Gcc-2.95.3 m68k-elf for uClinux

Compiling the new m68k uClinux 2.4 kernel requires an upgrade of the development tools to m68k-elf. The latest patched m68k toolchain is based on gcc-2.95.3. In preparation for building this compiler, download the following files.

- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/binutils-2.10.tar.bz2 (5.3MB)
- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/binutils-2.10-elfPICgot.patch (5kB)
- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/binutils-2.10-wdebug.patch (1kB)

- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/gcc-2.95.3.tar.gz (12.3MB)
- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/gcc-2.95.3-elfPICgot.patch (25kB)

- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/elf2flt-20010606.tar.gz (8kB)

- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/genromfs-0.3.1.tar.bz2 (16kB)
- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/genromfs-0.3.uclinuxdiff (4kB)

The build-m68k-elf.sh Automated Script

David McCullough has done a wonderful job at writing an automated script that builds the m68k toolchain. This automatically builds the following components,

- binutils-2.10 (Assembler, Linker etc)
- gcc-2.95.3 C and C++ Compilers
- optional support for uClibc and multilib libiaries
- elf2flt converter
- genromfs utility

This script file can be downloaded from

- http://www.uclinux.org/pub/uClinux/m68k-elf-tools/tools-20010610/build-m68k-elf.sh (12kB)

You are therefore given the option to download the script and automate the entire build process or manually build the tools yourself. While a great deal of effort has gone into making the script as fool proof as possible, changes to the uClibc library and other dependants can break the script causing headaches trying to hack all the headers together or trying to diagnose where the build has failed. The script also installs the tools in /usr/local which is not everyone's preferred location.

The top of the build-m68k-elf.sh script details instructions on how to built your toolchain and what files are needed. In order to build using the script, edit the following two variables found in the script.

```
UCLIBC="$BASEDIR/uClibc"
KERNEL="$BASEDIR/uClinux-2.0.x"
```

BASEDIR is set to your current directory, thus in most cases you only need to change uClinux-2.0.x to uClinux-2.4.x if you are compiling for version 2.4 of the uClinux kernel. Once you are satisfied with your settings, start the ball rolling with

```
./build-m68k-elf.sh build 2>&1 | tee errs
```

This will build everything except for the multi-lib versions of uClibc. To build these run,

```
./build-m68k-elf.sh uclibc
```

Manually Building the Toolchain

If you have the desire to build the tools yourself, you can proceed with the following instructions. To keep the buildtools uClinux version independent, we don't make the C++ compiler. If needed, this can be done later as an additional step.

binutils-2.10

First start by extracting the binary utilities, change to the binutil directory and apply the two binutil patches.

```
tar -xjf binutils-2.10.tar.bz2 ( or tar –xzf binutils-2.10.1.tar.gz )
cd binutils-2.10
patch –p1 < ../binutils-2.10-elfPICgot.patch
patch –p1 < ../binutils-2.10-wdebug.patch
```

Note : While the patches are aimed at binutils-2.10 as apposed to binutils-2.10.1, the patches have been tested to work on version 2.10.1 without any known side effects.

Configure the bin utilities for the m68k-elf target and give a suitable installation directory. Then make and install the tools.

```
./configure --target=m68k-elf --prefix=/opt/uClinux/
make
make install
```

Now add the binaries to your path for gcc. *During the gcc build process, a couple of libraries will be made for the target architecture and thus the newly created gnu archiver is required*. Either add a path or create symbolic links to your binaries in /usr/bin (preferred).

```
cd /opt/uClinux/bin
ln –s * /usr/bin/
```

gcc-2.95.3

Start on the gcc cross compiler by extracting the source and patching it.

```
tar –xzf gcc-2.95.3.tar.gz
cd gcc-2.95.3
patch –p1 < ../gcc-2.95.3-elfPICgot.patch
```

Configure gcc with a target of m68k-elf, set the install path, enable version specific runtime libaries and explicitly order it only to make the C compiler.

```
./configure --prefix=/opt/uClinux/m68k-elf --target=m68k-elf \
            --enable-multilib --enable-languages=c
```

If an attempt is make to build the m68k-elf cross compiler now, it will fail complaining of two missing files, stdlib.h and unistd.h. To prevent this edit `gcc-2.95.3/gcc/config/m68k/t-m68k-elf` and insert `TARGET_LIBGCC2_CFLAGS = -Dinhibit_libc`

Then make the C Cross Compiler and install it.

```
make
make install
```

elf2flt

Now we can make elf2flt and install it in the appropriate location. The elf2flt utility will convert an ELF file created by the toolchain into a binary flat (BFLT) file for use with uClinux. Start by extracting the tarball and copying the flat.h header file from your kernel's include directory.

```
tar –xzf elf2flt-20010606.tar.gz
cd elf2flt-20010606
cp ../uClinux-2.4.x/include/linux/flat.h .
```

