDev Wiki

Jump to: [navigation](#), [search](#)

Difficulty level:

Beginner

Introduction

What is a cross-compiler?

Generally speaking, a cross-compiler is a compiler that runs on platform A (the "host"), but generates executables for platform B (the "target"). The two "platforms" might differ in CPU, operating system, and/or [executable format](#).

Requirements

- A host system with a working [GCC](#) installation (dank Xcode vorhanden), and enough memory as well as hard drive space (auch meistens vorhanden). How much qualifies as "enough" is depending on the versions of the software involved, but GCC is a big piece of software, so don't be surprised when 128 or 256 MByte are not sufficient.
- A bash shell or comparable environment (Programme=>Diensprogramme=>Terminal). If you are not using a bash shell, you might have to modify some of the command lines below. If you have just installed the basic [Cygwin](#) package, you have to run the setup.exe again and install the following packages:
 - [GCC](#) (even if you have something like MinGW installed)
 - Make (vorhanden)
 - Flex (nicht erforderlich)
 - Bison (nicht erforderlich)
- If you plan to use GCC 4.3.0 or a later version, you will also have to install the following:
 - [GMP](#) (this package is also readily available for MSYS)
 - [MPFR](#)

GMP und MPFR müssen manuell von der GNU-FTP-Seite installiert werden!

It has also been tested successfully with various combinations of binutils 2.14 / 2.15 and [GCC](#) 3.2 / 3.3. The numbers refer to the versions being built, not the host compiler doing the build.

Erstellen von PrettyOS unter Mac OS X - Getestet mit Snow Leopard 10.6.2 und den Xcode Tools in Version 3.1 und 3.2.2 BETA

Problems have been reported on trying to build bintils 2.14 with [Cygwin GCC](#) < 3.3.3.3 as host compiler, as well as on trying to build binutils <= 2.15 with [GCC](#) 4.x as host compiler.

There are no known issues with building a crosscompiler under Linux.

For Mac OS users, the combinations of GCC 4.2.4 / Binutils 2.18 and GCC 4.4.0 / Binutils 2.19.1 is known to work. Also GCC trunk (4.5.0) and Binutils trunk (2.20) is known to work. When compiling GCC 4.3 or higher on OS X 10.4 and 10.5, you may get unresolved symbol errors related to libiconv. This is because the version shipped with OS X is seriously out of date. Install a new version (compile it yourself or use macports) and add `--with-libiconv-prefix=/opt/local` (or `/usr/local` if you compiled it yourself) to GCC's `./configure` line. Alternatively you may place the libiconv source in `gcc-x.y.z/libiconv` and it will be compiled as part of the GCC compilation process. (This trick also works for mpfr and gmp).

There have been reported problems with old versions of Binutils and earlier versions may therefore not work as expected.

Step 1 - Bootstrap

We build a toolset running on your host that can turn source code into object files for your target system.

We need the binutils and the [GCC](#) packages from <http://ftp.gnu.org/gnu/>. Download them to `/usr/src` (or wherever you think appropriate), and unpack them. (Finder=>Gehe zu...=>Gehe zum Ordner oder Shift+Command+G)

Note: The versioning scheme used is that each fullstop separates a full number, eg. 2.8.0 2.9.0 2.10.0 2.11.0, this may be confusing if you're used to Windows' program's schemes or even just basic math (eg. 1.01 1.02 etc).

Preparation

(Ich empfehle, diese Vorgänge als „root“ durchzuführen. Das geht ganz einfach mit „`sudo -i`“ und einer nachfolgenden Passworteingabe)

```
export PREFIX=/usr/cross
export TARGET=i586-elf
cd /usr/src
mkdir build-binutils build-gcc
```

The prefix will configure the build process so that all the files of your cross-compiler environment end up in `/usr/cross`, without disturbing your "normal" compiler setup. This tutorial has been shown to work in the same way for the `x86_64-elf` target, for building 64 bit executables (GCC 4.3.x). In order to do this, simply export TARGET as `x86_64-elf` instead of `i586-elf`.

binutils

```
cd /usr/src/build-binutils
../binutils-x.xx/configure --target=$TARGET --prefix=$PREFIX --
disable-nls --disable-werror
make all
make install
```

This compiles the binutils (assembler, disassembler, and various other useful stuff), runnable on your system but handling code in the format specified by \$TARGET.

