CREATING A CONFIGURABLE COMPILER DRIVER FOR SYSTEM V RELEASE 4

John F. Dooley
Vince Guarna

Motorola Computer Group
1101 E. University Avenue
Urbana, Illinois 61801
jdooley@urbana.mcd.mot.com
vguarna@urbana.mcd.mot.com

ABSTRACT

This paper discusses the design and application of the configurable C compiler driver component of System V Release 4. The configurable compiler driver is a table-driven *cc* command that allows users to define include file, library, and tool paths as well as command line option translations and defaults.

The configurable driver presents several advantages over more traditional compiler drivers. The resulting compilation system allows many compilers with differing command line interfaces to be used in large system builds with no makefile changes. It allows the user to easily switch among several different C compilers, while presenting the user with a single command line interface. By allowing changes to compiler component paths, it makes the development of cross-compilers easier. Finally, it allows easy user configuration for the inclusion of default options.

We discuss the motivation for the project, the architectural design, and applications of the driver, including experiences with using the driver for System V Release 4 system builds.

INTRODUCTION

The configurable compiler driver (CCD) is a driver for C that can serve an unlimited number of compilation environments for System V Release 4 (SVR4). CCD uses an ASCII configuration file to look up individual "personalities" that specify various aspects of its behavior including

(1) Component versions (preprocessor, compiler, optimizer, assembler, linker)

(2) Include file and library paths

(3) Command line option translations

(4) Default options

(5) Default #defines and #asserts

By setting a single environment variable, the user can determine which of the standard compilation configurations will be used when the *cc* command is used. Additionally, a second environment variable can be set to reference an arbitrary configuration file if the configurations supplied with the system do not meet the user's needs.

Requirements and Design Goals

The motivation for CCD emerges from several product requirements. First, Motorola's SVR4 is shipped with two C compilers — GNU (the default, chosen for its superior performance), and AT&T's C Issue 5 (CI5) compiler. Because we anticipated the need to use both compilers at various times, we needed a mechanism to switch between them conveniently.

Second, we anticipated the need to use cross compilation tools on 88K machines to reduce build times for 68K objects. This cross compilation environment effectively introduces another compiler to the language tool suite and further amplifies the need for a transparent switching mechanism.

Third was the need to develop a cross compilation environment for Motorola's real time products that was portable across both hosts and targets.

The most important requirement was complete compatibility with the existing System V Release 4 C compiler driver and its command line interface. A driver that required makefile changes would cause significant trouble for the SVR4 build team and would also be considered unacceptable by the user community. Therefore, the CI5 command line interface was chosen as the input language for the driver.

The fifth requirement was that the driver must exhibit the expected default behavior. Although the driver allows environment variables to be set to effect specific types of behavior from the compilation system, most users are not expected to use this functionality. Users familiar with C compilers are not likely to reference system documentation on the configurable system and will therefore not set the associated environment variables. The driver handles this gracefully by automatically referencing the default configuration file in this case.

Finally, CCD must be user customizable. Rather than build a simple merged driver that would serve the two SVR4 compilers with fixed, specific behavior, we decided that we would make runtime configurability available at the user level. Consequently, the driver's characteristics are completely determined by an ASCII file that can be modified by system administrators and overridden by individual users.

Implementation Decisions

The above requirements, along with a very limited time frame for implementation, led to the following implementation decisions. The driver would be table-driven, with a separate *translation table* for each compiler available on the system. The tables would be in a text configuration file that the user could copy and change. An environment variable would re-direct the driver to the new configuration file. A second environment variable would direct the driver to the correct table within the configuration file. There would be a simple translation language used with a few, simple keywords and control structures. The default table for translation would be the first table in the configuration file. For those compiler options that did not exactly match the CI5 command line interface, there would be a mechanism to pass them directly to the appropriate compilation phase.

ARCHITECTURE

Structure of the Driver

The configurable driver is broken up into four sections,

(1) the main program,

(2) the initialization section,

(3) the translation section, and

(4) the execution section.

