Building Existing Software with the EMAC OE SDK

It is very common to need to be able to build existing software projects for the target hardware rather than developing the software from scratch. This feature is especially important in an open source environment, where countless libraries and utilities are available for use and often times need to be compiled to match the target architecture. Fortunately, EMAC OE SDK toolchains are standard GCC packages designed and configured to make this process as easy as possible. In addition, most software projects are developed to allow for cross-platform development.

This guide provides an overview of the most common tasks associated with compiling existing software projects using the SDK. Note, however, that build methods differ significantly depending on the project design. Refer to the project documentation or support for information on how to cross-compile the software; the EMAC OE SDK can be treated as a standard GCC toolchain in this respect. Table 1 below denotes some conventions used in this guide.

Table 1. Conventions Used

`/path/to/sdk/`	Indicates the directory where the EMAC OE SDK is located.
`EMAC-OE-arm-linux-gnueabi-SDK_XX.YY/`	The name of the SDK directory; this will differ depending on what architecture is used. `XX` is the major version `YY` is the minor version `ZZ` is the current revision The major and minor version numbers will match the version of OE for which the SDK was created. The current version is 4.0.

Makefile-based Projects

Some projects have a build system based on a set of makefiles that are responsible for compiling and packaging the software. In general, configuring these projects to compile using the EMAC OE SDK is a simple process. In some instances, the software designers may have included a variable in the makefile which allows a cross-compiler prefix setting. In other cases, the `CC` and other makefile variables can be modified to direct `make` to compile using the EMAC OE SDK.

It may be advantageous to add the cross-compiler `bin` directory to the system `PATH` variable temporarily before compiling to simplify Makefile modifications. This can be done with a command such as the following:

```
$ export PATH=$PATH:/path/to/sdk/EMAC-OE-arm-linux-gnueabi-SDK_XX.YY/gcc-4.2.4-arm-linux-gnueabi/bi
```

Note that running the command above will only affect the current terminal session.

MTD Utilities Project Example

The MTD Utilities project (`mtd-utils`) is a good example of a Makefile-based build system that can easily be built using the EMAC OE SDK. Follow the steps below to accomplish this task.

The instructions in this section are valid as of the master git branch on 4/09/11. Future source changes may impact the required steps for compilation.

1. Begin by downloading the `mtd-utils` source code. Assuming that `git` is installed on the development system, run the following command to get the most recent version. *Note that this version will be different from the stable release installed on EMAC OE systems by default.*

```
$ git clone git://git.infradead.org/mtd-utils.git
```

2. After downloading the source, navigate to the source directory (`mtd-utils`) and open `Makefile` in the top-level directory. This file defines various make targets and compilation flags used to compile the source. Notice that the file `common.mk` is sourced with the line `include common.mk`. This is similar to the `global.properties` file in the EMAC OE SDK.
3. Close the `Makefile` editor and open the `common.mk` file. At the top of the file, `CC`, `AR`, and `RANLIB` are defined using the `CROSS` variable, which is not set in either `common.mk` or `Makefile`. An exert is shown below:

```
CC := $(CROSS)gcc
AR := $(CROSS)ar
RANLIB := $(CROSS)ranlib
```

   If the `CROSS` variable is defined when `make` is run, the specific toolchain to use can be specified. Also note that `CFLAGS` is defined using a `?=` assignment, which means that the assignment will only be made if `CFLAGS` is undefined:

```
CFLAGS ?= -O2 -g
```

4. Close `common.mk` without making any changes.
5. In order to compile the source correctly, the `CROSS` and `CFLAGS` variables should be defined. The tuning flags for the target architecture to be added to the `CFLAGS` variable should be used from the `ARCHFLAGS` variable in the `global.properties` file of the EMAC OE SDK. The following steps utilize tuning flags for an armv5te processor-based system. Although the path to the SDK toolchain could be included directly in the `CROSS` variable in this instance, this example works by adding the SDK toolchain to the system `PATH` variable. The path and prefix used will differ depending on the target architecture of the EMAC OE SDK; refer to `global.properties` in the SDK to determine the correct settings. The `DESTDIR` variable controls where the files are put when running the `install` make target. The `WITHOUT_XATTR` flag must be set to disable features of the software that are not available on EMAC OE.
    1. Before compiling, several source changes are required to match the setup of the SDK. These include changing references from `lzo2` to `lzo`, and changing the `lzo` include prefix. Run the following commands from the mtd-utils directory to make these changes:

```
$ for file in `find . -name Makefile`; do sed -i 's:lzo2:lzo:g' $file; done
$ for file in `find . -name '*.c'`; do sed -i 's:lzo/::g' $file; done
```

    2. Run the following commands to set the variables:

```
$ export PATH=$PATH:/path/to/sdk/EMAC-OE-arm-linux-gnueabi-SDK_XX.YY/gcc-4.2.4-arm-linux
$ export CROSS=arm-linux-gnueabi-
$ export CFLAGS="-O2 -g -march=armv5te -mtune=arm926ej-s"
$ export DESTDIR=Install
$ export WITHOUT_XATTR=1
```

6. Once the environment has been set up, `make` can be used to compile the source code:

```
$ make all
```

   The code should compile successfully with no errors. If compilation is not successful, check the steps above to ensure that all of the required changes have been made.