During the build of the elf2flt utility, the libbfd.a (Binary File Descriptor Library) and libiberty.a (GNU Library) is required. Edit the Makefile for elf2flt changing `prefix=` to `prefix=../binutils-2.10.1` so these two files can be found and linked with elf2flt. Now make the elf2flt utility and copy it to the following locations.

```
make
cp elf2flt /opt/uClinux/m68k-elf/bin
ln –f /opt/uClinux/bin/elf2flt /usr/bin
```

Copy the elf2flt linker script

```
cp elf2flt.ld /opt/uClinux/m68k-elf/lib
```

and move the GNU binutils m68k-elf linker to linker.real (There are two copies)

```
mv /opt/uClinux/bin/m68k-elf-ld m68k-elf-ld.real
mv /opt/uClinux/m68k-elf/bin/ld ld.real
```

Copy the linker script in place of the real linker.

```
cp ld-elf2flt /opt/uClinux/bin/m68k-elf-ld
cp ld-elf2flt /opt/uClinux/m68k-elf/ld
```

When called, the linker script will look for the -elf2flt argument. If it is not passed to the linker (e.g. compiling a kernel or library) the script will act transparent and pass all the arguments onto the real linker. However, if –elf2flt is present it will link the required files, then spawn the elf2flt utility to generate a flat binary file.

genromfs

Extract the genromfs (Generate ROM FileSystem) utility, patch the source, build and install it.

```
tar –xjf genromfs-0.3.1.tar.bz2
cd genromfs-0.3.1
patch -p1 < ../genromfs-0.3.uclinuxdiff
make
make install
```

And to finish it off, add all the binaries to your path. Either add a path or create links to your binaries in /usr/bin (preferred)

```
cd /opt/uClinux/bin
ln –sf * /usr/bin/
```

uClibc – The Standard C Library

We are now at a stage where we have a fully installed C Compiler, linker, elf2flt converter and genromfs utility. We can compile the kernel with these tools, but the uClibc library is required for compiling any userland utilities.

The uClibc library is available from CVS. For this you will need a CVS client installed.

Log into the uClinux.org anonymous CVS server using

```
echo anonymous > cvs –d:pserver:anonymous@cvs.uclinux.org:/var/cvs login
```

and download the uClibc library

```
cvs –z3 –d:pserver:anonymous@cvs.uclinux.org:/var/cvs co –P uClibc
```

We earlier built gcc with mulitilib support. This allows the use of one compiler for a subset of m68k architectures including the m68000, m5200 coldfire and mcpu32 by specifying each cpu type by a switch. Gcc will then link your source with the appropriate library based upon what switch you selected.

Compiler Switch	Library Directory
None	/opt/uClinux/m68k-elf/lib
-msoft-float	/opt/uClinux/m68k-elf/lib/msoft-float
-m5200	/opt/uClinux/m68k-elf/lib/m5200
-m5200 –msep-data	/opt/uClinux/m68k-elf/lib/m5200/msep-data
-m68000	/opt/uClinux/m68k-elf/lib/m68000
-m68000 –msep-data	/opt/uClinux/m68k-elf/lib/m68000/msep-data
-mcpu32	/opt/uClinux/m68k-elf/lib/mcpu32
-mcpu32 –msep-data	/opt/uClinux/m68k-elf/lib/mcpu32/msep-data

Therefore we must compile the uClibc library with the appropriate switch and place the compiled libraries in its designated path. The build-m68k-elf.sh script has a extra option invoked by `./build-m68k-elf.sh uclibc` which will compile uClibc for all the above CPUs and copy them to the appropriate place. Where this script comes unstuck, is if you have your tools installed in a location other than /usr.local. If you do use the script for compiling, make sure you have set up the uClinux kernel and toolchain paths.

In many instances you will only need to compile two or three different versions. For example if you have a uCsimm, then you will only need to target the m68000 and m68000 –msep-data. In this case you may opt to compile multi-lib support of uClibc manually.