Main

The main program is charged with driving the other three sections, with parsing the user-supplied command line, determining which compilation components to execute, and with controlling error reporting and cleaning up after compilation. The structure of the main program is shown in Figure 1.

```
Main:
        Find the configuration file;
        Initialize the translation table
        and component strings;
        Get the first user-supplied option;
        While (there are still options to translate) do
                Switch on each option, translating the option into
                the equivalent option for the target compilation system.

                The translated option is placed on the argv list
                for the appropriate target compilation component.

                If the option does not begin with a minus sign,
                assume it is a file name
                and put it onto the input file list.

                If it is an unknown option,
                add it to the linkage editor's argv list.

                If an option that is supposed to take an argument
                doesn't have one, then report the error and continue.
        end-while;

        For (each of the files on the input file list) do
                Preprocess the file if necessary,
                Compile the file,
                Include profiling code if requested,
                Optimize the file if requested, and
                Assemble the file if requested.
        end-for;

        If no errors occurred above, and if the -c option was not used,
        then link the file, clean up, and exit.
```

Figure 1. Configurable Driver Main Program

Initialization

During the initialization section, the configuration file is read in, the component names are defined, the component default strings are initialized, and the translation table is created. The configuration file contains one or more configuration tables, where each table contains all the information necessary to translate the configurable driver's option set into the equivalent options for the target compilation system.

No translations are done at this point, the translation table is just created, with each entry in the table being a sentence in the translation language. The set of indices into the translation table is the set of options recognized by the configurable driver. There may be options in the target compilation system that do not have translations from the configurable driver set. For example, the GNU -m machine dependent options have no equivalents in the driver option set, and are thus not directly included anywhere in the GNU translation table.

Translation

In the translation section, the user-supplied command line option is translated into the target option string and the option string is returned to the main program. The translation sentence is extracted from the configuration table that was created in the initialization section. The parser is a traditional recursive-descent parser. The translation language that is parsed is a small structured language, with simple conditional and sequential control structures and 32 keywords.

It's syntax is similar to C. The top level syntactic unit is the *sentence*, which is composed of three parts:

driver-option statement .

A more detailed description of the language is found below in the section on *Translation Expressions*.

Execution

When all user supplied options are translated, the driver goes through the list of input files and executes the components on each one, one at a time. It builds up each command line using the component name, translated pre-options, any silent or hidden options, the user's options and file, and the translated post-options. It then executes each component. Once all input files have been preprocessed, compiled, optimized, and assembled, the command line for the link editor is built and that component is executed.

CONFIGURATION FILE ORGANIZATION

The configuration file contains one or more configuration tables, each table including all the details necessary to translate the configurable driver options into the equivalent options for the target compilation system. The default SVR4 configuration file is located in */usr/ccs/lib/.compilerc*, and contains tables for the GNU and CI5 compilation systems.

Each configuration table has four parts,

(1) the name of the table,

(2) a section of component default specifications,

(3) a section of component name declarations and pre- and post-options, and

(4) a section of translation expressions.

Each of the last three table sections must be separated by a blank line.

Component Command Line Construction

Compilation command lines can be broken up into several distinct sections, but not all sections will be present for any given component invocation. The sections will always appear in the same relative order. The sections that appear depend largely on the user-supplied options, and occasionally on the values of user-supplied options.

For these reasons we have broken up the component command lines constructed by the configurable driver into several sections

(1) the component's name,

(2) a set of pre-options that are always added to the command line before any other options,

(3) one or more sets of other conditionally included pre-options,

(4) the user-supplied options and file names,

(5) conditionally supplied post-options, and

(6) post-options that are always included after all other options.

Component Default Specifications

The component default specifications are the conditionally included pre- and post-option sets placed on the command line. A default specification is a sentence in the translation language used by all objects in the configuration table. These default specifications invariably depend on which options the user has used on the *cc* command line. This means that these specifications are not translated until after all the user-supplied options are translated and we are constructing the component command line.

The results of these translations are generally called "hidden" or "silent" options because they are not usually visible to the user, and the user would be quite surprised to see them on the component command line.

Component Names and Pre- and Post-options

Component names are generally the absolute pathnames of each compilation component in the target compilation system. There are six possible components, preprocessor, compiler, profiler, optimizer, assembler, and linkage editor. Any of these may be omitted for a particular target; for example, the AT&T CI5 compilation system has a merged preprocessor and compiler so in the CI5 configuration table, the preprocessor component is omitted.

