Installing lcc

Christopher W. Fraser
AT&T Bell Laboratories Rm. 2C–464, 600 Mountain Ave., P.O. Box 636, Murray Hill, NJ 07974–0636

David R. Hanson
Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544

Contents

1. Introduction

`lcc` is the ANSI C compiler described in our book *A Retargetable C Compiler: Design and Implementation* (Benjamin Cummings, 1995, ISBN 0–8053–1670–1).

Extract the distribution into its own directory. All paths below are relative to this directory. The distribution holds the following subdirectories.

`src`
: source code

`etc`
: driver, man page

`doc`
: this document

`include`
: ANSI include files

`tst`
: test suite

`mips-* mips-*/tst`
: MIPS build area, test output

`sparc-* sparc-*/tst`
: SPARC build area directory, test output

`x86-dos x86-dos/tst`
: x86 build area, test output

Distributions include code generators for the MIPS, SPARC, and the Intel 386 and its successors.

Installation involves three steps performed in the following order.

1. Decide where to install the man page, the include files, the compiler, and `lcc`, the driver program; see Sec. 2.

2. Install a host–specific driver; see Sec. 3.

3. Install the compiler; see Sec. 4.

The value of the variable `rcsid` in `src/main.c` identifies the version of the distribution. If the file `LOG` appears, it describes the changes from the previous version.

doc/install.html is the HTML file for this document. doc/install.ps and doc/install.txt are PostScript and plain ASCII versions.

2. Paths

Installation consists of three files and one directory; these are summarized below along with paths used in typical installations.

/usr/local/man/man1/lcc.1	the man page
/usr/local/lib/rcc	the compiler
/usr/local/bin/lcc	the driver
/usr/local/include/ansi	include files (a directory)

These files can be placed in other, site−specific locations, but the compiler should be named rcc. If the driver isn't named lcc, edit the man page (etc/lcc.1).

Include files are in directories named include/*target*−*system*; the meaningful combinations are

mips−irix	IRIX Release 4.0
mips−ultrix	ULTRIX 4.3
sparc−sun	SunOS 4.1
sparc−solaris	Solaris 2.3
x86−dos	DOS 6.0, Windows 3.1

Choose the include files that are appropriate for your system, or make a copy of a closely related set and edit them.

For example, if the paths shown above are chosen and if include/mips−ultrix has the appropriate include files, install the man page and include files by

```
% cp etc/lcc.1 /usr/local/man/man1
% cp include/mips−ultrix/*.h /usr/local/include/ansi
```

3. Installing the Driver

The preprocessor, compiler, assembler, and loader are invoked by a driver program, lcc, which is similar to cc on most systems. It's described in the man page etc/lcc.1. The driver is built by combining the host−independent part, etc/lcc.c, with a small host−specific part. By convention, host−specific parts are named *hostname*.c, where *hostname* is the local name for the host on which lcc is being installed. etc holds many examples. Comments in most give the details of the particular host; pick one that is closely related to your host, copy it to etc/*yourhostname*.c, and edit it as described below. You should not have to edit etc/lcc.c.

Debug your version of the driver by running it with the −v −v options, which cause it to echo the commands it would execute, but not to execute them.

Here's etc/hart.c, which we'll use as an example in describing how to edit a host−specific part. This example illustrates all of the important features.

```
/* DECStations running ULTRIX at Princeton University */
```

```
#include <string.h>

char *cpp[] = {
        "/usr/gnu/lib/gcc-cpp", "-undef",
        "-DLANGUAGE_C", "-D_LANGUAGE_C", "-D__LANGUAGE_C",
        "-D_unix", "-D__unix", "-Dultrix", "-D_ultrix", "-D__ultrix",
        "-Dmips", "-D_mips", "-D__mips",
        "-Dhost_mips", "-D_host_mips", "-D__host_mips",
        "-DMIPSEL", "-D_MIPSEL", "-D__MIPSEL",
        "$1", "$2", "$3", 0 };
char *include[] = { "-I/usr/local/include/ansi", 0 };
char *com[] =  { "/usr/local/lib/rcc", "-target=mips-ultrix",
        "$1", "$2", "$3", 0 };
char *as[] =  { "/bin/as", "-o", "$3", "", "$1",
        "-nocpp", "-EL", "$2", 0 };
char *ld[] =  { "/usr/bin/ld", "-o", "$3", "/usr/lib/crt0.o",
        "$1", "$2", "", "", "-lm", "-lc", 0 };

int option(arg) char *arg; {
        if (strcmp(arg, "-g") == 0)
                as[3] = "-g";
        else if (strcmp(arg, "-p") == 0
        && strcmp(ld[3], "/usr/lib/crt0.o") == 0) {
                ld[3] = "/usr/lib/mcrt0.o";
                ld[7] = "/usr/lib/libprof1.a";
        } else if (strcmp(arg, "-b") == 0
        && access("/usr/local/lib/bbexit.o", 4) == 0)
                ld[6] = "/usr/local/lib/bbexit.o";
        else
                return 0;
        return 1;
}
```

