LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-PROC-209017

autokonf - A Configuration Script Generator Implemented in Perl

J. F. Reus

January 14, 2005

NECDC 2004
Livermore, CA, United States
October 4, 2004 through October 7, 2004

Proceedings from the NECDC 2004

autokonf
A Configuration Script Generator Implemented in Perl (U)

James F. Reus

reus@llnl.gov

Lawrence Livermore National Laboratory, Livermore, CA 94551

This paper discusses configuration scripts in general and the scripting language issues involved. A brief description of GNU autoconf is provided along with a contrasting overview of autokonf, a configuration script generator implemented in Perl, whose macros are implemented in Perl, generating a configuration script in Perl. It is very portable, easily extensible, and readily mastered. (U)

Introduction

The usefulness of a software package is not simply a matter of how well it solves a particular problem, its efficiency or its reliability. An important factor is how well it ports to a new site or to new or different hardware. If the package is difficult to build on a platform of interest then its usefulness is diminished. For example *ALE3D* is an engineering/physics code that may be configured to use a number of external solver packages: libraries tailored to the solution of systems of equations. If ALE3D is to use a particular solver then the solver library must be easily built and installed on all of the platforms supported by ALE3D. All additional libraries required by this solver package must also be easily built and installed. ALE3D itself must itself be easily configured to use this solver package if it is needed and can be built and installed. A solver package that is not portable or is difficult to build and install will have limited use with the code at best.

Libraries have particular configuration needs not generally faced by applications. For an application to be useful, the end user only cares that it can be built and installed properly. Users of libraries are generally application developers. They not only care that the library can be built and installed, but also the details of how was it built: 32- or 64-bit, debug or optimized. If it is a C++ library, it is important to know which compiler was used. For a library to be usable it must be compatible with not only the compiled application code, but also every other library used to form the application. The configuration process for libraries must allow for the configuration-time specification of all of these important factors.

Background

The process of building and installing a software package from source generally proceeds in the following fashion:

1. *Unwrap* - The distribution is unwrapped in a temporary working area. Most open source software is distributed in the form of a "tarball": a directory tree that was collected into a single file using an application such as *tar*, and compressed using an application such as *gzip*.
2. *Configure* - A setup or configuration script is run. This identifies the details of the target system, how source code is to be compiled, and which compilers are to be used. The availability of, and location of, other software packages that are required is specified or detected. Other details such as the enabling of optional features, may be done with command line options.
3. *Make* - The software package is constructed. Each of the source files is compiled, libraries are assembled and executables are formed. This is generally choreographed with a utility such as *make* or *build*.
4. *Install* - The software package is installed. Administrative rules at different sites may require different installation locations so the installation directory must be a configurable parameter. As with the previous step, the installation process is typically controlled by make or build.

These steps form the classic unwrap-configure-make-install cycle. Of particular interest in this paper is the second step: configuration.

Most open source software packages use a configuration script to analyze the target system hardware and software and adjust or generate makefiles and header files to suit the detected or specified system environment. Unfortunately, some important mathematical and scientific software packages handle the configuration step by requiring the person performing the installation to edit makefiles or make include files, uncommenting appropriate sections or inserting the required information as desired. Other packages require the installer to select from a list of known configurations. New platforms or environments may require a new configuration file to be developed.

Considerable effort may be required to produce configuration scripts that:

- Detect hardware characteristics such as processor type, data size and alignment, byte ordering and numeric precision.
- Detect software characteristics such as the host operating system type and version. Is threading supported? Is the file system case sensitive?

  Generally the processor type, operating system type, and version constitute the "platform" type.
- Detect the host name and domain.

  Generally the host name and domain constitute the "site".

- Decide which compilers are to be used and the language features supported. Use GNU or vendor compilers? Which compiler version? Is 64- or 32-bit code to be generated?

  C++ code compiled with one compiler is generally incompatible with code compiled with a C++ compiler from a different vendor. Sometimes different versions of a C++ compiler may produce incompatible code.

- Determine if the compiler supports required language features. Does the C compiler support a Boolean type? Is long long supported? Does the C compiler support new style ANSI headers?
- Detect the existence and location of header files and libraries of interest. Where is SILO installed? Is HDF5 available?

  Note that a specific version of a library may be important. Code that was compiled with the header file of one version of a library is frequently incompatible with code compiled with the header file of another.

- Detect the number of processors available. Does the system support a "switch" allowing for fast parallel communication?
- Provide mechanisms for the application of parameters appropriate to the site and platform type.
- Permit overriding detected characteristics such as using a configuration-time specified compiler set rather than the detected set.

  This is particularly important for libraries which may have to be built and installed using several compilers. Also note that certain tools such as debuggers may require the code to be built with certain compilers.

- Allow certain optional features to be enabled or disabled at configuration time. Is the package to be built parallel or serial, debug or production? Are design-by-contract features such as pre- and postcondition checks to be enabled?
- Support the configuration-time specification of the location of headers and libraries for other software packages.
- Generation of header files containing configuration specific information and the insertion of additional configuration information in makefiles[1].

