MXE (M cross environment)

eine Ausarbeitung von Thorsten Kattanek

Berlin, 29.05.2022

Inhaltsverzeichnis

Was ist MXE und wofür braucht man es?

MXE ist ein GNU-Makefile welches einen Cross-Compiler kompiliert und damit dann viele OpenSource Bibliotheken wie z.B. SLD und Qt kompiliert. Kurz gesagt: Ihr könnt mit MXE, Windows Binaries unter UNIX bzw. Linux erstellen ohne selber Windows zu benutzen.

MXE ist so ausgelegt das es auf jedem UNIX System läuft und auch angepasst werden kann. Es baut zusätzlich zum Cross-Compiler viele freie Bibliotheken, lädt die benötigten Pakete selbständig herunter und überprüft diese mit einer eindeutigen Prüfsumme. MXE kann auch die Versionsnummern automatisch aktualisieren und verwendet dafür direkt die jeweiligen Quellpakete. Somit ist Sichergestellt das der gesamte Build Prozess transparent ist.

Um wiederholte Builds zu beschleunigen lässt sich MXE gut in autotools, cmake, qmake und in Handgeschrieben Makefiles integrieren. Ich selber benutze dafür lieber ein Bash Skript da ich die Windows Version maximal zum Release erstelle.

Die Geschichte von MXE

Hier ein kleiner Überblick über die Entwicklung des MXE Projekts. Das Projekt wurde am 12. Juni 2007 durch Volker Grabsch unter dem Namen mingw-cross-env (MinGW cross compiling environment) gestartet. Bereits 7 Tage später wurde das 1. Release veröffentlicht. Dieses Release unterstützte SLD 1.2 und alles was dazugehört. Hier kam noch der GCC-3 zum Einsatz.

Im Dezember wurde dann auf GCC-4 umgestellt und es kamen weitere Bibliotheken dazu die unterstützt wurden. Im Januar 2018 kam dann ein Tutorial hinzu. Ab der Version 2.4 die im März 2009 erschien, kamen wichtige Bibliotheken hinzu wie wxWidgets und GTK+. Im Oktober 2009 kam eine weitere wichtige Bibliothek dazu und zwar die Qt Bibliotheken.

Im laufe der Zeit kamen immer weitere Bibliotheken hinzu. Die folgenden Release beschäftigen sich hauptsächlich damit neue Bibliotheken hinzuzufügen und diese aktuell zu halten. Außerdem wird das Buildsystem immer weiter verbessert.

Am 12.04.2012 wurde dann das Projekt in MXE (M cross environment) umbenannt. Im Juli 2012 wurde die Unterstützung für 32 und 64 Bit hinzugefügt, auch wird jetzt Qt5 Supported. Ab Mai 2015 wurde der Stable Zweig entfernt und es wird nur noch der master Zeig verwendet. Und dann wurde noch von MinGW zu MinGW-w64 gewechselt zu Gunsten für Qt5 und GLib welche wohl 2 der wichtigsten Pakete von MXE darstellen.

Bis heute wurde das MXE Projekt von vielen Contributors unterstützt. Stand 26.03.2022 sind es 188 an der Zahl.

Referenzen

MXE wird von vielen Projekten benutzt um ihre Windows Binaries zu erstellen. Hier seien nur Beispielhaft einige genannt.

- Krita https://krita.org
- digiKam https://www.digikam.org
- OpenSCAD https://www.openscad.org
- ScummVM https://github.com/scummvm/scummvm
- RetroArch https://www.retroarch.com
- Warzone 2100 https://wz2100.net
- Emu64 https://github.com/ThKattanek/emu64

und weitere kann man nachlesen unter: https://mxe.cc/#used-by

Betriebsysteme

MXE läuft auf jedes UNIX/Linux System z.B Debian, Fedora, FreeBSD, Gentoo, ... um nur einige zu nennen. Als Zielplattform kann zwischen 32-Bit Windows und 64-Bit Windows gewählt werden, wobei jedes nochmal als Static oder Shared gebaut werden kann.

Unterstützte Bibliotheken

Zum jetzigen Zeitpunkt unterstützt MXE 482 Bibliotheken. Das sind einfach zu viele um alle hier aufzuzählen. Deshalb hier nur ein kleiner Ausschnitt an Bibliotheken:

- Qt4 + Qt5
- SDL + SDL2
- OpenGL
- SFML
- wxWidgets
- GTK+
- ffmpeg

Am besten ihr schaut selber mal nach, ob die Bibliotheken dabei sind die ihr so in euren Projekten benutzt. Die gesamte Liste findet man unter https://mxe.cc/#packages

Einrichtung

An einem Beispiel zeige ich euch wie ihr MXE auf euer System installiert. Ich benutze dafür hier Kubuntu also Ubuntu mit KDE als Desktop.

