OS Specific Toolchain

From OSDev Wiki

Difficulty level
Advanced

This tutorial will guide you through creating a toolchain comprising binutils and gcc that specifically targets your operating system. The instructions below teach binutils and gcc how to create programs for a hypothetical OS named 'MyOS'.

Until now you have been using a cross-compiler configured to use an existing generic bare target. This is very convenient when starting out as you get a reliable target and the compiler doesn't make any bad assumptions because it thinks it is targeting an existing operating system. However, when you proceed it becomes useful if the compiler knows it is targeting your operating system and what its customs are. For instance, you can make the compiler define a `__myos__` preprocessor macro, know which directories to search for include files in, what special `crt*.o` files are used when linking against libc, and so on. It also becomes much easier to cross-compile software to your OS when you simply have to invoke `x86_64-myos-gcc hello.c -o hello` to cross-compile a program. Additionally part of the instructions here can be applied to other software packages that also use the GNU build system, which will help you port existing software.

This tutorial teaches you how to set up a cross-compiler that specifically targets your OS. This is actually the first step of porting binutils and gcc to your operating system: Any information you give GCC about your OS will help it run on your OS. Once your OS Specific Toolchain has been set up and you have built your OS with it, you can continue by using the cross-compiler to cross-compile the compiler itself to your OS, assuming your libc and kernel is powerful enough. For more information see Porting GCC to your OS.

Contents

Introduction

You need the following before you get started:

- A build environment that can successfully build a GCC Cross-Compiler.
- autoconf (exactly version 2.64)
- automake
- libtool
- The latest binutils source code. (**Note**: This tutorial has not yet been updated to cover binutils 2.25)
- The latest gcc source code.
- Knowledge of the internals of binutils and gcc.
- Knowledge of autoconf and automake.
- The dependencies of binutils and gcc as detailed in GCC Cross-Compiler.

Additionally you will need a C Library as described in a later section. As detailed in Hosted GCC Cross-Compiler, it doesn't need to support much and the functionality can be stubbed, but libgcc will need to believe you have a libc.

You should decide exactly what targets you'll add to binutils and gcc. If you have been using a generic `i686-elf` or `x86_64-elf` or such target, you'll simply want to swap `-elf` with `-myos` and get `i686-myos` and `x86_64-myos`. Naturally, don't actually write myos, but rather use the name of your OS converted to lower case. See Target Triplet.

This tutorial currently only have instructions for adding a new x86 and x86_64 target for myos, but it serves as a good enough example that it should be trivial to add more processors by basing it on these instructions and what other operating systems have done.

Modifying Binutils

config.sub

This is a file you will modify in the same way for each package. It is a GNU standard file produced by including the line 'AC_CANONICAL_SYSTEM' in a configure.ac that is processed by autoconf, and is designed to convert a canonical name of the form `i686-pc-myos` into separate variables for the processor, vendor and OS, and also rejects systems it doesn't know about. We simply need to add 'myos' to the list of acceptable operating systems. Find the section that begins with the comment "`First accept the basic system types`" (it begins '`-gnu*`') and add '`-myos*`' to the list. Find a line with some free room and add your entry there.

bfd/config.bfd

This file is part of the configuration for libbfd, the back-end to binutils which provides a consistent interface for many object file formats. Generally, each platform-specific version of binutils contains a libbfd which only supports the object files normally in use on that system, as otherwise the library would be massive (libbfd can support a _lot_ of object types). We need to associate our os with some particular object types. There is a long list starting 'WHEN ADDING ENTRIES TO THIS MATRIX' with the first line as 'case "${targ}" in'. We need to add our full canonical name to this list, by adding some cases such as:

```
  i[3-7]86-*-myos*)
    targ_defvec=bfd_elf32_i386_vec
    targ_selvecs=
    targ64_selvecs=bfd_elf64_x86_64_vec
    ;;
#ifdef BFD64
  x86_64-*-myos*)
    targ_defvec=bfd_elf64_x86_64_vec
    targ_selvecs=bfd_elf32_i386_vec
    want64=true
    ;;
#endif
```

Note: This does not compile with binutils-2.25, however changing bfd_elf32_i386_vec to i386_elf32_vec helped. (Possibly similar solutions exist for the other targets)

Be sure to follow the instructions in the comment block above the list and add your entry beneath the comment "`#START OF targmatch.h`". If you like, you could support different object formats (look at other entries in the list, and the contents of 'bfd' for hints) and also provide more than one to the `targ_selvecs` line. For instance, you can support coff object files if you add `i386coff_vec` to the `targ_selvecs` list. For some reason, all the `x86_64` entries in the file file are wrapped in `#ifdef BFD64`, it's probably prudent to do it yourself as well.