All important system hardware characteristics and software features should be automatically detected unless an appropriate site-specific file specifies the characteristic or a configuration-time override is supplied. Most importantly every software package should be configurable to use specific compilers and options, to use particular versions of headers and libraries found at indicated locations, and in the end to be installed in specified directories.

Implementing such a setup or configuration script can be a project in its own right.

---

[1] This is frequently done by translating files such as *makefile.in* to *makefile* by substituting notations resembling @NAME@ with a string appropriate to a detected or specified characteristic.

GNU autoconf

Rather than going to the considerable effort to write such complex configuration scripts, the authors of many open source software packages use GNU autoconf to generate configuration scripts. This tool relies on the fact that most functions of a configuration script are common to all software packages, only differing in detail. GNU autoconf takes a configuration specification file as its input, and generates a configuration script as its output.

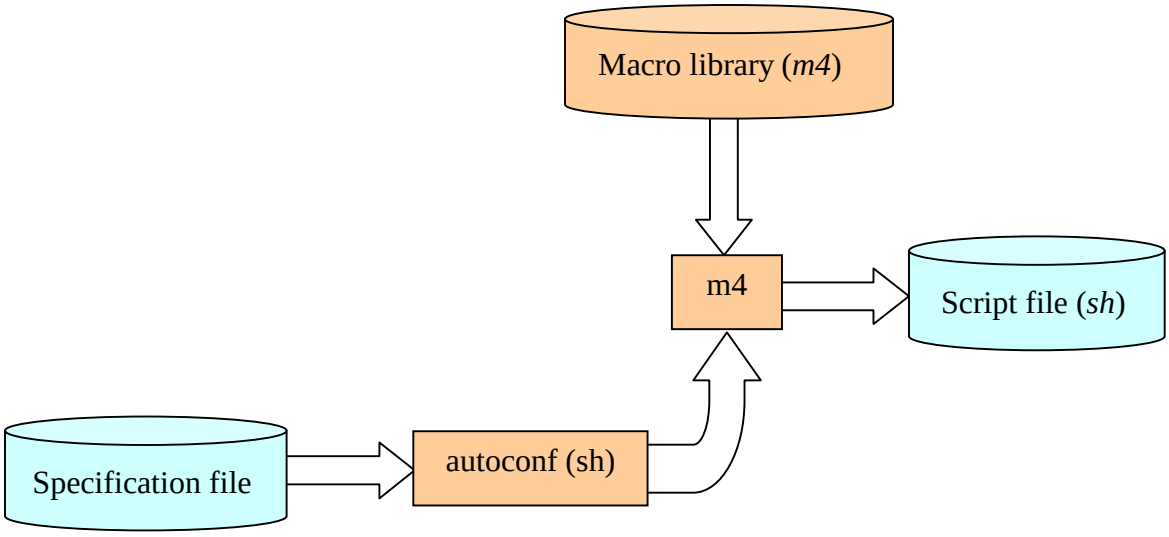

Fig. 1

The input specification file[2] is, in theory, a list of characteristics, features and external packages that are of interest. In reality, it is a text file containing a mix of shell code and macro references (the tests) that are used to generate the desired output configuration shell script[3]. Note that the autoconf utility is actually a fairly small Bourne-shell script that simply hands-off the real work to the GNU m4 macro processor.

GNU autoconf has been quite successful and is used by many open source projects in the Linux/UNIX community. It has an extensive library of macros[4], it generates a Bourne-shell script and many developers in the Linux/UNIX community are familiar with Bourne-shell scripting.

---

[2] A GNU autoconf specification file is generally named *configure.in*.

[3] GNU autoconf generally produces a Bourne-shell script named *configure* as its output.

[4] GNU autoconf macros are used to implement the various tests used to detect hardware and software system characteristics that are of interest.

While the use of autoconf to generate configuration scripts is a great improvement over selecting/editing makefiles or handwriting configure scripts, it has a number of shortcomings:

- **GNU autoconf macros are written using m4**

  GNU m4 is a powerful macro processing language but few developers are fluent in m4[5]. Software package developers end up implementing new tests as shell code rather than implementing new macros. For example only 101 of 1461 lines (7%) of SILO's configure.in file actually contain autoconf macros. Much of the remainder is handwritten shell code implementing new tests.

- **GNU autoconf generates a UNIX-centric configuration script**

  The configuration script generated by autoconf is in implemented in Bourne-shell. While every UNIX-like platform supports this shell dialect, the Bourne-shell is not generally found on non-UNIX platforms such as Windows. Even then, Bourne-shell interpreters ported to Windows platforms have not been completely successful and frequently have problems when dealing with large complex scripts. Such Bourne-shell interpreters frequently have trouble dealing with the command line requirements of native Windows utilities[6]. Software package developers often resort to using GNU autoconf to generate a configuration script for UNIX-like platforms and a separate *project* file for windows platforms.

- **Little platform- or site-specific support**

  Developers generally resort to handwritten shell code to deal with platform- and site-specifics.