Vorraussetzungen

Um mit MXE zu starten benötigt es folgende Komponenten die Ihr auf euer System installiert haben müsst. Und das sind folgende:

- Autoconf ≥ 2.68
- Automake ≥ 1.11.3
- Bash
- Bison
- Bzip2
- Flex ≥ 2.5.31
- GCC (gcc, g++)
- gdk-pixbuf
- Git ≥ 1.7
- GNU Coreutils
- GNU Gettext
- GNU gperf
- GNU Make ≥ 3.81
- GNU Sed
- GNU Tar
- Intltool ≥ 0.40
- LibC for 32-bit
- Libtool ≥ 2.2
- Lzip
- OpenSSL-dev ≥ 1.01
- p7zip (7-Zip)
- Patch
- Perl
- Perl XML::Parser
- Python
- Ruby
- UnZip
- Wget
- XZ Utils
- zlib

Ab jetzt muss alles in der Konsole / Terminal ausführen.

Unter Debian / Ubuntu reicht dann folgender Aufruf:

```
sudo apt-get install autoconf automake autopoint bash bison bzip2 flex g++ g++-multilib gettext git gperf intltool libc6-dev-i386 libgdk-pixbuf2.0-dev libltdl-dev libssl-dev libtool-bin libxml-parser-perl lzip make openssl p7zip-full patch perl python ruby sed unzip wget xz-utils
```

Wer ein anderes Systeme benutzt kann hier nachschauen wie ihr eure Pakete installieren müsst.
https://mxe.cc/#requirements

Jetzt können wir MXE von GitHub klonen. Dazu in das Verzeichnis gehen in das MXE erzeugt werden soll, es wird beim klonen ein neues Verzeichnis mit dem Name „mxe“ angelegt.

```
git clone https://github.com/mxe/mxe.git
cd mxe
```

Cross-Compiler und Bibliotheken erzeugen

Weg 1 -- Manuell --

Jetzt könnt ihr anfangen alles zu erstellen was ihr für eure Projekte benötigt. Wir fangen mal mit dem wichtigsten an, dem Cross-Compiler.

```
make cc --jobs=4 JOBS=3
```

cc ist das Paket was erstellt werden soll, zudem werden auch automatisch alle Abhängigkeiten erstellt. --jobs gibt an wie viele Pakete Gleichzeitig gebaut werden sollen hier sind es 4 Pakete. JOBS gibt an wie viele Compiler Prozesse für jedes Paket genutzt werden sollen. Dieser Aufruf erzeugt aber lediglich die 32-Bit Variante als Static (i686-w64-mingw32.static).

Möchten wir für ein anderes Target die Bibliotheken erzeugen können wir die Variable "MXE_TARGETS" dem makefile übergeben. Das sieht dann so aus:

```
make cc --jobs=4 JOBS=3 MXE_TARGETS=x86_64-w64-mingw32.static
```

Für mehrere gleichzeitig geht dann sowas hier:

```
make cc --jobs=4 JOBS=3 MXE_TARGETS='i686-w64-mingw32.shared x86_64-w64-mingw32.shared'
```

folgende Targets stehen zur Verfügung:

- i686-w64-mingw32.static
- i686-w64-mingw32.shared
- x86_64-w64-mingw32.static
- x86_64-w64-mingw32.shared

Um jetzt eine beliebige Bibliothek zu erzeugen gehen wir geauso vor wie mit dem Cross-Compiler. In diesem Beispiel wird SLD und SFML erzeugt als 32-Bit Static. Wie man sieht kann man mehrere Pakete angeben die dann alle erzeugt werden.

```
make sdl smfl --jobs=4 JOBS=3
```

Möchte man alle Pakete erzeugen reicht dieser Aufruf:

```
make --jobs=4 JOBS=3
```

Kostet viel Zeit und ist eigtl. unnötig. Das wäre jetzt der eine Weg wie ihr euren MXE Cross-Compiler incl. Bibliotheken bauen könnt.

