Fix Me! Elaborate and test on all development platforms!

Unofficial Method

This document is an experiment, not an instruction

Newbies, this is not the page you're searching for.

Using the AutoTools application skeleton

This page describes the use of the autotools-based mios application skeleton.

Announcement forum post: http://www.midibox.org/forum/index.php?topic=10338.0

You can download the skeleton in the annoucement forum post.

Background

Traditional build/install procedure

As of today, quite all applications uses the same standard procedure to be build and installed. That procedure is composed of the following steps :

1. Configuration of the build
2. Build itself (Compile/Assemble/Link)
3. Installation of the build artefacts

To each of those three steps is associated a standard command. The achieve the build and installation of a program/library we do :

```
./configure <configure informations>
make all
make install
```

> Note that the 'all' in 'make all' may be ommited.

However writing the **configure** script and the **Makefile** (that consumes the make command) can be very tedious, moreover if you want them to be portable and easily maintainable... That's when the Autotools come into action!

What are the AutoTools ?

The, so-called, AutoTools are a set of gnu tools to automate the build of gnu-like applications. They are made as a layer above traditional bash shell and make tool. They use simple configuration files to generate the scripts and Makefile(s) required to build the application.

AutoTools comprise (amongst others) :

- **autoconf**: Generates a configure script
- **automake**: Generates Makefile files

Why another application skeleton ?

We can note the following facts about the actual C skeleton :

1. The skeleton comprise lots of files which are generic to the application. Thus, if you make multiple applications, each and every application directory contains copies of those files. This is really difficult to maintain and very error-prone.
2. The skeleton uses its own makefile generation system. This...
    - Does not enable directory hierarchies
    - Does not encompass by default all the currently wanted targets (all, clean, install, dist, dist-check...)
    - Is not easily extensible (don't want to have to make rules for simple things and want to make simple rules for complex things)
    - Does not take profit of an externally-maintained build system (MidiBox people maintains it and makes it evolve, which add extra work to the MidiBox community)

Let's detail the exact implications in the try to make it better :

- (1) required the MIOS wrapper itself to be put elsewhere than in the skeleton. Why not having a libcmios library ? Hopefully gplib can make shared libraries and gplink can grasp object files from them and link them statically inside the application. Cool!
- The MIOS-specific linker scripts also had to be put in a shared location. I decided to follow the usual rules of sharing for cross-compilation... having a directory that is specific for a host platform that contains headers and libraries. (${prefix}/host/)
- The MIOS-specific asm fix script was a problem and should be transparently called.

What is a toolchain ?

A toolchain is the set of tools to go from source files to executable application :

(C source file) ⇒ compile ⇒ (Assembly source file) ⇒ assemble ⇒ (Object file) [⇒ librarian ⇒ (Reusable library)] ⇒ link ⇒ (Executable application)

Thus a toolchain generally includes :

- A compiler (for us **sdcc**)
- An assembler (for us **gpasm**)

- A librarian (for us **gplib**)
- A linker (for us **gplink**)

To enable integration with the AutoTools, you must have your toolchain to follow the naming style of the gnu toolchain. That is : tools must be name as the standard gnu tools prefixed by the platform host (which follows the scheme `processor-vendor-os`). So :

- C compiler is named **processor-vendor-os-gcc**
- Asm compiler is named **processor-vendor-os-as**
- Librarian tool is named **processor-vendor-os-ar**
- Linker is named **processor-vendor-os-ld**

Also those tools needs to take the standard options of the gnu tools. So we need to have a wrapper script around sdcc, gpasm, gplib and gplink, that would take care of transparently fixing the asm, and that would use the GCC command line style.

What are the MIOS platform hosts ?

Host processors are :

- **pic18f452** (vendor: microchip)
- **pic18f4520** (vendor: microchip)
- **pic18f4620** (vendor: microchip)
- **pic18f4685** (vendor: microchip)

Host OS is (obviously :)) **mios**

Which makes four possible targets/hosts, which are:

- pic18f452-microchip-mios
- pic18f4520-microchip-mios
- pic18f4620-microchip-mios
- pic18f4685-microchip-mios

> **Support for PIC18F4685 :** You need sdcc-svn and gputils-0.13.5 versions at least to get support for the PIC18F4685 processor (`sdcc-svn` and `>=gputils-0.13.5` packages on Gentoo Linux)

Settings up the tools

Requirements

You obviously need the AutoTools so...

