# Cross Compilation

This has become a necessary thing to do given the environment most people are working nowadays. I think the rise of embedded systems started off the trend, where the target system is usually very slow compared to the desktop systems and somebody started to wonder, “Wouldn't it be nice if I can compile applications for my embedded ARM processor on my powerful 3.0GHz Intel dual-core processor?”. The rest, as they say it, is history. Other examples where cross compilation is handy:

- compiling x86_64 Linux kernel, system utilities and applications on 32-bit machines (and vice-versa)
- compiling Windows application on Linux (Yes! You can now throw away your WindowsXP OEM installer!)

At the moment (2010), I'm interested in rewriting my old C++Builder applications using wxWidgets libraries so that I get cross-platform application and I don't even have to have Window$XP to be able to write & compile a program for my friends who are stuck with M$Window$ (Yeah! I know... I pity them too!  ) However, most of them just want the executable (without having to recompile), so I need to build a cross compiler based on MinGW as well. There's a build script available on MinGW website to build such tool but I prefer to build on my own (so what else is new?  ). Before I get to that, I've included here my previous effort on building a cross compiler.

# GNU ARM Cross Compiler on Windows

It's a pill! A pain in the neck! Let's face it, M$Window$ is not the best platform for hobbyists' embedded system development. I'm currently working on a Gumstix-based robot controller and I used buildroot to build all my cross compiler and related filesystems. However, the project requires a Windows-based development environment to be available as well. It seems that nobody realized that our existing Windows toolchain is using different kernel headers, different uClibc version and different gcc version compared to the one built using gumstix-buildroot!

So, I set out on a quest to rebuild the Windows toolchain, based on information from the guy who built the previous one. It is a bit inaccurate (some details are incorrect), and it took me almost 3 full days to figure out what he's trying to say. So, here's my note on that.

First, the requirements - we need the Cygwin environment to build the toolchain on Windows. It's also theoretically possible to build using MinGW but at the time of writing, I only found a guide on how to build a newlib-based MinGW toolchain instead of uclibc. Besides, I think MinGW sources are patched for Win32. Bottom line, I opted to install Cygwin environment with all the common development packages (gcc, bison, flex, make, etc.).

To make life a lot easier, I did not exactly start from scratch. I used the libraries (patched & built) from my gumstix-buildroot's staging_dir folder and the binutils & gcc sources (patched) from my gumstix-buildroot's toolchain_build_arm_nofpu folder. The buildroot path on my Linux machine is /home/share/m6build - I read somewhere that for gcc version 3.x, there are some hardcoded paths involved. So, when I extracted staging_dir in my Cygwin environment, I created the same path -

/home/share/m6build/build_arm_nofpu/staging_dir. In there, we have uClibc 0.9.28 and 2.6.17 kernel headers. The binutils version is 2.17 and the gcc version is 3.4.5 - I copied the patched sources because when using the original source, I keep failing gcc compilation (when checking for main in -lm, I got the error message: “Test links …. after GCC_NO_EXECUTABLES…” something like that).

So, with all the copied files extracted accordingly, set the following variables:

```
STAGEDIR=/home/share/m6build/build_arm_nofpu/staging_dir
XTOOLDIR=/home/share/m6xtool
TARGET=arm-xscale-linux-uclibc
```

Build binutils (use a separate build directory) with:

```
../binutils-2.17/configure --prefix=$XTOOLDIR --target=$TARGET \
  --with-cpu=arm5vte --with-arch=arm5vte --with-tune=xscale \
  --with-float=soft --disable-nls --enable-interwork \
  --disable-multilib --with-headers=$STAGEDIR/include \
  --with-libs=$STAGEDIR/lib --with-sysroot=$STAGEDIR
make
make install
```

And, build gcc (use a separate build directory) with:

```
../gcc-3.4.5/configure --prefix=$XTOOLDIR --target=$TARGET \
  --with-arch=arm5vte --with-tune=xscale --with-float=soft \
  --disable-nls --enable-interwork --disable-multilib \
  --enable-threads=posix --disable-__cxa_atexit \
  --enable-languages=c,c++ --with-headers=$STAGEDIR/include \
  --with-libs=$STAGEDIR/lib --with-sysroot=$STAGEDIR
make all
make install
```

And, voila! We got a Windows-based cross compiler. The programs created this way requires Cygwin dll's to work. So, I copied cygwin1.dll and cygiconv-2.dll from the cygwin setup to the cross compiler folder. The dll's need to be in one of the specified %path% to make it work. I am using MinGW's make program instead of Cygwin's because the latter is using Cygwin paths (/cygdrive/c/test) instead of Window$ path (c:\test).