gas/configure.tgt

This file tells the gnu assembler what type of output to generate for each target. It automatically matches the i686 part of your target and generates the correct output for that. We just need to tell it what type of object file to generate for myos. In the section starting '`Assign object format ... case ${generic_target} in`' you need to add a line like

```
  i386-*-myos*)    fmt=elf ;;
```

You should use '`i386`' in this line even if you are targeting x86_64. This is the only file where you should do it. It is basically because the variable 'generic_target' is not your canonical target name, but rather a variable generated further up in the configure.tgt file, and it sets the first part to `i386` for any `i[3-7]86` or `x86_64`.

ld/configure.tgt

This file tells the gnu linker what 'emulation' to use for each target. An emulation is basically a combination of linker script and executable file format. We are going to define our own emulation called `elf_i386_myos`. We need to add an entry to the case statement here after '`Please try to keep this table in alphabetical order ... case "${targ}" in`':

```
i[3-7]86-*-myos*)
                        targ_emul=elf_i386_myos
                        targ_extra_emuls=elf_i386
                        targ64_extra_emuls="elf_x86_64_myos elf_
                        ;;
x86_64-*-myos*)
                        targ_emul=elf_x86_64_myos
                        targ_extra_emuls=elf_i386_myos elf_x86_6
                        ;;
```

- **elf_i386_myos** is a 32-bit target for your OS.
- **elf_x86_64_myos** is a 64-bit target for your OS.
- **elf_i386** is a bare 32-bit target as you had with `i686-elf`.
- **elf_x86_64** is a bare 64-bit target for your OS as you had with `x86_64-elf`.

This setup provides you with a 32-bit toolchain that also can produce 64-bit executables, and a 64-bit toolchain that can also produce 64-bit executables. This comes in handy if you use objcopy, for instance. You can also add `targ_extra_emuls` entries to specify other targets ld should support. See the `ld/configure.tgt` file for examples.

ld/emulparams/elf_i386_myos.sh

Now we need to actually define our emulation. There is a generic file called `ld/genscripts.sh` which creates the required linker scripts for our target (you need more than one, depending on shared object usage and the like: I have 13 for a single target). It uses a linker script template (from the ld/scripttempl directory) to do this, and it creates the actual emulation C file from an emulation template (from the ld/emultempl directory). These templates are customised by running a script in the ld/emulparams directory which sets various variables. You are welcome to define your own emulation and linker templates, but I find the ELF ones adequate, given that they can be customised by simply adding a file to the emulparams directory. This is what we are going to do now. The content of the file could be something like:

```
. ${srcdir}/emulparams/elf_i386.sh
GENERATE_SHLIB_SCRIPT=yes
GENERATE_PIE_SCRIPT=yes
```

This script is included by `ld/genscripts.sh` to customize its behavior through shell variables. We include the base `elf_i386.sh` script as it sets reasonable defaults. Finally, we override the variables whose defaults we disagree with.

There are a large number of variables that can be set here to customize your toolchain. Read the documentation and look at existing emulations for further information. These are some of the variables that can be set:

- **GENERATE_SHLIB_SCRIPT**=yes|no Whether to generate a linker script for shared libraries. We enable this as you might want it later. The base 32-bit elf script defaults this to disabled for some reason.
- **GENERATE_PIE_SCRIPT**=yes|no Whether to generate a linker script for position independent executables. We enable this as you might want it later. The base 32-bit elf script defaults this to disabled for some reason.
- **SCRIPT_NAME**=name Controls which ld/scripttempl/name.sc script generates our linker scripts.
- **TEMPLATE_NAME**=name Controls which ld/emultempl/name.em script generates our bfd emulation C implementation.
- **OUTPUT_FORMAT**=name The name of the BFD output target we use.
- **TEXT_START_ADDR**=0xvalue Controls where the executable begins in memory.

You can read the base `elf_i386.sh` script for the defaults of these variables, you can then decide for yourself if you wish to override them for your operating system.

ld/emulparams/elf_x86_64_myos.sh

This file is just like the above `ld/emulparams/elf_i386_myos.sh` but for x86_64.

```
. ${srcdir}/emulparams/elf_x86_64.sh
```

There is no reason to set `GENERATE_SHLIB_SCRIPT` and `GENERATE_PIE_SCRIPT` here as the x86_64 base script enables them by default.