The fundamental difficulty with GNU autoconf is its reliance on the Bourne-shell. The Bourne-shell scripting language lacks the necessary internal features to permit the configuration script to easily stand on its own. It relies on additional external utilities to do much of the necessary work. Availability of these utilities cannot be depended upon outside of the family of UNIX-like platforms and must be installed as a prerequisite[7]. Even among the various UNIX and Linux environments where such external utilities may be found, they often differ in detail, have different options and frequently handle borderline cases in different fashions.

---

[5] GNU autoconf actually requires GNU m4. While this dialect of m4 isn't massively different from the standard, it has some subtle semantic differences and extensions required by autoconf.

[6] Native Windows utilities generally use the forward slash (`/`) as an option indicator and the backslash (`\`) as a pathname component separator. This doesn't generally work well with the Bourne-shell, which uses the backslash as an escape character.

[7] Many of these utilities are available for Windows-like platforms as components of the open source *cygwin* project.

There are several desirable characteristics of a configuration script generator:

1. The generator should be implemented using a popular language. This language should be well suited to simple parsing and script code generation.
2. The configuration script language should have wide operating system and hardware support.
3. The configuration script should limit its reliance on additional utilities. The language used should supply the necessary operations as intrinsics.
4. The script generator and the generated script should be the same language. While not a requirement, this can simplify the implementation and maintenance.

Scripting Languages

When generating a configuration script there is a choice of scripting languages available. There are several characteristics that are needed: broad user base, portability, and powerful intrinsic operations. An obscure language may be ideal from a simply implementation basis but would be a poor choice as most potential users would face additional training. To maximize the size of this population, a popular scripting language should be chosen such as: Bourne-shell, C-shell, Perl, Python, or Visual Basic (VB).

Both Bourne-shell and C-shell are UNIX-centric and generally lack the intrinsic operations needed for this task. They depend largely on additional utilities such as *cp*, *mv*, *rm*, *sed* or *test* to do real work. Visual Basic has the necessary powerful built-in operations but is limited to Windows-like platforms.

To achieve wide portability and broad developer base the choice is rather limited: Perl or Python. Both Perl and Python are well known and supported by a broad range of operating systems and hardware. They both have powerful intrinsic operations and don't rely on additional external utilities to do the needed work. Perl is of particular interest since it is well designed for parsing, is easily generated[8], and is very widely available due to its popularity with those that write web CGI scripts.

Unfortunately the implementation of GNU autoconf makes generation of a Perl script quite difficult. All of the macros are implemented with Bourne-shell generation in mind. Since in practice GNU autoconf is really just its macros, modifying GNU autoconf to generate Perl would actually be a re-write. Tempting, but there is another problem: the input specification files used by GNU autoconf are actually Bourne-shell code with embedded autoconf macro calls.

The connection between GNU autoconf and the Bourne-shell is quite difficult to break.

[8] Python has interesting code formatting requirements that can make code generation, particularly through macro expansion, rather difficult.

Autokonf - An Alternative

Like GNU autoconf, autokonf is a configuration script generator. It is used in much of the same way: it takes a configuration specification file as its input and using a macro expansion process it generates a configuration script as its output.

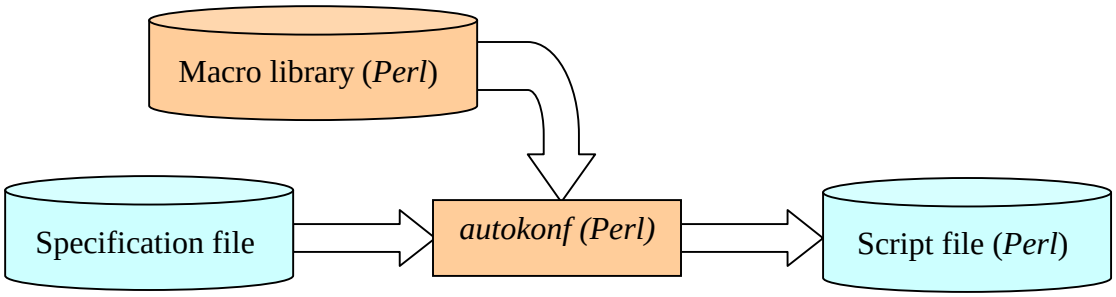

Fig. 2

Normally autokonf expects to be given a specification file named *konfigure.in* and will generate a script file named *konfigure*. These names were chosen to permit coexistence with the input and output file names used by GNU autoconf but if given an input specification file named *configure.in* autokonf will generate a script file named *configure*.

Autokonf is implemented in simple Perl with no reliance on specialized modules. The input specification file is Perl with embedded macro references. The generated configuration script is in Perl using no specialized modules. The result is a very portable system; Perl runs everywhere, with no need for a special environment such as Cygwin or MKS for Windows-like platforms.

New Features

Autokonf provides a number of important features:

- A hierarchical approach is used when searching for both site and platform specific files.
- Improved hardware detection. For example the generated configuration script can distinguish between Pentium 3 and 4 processors.
- A mechanism to control the compiler names and order the generated configuration script is to try using. For example some applications might prefer to use GNU compilers if available but others might prefer commercial compilers. Note that this is specified independently for each language.
- Search paths to consider when looking for packages.