***NOTE*** The cross toolchain is based on uclibc! C++ libraries (i.e. iostream, etc.) doesn't work - although we can still compile C++ codes (basic constructs). I haven't actually tested this - it's just based on my theory.

# MinGW Cross Compiler on Linux

MinGW is **Min**imalist **G**NU for **W**indows. It was originally created using a very early version of Cygwin suite, and these two are currently the most widely used Win32 build system based on GNU utilities. I have successfully built this on my Slackware 13.1. Prior to that I did try to build everything from scratch, getting the latest MinGW packages. It turns out that newer MinGW tools has lots of extra

packages compared to the previous one (I think it's gcc-4 thingy). That build seems to be too much for me at the moment - so, I settled for the old MinGW packages based on gcc-3. I repackaged them for an old Win32 project that I was working on. The versions used are:

- mingw-runtime-3.13
- win32api-3.10
- binutils-2.16.91-20060119-1
- gcc-3.4.2-20040916-1

I changed my plan and tried to build using similar technique to what I have done before (with ARM cross compiler for Window$). Notice that I used the pre-built uClibc and kernel headers (instead of building them from source) - I figured I can do the same by taking the binary packages for mingw-runtime (the library) and win32api (the system). So, I need to just get the sources for binutils and gcc. Unfortunately, I can no longer find the exact version mingw-runtime and win32api binaries at MinGW website. So I searched for them and found this.

First thing to do is setup the build environment:

```
XTOOLDIR=/home/ftp/software/mingw-tool
TARGET=i686-pc-mingw32
```

I extracted mingw-runtime and win32api to $XTOOLDIR. This is also where the built tools will be installed. The name for target triplets can be found from the internet. Alternatively, it can be `i386-pc-mingw` or something like that... I think. The one I used works for me. Then I can start building `binutils`. On that note, I forgot to mention in the previous section about this 'bug' in binutils. Apparently, there's this version checker in binutils that thinks makeinfo version > 4.9 as 'older' than 4.4 (the minimum build requirement). One solution is, if you're sure you have makeinfo that is at least (or newer than) version 4.4, to edit the created Makefile in binutils build path directly and set MAKEINFO to /path/to/makeinfo! As mentioned previously, it is advised to build binutils in a separate build folder.

```
CFLAGS="-O1" CXXFLAGS="-O1" ../binutils-2.16.91-20060119-1/configure --
prefix=$XTOOLDIR --target=$TARGET \
    --disable-nls --disable-interwork --disable-multilib \
    --with-headers=$XTOOLDIR/include --with-libs=$XTOOLDIR/lib --with-
sysroot=$XTOOLDIR
# edit Makefile for makeinfo bug!
CFLAGS="-O1" CXXFLAGS="-O1" make
CFLAGS="-O1" CXXFLAGS="-O1" make install
```

This one I just found out while building `binutils` - I also need to change the configure line as shown above (notice the two environment variables). Optimization options at `-O2` will give out array bounds error! I have no idea why that is the case, but as long as this works, eh? 😮 Next is building gcc - remember to use a separate build directory. I don't want to include MinGW in my standard path, so I have to include the path at command line.

```
PATH=$XTOOLDIR/bin:$PATH ../gcc-3.4.2-20040916-1/configure --
prefix=$XTOOLDIR --target=$TARGET \
    --disable-nls --enable-interwork --disable-multilib \
    --enable-threads --enable-languages=c,c++ --with-
```

```
headers=$XTOOLDIR/include \
    --with-libs=$XTOOLDIR/lib --with-sysroot=$XTOOLDIR
# create dummy folder to satisfy fixinc.sh
mkdir -pv $XTOOLDIR/usr/include
PATH=$XTOOLDIR/bin:$PATH make
PATH=$XTOOLDIR/bin:$PATH make install
```

And, there you go... a MinGW-based Win32 compiler executable on Linux!

## Testing the Cross Compiler

Here's a simple code you can use to test the cross compiler:

xtest.c

```
#include <windows.h>

int main(int argc, char *argv[])
{
    MessageBox(NULL, "I LOVE KIKI!", "Major Declaration", MB_OK);
    return 0;
}
```

And, a makefile - just type make! And, of course, you have to modify the paths accordingly.