ld/Makefile.am

We now just need to tell make how to produce the emulation C file for our specific emulation. Putting the '`targ_emul=elf_i386_myos`' line into `ld/configure.tgt` above implies that your host linker will try to link your target ld executable with an object file called `eelf_i386_myos.o`. There is a default rule to generate this from `eelf_i386_myos.c`, so we just need to tell it how to make this `eelf_i386_myos.c` file. As stated above, we let the genscripts.sh file do the hard work. You need to add makefile rules:

```
# Add this after eelf_i386.c:
eelf_i386_myos.c: $(srcdir)/emulparams/elf_i386_myos.sh \
  $(ELF_DEPS) $(srcdir)/scripttempl/elf.sc ${GEN_DEPENDS}
	${GENSCRIPTS} elf_i386_myos "$(tdir_elf_i386_myos)"

# Add this after eelf_x86_64.c:
eelf_x86_64_myos.c: $(srcdir)/emulparams/elf_x86_64_myos.sh \
  $(ELF_DEPS) $(srcdir)/scripttempl/elf.sc ${GEN_DEPENDS}
	${GENSCRIPTS} elf_x86_64_myos "$(tdir_elf_x86_64_myos)"
```

Note: Some parts of the line use normal brackets () whereas other parts use curly braces {}.

Note: The third line must start with single tab, not spaces, as this is a Makefile.

You also need to add `eelf_i386_myos.c` to the `ALL_EMULATION_SOURCES` list; and you also need to add `eelf_x86_64_myos.c` to the `ALL_64_EMULATION_SOURCES` list.

Note: You must run `automake` in the `ld` directory after you modify `Makefile.am` to regenerate `Makefile.in`.

Modifying GCC

config.sub

Similar modification to config.sub in binutils.

gcc/config.gcc

This file defines what needs to be built for each particular target and what to include in the final executable. There are two main sections: one which defines generic options for your operating system, and those which define options specific to your operating system on each individual machine type.

For the first part, find the '`case ${target} in`' line just after '`# Common parts for widely ported systems`' (around line 617) and add something like:

```
*-*-myos*)
  gas=yes
  gnu_ld=yes
  default_use_cxa_atexit=yes
  ;;
```

This states that our operating system by default uses the GNU linker and assembler and that we will provide __cxa_atexit (you will need to provide this in your standard library).

The second section we need to add to is the architecture-specific one. Find the '`case ${target} in`' line just before '`tm_file="${tm_file} dbxelf.h elfos.h newlib-stdint.h"`' (around line 886) and add something like:

```
i[34567]86-*-myos*)
    tm_file="${tm_file} i386/unix.h i386/att.h dbxelf.h elfos.h
    ;;
x86_64-*-myos*)
    tm_file="${tm_file} i386/unix.h i386/att.h dbxelf.h elfos.h
    ;;
```

This defines which target configuration header files gets used. You can make `i386/myos32.h` and `i386/myos64.h` files if desired.

gcc/config/myos.h

This header allows you to customize your toolchain using preprocessor macros. The relevant parts of GCC will include this header (as controlled by `gcc/config.gcc`) and modify the behavior according to your customizations.

```
/* Useful if you wish to make target-specific gcc changes. */
#undef TARGET_MYOS
#define TARGET_MYOS 1

/* Default arguments you want when running your
   i686-myos-gcc/x86_64-myos-gcc toolchain */
#define LIB_SPEC "-lc -lg -lm" /* link against C standard librar
                               /* modify this based on your need

/* Don't automatically add extern "C" { } around header files. *
#undef  NO_IMPLICIT_EXTERN_C
#define NO_IMPLICIT_EXTERN_C 1

/* Additional predefined macros. */
#undef TARGET_OS_CPP_BUILTINS
#define TARGET_OS_CPP_BUILTINS()      \
  do {                                \
    builtin_define ("__myos__");      \
    builtin_define ("__unix__");      \
    builtin_assert ("system=myos");   \
    builtin_assert ("system=unix");   \
    builtin_assert ("system=posix");   \
  } while(0);
```

TODO: It would make sense to document the handling of `STARTFILE_SPEC`, `ENDFILE_SPEC`here as well.

libstdc++-v3/crossconfig.m4

This file describes how the libstdc++ configure file will examine your operating system and adjust the provided features of libstdc++ accordingly. Add a case similar to

```
  *-myos*)
    GLIBCXX_CHECK_COMPILER_FEATURES
    GLIBCXX_CHECK_LINKER_FEATURES
    GLIBCXX_CHECK_MATH_SUPPORT
    GLIBCXX_CHECK_STDLIB_SUPPORT
    ;;
```

TODO: Examine this design and find out what actually needs to be done here.