Most of the host−specific code is data that gives prototypes for the commands that invoke the preprocessor, compiler, assembler, and loader. Each command prototype is an array of pointers to strings terminated with a null pointer; the first string is the full path name of the command and the others are the arguments or argument placeholders, which are described below.

The `cpp` array gives the command for running the preprocessor. `lcc` is intended to be used with an ANSI preprocessor, such as the GNU C preprocessor available from the Free Software Foundation. If the GNU preprocessor is used, it must be named `gcc-cpp` in order for `lcc`'s `-N` option to work correctly.

Literal arguments specified in prototypes, e.g., `"-Dmips"` in the `cpp` command above, are passed to the command as given.

The strings `"$1"`, `"$2"`, and `"$3"` in prototypes are placeholders for *lists* of arguments that are substituted in a copy of the prototype before the command is executed. `$1` is replaced by the *options* specified by the user; for the preprocessor, this list always contains at least `-Dunix` and `-D__LCC__`. `$2` is replaced by the *input* files, and `$3` is replaced by the *output* file.

Zero−length arguments after replacement are removed from the argument list before the command is invoked. So, e.g., if the preprocessor is invoked without an output file, `"$3"` becomes `""`, which is removed from the final argument list.

For example, to specify a preprocessor command prototype to invoke `/bin/cpp` with the options `-Dmips` and `-Dultrix`, the `cpp` array would be

```
char *cpp[] = { "/bin/cpp", "-Dmips", "-Dultrix",
        "$1", "$2", "$3", 0 };
```

The `include` array is a list of `-I` options that specify which directives should be searched to satisfy include directives. These directories are searched in the order given. The first directory should be the one to which the ANSI header files were copied in Sec. 2. The driver adds these options to `cpp`'s arguments when it invokes the preprocessor, except when `-N` is specified.

Design this list carefully. Mixing ANSI and pre−ANSI headers (e.g., by listing `/usr/include` after the directory of ANSI headers shown above) may mix incompatible headers. Unless the default list holds *only* `/usr/include` or *only* the ANSI headers, many users may be forced to use `−N` and `−I` incessantly.

`com` gives the command for invoking the compiler. This prototype can appear as shown above, with two important changes. The command name should be edited to reflect the location of the compiler chosen in Sec. 2, and the option `−target=mips−ultrix` should be edited to the *target−system* for your host. `lcc` can generate code for *all* of the *target−system* combinations listed in Sec. 2. The `−target` option specifies the default combination. The driver's `−Wf` option can be used to specify other combinations; the man page elaborates.

`as` gives the command for invoking the assembler.

`ld` gives the command for invoking the loader. For the other commands, the list `$2` contains a single file; for `ld`, `$2` contains all '.o' files and libraries, and `$3` is `a.out`, unless the `−o` option is specified. As suggested in the code above, `ld` must also specify the appropriate startup code and default libraries.

The `option` function is described below; for now, use an existing `option` function or one that returns 0.

After specifying the prototypes, compile the driver by

```
% cd etc
% make HOST=hart
```