Implementation Details

The autokonf script reads the input specification file scanning it for macros. The parsing process is simplified by a simple rule:

All autokonf macros are upper case identifiers that start with `AK_`.

The configuration script is scanned in multiple passes. The first pass looks for and extracts *argument processing* and *section* macros.

All argument processing macros start with `AK_ARG_`.

All section macros start with `AK_SECTION_`.

Argument Processing Macros

Argument processing macros are handled in a rather special fashion, as they are not expanded in place. Rather they are removed from where they are found and additional argument processing code is inserted at the appropriate place in the generated script file. Similarly the section macros are not expanded in place but simply partition the text into regions that indicate where the subsequent code is to land in the generated script. The following argument processing macros are currently supported:

AK_ARG_ENABLE(“*feature***”, “**help string**”);**
AK_ARG_DISABLE(“*feature***”, “**help string**”);**
AK_ARG_WITH(“*package***”, “***help string***”);**
AK_ARG_WITH(“*package***=[***pathname, …***]”, “***help string***”);**
AK_ARG_WITHOUT(“*package***”, “***help string***”);**
AK_ARG_WITHOUT(“*package***=[***pathname, …***]”, “***help string***”);**

The `AK_ARG_ENABLE` and `AK_ARG_DISABLE` macros are used to create feature control options in the configuration script: `--enable-`*feature*, which is used to enable some feature, and `–-disable-`*feature*, which is used to disable some feature. Some popular features include: debug, production, parallel, serial, shared, and/or static. Note that the `AK_ARG_ENABLE` and `AK_ARG_DISABLE` macros have some interesting properties as the order of specification can control default behavior of the generated configuration script. For example, the following *konfigure.in* fragment:

```
AK_ARG_DISABLE(“shared”,”Produce static library.”);
```

with no corresponding `AK_ARG_ENABLE(“shared”,…)` reference not only results in a configure script that supports an `–-disable-shared` option but also has the “shared” feature enabled by default. If both macros are used, then the order indicates the default:

```
AK_ARG_DISABLE(“shared”,”Produce static library.”);
AK_ARG_ENABLE(“shared”,”Produce dynamic library.”);
```

then a script is produced that supports both `–-disable-shared` and `–-enable-shared` options and has the “shared” feature enabled by default. However if the order is reversed:

```
AK_ARG_ENABLE(“shared”,”Produce dynamic library.”);
AK_ARG_DISABLE(“shared”,”Produce static library.”);
```

then the “shared” feature will be disabled by default. In summary: The first `AK_ARG_ENABLE` or `AK_ARG_DISABLE` macro for a particular feature sets the default state

of the feature to the opposite condition. If `AK_ARG_ENABLE` for a particular feature is encountered first, then the feature is disabled by default, but if the `AK_ARG_DISABLE` macro is encountered first, then the feature is enabled by default.

The `AK_ARG_WITH` or `AK_ARG_WITHOUT` macros operate in a similar fashion: The first `AK_ARG_WITH` or `AK_ARG_WITHOUT` macro for a particular package sets the default state of the package to the opposite condition. . If `AK_ARG_WITH` for a particular package is encountered first, then by default the package will not be used, but if the `AK_ARG_WITHOUT` macro is encountered first, then by default the package will be used.

The "default" behavior of the `AK_ARG_ENABLE`, `AK_ARG_DISABLE`, `AK_ARG_WITH` and `AK_ARG_WITHOUT` macros may seem backwards but it does make sense. For example, if you provide for an `--enable-`*feature* option using an `AK_ARG_ENABLE` option but didn't use `AK_ARG_DISABLE` to provide for a `--disable-`*feature* option, then the user of the configuration script has a way to enable the feature but no way to disable it, so it makes sense to have it disabled by default.

Section Macros

Section macros are used to locate subsequent code at certain points in the generated configuration script. The following section macros are currently supported:

```
AK_SECTION_EPILOG;
AK_SECTION_PROLOG;
AK_SECTION_TESTS;
```

Prolog code is performed early in the configuration processes, after the hostname and platform type have been determined and cached results have been loaded, but before site and platform specific files have been loaded and compilers to be used for testing have been determined. Normally the bulk of the work is done by code in the tests section. Epilog code is performed when the configuration process is nearly complete, right before cleanup. The default is of course to locate the macro expanded text in the "tests" part of the generated configuration script, as most applications do not require prolog or epilog sections.

Normal Macros

Additional passes made over the input are used to expand "normal" macros. Unlike the argument processing and section macros, normal macros are not "built-into" the autokonf script. At this time autokonf implements a very simple form of macro processing; it simply translates macro references to function calls[9].