Component names can be changed by the user to use different pathnames. For example, if one is developing a new assembler (e.g. for a cross compilation environment) one can change the assembler component name and call the new assembler. This requires making a copy of the configuration file, making the necessary changes and using the copy. The user must then set the CCMAP and CCCOMPILER environment variables to point to the new configuration file and configuration table, respectively. This ability to easily change compilation systems while maintaining the same user interface is one of the major advantages of the configurable driver.

Pre- and post-options are those that are generally included unconditionally first and last on the component's command line. The pre- and post options are sentences in the translation language used by all objects in the configuration table. For any component either or both pre- and post-options may be omitted.

Translation Expressions

The translation language that is parsed is a small structured language, with simple conditional and sequential control structures and 32 keywords. It's syntax is similar to C. The top level syntactic unit is the *sentence*, which is composed of three parts:

driver-option statement .

A *statement* can be any one of the following:

(1) a simple expression, which translates one configurable driver option into an equivalent target option. For example,

```
c %e %%.
V %e -v.
```

The first expression says to note when the -c option is used, but not put anything on the component command lines because -c is handled internally by the driver.

The second simple expression says to translate the -V driver option to the -v target option.

(2) an *if-then-else* expression, which tests for the presence of a particular user-supplied option and includes one or more target options depending on the result of the test; since all user-supplied options must have been translated before the *if-then-else* test can be made, *if-then- else* expressions are generally found in the component default specifications described above. For example,

```
%if (!G) %e "-lc".
```

says that if the -G driver option is not present, include -lc on the command line.

(3) a multi-way *switch* expression, which tests the value of an argument to a user-supplied driver option and includes one or more target options based on the value of the argument.

For example,

```
X %s (%a) {
    t:  { %e -traditional }
  a,c:  { %e -ansi }
  default: {%e %% } }.
```

This option says that if the user specifies the -Xt option, replace it with -traditional; if the user specifies either -Xa or -Xc, replace it with -ansi. If the command line option contains any other argument besides a, c, or t, it is ignored.

(4) a *message*, which simply prints an informational message. An example of a message expression is

```
J %m "The -J option is not supported, skipping it".
```

(5) a *block* of one or more statements, separated by semi-colons, and enclosed in curly braces.

Pre- or post-option specifications make good examples of block statements, as in this one for the GNU compiler

```
%dc /usr/ccs/lib/gcc-cc1
    "{ %e %t.i; %e "-quiet -dumpbase"; %e %f; }."
    ;
    "{ %if (#) %e -version;
       %if (S) %e -o %b.s %else %e -o %t.s }. "
```

In this example %dc informs the parser that the following expression contains the name of the compiler. */usr/ccs/lib/gcc-cc1* is the name of the GNU compiler program. This is followed by a descriptor containing the pre-options for the compiler.

The descriptor is a block statement containing three simple expressions; the first includes the current temporary file name with a *.i* extension, the two GNU options *-quiet* and *-dumpbase*, and finally the name of the input file. The single semi-colon is the separator between the pre- and post-options.

The post-option descriptor is a block statement containing two if-then- else expressions. The first if replaces the -# driver option, if present, with the -version GNU option. The second if puts the string "-o *basename*.s" on the compiler command line if the -S driver option was used, and puts the string "-o *temporary-file-name*.s" on the command line otherwise. *Basename* is the entire name of the current input file, without the extension. There is a translation expression for each of the configurable driver's 37 possible options.

The translation table for the GNU compiler is included in the Appendix.

PRACTICAL APPLICATIONS AND EXPERIENCES

Experiences with the configurable driver have shown it to be a valuable aid in compilation. One area is the build process for SVR4. Early baselines of this system were built with the CI5 compiler. The change to GNU C as the default compiler was made early in the development of Motorola's version of SVR4. During that transition, no compiler-specific makefile changes were necessary to accommodate GNU C as called by CCD. This resulted in significant savings in build times and enabled the compiler team to focus on compiler problems, rather than build problems.

Interestingly, there were problems that arose from the use of GNU, but these problems were usually solved by adding new entries to the GNU translation table rather than changing makefiles. One example is GNU's handling of the **inline** symbol.