Makefile

```
# makefile for testing cross-mingw-toolchain

XTOOL_DIR        = /home/ftp/software/mingw-tool
XTOOL_TARGET       = $(XTOOL_DIR)
CROSS_COMPILE       = $(XTOOL_TARGET)/bin/i686-pc-mingw32-

OLIBS = xtest.o
TPROG = xtest.exe

TARGET_ARCH = -Wall --static
CC = $(CROSS_COMPILE)gcc-3.4.2

CFLAGS += -I$(XTOOL_DIR)/include
CFLAGS += $(TARGET_ARCH)
LDFLAGS += -L$(XTOOL_DIR)/lib
# below is to remove console at runtime
LDFLAGS += -Wl,-subsystem,windows
OFLAGS +=

DELETE = rm -rf
```

```
all: $(TPROG)

%.o: %.c %.h
    $(CC) $(CFLAGS) -c $<

%.o: %.c
    $(CC) $(CFLAGS) -c $<

%: %.c
    $(CC) $(CFLAGS) -o $@ $(LDFLAGS) $(OFLAGS)

$(TPROG): $(OLIBS)
    $(CC) $(CFLAGS) -o $@ $+ $(LDFLAGS) $(OFLAGS)

clean:
    -$(DELETE) $(TPROG) $(OLIBS)
```

## Adding wxWidgets Library

Downloaded wxwidgets source (got the same version I have installed on Slackware wxGTK-2.8.11).

```
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH ./configure --prefix=$XTOOLDIR \
  --host=i686-pc-mingw32 --build=i686-slackware-linux --disable-unicode --
disable-shared
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH make
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH make install
```

The installed wx libraries has the `.dll.a` extension that makes them unreadable. I used a simple script to create suitably-named links:

```
for a in *.dll.a; do ln -sf $a ${a//.dll.a/.a}; done
```

**Update** Use –`disable-shared` to avoid dll dependencies. So, I do not have to do the above.

MinGW dependencies appear because wxWidgets' config added `-mthreads` and according to gcc: "-mthreads: Support thread-safe exception handling on Mingw32. Code that relies on thread-safe exception handling must compile and link all code with the -mthreads option. When compiling, -mthreads defines -D_MT; when linking, it links in a special thread helper library -lmingwthrd which cleans up per thread exception handling data." So, in my makefile I use sed to filter it out of wx-config's output.

**Update20101018** I was trying to get wxWidgets-2.9.1 because of some deprecating functions in 2.8.11 However, I found out the hard way that my mingw-gcc (version <4.0) doesn't support the tr1 c++ library which is required to compile 2.9.1! Tough luck... still deciding whether to stick with the old wxGTK!

# MinGW Cross Compiler on Linux (Reloaded)

I've had some problems building a gcc-4.x-based MinGW Cross Compiler before and I was just thinking I want to give it another try. When I really needed wxWidgets-2.9, I started out using the same method I tried last time. First, I refer here for the suitable versions to create a working MinGW toolchain. Get all binaries (dev/dll) except for gcc and binutils (these... we need to build) - the list are as follows:

```
binutils-2.21-3-mingw32-src.tar.lzma
gcc-4.5.2-1-mingw32-src.tar.lzma
gmp-5.0.1-1-mingw32-dev.tar.lzma
libgcc-4.5.2-1-mingw32-dll-1.tar.lzma
libgmp-5.0.1-1-mingw32-dll-10.tar.lzma
libgomp-4.5.2-1-mingw32-dll-1.tar.lzma
libiconv-1.13.1-1-mingw32-dev.tar.lzma
libiconv-1.13.1-1-mingw32-dll-2.tar.lzma
libintl-0.17-1-mingw32-dll-8.tar.lzma
libmpc-0.8.1-1-mingw32-dll-2.tar.lzma
libmpfr-2.4.1-1-mingw32-dll-1.tar.lzma
libpthread-2.8.0-3-mingw32-dll-2.tar.lzma
libssp-4.5.2-1-mingw32-dll-0.tar.lzma
libstdc++-4.5.2-1-mingw32-dll-6.tar.lzma
mpc-0.8.1-1-mingw32-dev.tar.lzma
mpfr-2.4.1-1-mingw32-dev.tar.lzma
pthreads-w32-2.8.0-3-mingw32-dev.tar.lzma
w32api-3.17-2-mingw32-dev.tar.lzma
mingwrt-3.18-mingw32-dev.tar.gz
mingwrt-3.18-mingw32-dll.tar.gz
```