Please refer to :

- Linux : Installing the AutoTools on Gentoo Linux
- Mac OSX : Installing the AutoTools on Mac OSX
- Cygwin/Windows : Installing the AutoTools on Cygwin/Windows

Making choices

It is time to choose where you want to install all your MIOS-related things.

Basically, here where the different things will go :

- `${prefix}/bin` will contain the tools from the toolchain
- `${prefix}/mios/${host}` will contain platform-dependent stuff
- `${prefix}/mios/*` will contain platform-independent stuff

where `${prefix}` is by default `/usr/local`.

A more complete view of the filesystem tree is :

- `${prefix}/`
  - `bin/` tools (eg. pic18f4685-microchip-mios-gcc)
  - `mios/`
    - `include/` libraries' header files (eg. cmios.h, can.h...)
    - `share/` shared files (eg. documentations, licenses...)
    - `${host}/`
      - `lib/` : libraries (eg. libcmios.a, libcan.a...)
      - `scripts/` : linker scripts

You can specify the `${prefix}` value by passing `–prefix=value` to your `./configure` command-line calls. For instance :

```
./configure ... --prefix=/usr
```

Note that if you chose to configure a certain prefix for one of your library, you will have to put the exact same prefix in the configure of all your libraries and applications.

So you have to choose a place in your filesystem and to abide by your choice! (I personally let the standard `/usr/local/` prefix which makes me less to type on the command line.)

> In fact, the choice of the `${prefix}` for `mios-pic16-toolchain` is independent from anything else. But that is the sole one.

Installing mios-pic16-toolchain

Details...

```
cd mios-pic16-toolchain
./configure --target=pic18f4685-mios
make
sudo make install
```

Installing libcmios

Fix Me! Details...

```
cd libcmios
./configure --host=pic18f4685-mios
make
sudo make install
```

Getting started

Here we will describe some of the features of the skeleton, by giving a step-by-step guide to use the skeleton.

The skeleton, in its delivery form, already contains a simple print “Hello, world!” on the LCD. So, we will do a little more : fetch an encoder and print its value on the LCD. Let's call our application the 'one-encoder' application.

By doing this we will see how to :

- Make the basic configuration of the skeleton
- Use an external library (namely libcmios)
- Add more C headers and sources
- Tweak the data distributed by our package (documentation and license)

So let's start our fabulous application. First copy the skeleton folder and rename its copy `one-encoder`

```
cp -r mios-skeleton-1.9f-r1/ one-encoder
cd one-encoder/
```

Skeleton overview

Here are the files provided in the skeleton :

- `config.sub`
- `configure.in`
- `Makefile.am`
- `README`
- `src/`
    - `main.c`
    - `Makefile.am`

Three files are controlling the build and installation behavior :

- `configure.in` - Package configuration tweaking
- `Makefile.am` - Root directory build tweaking

- `src/Makefile.am` - Source directory build tweaking

There is **another file** in the root directory (`config.sub`), and there will be more (generated) files after the first AutoTools launch. However you will **never have to modify them**. Just understand that `*.am` files are used to generated `*.in` files, which in turn are used to generate * files. (For instance, `Makefile.am` is the source to generate `Makefile.in` which itself is the source to generate `Makefile`.)

You will have to **launch the first auto-configuration**. This will be done by calling the `auto(re)conf` command in a certain way.

After that, each and every time you will **modify one of those three files**, then, at the time you will launch the **make command again**, the package will **automatically be reconfigured** (in the sense of calling `autoconf`).

But, for now, let's copy the skeleton and tweak a bit its `configure.in` file...