Note: You need to run `autoconf` in the `libstdc++-v3` directory.

libgcc/config.host

Find the '`case ${host} in`' just prior to '`extra_parts="$extra_parts crtbegin.o crtend.o crti.o crtn.o"`' (around line 318) and add the cases:

```
i[34567]86-*-myos*)
        extra_parts="$extra_parts crtbegin.o crtend.o"
        tmake_file="$tmake_file i386/t-crtstuff t-crtstuff-pic t
        ;;
x86_64-*-myos*)
        extra_parts="$extra_parts crtbegin.o crtend.o"
        tmake_file="$tmake_file i386/t-crtstuff t-crtstuff-pic t
        ;;
```

fixincludes/mkfixinc.sh

You should disable fixincludes for your operating system. Find the case statement and add a pattern for your operating system. For instance:

```
# Check for special fix rules for particular targets
case $machine in
    *-myos* | \
    i?86-*-cygwin* | \
        # (... snip ...)
    powerpcle-*-eabi* )
        #  IF there is no include fixing,
        #  THEN create a no-op fixer and exit
        (echo "#! /bin/sh" ; echo "exit 0" ) > ${target}
        ;;
```

A number of operating systems (especially older and obscure ones) provide troublesome systems headers that fail to strictly comply with various standards. The GCC developers consider it their job to fix these headers. GCC will look into your system root, apply a bunch of patterns to detect headers it doesn't like, then it copies that header into a private gcc system directory (that overrides your standard system directory) and attempts to fix the header. Sometimes fixincludes even break working headers (some people refer to it as breakincludes).

This is rather inconvenient as your libc will likely happen to trigger these patterns (and false positives often happens). Any time you change your system headers, you have to rebuild your compiler so the fixed versions get updated. The first time you encounter this, it will show up as a system header that does nothing different even though you edit it.

This addition to the `mkfixinc.sh` file forcefully disables fixincludes for your operating system. It's your job to provide working system headers, not the compiler developers'.

Further Customization

TODO: Document more various tips and tricks for further customization of OS specific toolchains.

Changing the Default Include Directory

If you belong to the group that has decided that the /usr directory is big and evil, you may wish to change the default include directory path. If you wish to do so, you can simply add this to your gcc/gcc/config/myos.h:

```
/* Standard include directory. */
#undef STANDARD_INCLUDE_DIR
#define STANDARD_INCLUDE_DIR "/include"
```

Changing the Default Library Directory

TODO: Document this, but it is done by changing binutils and is a bit more complex

Start Files Directory

You can modify which directory GCC looks for the crt0.o, crti.o and crtn.o in. The path to that directory is stored in `STANDARD_STARTFILE_PREFIX`. For instance, if you wish to have different locations depending on the processor, you can add the following to gcc/gcc/config/i386/myos64.h:

```
#undef STANDARD_STARTFILE_PREFIX
#define STANDARD_STARTFILE_PREFIX "/x86_64-myos/lib/"
```

Selecting a C Library

Main article: C Library

At this point, you have to decide which C Library to use. You have options:

- Create your own C library.
- Pick an existing C Library such as Newlib.

Building

Main article: Hosted GCC Cross-Compiler

Your OS specific toolchain is built differently from the introductory `i686-elf` toolchain as it has a user-space and standard library. In particular, you need to ensure your libc meets the minimum requirements for libgcc. You need to install the standard library headers into your System Root before building the cross-compiler. You need to tell the cross-binutils and cross-gcc where the system root is via the configure option `--with-sysroot=/path/to/sysroot`. You can then build your libc with your cross-compiler and then finally libstdc++ if desired.

Conclusion

You now have a `i686-myos` toolchain that can be used instead of your old `i686-elf` toolchain. Your new toolchain is effectively just a renamed `i686-elf` with customizations. You should switch all your operating system build scripts to use this new compiler, even the kernel and libk, as your new compiler is capable of providing a freestanding environment.

You will certainly wish to package up your custom toolchain (and be able to create a diff between the upstream version and your custom version for others to audit). Contributors should be able to download tarballs of your myos-binutils and myos-gcc packages, so they can build themselves your custom toolchain.

Common errors

Whitespaces

Some files need tabs, some files need spaces and some files accept happily any mixture. Use an editor that can display special chars such as tabs and spaces, to be sure you use the right form. Whitespace errors may result in 'make' reporting missing separators. Some editors will replace a tab with four spaces, which will also cause invalid separator issues.

Autoconf

There are several steps that conclude in running '`autoconf`' or '`automake`', '`autoreconf`', be sure you did not miss them. The order of autoconf/-reconf calls in a package is important. These errors may result in missing subdirectories of the build-* directory and/or 'make' reporting missing targets.

See Also

Articles

- GCC