Practical Auto-confiscation

Revision 1.2

Part-I

Portability

- Is always an issue! When you hear “not in our company” someone is:
  - Inexperienced – “It never happened to me”.
  - Naive – “it won’t happen to me”.
- Eventually some code will need porting:
  - Different processor (handheld, 64bit, etc.)
  - Different compiler (gcc is too slow, you must use the new Wizbang-7).
  - Another OS.
  - New/old version of the same OS (NPTL anybody?)
- Unix crowd has learned this ages ago... “MS-camp” are learning it now.

Strategy

- Sticking to standards will solve 90% of the problems:

  Formal standards - POSIX + SUS, IETF, W3C, OMG

  Portable free code - ACE, Qt/Gtk, etc.

- But those pesky 10% differences are not fun.
- We aim at the **“last-mile”** problem.

note

- If your project manager “selected” for you: `COM`, `MFC`, `ADO`, ...
- Than prepare to walk the full 100 miles :-(

Tactics

- Portability of both **code** and **build process**.
- Tough requirements:
  - Linear categorization (Linux/HPUX/Solaris/Windows) is naive.
    Life are multi-valued:
    $kernel \times processor \times compiler \times libraries$.
  - Should have minimal prerequisites on build host.
  - Non-interactive builds should be possible.
  - But user should be able to supply some options.
  - Extensible (future proof :-)

Example Scenario

Library interface `luser.h`

```
#ifdef  LUSER_H
#define LUSER_H

void shout(const char *msg);

#endif  /* LUSER_H */
```

Library implementation `luser.c`

```
#include <luser.h>
#include <stdio.h>

void shout(const char *msg)
{
  printf("Shouting on the luser: %s\n", msg);
}
```

A useful program `lart.c`

```
#include <luser.h>

int main()
{
  shout("Watch my differential SCSI cable!");
}
```

Library Problems

- How a shared library is built:
  `gcc -shared ...` But what if it's not gcc?
- The library file name:
  `libuser.so`? `libuser.sl`? `libuser.dll`?
- Pre-install usage:
  Debugging, testing, etc. Before committing.
- How the run-time loader will find it:
  `LD_LIBRARY_PATH`? `SHLIB_PATH`? `LIBPATH`? `PATH`?
- Installation procedures:
  Locations, `ldconfig`?

More Problems

- Writing a makefile is easy.
- Writing a **good** makefile is hard:
  - Maintenance targets (install, clean, distclean, dist, ...)
  - Dependency tracking (per-file please).
  - Recursive builds (and use the same make program).
  - User install options (paths, names, ...)
- Writing a portable makefile is harder (tools location, names, ...)

The Players

- Central tools:

`libtool`	For handling shared/static libraries.
`autoconf`	For build configuration (generates `./configure`)
`autoheader`	For code portability.
`automake`	Write makefiles for us.
...	Optional tools: `gettext`, regression testing, ...

- Each can be used separately but integrated operation is easier.
- They are not required on the final build host.
- The main input for `autoconf`, `autoheader` and `automake` is:

configure.ac [1]

[1] Used to be `configure.in` in older versions

A Draft of `configure.ac`

- `autoscan` parses our project and creates `configure.scan`

```
# Process this file with autoconf to produce a configure script.
AC_INIT(FULL_PACKAGE_NAME, VERSION, BUG_REPORT_ADDRESS)
AC_CONFIG_SRCDIR([luser.h])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.

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

# Checks for library functions.

AC_CONFIG_FILES([])
AC_OUTPUT
```

What in this file?

- `configure.ac` contains comments and `m4` macros:
- The basic process is:

  `configure.ac` $\Longrightarrow$ `autoconf` (`m4` processing) $\Longrightarrow$ `./configure`
- The first macro must be `AC_INIT...` which expands to:

```
#! /bin/sh
...
```

- The last macro must be `AC_OUTPUT` which creates the outputs.

`configure.ac` After Editing

- Since the example code is portable – no need for `AC_CONFIG_HEADER`[1]
- But we need a portable build:

```
# Process this file with autoconf to produce a configure script.
AC_INIT(lurser, 1.1, [oron@actcom.co.il])
AC_CONFIG_SRCDIR([luser.h])

# Checks for programs.
AC_PROG_CC

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

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

- We can run `autoconf` now...

[1] Let's forget about the `const` for a while...

First run of `./configure`

- `autoconf` generated `./configure`
- So we try it:

```
$ ./configure
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for an ANSI C-conforming const... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: error: cannot find input file: Makefile.in
```

- But something is missing...

Outputs of ./configure

- `./configure` won't generate outputs without inputs :-)
- The generation pattern is[1]:

  `Makefile.in` $\Longrightarrow$ `Makefile`
  `config.h.in` $\Longrightarrow$ `config.h`
  `foo``.in` $\Longrightarrow$ *`foo`*

- In our specific example we need only a `Makefile.in`.
- But instead of writing it – let's call `automake`.

[1] `config.h` is generated only if the `AC_CONFIG_HEADER` macro is used (not in our case).

`automake`

- The basic process is:

  `Makefile.am` $\Longrightarrow$ `automake` (`perl` processing) $\Longrightarrow$ `Makefile.in`

```
$ automake
automake: configure.ac: 'AM_INIT_AUTOMAKE' must be used
automake: no proper implementation of AM_INIT_AUTOMAKE was found,
automake: probably because aclocal.m4 is missing...
automake: You should run aclocal to create this file, then
automake: run automake again.
configure.ac: required file './install-sh' not found
configure.ac: required file './mkinstalldirs' not found
configure.ac: required file './missing' not found
automake: no 'Makefile.am' found or specified
```

- Problems:
  - `automake` needs some tools for its generated makefiles.
  - `./configure` must be told to check for these – `AM_INIT_AUTOMAKE`

Missing Macros

- Extending `autoconf`:
  - It can read macros from an `aclocal.m4` and `acsite.m4` files.
  - `aclocal.m4` is generated by the `aclocal` command.
- How does `aclocal` work?
  - It reads our `configure.ac`.
  - And fetches needed macros from a macro repository.
- In our case we:
  - Add the `AM_INIT_AUTOMAKE` macro to the first section of `configure.ac`.
  - Run `aclocal` to fetch it.

Missing files

- Some are required for proper operation.
- Others are meant to comply with GNU packaging rules[1].
- Created:
  - By running `automake` with the '`--add`' flag: `INSTALL`, `COPYING`, `mkinstalldirs`, `install-sh`, `depcomp`, ...
  - Manually: `README`, `AUTHORS`, `NEWS`, `ChangeLog`.
- So after running `automake -a`:

We are ready to learn about `Makefile.am`.

[1] We can force automake to ignore them by the `--foreign` option.

Makefile.am

- Is technically a makefile:
  - Hand crafted rules may be added if we wish.
  - Makefile syntax must be obeyed.
- But we normally only want to specify:
  - What to build? `PROGRAMS`, `SCRIPTS`, `LIBRARIES`, `MANS`, `DATA`, ...
  - Where to install it? `bin`, `sbin`, `lib`, `mans`, `data`, `sysconfig`, ...
- This is described via “specification variables” syntax:

```
where_WHAT = ...
```

A Complete and Trivial Example

- We have `hello.c` (assume we wrote `configure.ac`):

```
#include <stdio.h>

int main()
{
  printf("hello K&R\n");
  return 0;
}
```

- Here is our `Makefile.am`:

```
bin_PROGRAMS     = hello
```

- The bootstrap sequence is:

```
$ touch README AUTHORS NEWS ChangeLog
$ aclocal; autoconf; automake -a
```

Revisit the Original Example

- But we wanted to build `libuser` and `lart`...
- Here is `Makefile.am`:

```
bin_PROGRAMS            = lart

lib_LIBRARIES           = libuser.a
libuser_a_SOURCES       = luser.c
include_HEADERS         = luser.h

lart_LDADD              = libuser.a
```

  - "bad" characters in target name are replaced with an '_'
  - The `HEADERS` target is meant to install headers.
  - Other (internal) headers should be added to the `SOURCES` directive.
  - The `LDADD` adds parameters and flags to the link phase.

What About Dynamic Libraries?

- We need to use `libtool` to have portable solution.
- With `autoconf` + `automake` all we need is:
  - Change the `LIBRARIES` into `LTLIBRARIES`.
  - Rename each `libfoo.a` into `libfoo.la`.
  - Run `libtoolize` to bring required tools (`ltmain.sh`, `config.guess`, `config.sub`).
  - Add `AC_PROG_LIBTOOL` to `configure.ac`.
- Now for a test:

```
aclocal; autoconf; automake
./configure
make
make dist
make distcheck
```

Part-II

And Now to Something Completely Different

- Let's *auto-confiscate* a non-trivial software package – OpenGUI[1]:
  - A fast, non-X11, GUI library.
  - Many platform – Linux, Solaris, Windows, QNX, DOS.
  - On DOS/Windows Several compilers – Cygwin, Watcom, Visual-c, Borland.
  - Build at three color depths (compile time): 8bpp, 16bpp, 32bpp.
  - Optionally interface with Mesa (OpenGL) using some glue code.
- Current build environment for library only (without examples):

13 `makefile.*` + 6 `*.mak` + 1 `*.bat`

[1] OpenGUI home http://www.tutok.sk/fastgl/

Crisis? What Crisis?

References

[1] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor, *GNU Autoconf, Automake, and Libtool*. SAMS, October 2000.