The autokonf script repeatedly scans the text for a macro name: any upper case identifier starting with AK_. When the autokonf script encounters such a macro name it replaces the macro reference with a function call and picks-up the function definition

[9] Earlier versions of autokonf implemented a more general form of macro expansion but it proved more difficult to use and added little power to satisfy most of the perceived need.

from a text file. Since macro references are translated into function calls they should only be used in the same syntactical context as a Perl function. The details are as follows:

1. The macro reference is replaced with a call to a function with the same name but in lowercase. The macro reference:

   ```
   AK_BANANA("peel");
   ```

   Is replaced with the function call:

   ```
   ak_banana("peel");
   ```

   Note that the parameters are not modified. If the macro reference has no parameters then the parentheses are optional. If there are none then the autokonf will simply add empty parentheses.

   For example the macro reference:

   ```
   AK_CONST;
   ```

   Is replaced with the function call:

   ```
   ak_const();
   ```

2. If this is the first time the particular macro has been encountered then the autokonf script looks for a text file whose name is simply the macro name with the leading AK_ stripped away and shifted to lowercase. So when dealing with the macro:

   ```
   AK_CONST;
   ```

   The autokonf script will attempt to locate a text file named *const*. Autokonf will append the contents of this text file to the end of the text being scanned. Note that the function implementing the macro may itself contain macro references.

The following figure graphically illustrates how a normal macro is "expanded":

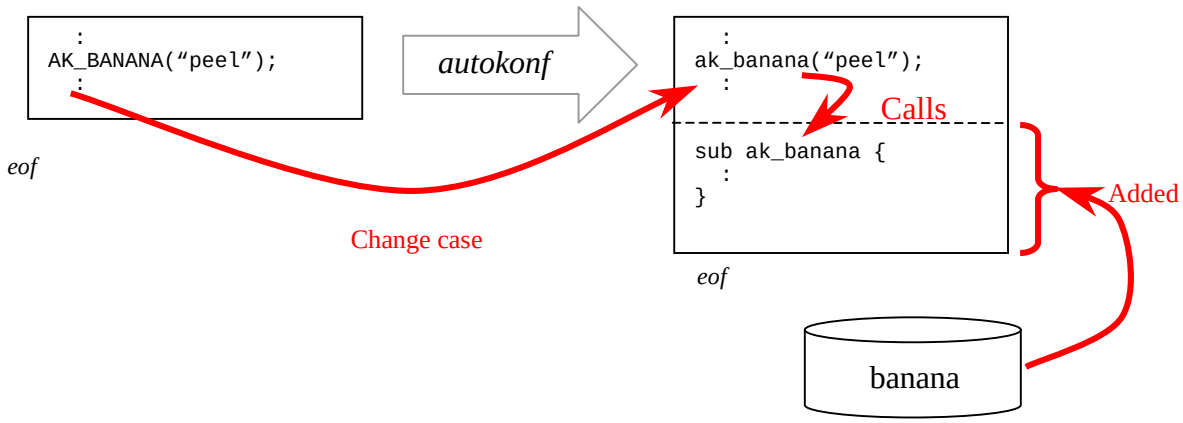

Fig. 3

The macro expansion process is repeated until no identifiers starting with `AK_` are found. Note that the expansion process is cyclic since newly introduced functions may (an frequently do) contain macro references. The expansion process is guaranteed to terminate as the only way to introduce new macros references is when a macro is expanded for the first time and the implementing function is added. Since there is a finite number of macros available there will be a finite number of such functions added and so a finite number of calls to expand. The autokonf tool need not check for macro expansion cycles.

The macro expansion process may seem simple, and indeed it is, but it has proven powerful enough to produce very capable configuration scripts. The build utility, a very capable replacement for make, uses a configuration file generated by autokonf. Similarly a configuration script has been produced using autokonf for ALE3D, a powerful parallel engineering/physics code.

The simplicity of the scheme has proven a benefit as the autokonf development process has been plagued with few bugs. It has also proven easy to understand, maintain and to extend.

Normal Macros Supplied with autokonf

At this time over 100 predefined “normal” macros are supplied with autokonf:

AK_C_ANDORNOT;
AK_C_BIGENDIAN;
AK_C_CHAR_UNSIGNED;
AK_C_CLEANUP;
AK_C_CONST;
AK_C_DASHED_TYPEOF;
AK_C_DUMP_SRC_LOG;
AK_C_ENDIAN;
AK_C_INCLUDE;
AK_C_INCLUDE(*filename*, … **);**
AK_C_INLINE;
AK_C_LONG_DOUBLE;
AK_C_PROTOTYPES;
AK_C_STRINGIZE;
AK_C_TYPEOF;
AK_C_USED;
AK_C_WORKS;
AK_CACHE(*variable*, … **);**
AK_CACHE_SAVE;
AK_CASE_SENSITIVE_FILESYSTEM;
AK_CHECK_FILE(*pathname*, … **);**
AK_CHECK_HEADER(*pathname*, … **);**
AK_CHECK_HEADER(*pathname*, … **,[***function*, … **]);**
AK_CHECK_LIBRARY(*name***);**
AK_CHECK_LIBRARY(*name*,*function***);**
AK_CHECK_LIBRARY(*name*,**[***pathname*, … **]);**
AK_CHECK_LIBRARY(*name*,*function*,**[***pathname*, … **]);**
AK_CHECK_PROG(*name***);**
AK_CHECK_SIZEOF(*type***);**
AK_CHECKING(*message***);**
AK_CK_SEVERE;
AK_JAVA_DUMP_SRC_LOG;
AK_JAVA_USED;
AK_JAVA_WORKS;
AK_LANG_C;
AK_LANG_CXX;
AK_LANG_F77;
AK_LANG_F90;
AK_LANG_RESTORE;
AK_LANG_SAVE;
AK_LIBRARY_PATH;
AK_LIBRARY_PATH(*pathname*, … **);**
AK_MAX_INT;
AK_MAX_LONG;
AK_MAX_SHORT;
AK_MIN_INT;
AK_MIN_LONG;
AK_MIN_SHORT;
AK_MSG_CHECKING(*message***);**
AK_MSG_CHECKING(*message*,*result***);**
AK_MSG_RESULT(*result***);**
AK_OUTPUT(*filename***);**
AK_PARALLEL_SWITCH;
AK_PARALLEL_SWITCH(*pathname*, … **);**
AK_PARALLEL_USED;
AK_PARAMDIR;
AK_RETADROFS;
AK_RUN_COMMAND(*command***);**
AK_SEMUNION;
AK_STACKDIR;

```
AK_CK_TYPE(type, ... );
AK_CK_TYPES;
AK_CPP_ELIF;
AK_CXX_CLEANUP;
AK_CXX_DUMP_SRC_LOG;
AK_CXX_DYNAMIC_CAST;
AK_CXX_USED;
AK_CXX_WORKS;
AK_DUMP_LOG;
AK_F77_CLEANUP;
AK_F77_DUMP_SRC_LOG;
AK_F77_USED;
AK_F77_WORKS;
AK_F90_CLEANUP;
AK_F90_DUMP_SRC_LOG;
AK_F90_USED;
AK_F90_WORKS;
AK_FUNC_CLOSEDIR_VOID;
AK_GNU_C;
AK_H_DEFINE(name);
AK_H_DEFINE(name,value);
AK_H_DEFINE_NO_PREFIX(name);
AK_H_DEFINE_NO_PREFIX(name,value);
AK_H_UNDEF(name);
AK_H_UNDEF_NO_PREFIX(name);
AK_HEADER_DIRENT;
AK_HEADER_STAT;
AK_HEADER_STDBOOL;
AK_INTEL_C;
AK_SUBST(name);
AK_SUBST(name,variable);
AK_SUBST_FILE(name,pathname);
AK_TRANSFORM(filename);
AK_TRANSFORM(filename,filename);
AK_TRY_C_COMPILE;
AK_TRY_C_COMPILE(filename);
AK_TRY_C_LINK;
AK_TRY_C_LINK(objfile, ... );
AK_TRY_C_RUN;
AK_TRY_COMPILE;
AK_TRY_CXX_COMPILE;
AK_TRY_CXX_COMPILE(filename);
AK_TRY_CXX_LINK;
AK_TRY_CXX_LINK(objfile, ... );
AK_TRY_CXX_RUN;
AK_TRY_F77_COMPILE;
AK_TRY_F77_COMPILE(filename);
AK_TRY_F77_LINK;
AK_TRY_F77_LINK(objfile, ... );
AK_TRY_F77_RUN;
AK_TRY_F90_COMPILE;
AK_TRY_F90_COMPILE(filename);
AK_TRY_F90_LINK;
AK_TRY_F90_LINK(objfile, ... );
AK_TRY_F90_RUN;
AK_TRY_LINK;
AK_TRY_RUN;
AK_TYPE_PID_T;
```

Most of the macros implement tests. They typically generate a source file, compile, link, and sometimes execute it[10]. In this fashion some characteristic of the hardware or software environment, or even a compiler characteristic may be detected. A different sort of macro, such as `AK_C_INCLUDE`, alter subsequent tests, in the case of the `AK_C_INCLUDE` macro, a specified header file is to be included in all subsequent tests using the C compiler.

Some macros such as `AK_H_DEFINE` and `AK_H_UNDEF` control how a detected characteristic is to be placed in a generated header file[11]. Macros such as `AK_SUBST` specify how the value of a configuration script variable is to replace an `@NAME@` construct when certain files[12] are transformed at the end of the configuration process.

Writing New Macros

The simple form of macro expansion used by autokonf was chosen to make it easy to implement new tests. All of the more than 100 normal macros supplied with the current revision of autokonf use the simple macro expansion mechanism described above. To

[10] Not all tests require the execution of a test program. A number of them simply require the source file to be compiled.

[11] This header file is of course generated by the configuration script and is generally named *config.h* or *konfig.h*.

[12] `AK_TRANSFORM` macros are used to specify the files to be transformed.

implement a new test a developer only needs to implement a Perl function which when called examines its parameters and performs the test as indicated.