Start with preparing uClibc. uClibc has support for a variety of architectures each having it's own configuration file stored away in the Configs directory.

```
cp /extra/Configs/Config.m68k Config
```

Edit the Config file, uncommenting the `#CROSS=m68k-elf-` line. Point `KERNEL_SOURCE` to the directory containing your kernel source and add any architecture specific flags (eg. –m68000) to the `ARCH_CFLAGS` towards the end of the config file. If you are using Coldfire, you will also need to tell the assembler to use m5200 (eg. `-m5200 -Wa,-m5200`). Then start building uClibc.

```
make
```

Once completed, copy the libraries into the designated library directory. For example if we were compiling for –m68000, we would of added –m68000 to the `ARCH_CFLAGS` of the uClibc Config file. After compilation we would copy crt0.o, libc.a, libcrypt.a, libm.a, libresolv.a and libutil.a (all found in the libs directory) to /lib/m68000/.

```
cp lib/*     /opt/uClinux/m68k-elf/lib/m68000/
```

Then complete this process for each desired cpu flag.

Examining the m68k-elf C Cross Compiler

The m68k-elf cross compiler is the latest version to hit uClinux development workstations. Partnered with the elf2flt utility, it offers several different types of binaries with many advantages over the older m68k-coff and m68k-pic-coff toolchain combo.

Fully Relocated Binaries

These non-pic (Position Independent Code) binaries are compiled with an origin of zero. Position Independence is achieved at run time using a relocation table appended to the end of the data segment with the offset of each location-dependent address within the program. At execution, the binflat loader copies the text and data segments into RAM, and then adds the start address of the relevant text or data segment of where the binary is loaded at to each address specified in the relocation table.

The outcome is a relatively simple binary which can be loaded (fully relocated) anywhere in memory and relocations performed before control is passed to it. As parts of the code is modified at runtime, the binary must be loaded entirely into RAM in order for the relocation touch-ups to occur. Multiple copies must have both different text and data segments for each instance. Its not uncommon for a single line printf("Hello World\n"); program to have 150 relocations. In this case as each relocation entry is 4 bytes there is a 600 byte relocation table overhead appended to the end of the Hello World binary. At run time, these 150 relocations must be individually touched up before the tread can be started.

The advantages fully relocated binaries have over PIC is it is supported on a wider number of platforms. This is good if your platform doesn't support PIC. It also has fewer, less reachable limits than PIC. For example m68k processors only support a 16 bit offset, thus restricting the GOT (Global Offset Table) to a bit over 8000 relocations. Therefore if you needed more than 8000 relocations, a fully relocated binary may be a good choice.

PIC (Position Independent Code) -msep-data

The m68k-elf compiler has an option (fpic / fPIC) to generate position independent code. This is achieved by making all jump and subroutine calls PC relative rather than absolute. Data access is performed in the form of a GOT (Global Offset Table). A GOT is a 16 bit look-up table which contains 32 bit address pointers and resides at the start of the data segment, terminated by a minus 1.

The only problem with (fpic / fPIC) is its dependence that the data segment must immediately follow the text segment. This is what occurs when the program is first compiled at location zero, thus branches can be safety made.

-msep-data, short for separate data allows for the data and text segments to be separated and placed in different regions of memory. Separate text and data segments come in useful with XIP (eXecute In Place) where the code (.text) is executed from FLASH/ROM and the data segment is loaded into RAM where the program can modify it's variables. -msep-data will turn on the –fPIC option. –fPIC and –fpic should not be used with m68k-elf.

Therefore the main advantage of msep-data is the ability for binaries to do XIP. It also has a lot less relocations, which helps to keep the file size smaller and saves in memory.

Using m68k-elf-gcc

The m68k-elf-gcc cross compiler will generate m68020 code by default. Therefore to prevent privileged instructions or illegal instruction stops, you must specify your target processor. For most architectures including the Dragonball processors, you will need to add –m68000. For the Coldfire processors you will need to add –m5200.

Fully Relocated Binaries

```
m68k-elf-gcc –Wl,-elf2flt –m68000 –o demo demo.c –lc.
```

Advantages

- Works for most targets
- Has fewer limits – Good for larger programs

Disadvantages

- Includes quite a few relocations that cause a larger executable size and takes longer to load & relocate.
- Doesn't support XIP, thus .TEXT, .DATA & .BSS must be in RAM. This must be duplicated for multiple instances.

PIC Separate-Data Binaries

```
m68k-elf-gcc –Wl,-elf2flt –m68000 –msep-data –o demo demo.c –lc.
```

Advantages

- Supports XIP. Can be executed from FLASH/ROM. Multiple instances only need one copy of .TEXT segment.
- Smaller executable size.
- Less relocations.

Disadvantages

- Maximum of about 8000 relocations.

Compressed Binaries

```
m68k-elf-gcc –Wl,-elf2flt –m68000 [-msep-data] –o demo demo.c –lc.
elf2flt –z –o demo demo.elf
```

Advantages

- Smaller executable size.
- Good for less frequently used executables (e.g. flashloader etc)

Disadvantages

- Latency at load to decompress executable.
- Doesn't support XIP as .text must be decompressed into RAM.