Configuration of the application

Open the `configure.in` file with your favorite editor. Here is its content :

1 |configure.in

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.61)
AC_INIT(MIOS C Skeleton, 1.9f-r1, http://www.ucapps.de/, mios-c-
skeleton)
AC_MIOS_HOSTS([
        pic18f452-mios pic18f4520-mios pic18f4620-mios pic18f4685-mios
])
AC_MIOS_HOST_PREFIX
AM_INIT_AUTOMAKE([foreign])

# Checks for programs.
AM_CHECK_MIOS_TOOLCHAIN

# Checks for libraries.
AC_LIB_PREFIX

# Checks for header files.
AC_CHECK_HEADERS([pic18fregs.h], [], [
        AC_MSG_FAILURE([Can't compile without PIC header files])])
AC_CHECK_HEADERS([cmios.h], [], [
        AC_MSG_FAILURE([Can't compile without cmios.h])])

# Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
AC_C_VOLATILE
```

```
# Checks for library functions.

AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADERS([src/config.h])
AC_CONFIG_FILES([
        Makefile
        src/Makefile
])
AC_OUTPUT
```

Modify line 4 to be as follows :

4 |configure.in

```
AC_INIT(My One Encoder App., 0.1, http://www.example.com/, one-encoder)
```

First launch of the AutoTools

Let's tell AutoTools to make the first generation and install the missing files :

```
autoreconf --install
```

This will install the `compile`, `missing`, `install-sh`, `config.guess` and `depcomp` scripts. You don't need take any attention to them. There are here for the AutoTools to work correctly.

This will also generate the `configure` script and `Makefile.in`.

Modifying src/Makefile.am

Here is the content of `src/Makefile.am` :

1 |src/Makefile.am

```
## Process this file with automake to produce Makefile.in
bin_PROGRAMS = mios-skeleton
mios_skeleton_SOURCES = main.c # Add C headers and sources here

LDADD = -lcmios

CFLAGS = --disable-warning 85
CFLAGS += --fommit-frame-pointer
CFLAGS += --opt-code-speed
CFLAGS += --optimize-goto
CFLAGS += --optimize-cmp
CFLAGS += --optimize-df
```

```
CFLAGS += --obanksel=2

#CFLAGS += --keep-asm
```

Modify lines 2 and 3 of the file so that it looks like the following :

2 |src/Makefile.am

```
bin_PROGRAMS = one-encoder
one_encoder_SOURCES = main.c
```

By doing that we modified the name of the build application.

Building the application

Your application has now all the things needed to be build using the traditional build procedure. So let's see what are the options of the configure script :

```
./configure --help=short
```

This command shows all the options specific to your application. As you can see, there isn't any specific options (yet!).

So, for now, configure the application in a basic manner :

```
./configure --host=pic18f4685-mios
```

You might notice that by running the `configure` script, the `Makefile` files have now been generated.

Now build your application :

```
make
```

Check the `src/` folder's content. You will find there your `one-encoder.hex` file ready to be uploaded to your MidiBox!

Isn't it simple and easy ??

Using external libraries

Specifying a library :

5 |src/Makefile.am

```
LDADD = -lcmios
```

Checking for the availability of a header file at configure time :

20 |configure.in

```
AC_CHECK_HEADERS([cmios.h], [], [
        AC_MSG_FAILURE([Can't compile without cmios.h])])
```

Write about :

- Availability of HAVE_CMIOS_H in config.h
- Adding an –enable-xxx configure flag for an optional dependency (as in libcan for optional libdebug)

Adding C header and source files

3 |src/Makefile.am

```
one_encoder_SOURCES = main.c test.h test.c
```

Write test.h, test.c and call test.c from main.c

Write about :

- Use of _HEADERS instead of _SOURCES to distribute headers

Distribute data

1 |Makefile.am

```
## Process this file with automake to produce Makefile.in
SUBDIRS = src
dist_doc_DATA = README
```

Write about :

- Distribute a whole documentation directory like a single file

Skeleton reference

configure.in

Makefile.am

SUBDIRS variable

*_PROGRAMS variables

*_LIBRARIES variables

*_SOURCES variables

*_HEADERS variables

*_DATA variables

LDADD and LDFLAGS variables

CFLAGS variable

Distinctive main.c

Distinctive C tables

MIOS_ENC_TABLE

MIOS_MPROC_TABLE