Extract all binaries to a staging location like `/home/share/tool/mingw-staging`. Get another place to extract the source files - I use `$HOME/temp/mingw-build`. With that, I then set some useful environment variables - to make life easier when testing the *configure && make && make install* ~~a few~~ many times.

```
export XTOOLDIR=/home/share/tool/mingw
export STAGEDIR=/home/share/tool/mingw-staging
export TARGET=i686-pc-mingw32
export BUILDHOST=i686-slackware-linux
```

So, the final cross compiler will be in `$XTOOLDIR` - but we have to copy everything in `$STAGEDIR` (except `$STAGEDIR/mingw`, which is created per request as shown later) into `$XTOOLDIR` to get the complete toolchain (a few files will be duplicated but **SKIP** those). Go to the build location and start building binutils. It is recommended to build both binutils and gcc in a separate location, as show below.

```
cd $HOME/temp/mingw-build
# binutils package should already be extracted here
mkdir binutils-build
cd binutils-build
```

```
../binutils-2.21/configure --prefix=$XTOOLDIR --build=$BUILDHOST --
host=$BUILDHOST \
    --target=$TARGET --disable-nls --disable-interwork --disable-multilib \
    --with-headers=$STAGEDIR/include --with-libs=$STAGEDIR/lib --with-
sysroot=$STAGEDIR
make
make install
```

Obviously, the above shown code is the working one - which required a few tries before getting there. I needed to specify –`build` and –`host` because I'm on chroot installation on 64-bit machine, and the compiler detects the kernel rather than binaries. Everything should go well, and then we proceed to building gcc4.

```
cd $HOME/temp/mingw-build
# gcc package should already be extracted here
mkdir gcc-build
cd gcc-build
export PATH=$XTOOLDIR/bin:$PATH
../gcc-4.5.2/configure --prefix=$XTOOLDIR --build=$BUILDHOST --
host=$BUILDHOST \
    --target=$TARGET --disable-nls --enable-interwork --disable-multilib \
    --enable-threads --enable-languages=c,c++ --with-
headers=$STAGEDIR/include \
    --with-libs=$STAGEDIR/lib --with-sysroot=$STAGEDIR
# create dummy folder to satisfy fixinc.sh
( cd $STAGEDIR; mkdir mingw; cd mingw; ln -sf ../include include )
make
make install
```

Finally, when everything is completed, copy all the contents of staging ($STAGEDIR) to the final path ($XTOOLDIR) as mentioned previously. Remember to skip $STAGEDIR/mingw (including it doesn't actually make any difference!).

## Testing the Cross Compiler

*use the codes above...*

## Adding wxWidgets Library

I've aso included wxWidgets library in the MinGW toolchain - I used wxWidgets-2.9.2 for this (the reason for this initiative in the first place).

```
cd $HOME/temp/wxWidgets-2.9
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH ./configure --prefix=$XTOOLDIR \
    --build=$BUILDHOST --host=$TARGET --disable-unicode --disable-shared
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
```

```
PATH=$XTOOLDIR/bin:$PATH make
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH make install
```

Now I have that shiny wxWidgets-2.9-based MinGW32 cross-compiler on my pure 64-bit Linux machine! 😎

# Build Script

Having created `MinGW cross compiler for Linux` more than once naturally prompted me to create a build script.