where `hart` is replaced by *yourhostname*. Run the resulting `a.out` with the options `−v −v` to display the commands that would be executed, e.g.,

```
% a.out -v -v foo.c baz.c mylib.a -lX11
a.out $Revision: 3.1 $ $Date: 1994/09/07 11:37:52 $
foo.c:
/usr/gnu/lib/gcc-cpp -undef -DLANGUAGE_C -D_LANGUAGE_C -D__LANGUAGE_C
   -D_unix -D__unix -Dultrix -D_ultrix -D__ultrix -Dmips -D_mips
   -D__mips -Dhost_mips -D_host_mips -D__host_mips -DMIPSEL -D_MIPSEL
   -D__MIPSEL -Dunix -D__LCC__ -v -I/usr/local/include/ansi foo.c
   | /usr/local/lib/rcc -target=mips-ultrix -v - /tmp/lcc12511.s
/bin/as -o foo.o -nocpp -EL /tmp/lcc12511.s
baz.c:
/usr/gnu/lib/gcc-cpp -undef -DLANGUAGE_C -D_LANGUAGE_C -D__LANGUAGE_C
   -D_unix -D__unix -Dultrix -D_ultrix -D__ultrix -Dmips -D_mips
   -D__mips -Dhost_mips -D_host_mips -D__host_mips -DMIPSEL -D_MIPSEL
   -D__MIPSEL -Dunix -D__LCC__ -v -I/usr/local/include/ansi baz.c
   | /usr/local/lib/rcc -target=mips-ultrix -v - /tmp/lcc12511.s
/bin/as -o baz.o -nocpp -EL /tmp/lcc12511.s
/usr/bin/ld -o a.out /usr/lib/crt0.o foo.o baz.o mylib.a -lX11 -lm -lc
rm /tmp/lcc12511.s
```

Leading spaces indicate lines that have been folded manually to fit this page. Note the use of a pipeline to connect the preprocessor and compiler. `lcc` arranges this pipeline itself; it does not call the shell. If you want `lcc` to use temporary files instead of a pipeline, define `PIPE=0` in `CFLAGS` when making the driver:

```
% make CFLAGS='-DPIPE=0' HOST=hart
```

The option `−pipe` forces `lcc` to use a pipeline between the preprocessor and the compiler regardless of `PIPE`'s value.

As the output shows, `lcc` places temporary files in `/tmp`. Alternatives can be specified by defining `TEMPDIR` in `CFLAGS` when making the driver, e.g.,

```
% make CFLAGS='-DTEMPDIR=\"/usr/tmp\"' HOST=hart
```

causes `lcc` to place temporary files in `/usr/tmp`. Once the driver is completed, install it by

```
% cp a.out /usr/local/bin/lcc
```

where the destination is the location chosen for `lcc` in Sec. 2.

The `option` function is called for the options `-g`, `-p`, `-pg`, and `-b` because these compiler options might also affect the loader's arguments. For these options, the driver calls `option(arg)` to give the host-specific code an opportunity to edit the `ld` command, if necessary. `option` can change `ld`, if necessary, and return 1 to announce its acceptance of the option. If the option is unsupported, `option` should return 0.

For example, in response to `-g`, the `option` function shown above changes `as[3]` from `""` to `"-g"`, which specifies the debugging option to the assembler. If `-g` is not specified, the `""` argument is omitted from the `as` command because it's empty.

Likewise, the `-p` causes `option` to change the name of the startup code and add the name of the profiling library. Note that `option` has been written to support simultaneous use of `-g` and `-p`, e.g.,

```
% a.out -v -v -g -p foo.s baz.o -o myfoo
a.out $Revision: 3.1 $ $Date: 1994/09/07 11:37:52 $
/bin/as -o foo.o -g -nocpp -EL foo.s
/usr/bin/ld -o myfoo /usr/lib/mcrt0.o foo.o baz.o
   /usr/lib/libprof1.a -lm -lc
rm /tmp/lcc12516.s
```

On Suns, the driver also recognizes `-Bstatic` and `-Bdynamic` as linker options, and recognizes but ignores Sun's '`-target` *name*' option.

The option `-Wo`*arg* causes the driver to pass *arg* to `option`. Such options have no other effect; this mechanism is provided to support system-specific options that affect the commands executed by the driver.

The `-b` option causes the compiler to generate code to count the number of times each expression is executed. The `exit` function in `etc/bbexit.c` writes these counts to `prof.out` when the program terminates. If `option` is called with `-b`, it must edit the `ld` command accordingly, as shown above. This version of `option` uses the `access` system call to insure that `bbexit.o` is installed before editing the `ld` command. To install `bbexit.o` execute

```
% make bbexit.o
% cp bbexit.o /usr/local/lib/bbexit.o
```

If necessary, change `/usr/local/lib` to reflect local conventions. The `exit` function in `etc/bbexit.c` works on the systems listed in Sec. 2, but may need to be modified for other systems.