Weg 2 per "settings.mk"

Dieser weg benötigt eine Datei die ihr ins mxe Verzeichnis ablegen müsst. Diese muss "settings.mk" heißen. Die Datei sollte wie folgt aufgebaut sein.

```
JOBS := 3
MXE_TARGETS := i686-w64-mingw32.static x86_64-w64-mingw32.static
LOCAL_PKG_LIST := sdl2 sdl2_gfx sdl2_image sdl2_mixer sdl2_net sdl2_ttf
.DEFAULT_GOAL  := local-pkg-list
local-pkg-list: $(LOCAL_PKG_LIST)
```

Dann reicht es wenn ihr nur noch make -j4 eingebt und ihr könnt auch enspannt zurücklehen und warten bis alles fertig erstellt wurde. Dieser Weg macht es leichter wenn ihr eurer Cross Build hin und wieder neu aufsetzten wollt.

Ich habe meine settings.mk an einem anderen Ort gespeichert und kann so schnell alles wieder neubauen und habe dann immer gleich alle meine Bibliotheken die ich benötige.

Es ist aber euch überlassen wie ihr eure Bibliotheken bauen lasst. Man kann auch beide Wege mischen. Alles kein Problem.

Updaten

Um alle Pakete auf den neusten Stand zu bringen reicht folgender aufruf.

```
make update
```

Möchte man nur eine Liste aller Updates ruft man folgendes auf.

```
make update UPDATE_DRYRUN=true
```

Bereinigen

Wenn man alle Libaries erzeugt hat kann man alles etwas entschlacken und unötigen Ballast abwerfen. Das macht man mit

```
make clean-junk
```

Hiermit werden alle nicht nenötigten Dateien entfernt, sowie alle unbenutzten Pakete, teporäre Verzeichnisse und alle erzeugten Logs.

Vorsicht! make clean löscht auch das usr Verzeichnis und löscht damit auch alle erstellten Bibiotheken.

Benutzung

Wichtig ist noch, das vor jeder Benutzung von mxe immer der Pfad zu mxe dem System bekannt gemacht wird. Endweder Manuel wie hier einmal zu beginn in der Konsole. Wird aber beim beenden dieser nicht gehalten und muss immer wieder gemacht werden.

```
export PATH=~/mxe/usr/bin:$PATH
```

Oder man bindet ihn fest ins System mit ein. Unter Debian / Ubuntu legt man dazu unter /etc/profile.d eine neue Datei mit dem Namen mxe_env.sh (könnt ihr auch nennen wie ihr wollt nur .sh muss am Ende stehen) ein und schreibt den oberen Passus darein. Nach einem Neustart könnt ihr mit echo $PATH prüfen ob der mxe Pfad jetzt global definiert ist.

Ein "cmake Projekt"

Dafür legen wir uns einfach ein kleines Projekt an um das cmake build zu testen. Dazu ertsmal ein leeres Verzeichnis anlegen:

```
mkdir hello_world_cmake
cd hello_world_cmake
nano main.cpp
```

Hier ein kleines „Hello World“ Programm welches als Test dienen soll:

```
#include <iostream>

using namespace std;

int main()
{
        cout << "Hello World!" << endl;
        return 0;
}
nano CMakeLists.txt
```

```
cmake_minimum_required(VERSION 2.8)

project(hello_world)

set(SOURCE main.cpp)
add_executable(hello_world ${SOURCE})
```

Native Variante

```
mkdir build_native
cd build_nativecd
cmake ..
make
file ./hello_world
./hello_world
```

Windows Cross Compile Variante mit MXE

```
mkdir build_win
i686-w64-mingw32.static-cmake ..
make
file ./hello_world.exe
wine hello_world.exe
```

Ein "qmake Projekt"

Dafür legen wir uns einfach ein kleines Projekt an um das cmake build zu testen. Dazu erstellen wir erst einmal ein leeres Verzeichnis und eine Datei main.cpp

```
mkdir hello_world_qmake
cd hello_world_qmake
nano main.cpp
```

In die main.cpp kommt dieses kleine „Hello World“ Programm

```
#include <iostream>

using namespace std;

int main()
{
        cout << "Hello World!" << endl;
        return 0;
}
```

Dann erstellen wir uns noch die „hello_world.pro“ unsere Qt Projektdatei.

```
TARGET = hello_world
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += \
        main.cpp
```

Dann kann auch schon qmake von mxe aufgerufen werden und anschließend make:

```
i686-w64-mingw32.static-qmake-qt5 ..
make
```

Probleme unter Ubuntu 22.04

Kommt es zu folgender Fehlermeldung:

```
python --version
bash: Zeile 1: python: Befehl nicht gefunden
```

Aber Python3 ist installiert, dann hilft folgendes. Einfach einen Links setzen !

```
sudo ln /usr/bin/python3 /usr/bin/python
```

Quellen

http://mxe.cc