By default, GNU C recognizes **inline** as a keyword to specify function inlining. Additionally, this behavior can be turned off on the command line with the **-fno-inline** option. Treating **inline** as a keyword would normally have no effect except for the fact that a few source files in the SVR4 command suite (**vi**, for example) use the symbol **inline** as an identifier — GNU C generates parse errors for these files. This problem is easily solved by adding **-fno-inline** to the list of default options in the GNU translation table; however, there is a conflicting issue. For performance reasons, it is desirable to use the inlining feature, particularly for kernel builds. It is therefore sometimes important to have the compiler recognize **inline** when experimenting with kernel changes. The result is that a second GNU table was added to the configuration file — one that operates conventionally (no keyword) and one that can be used for performance work.

We have done some experimentation using the driver for cross compilations on 88K SVR4 machines (generating 68K SVR4 binaries). Libraries, include files, a cross compiler, a cross assembler, and a cross linker were loaded onto an 88K machine and a configuration table named **gnu68** was created. By going into the source base for some of the SVR4 commands, binaries for both 88K and 68K targets were successfully generated from the same source base, using the same makefiles, just by changing the **CCCOMPILER** variable. Although many aspects of the system build process make complete cross compilation difficult, we are investigating ways to use 88K machines in conjunction with the driver to reduce 68K build times.

The configurable driver also provides a convenient mechanism to do other useful tasks. For example, a programmer may wish to make sure that all compilations are performed with debugging or optimization turned on. This is trivially accomplished by adding the appropriate flag (**-g**, **-O**) to a local version of the configuration file. Conversely, someone may want to make sure that debugging or optimization is never turned on. This is easily accomplished by mapping the appropriate option to null in the translation table. For large, hierarchical applications with many makefiles, this could save a significant amount of time.

Performance Considerations

Several tests were run to compare the performance of the configurable driver with the GNU driver and the CI5 driver. All tests were run on an 88K SVR4 system, with 16MB of main memory, 2 88100 processors and 8 88200 CMMU's. All times were measured using the elapsed (real) time using the Unix */bin/time* function, and all compilations used the -O compiler option. The test using *make* also used the -c option.

In the first test, a C file with an empty *main()* function was compiled. In this test the compilation using the CI5 compiler with the configurable driver was approximately 7% slower than just using the CI5 driver. When the compilation was done with the configurable driver using the GNU compiler, the configurable driver was approximately 14% slower than GNU.

In a larger test, the 022.li SPEC benchmark files were compiled using the *make compile* target for the benchmark. This benchmark contains 22 C program files ranging in size from several hundred to over thirteen thousand bytes in length. In this longer test the compilation using the CI5 compiler with the configurable driver was only 2% slower than just using the CI5 driver. When the compilation was done with the configurable driver using the GNU compiler, the configurable driver was only 3% slower than GNU. Thus, the configurable compiler driver does not add appreciably to compilation times.

The extra time required by CCD is attributable to the I/O necessary to read the configuration file and build the translation table. This work all occurs once, during initialization. Work is under way to improve the performance of this section.

SUMMARY and FUTURE WORK

CCD has shown to be a useful tool in the compilation environment and we expect to enhance it and expand its use in several ways.

One enhancement we have made is the recognition and expansion of environment variables in the configuration file. This is particularly useful in Unix baseline building. Part of the bootstrap process in the SVR4 baseline building procedure includes the definition of the environment variable **$ROOT**. This variable is used to control where include files are retrieved; further, the variable is changed during the build process. Because CCD does not recognize environment variable syntax in the configuration file, two tables are required for the GNU compiler instead of one, one with $ROOT as the first value and one without it.

We also want to expand the use of the driver to other languages such as C++ and Fortran. Because of the likelihood of our supporting multiple offerings for these languages, the benefits of extending the driver for these cases are probably worthwhile. Additionally, the availability of configurable drivers for all of our language tools increases the ability of third party software vendors to introduce language products for our platforms in a seamless manner. This should result in the long-term enhancement of our programming environment.

Finally, we are making the configurable driver available for new and existing SVR3 platforms. This is especially useful for the 68K environment where a transition is being made to a new C compiler.