For example, consider the `AK_C_CONST` macro that tests the C compiler for support of the const qualifier. This macro is typical of a test without parameters. A source file is generated and compiled, if successful then it can be inferred that the C compiler supports the *const* qualifier.

```
sub ak_c_const {
   my @params = @_;
   if (0 <= $#params) {
      print STDERR "$AK_scriptName: too many parameters for AK_"."C_CONST macro \n";
      exit 1;
   }
   $AK_macDepth += 1;
   {  my $how = "";
      my $ok  = 0;
      AK_MSG_CHECKING("   Checking for C \"const\" qualifier");
      if ( ! defined $ak_c_const_isSupported) {
         $ak_c_const_isSupported = 0;
         print AK_logFile __FILE__,":",__LINE__,"[",AK_whoAmI(),"]",
                          " -- generate code testing for \"const\"\n";
         if ( ! open(AK_srcFile,"> konftest.c") ) {
            print AK_logFile __FILE__,":",__LINE__,"[",AK_whoAmI(),"]",
                             " -- can't create/write \"konftest.c\"\n";
         }
         else {
            print AK_srcFile "/* Generated by macro: AK_C_CONST */\n";
            if (0 <= $#AK_includeFiles) {
               my $i = 0; foreach $i (0 .. $#AK_includeFiles) {
                  my $headerFileName = $AK_includeFiles[$i];
                  print AK_srcFile "#include <$headerFileName>\n";
               }
               print AK_srcFile "\n";
            }
            print AK_srcFile "const int zero = 0;\n";
            close AK_srcFile;
            $ok = 1;
         }
         if ($ok) {
            if (AK_TRY_C_COMPILE) {
               $ak_c_const_isSupported = 1;
            }
            else {
               AK_C_DUMP_SRC_LOG;
            }
            AK_C_CLEANUP;
         }
      }
      else {
         $how = " (known)";
      }
      if ($ak_c_const_isSupported) {
         AK_H_DEFINE("SW_C_with_const");
         AK_H_UNDEF("SW_C_without_const");
         AK_MSG_RESULT("supported$how");
      }
      else {
         AK_H_UNDEF("SW_C_with_const");
         AK_H_DEFINE("SW_C_without_const");
         AK_H_DEFINE_NO_PREFIX("const");
         AK_MSG_RESULT("NOT supported$how");
      }
      AK_CACHE('$ak_c_const_isSupported');
   }
   $AK_macDepth -= 1;
   return $ak_c_const_isSupported;
}
```


Of course the actual implementation should be well commented.

Reus, J. F.

An Example Input Specification File

The following is a realistic example of an input configuration file; in fact it is the file from which the configuration script for the build utility is generated.

```
if ( ! AK_C_WORKS) {
   AK_announceMsg("Requires working C compiler and linker");
   exit 1
}
if ( ! AK_C_PROTOTYPES) {
   AK_announceMsg("Requires C function prototype support");
   exit 1
}
if ( ! AK_CPP_ELIF) {
   AK_announceMsg("Requires C preprocessor support for \"#elif\" directive");
   exit 1
}

if ($AK_enabled_debugMode) {
   $CC_SER_DBG_FLAGS = "$CC_SER_DBG_FLAGS –DBUILD_DBG";
}

if ( ! AK_CHECK_HEADER("stdio.h")) {
   AK_announceMsg("Require stdio.h");
   exit 1
}
if ( ! AK_CHECK_HEADER("sys/types.h")) {
   AK_announceMsg("Require sys/types.h");
   exit 1
}
if (AK_CHECK_HEADER("iso646.h")) {
   AK_C_INCLUDE("iso646.h");
}
if (AK_CHECK_HEADER("inttypes.h")) {
   AK_C_INCLUDE("inttypes.h");
}
else {
   if (AK_CHECK_HEADER("stdint.h")) {
      AK_C_INCLUDE("stdint.h");
   }
}
AK_CHECK_HEADER("sys/param.h");
AK_CHECK_HEADER("sys/sysctl.h");
AK_CHECK_HEADER("unistd.h");
AK_CHECK_HEADER("stdlib.h",["abort","malloc","free"]);

AK_CK_TYPES;
AK_CK_TYPE("size_t");
AK_CK_SEVERE;
AK_C_TYPEOF;
AK_C_DASHED_TYPEOF;
AK_C_CONST;
AK_C_ANDORNOT;

AK_SUBST("BinDir","AK_pathPrefixBin");
AK_SUBST("IncludeDir","AK_pathPrefixInclude");
AK_SUBST("LibDir","AK_pathPrefixLib");
AK_SUBST("HtmlDir","AK_pathPrefixHtml");
$Html1Dir = "$AK_pathPrefixHtml/html1"; AK_SUBST("Html1Dir");

AK_OUTPUT("./konfig.h");
AK_TRANSFORM("makefile.in","makefile");

AK_CACHE_SAVE;
```

Of course the actual file contains helpful comments. These have been removed so that the contents fill fit a single page.