--disable-nls tells binutils not to include native language support. This is basically optional, but reduces dependencies and compile time. It will also result in English-language diagnostics, which the people on the [Forum](#) understand when you ask your questions. ;-)

gcc

Now, you can build [GCC](#). (Use v3.3 or later - v3.2.x has a bug with internal `__malloc` declarations resulting in an error during compilation. This could be fixed by patching four occurrences in three different source files, but I lost the diff output and am not in a mind of re-checking. ;-)

```
cd /usr/src/build-gcc
export PATH=$PATH:$PREFIX/bin
../gcc-x.x.x/configure --target=$TARGET --prefix=$PREFIX --
disable-nls --enable-languages=c,c++ --without-headers --disable-
werror
make all-gcc
make install-gcc
```

(bitte nicht nur „make“ oder „make all“ eingeben!!)

The path has to be extended since [GCC](#) needs the binutils we built earlier at some point of the build process. You might want to add these extensions to your \$PATH permanently, so you won't have to use fully qualified path names every time you call your cross-compiler.

--disable-nls is the same as for binutils above.

--without-headers tells [GCC](#) not to rely on any C library (standard or runtime) being present for the target.

--enable-languages tells [GCC](#) not to compile all the other language frontends it supports, but only C (and optionally C++).

If you are compiling [GCC](#) <= 3.3.x, you need **--with-newlib** as additional parameter. Those older versions have [a known bug](#) that keeps `--without-headers` from working correctly. Additionally setting `--with-newlib` is a workaround for that bug.

Summary

Now you have a "naked" cross-compiler. It does not have access to a C library or C runtime yet, so you cannot use any of the standard includes or create runnable binaries. But it is quite sufficient to compile your self-made kernel.

Usage

Once you are finished, your toolset resides in `/usr/cross`. For example, you have a `gcc` executable in `/usr/cross/bin/$TARGET-gcc` (and `/usr/cross/$TARGET/gcc` as well), which spits out binaries for your `TARGET`. Add `/usr/cross/bin` to your `PATH` environment variable, so that `gcc` invokes your system compiler, and `$TARGET-gcc` invokes your cross-compiler.

You could also use the `-b` and `-V` options of GCC. Let's assume you have a `gcc 3.3.3` as system compiler (`/usr/bin/gcc`), and you just created a `gcc 3.4.3` cross-compiler for `i586-elf` (`/usr/cross/bin/i586-elf-gcc`). Instead of calling `i586-elf-gcc`, you could also call `gcc -b i586-elf -V 3.4.3...`

This sounds strange at first, but it is rather simple: These options, passed to your system compiler, tell it that it's not really the system compiler you want - but rather the one for the target passed by the `-b $TARGET` option, and in the version passed by the `-V $VERSION` option. The system compiler turns these two options into a new executable name (namely `gcc-$TARGET-$VERSION`), and calls that one instead of compiling the source itself. Neat, huh? ;-)

libgcc

As a useful addition (wirklich dringend empfohlen, geht ganz schnell), you may also like to build `libgcc`, the GCC low-level runtime library. Linking against `libgcc` provides integer, floating point, decimal, stack unwinding (useful for exception handling) and other support functions. Once you have built and installed the GCC Cross-Compiler, keep your Bash window open and type:

```
make all-target-libgcc
make install-target-libgcc
```

Ab hier ist die Installation komplett und PrettyOS kann erstellt werden.

- This page was last modified 18:57, 1 February 2010.
- This page has been accessed 48,118 times.
- [Privacy policy](#)
- [About OSDev Wiki](#)
- [Disclaimers](#)