A new version of the configurable driver is being developed to address problems introduced by using the driver for cross development for Motorola real-time products. Pushing the limits of the original requirements for the driver has forced us to re-examine the driver and change it.

This new version (CCD2) is a complete re-write of the configurable driver that eliminates many of the limitations of the original. CCD2 is still table driven, but is not restricted to using the CI5 command line interface - a limitation that became acute when we wanted to develop cross compilation products for the real-time market. It also removes the six compilation phase restriction of CCD1 and gives the user greater flexibility in tailoring the command line interface. Finally, it enhances the translation language by allowing for the complete expansion of environment variables, and for the use of user-defined variables.

REFERENCES

[1] Holub, Allen I., **Compiler Design in C**, Prentice-Hall, 1990.

[2] Wirth, Niklaus, **Algorithms + Data Structures = Programs**, Chapter 5, Prentice-Hall, 1976.

APPENDIX

```
/*
  This is the 88k version of the configuration file.
  Copyright 1991, 1992, 1993 Motorola, Inc.
*/

/* The default "gnu" table is for the GNU-C 2 compiler. */
Env gnu {
%PREDEFINES "-D__GNUC__=2 -D__m88k__ -D__unix__
-D__OPEN_NAMESPACE__ -D__CLASSIFY_TYPE__ -D___CLASSIFY_TYPE___
-D__m88000__"
%PREASSERTS "-Acpu(m88k) -Asystem(unix) -Amachine(m88k)"
%PREPROC "{ %e "-lang-c -trigraphs"; %if (O) %e "-D__OPTIMIZE__ ";
    %if (v) %e "-Wall -Wtraditional -pedantic";
    %if (X) %s (%a) {
            a: { %e %% }
            c: { %e "-pedantic -D__STRICT_ANSI__" }
            p: { %e "-pedantic -D__STRICT_ANSI__ -D_POSIX_SOURCE" }
            x: { %e "-pedantic -D__STRICT_ANSI__ -D_POSIX_SOURCE
                    -D_XOPEN_SOURCE" }
           %default: { %e %% }
           }
    }."
```

```
%COMPILE "{ %e "-funsigned-bitfields -fwritable-strings" }."
%STARTUP_PATH "/usr/ccs/lib"
%LOADR1 "{ %e "%STARTUP_PATH/crti.o";
       %if (X) %s (%a) {
               a: { %e "%STARTUP_PATH/values-Xa.o" }
               c,p,x: { %e "%STARTUP_PATH/values-Xc.o" }
               n: { %e %% }
               %default: { %e "%STARTUP_PATH/values-Xt.o" }
              }
       %else %e "%STARTUP_PATH/values-Xt.o" }."
%LOADR2 "{ %if (q) %s (%a) {
                               g: { %e "/usr/ccs/lib/gmon.o" }
                               l: { %e "-lprof -lelf -lm" }
                               p: { %e %% }
                     };
  %if (Y) { %e "-Y P,"; %e %r }
  %else %if (p)
    %if (K) %s (%a) {
            minabi: { %e "-I /usr/lib/ld.so.1" ;
                   %e "-Y P,/usr/ccs/lib/minabi/libp:
                          /usr/ccs/lib/libp:/usr/lib/libp:
                          /usr/ccs/lib/minabi:/usr/ccs/lib:/usr/lib"}
            %default: { %e  "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:
                              /usr/ccs/lib:/usr/lib" } }
    %else %e "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:
                 /usr/ccs/lib:/usr/lib"
  %else %if (q) %s (%a) {
           g,p: {
             %if (K) %s (%a) {
           minabi: { %e "-I /usr/lib/ld.so.1" ;
                    %e "-Y P,/usr/ccs/lib/minabi/libp:
                           /usr/ccs/lib/libp:/usr/lib/libp:
                           /usr/ccs/lib/minabi:/usr/ccs/lib:
                           /usr/lib"}
       %default: { %e  "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:
                          /usr/ccs/lib:/usr/lib" } }
             %else %e "-Y P,/usr/ccs/lib/libp:/usr/lib/libp:/usr/ccs/lib:/usr/lib"
             }
              l: { %if (K) %s (%a) {
                                minabi: { %e "-I /usr/lib/ld.so.1" ;
                    %e "-Y P,/usr/ccs/lib/minabi:
                           /usr/ccs/lib:/usr/lib" }
                       %default: { %e %% } }
      %else %e "-Y P,/usr/ccs/lib:/usr/lib"
      }
    }
  %else %if (K) %s (%a) {
                minabi: { %e "-I /usr/lib/ld.so.1" ;
                       %e "-Y P,/usr/ccs/lib/minabi:
                              /usr/ccs/lib:/usr/lib" }
                %default: { %e %% } }
  %else %e "-Y P,/usr/ccs/lib:/usr/lib" }."
%STARTUP  "{ %if (p) %e "%STARTUP_PATH/mcrt1.o"
           %else %if (q) %s (%a) {
                        g: { %e "%STARTUP_PATH/gcrt1.o" }
```

```
                    l: { %e "%STARTUP_PATH/pcrt1.o" }
                    p: { %e "%STARTUP_PATH/mcrt1.o" }
                  }
        %else %if (!G) %e "%STARTUP_PATH/crt1.o" }."

%dp /usr/ccs/lib/gcc2/cpp "{ %e "-nostdinc";
       %e -undef; %if (# %or V) %e -v }." ;
       "{ %if (X) %s (%a) {
                a,c,p,x: { %e %% }
               %default: { %e "-Dm88k -Dunix -Dm88000" }}
          %else %e "-Dm88k -Dunix -Dm88000";
              %e "-I/usr/ccs/lib/gcc2/include -I/usr/include";
              %if (P) %e -o %b.i
              %else %if (E) %e "-o -"
              %else %e %t.i }."
%dc /usr/ccs/lib/gcc2/cc1 "{ %e %t.i; %e "-quiet -dumpbase"; %e %f }." ;
        "{ %if (# %or V) %e "-version";
          %if (S) %e -o %b.s %else %e -o %t.s }."
%da /usr/ccs/bin/as "{ %e -o %b.o; %if (# %or V) %e -V }." ;
                "{ %if (S) %e %f %else %e %t.s }."
%dl /usr/ccs/bin/ld ; "{ %if (Q) %e "-Qy";
                %if (# %or V) %e -V;
                %if (!G) %e "-L/usr/ccs/lib/gcc2 -lgcc -lc";
            %e " %STARTUP_PATH/crtn.o"}."

A %s (%a) { -: { %e %% }
      default: { %e -A %all } }.
B %e -B %a.
C %e -C.
c %e %%.
D %e -D %all.
d %e -d %a.
e %e -e %a.
E %e -E.
f %e %%.
G %e -G.
g %e -g.
h %e -h %a.
H %e -H.
I %e -I %a.
J %m "The -J option is not supported, skipping it".
K %s (%a) {
          PIC,pic: { %e -fpic }
          fpe: { %m "Ignoring the -Kfpe option" }
          mau: { %m "Ignoring the -Kmau option" }
       minabi: { %e %% }
          %default: { %e %% } }.
L %e -L %a.
l %e -l %a.
O %e -O2.
o %e -o %a.
P %e %%.
p %e -p.
Q %e %%.
q %s (%a) { g: { %e -p }
```

```
          l: { %e "-mlprof -g -a" }
        p: { %e -p }
    %default: { %e %% } }.
S %e %%.
u %e -u %a.
U %e -U %a.
V %e %%.
v %e "-Wall -Wshadow -Wcast-align -Winline
     -Waggregate-return -Wwrite-strings
     -Wcast-qual -Wpointer-arith -Wstrict-prototypes
     -Wmissing-prototypes -Wnested-externs
     -Wtraditional -Wconversion -pedantic".
W %e %%.
X %s (%a) {    t,n: { %e %% }
                  a: { %e -ansi }
           c,p,x: { %e "-ansi -pedantic" }
        %default: { %e %% } }.
Y %s (%a) { S,L,F,U: { %e %% }
        I: { %e "-nostdinc -I/usr/include -I."; %e -I; %e %r }
        P: { %e -Y; %e %a } }.
z %e -z %a.
# %e %%.
}
```