7. After successfully compiling the project, the `install` make target can be used to package all of the software

into the `Install` directory as specified by the `DESTDIR` variable:

```
$ make install
$ ls -R Install
Install:
usr

Install/usr:
sbin  share

Install/usr/sbin:
docfdisk      flash_erase     flash_lock      flash_unlock  jffs2dump   nanddump   nftldump
doc_loadbios  flash_eraseall  flash_otp_dump  ftl_check     mkfs.jffs2  nandtest   nftl_format
flashcp       flash_info      flash_otp_info  ftl_format    mtd_debug   nandwrite  recv_image

Install/usr/share:
man

Install/usr/share/man:
man1

Install/usr/share/man/man1:
mkfs.jffs2.1.gz
```

While the procedure in this example is specific to `mtd-utils`, many makefile-based projects will require similar steps for cross-compiling.

Autotools-based Projects

The GNU build system is known as Autotools. Autotools is a group of applications that are designed to provide a configurable build system to allow compilation on different platforms and environments. A `configure` script and set of input files are used to generate makefiles based on options passed to the configure script and deduced from the system environment. This includes tests for compiler options, library functions, install configuration, and other assorted variables.

The `configure` script is the most important step in building an autotools-based project. Although available options for `configure` vary depending on the project design, there are common options shared between most autotools projects.

libConfuse Example Project

The libConfuse project is a simple C library written for parsing configuration files. It uses an autotools build system for configuration. The steps below demonstrate how to build libConfuse and should be used as an example for building other autotools-based projects.

1. The source code for the libConfuse project can be downloaded as described on the project website http://www.nongnu.org/confuse/ (http://www.nongnu.org/confuse/) . For this example, release 2.7 is used: http://savannah.nongnu.org/download/confuse/confuse-2.7.tar.gz (http://savannah.nongnu.org/download/confuse/confuse-2.7.tar.gz) . After downloading the source, extract the archive and navigate to the top-level directory of the project (i.e. `confuse-2.7`).
2. Read through the `README` and `INSTALL` files for information on the project and general information on how to build it. Also, look at the help output from `configure` to see a summary of the available options:

```
$ ./configure --help
```

3. Before beginning, the system `PATH` variable should be changed to include the SDK toolchain as in the makefile-based project example. The `CFLAGS` variable should also be set. If the `CC` variable is set, it should be set to `arm-linux-gnueabi-gcc` (or the appropriate value for the target); if not set the compiler name will be

detected from the options passed to configure. The `CFLAGS` architecture tuning values should be set according to the `global.properties` file in the EMAC OE SDK. The `DESTDIR` variable determines where the files will be installed to when the `install` make target is run. `DESTDIR` must be an absolute path for the libConfuse project, so the current working directory is added to the variable using the `pwd` command. The following commands are an example of how to set up the environment correctly:

```
$ export CFLAGS="-O2 -march=armv5te -mtune=arm926ej-s"
$ export DESTDIR=`pwd`/Install
```

4. After setting the environment, `configure` can be run with the appropriate options to configure the build system and generate the makefiles. The code below shows an example configuration used by EMAC OE. Be sure to set the host and target correctly based on the architecture:

```
$ ./configure --build=i686-linux --host=arm-linux-gnueabi --target=arm-linux-gnueabi \
        --prefix=/usr --exec_prefix=/usr --bindir=/usr/bin --sbindir=/usr/sbin \
        --datadir=/usr/share  \
        --infodir=/usr/share/info \
        --mandir=/usr/share/man --enable-shared
```

   The configuration should complete successfully. If any problems are reported that result in an error, check the environment settings and `configure` options again.

5. Now that `configure` has generated all of the makefiles for the project, `make` can be used to compile the source code:

```
$ make all
```

   If any errors are encountered during compilation, examine the output of `configure` and make sure that all of the environment variables and `configure` options were specified correctly.

6. Once compilation is complete, the `install` target can be used to package all of the necessary files together so that they can be transferred to the target board.

```
$ make install
$ ls -R Install
Install:
usr

Install/usr:
include  lib  share

Install/usr/include:
confuse.h

Install/usr/lib:
libconfuse.a  libconfuse.la  libconfuse.so  libconfuse.so.0  libconfuse.so.0.0.0  pkgconfig

Install/usr/lib/pkgconfig:
libconfuse.pc

Install/usr/share:
locale

Install/usr/share/locale:
fr  sv

Install/usr/share/locale/fr:
LC_MESSAGES

Install/usr/share/locale/fr/LC_MESSAGES:
confuse.mo

Install/usr/share/locale/sv:
LC_MESSAGES

Install/usr/share/locale/sv/LC_MESSAGES:
confuse.mo
```

Note that not all of the files installed would be necessary to install on the board, such as the man pages and pkgconfig information.

See Also

TODO