The Generated Perl Script

To provide some organization and to simplify code generation the Perl configuration script generated by autokonf is arranged into a number of parts:

Part 0 – Basic initialization.
1 – Initialization of defaults implied by any `AK_ARG_` macros used.
2 – Command line argument processing.
3 – Extracting certain variables from the environment.
4 – Fundamental tests: Is this a UNIX- or a Windows-like system?
5 – Test shell and Perl interpreters.
6 – Select a C compiler.
7 – Identify host name and platform type.
8 – Load the *cache file* if one exists.
9 – The prolog code (if `AK_SECTION_PROLOG` macro was used).
10 – Source the appropriate site and platform specific files.
11 – Select compilers and set various compiler variables.
12 – Deal with path prefix.
13 – The macro processed testing code from *konfigure.in* including all inserted macro functions.
14 – Code to emit the *konfig.h* file as specified by `AK_OUTPUT` macro.
15 – Perform file transformations replacing all `@NAME@` constructs in files specified by `AK_TRANSFORM` macros.
16 – Copy directory tree if "out-of-directory" configure is being performed.
17 – The epilog code (if `AK_SECTION_EPILOG` was used).
18 – Cleanup.
19 – Public built-in functions.
20 – Built-in private functions.

Note that most of the parts are automatically generated by autokonf to deal with problems common to all configurations scripts. They are of course generated with the particular configuration issues in mind. For example generation of Perl code for the selection of a C++ compiler and setting of C++ specific variables in parts 3 and 11 and processing of C++ specific options in part 2 is only done if macros using C++ are encountered in the specification file scanned by autokonf. Parts 9, 13, and 17 are drawn directly from the specification file. Part 13 is where macro processed testing code lands along with the appropriate functions implementing the macros.

Generated Files

In previous sections autokonf was described as a configuration script generator that generates a single output file: a rather complex and portable Perl script. When given a specification file such as *configure.in* as input, autokonf actually generates three output files:

configure	A Bourne-shell script; uses Perl to execute the *configure.pl* script.
configure.exe	A WIN32 binary executable; uses Perl to execute the *configure.pl* script.
configure.pl	The actual Perl configuration script as described in this document.

This is done so that the configuration script may be invoked in the same fashion on different platforms. For example the command:

```
configure –-enable-evpc
```

should work on a Windows platform just as well as on a UNIX or Linux platform. The issue is how the system is to recognize that we want to use the Perl interpreter to run the generated script. On a UNIX platform we can simply use a “sha-bang” line at the start of the script to indicate that pathname of the interpreter to use.

```
#!/usr/bin/perl
```

The trouble is that the Perl interpreter is not installed in the same place on all systems and the pathname must be given accurately. A number of schemes may be used to allow the interpreter to be located using the PATH environment variable such as:

```
#!/bin/sh
perl –x –S $0 “$@”
exit
# 4 \     These lines are here so line
# 5  \    numbers reported by Perl will
# 6   \   be off by exactly 10 (the line
# 7    >  numbers are wrong because of
# 8   /   the portable way we are executing
# 9  /    this script).
#10 /
#!/usr/bin/perl
```

However this scheme only works on UNIX-like systems since it relies on the Bourne-shell to find the Perl interpreter (note the first line). Windows-like platforms generally rely on filename extensions along with “associations” to determine how a file is to be executed.

Rather than depend on the host system having the proper association a different scheme is used by autokonf. The Perl configuration script is invoked by a “front-end” supplied by autokonf. On a Windows-like platform a command such as “configure” is generally expected to be a program in a file named *command.exe*. Actually the situation is a bit more complex involving the PATH variable and associations but if a file named

configure.exe is located in the current directory or with a specified path then it will be executed. On a UNIX-like platform a command such as *configure* is generally expected to be implemented as a file named *configure*. Autokonf exploits this fundamental difference by providing two front-end files: *configure* – a Bourne-shell script used by UNIX-like systems to invoke the Perl interpreter on the real configuration script and *configure.exe* a binary executable used on Windows-like systems to invoke the Perl interpreter.

Future Work

The autokonf configuration script generator is still in its fairly early stages of development. While it is actually usable as it currently stands there remains more work to be done:

- Implementation of most of the macros supplied with GNU autoconf. Experience has shown that some of the GNU autoconf macros are not really applicable to the autokonf environment.
- Provide true macro expansion in addition to the current model. While early versions of autokonf supported such a mechanism, the implementation had made it rather difficult to use. New mechanisms have been proposed that should eliminate most of the problems encountered.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

Bruegger, Bud. P., *GNU autoconf Solves Only Part of the Problem*. 12 Feb. 2002. <http://freshmeat.net/articles/view/146>.

DuBois, Paul, *imake-Related Software and Documentation*. 29 Nov. 2001. <http://www.snake.net/software/imake-stuff>.

DuBois, Paul, *Software Portability with imake,* (O'Reilly & Associates, 1994).

McCall, Andrew, *Stop the autoconf insanity! Why we need a new build system*. 21 Jun. 2003. <http://freshmeat.net/articles/view/889>.

MacKenzie, David, *GNU autoconf*. 24 Dec. 2002. Free Software Foundation. <http://www.gnu.org/software/autoconf>.

<http://www.cims.nyu.edu/cgi-comment/info2html?(autoconf.info)Why+Not+Imake>