If `option` supports `-b`, you should also install `etc/bprint.c`, which reads `prof.out` and generates a listing annotated with execution counts. After `lcc` is installed, install `bprint` with the commands

```
% make bprint
% cp bprint /usr/local/bin/bprint
% cp bprint.1 /usr/local/man/man1
```

The `makefile` uses `lcc` to compile `bprint.c`; you must use `lcc` or another ANSI C compiler, e.g., `gcc`, because `bprint.c` is written in ANSI C. Also, `bprint.c` *includes* `"../src/profio.c"`, so it must be compiled in `etc`.

To complete the driver, write an appropriate `option` function for your system, and make and install the driver as described above.

4. Installing the Compiler Proper

The compiler proper, `rcc`, is built by compiling it with the host C compiler and then using the result to re−compile itself. A test suite is used to verify that the compiler is working correctly. The examples below illustrate this process on a MIPS under Ultrix. You must have the driver, `lcc`, installed in order to test `rcc`. If any of the steps below fail, contact us (see Sec. 5).

The object files, `rcc`, and the generated code for the programs in the test suite are placed in the directory *target−system* where *target* and *system* are the names of your target machine and its operating system, respectively. There are directories for the supported *target−system* combinations, e.g., `mips−ultrix`.

The default target in `src/makefile` is `rcc`. `lcc` is built by executing `make` from the apppropriate *target−system* directory and specifying system−specific values for `CFLAGS` and `LDFLAGS`, if necessary. For example, to build `rcc` for a MIPS running Ultrix, execute the commands

```
% cd mips−ultrix
% make −f ../src/makefile
cc −c −O ../src/alloc.c
...
cc −c −O ../src/x86.c
cc −o rcc  alloc.o bind.o dag.o ... mips.o sparc.o x86.o
```

There may be a few warnings, but there should be no errors. If your host is an SGI machine running IRIX 4.0 or later, you might need `CFLAGS=−cckr`. If `cc` doesn't automatically search the directory that holds the source file, specify `CFLAGS=−I../src`. If you use `gcc`, specify `CFLAGS="−ansi −fno−builtin"`.

Once `rcc` is built with the host C compiler, run the test suite to verify that `rcc` is working correctly. The commands in `src/makefile` run the shell script `src/run` on each C program in the test suite, `tst/*.c`. It uses the driver, `lcc`, so you must have the driver installed before testing `rcc`. The *target−system* combination is read from the variable `TARGET`, which is specified when invoking `make`:

```
% make −f ../src/makefile TARGET=mips−ultrix test
../rcc mips−ultrix 8q:
../rcc mips−ultrix array:
../rcc mips−ultrix cf:
../rcc mips−ultrix cq:
../rcc mips−ultrix cvt:
../rcc mips−ultrix fields:
../rcc mips−ultrix front:
../rcc mips−ultrix incr:
../rcc mips−ultrix init:
../rcc mips−ultrix limits:
../rcc mips−ultrix paranoia:
../rcc mips−ultrix sort:
../rcc mips−ultrix spill:
../rcc mips−ultrix stdarg:
../rcc mips−ultrix struct:
../rcc mips−ultrix switch:
../rcc mips−ultrix wf1:
../rcc mips−ultrix yacc:
```

For each C program in the test suite, `src/run` compiles the program and uses `diff` to compare the generated assembly code with the expected code (the MIPS code expected for `tst/8q.c` is in `mips−ultrix/tst/8q.s.bak`, etc.). If there are differences, the script executes the generated code with the input given in `tst` (the input for `tst/8q.c` is in `tst/8q.0`, etc.) and compares the output with the expected output (the expected output from `tst/8q.c` on the MIPS is in `mips−ultrix/tst/8q.1.bak`, etc.). The script also compares the diagnostics from the compiler with the expected diagnostics.

On some systems, there may be a few differences between the generated code and the expected code. These differences occur because the expected code is generated by cross compilation on a MIPS and the least−significant bits of some floating−point constants differ from those bits in constants generated on your system. There should be no differences in the output from executing the test programs.

The `../rcc` and `mips-ultrix` preceding the name of each test program in the output above indicate the compiler and the target, e.g., '`../rcc` is generating code for a `mips` running the `ultrix` operating system.'