_build.sh

```
#!/bin/bash

TARGET_PATH=$1
[[ "$TARGET_PATH" == "" ]] && TARGET_PATH="/home/share/tool"
[[ ! -d "$TARGET_PATH" ]] && mkdir -p $TARGET_PATH
BUILD_PATH=$TARGET_PATH/mingw-build
STAGE_PATH=$TARGET_PATH/mingw-staging
THIS_PATH=$(pwd)

# setup environment
export XTOOLDIR=$TARGET_PATH/mingw
export STAGEDIR=$STAGE_PATH
export TARGET=i686-pc-mingw32
export BUILDHOST=i686-slackware-linux

# staging should not be existing!
[[ -d "$STAGE_PATH" ]] && echo "$STAGE_PATH exists! Check path!" &&
exit 1

# create build and staging directories
mkdir -p $BUILD_PATH $STAGE_PATH

# copy the two source files into build
for src in *-mingw32-src.tar.lzma ; do cp $src $BUILD_PATH ; done

# copy the rest into staging
for pkg in *.lzma *.gz ; do [[ "${pkg//-mingw32-src/}" != $pkg ]] &&
continue ; cp $pkg $STAGE_PATH; done

# extract the files in staging
cd $STAGE_PATH ; for pkg in *.lzma *.gz ; do tar xf $pkg ; rm $pkg ;
done
```

```
# build binutils
cd $BUILD_PATH
tar xf binutils-2.21-3-mingw32-src.tar.lzma
# binutils package should already be extracted here
mkdir binutils-build
cd binutils-build
../binutils-2.21/configure --prefix=$XTOOLDIR --build=$BUILDHOST --
host=$BUILDHOST \
    --target=$TARGET --disable-nls --disable-interwork --disable-
multilib \
    --with-headers=$STAGEDIR/include --with-libs=$STAGEDIR/lib --with-
sysroot=$STAGEDIR
make
make install

# build gcc
cd $BUILD_PATH
tar xf gcc-4.5.2-1-mingw32-src.tar.lzma
tar xf gcc-4.5.2.tar.bz2
# gcc package should already be extracted here
mkdir gcc-build
cd gcc-build
export PATH=$XTOOLDIR/bin:$PATH
../gcc-4.5.2/configure --prefix=$XTOOLDIR --build=$BUILDHOST --
host=$BUILDHOST \
    --target=$TARGET --disable-nls --enable-interwork --disable-
multilib \
    --enable-threads --enable-languages=c,c++ --with-
headers=$STAGEDIR/include \
    --with-libs=$STAGEDIR/lib --with-sysroot=$STAGEDIR
# create dummy folder to satisfy fixinc.sh
( cd $STAGEDIR; mkdir mingw; cd mingw; ln -sf ../include include )
make
make install

cd $THIS_PATH

# build wxwidgets if available
WXWIDGETS_NAME="wxWidgets"
WXWIDGETS_VERS="2.9.4"
WXWIDGETS_FULL="${WXWIDGETS_NAME}-${WXWIDGETS_VERS}"
if [[ -f "${THIS_PATH}/${WXWIDGETS_FULL}.tar.bz2" ]] ; then
echo "Building ${WXWIDGETS_FULL}..."
cd $BUILD_PATH
tar xf ${THIS_PATH}/${WXWIDGETS_FULL}.tar.bz2
cd ${WXWIDGETS_FULL}
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH ./configure --prefix=$XTOOLDIR \
    --build=$BUILDHOST --host=$TARGET --disable-unicode --disable-
shared
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
```

```
PATH=$XTOOLDIR/bin:$PATH make
CFLAGS="-I$XTOOLDIR/include" CXXFLAGS="-I$XTOOLDIR/include"
PATH=$XTOOLDIR/bin:$PATH make install
echo "Done."
fi
```

Notice that the script still requires the source/binary tarballs to be downloaded manually. Refer above on which files are needed.

# MinGW Cross Compiler on Devuan

*LastUpdate:20210522*

I need to recompile my1sim85 on Devuan. Since `mingw` status is still a suspect, I decided to go with mingw-w64 for this because that is already in Devuan repo. Simply do

```
# apt install mingw-w64
```

and we have a 64-bit cross compiler available.

To compile wxWidgets library, I use a modified build script

wxWidgets.build

```
#!/bin/bash

TOOL_PATH=$1
[ -z "$TOOL_PATH" ] && TOOL_PATH="/home/share/tool"
[ ! -d "$TOOL_PATH" ] && mkdir -p $TOOL_PATH
THIS_PATH=$(pwd)

export XTOOLDIR=$TOOL_PATH/xtool-mingw
export TARGET=i686-w64-mingw32
export BUILDHOST=i686-devuan-linux

THAT_NAME="wxWidgets"
THAT_VERS="3.0.5"
THAT_FULL="$THAT_NAME-$THAT_VERS"
THAT_EXTN="tar.bz2"
THAT_BALL="$THAT_FULL.$THAT_EXTN"
THAT_FILE="$THIS_PATH/$THAT_BALL"
[ ! -f "$THAT_FILE" ] && echo "** Tarball $THAT_BALL not found!" &&
exit 0

TEMP_PATH=$THIS_PATH/temp_$(echo $THAT_NAME|tr '[A-Z]' '[a-z]')
echo "-- Building ${THAT_FULL}..."
[ ! -d "$TEMP_PATH" ] && mkdir -p $TEMP_PATH
```

```
cd $TEMP_PATH
tar xf $THAT_FILE
cd $THAT_FULL
./configure --prefix=$XTOOLDIR \
  --build=$BUILDHOST --host=$TARGET --disable-unicode --disable-shared
make
make install
echo "Done."
cd $THIS_PATH
```

The above script expects wxWidgets tarball to be available in current path and, by default, installs the wxWidgets library in `/home/share/tool/xtool-mingw`.