Next, build `rcc` again using the just−built `rcc`:

```
% make -f ../src/makefile TARGET=mips-ultrix triple
rm -f *.o
make -f ../src/makefile CC='lcc -B./ -d0.1 -A'
   CFLAGS='-Wf-target=mips-ultrix -I../src/../include/mips-ultrix
   -I../src'  LDFLAGS=''
lcc -B./ -d0.1 -A -c -Wf-target=mips-ultrix
   -I../src/../include/mips-ultrix -I../src ../src/alloc.c
...
lcc -B./ -d0.1 -A -o rcc  alloc.o bind.o dag.o decl.o ... x86.o
strip rcc
od +8 od2
rm -f *.o
make -f ../src/makefile CC='lcc -B./ -d0.1 -A'
   CFLAGS='-Wf-target=mips-ultrix -I../src/../include/mips-ultrix
   -I../src'  LDFLAGS=''
lcc -B./ -d0.1 -A -c -Wf-target=mips-ultrix
   -I../src/../include/mips-ultrix -I../src ../src/alloc.c
...
lcc -B./ -d0.1 -A -o rcc  alloc.o bind.o dag.o decl.o ... x86.o
strip rcc
od +8 od3
cmp od[23] && rm od[23]
```

This command builds `rcc` twice; once using the `rcc` built by `cc` and again using the `rcc` built by `lcc`. After building each version, an octal dump of the resulting binary is made, and the two dumps are compared. They should be identical, as shown at the end of the output above. If they aren't, our compiler is generating bad code; contact us.

The final version of `rcc` should also pass the test suite; i.e., the output from

```
make -f ../src/makefile TARGET=mips-ultrix test
```

should be identical to that from the previous `make test`.

Now install the final version of `rcc`:

```
% cp rcc /usr/local/lib/rcc
```

where the destination is the location chosen for `rcc` in Sec. 2.

On some systems, you may be able to use environment variables and `make`'s `-e` option to avoid specifying `TARGET` on each `make` command, and the `make` commands described above can be done with a single command:

```
% setenv TARGET mips-ultrix
% cd mips-ultrix
% make -e -f ../src/makefile test triple test clean
```

`make clean` cleans up, but does not remove `rcc`, and `make clobber` cleans up and removes `rcc`.

The code generators for the other targets can be tested by running `make` from the appropriate target−specific directory and setting some environment variables to control what `src/run` does. For example, if you built `mips−ultrix/rcc` and installed it in `/usr/local/lib/rcc`, you can test the SPARC code generator for the SunOS operating system as follows.

```
% setenv REMOTEHOST noexecute
% setenv BUILDDIR /usr/local/lib/
% cd sparc−sun
% make −f ../src/makefile RCC= TARGET=sparc−sun test
/usr/local/lib/rcc sparc−sun 8q:
/usr/local/lib/rcc sparc−sun array:
/usr/local/lib/rcc sparc−sun cf:
/usr/local/lib/rcc sparc−sun cq:
/usr/local/lib/rcc sparc−sun cvt:
/usr/local/lib/rcc sparc−sun fields:
/usr/local/lib/rcc sparc−sun front:
/usr/local/lib/rcc sparc−sun incr:
/usr/local/lib/rcc sparc−sun init:
/usr/local/lib/rcc sparc−sun limits:
/usr/local/lib/rcc sparc−sun paranoia:
/usr/local/lib/rcc sparc−sun sort:
/usr/local/lib/rcc sparc−sun spill:
/usr/local/lib/rcc sparc−sun stdarg:
/usr/local/lib/rcc sparc−sun struct:
/usr/local/lib/rcc sparc−sun switch:
/usr/local/lib/rcc sparc−sun wf1:
/usr/local/lib/rcc sparc−sun yacc:
```

As above, `src/run` compares the SPARC code generated with what's expected. There should be no differences. Setting `REMOTEHOST` to `noexecute` suppresses the assembly and execution of the generated code. `BUILDDIR` gives the directory that holds `rcc`, and specifying `RCC=` to `make` insures that `rcc` is not rebuilt in the `sparc−sun` directory.

If you set `REMOTEHOST` to the name of a SPARC machine to which you can `rlogin`, `src/run` will `rcp` the generated code to that machine and execute it there, if necessary. See `src/run` for the details.

Once everything is installed, you can use `lcc` as a cross compiler. The options `−S` and `−Wf−target=`*target−system* generate assembly code for the specified target, which is any of those listed in Sec. 2. For example,

```
% lcc −Wf−target=sparc−sun −S tst/8q.c
```

generates SPARC code for `tst/8q.c` in `8q.s`.

`lcc` can also generate code for a 'symbolic' target. This target is used routinely in front−end development, and its output is a printable representation of the input program, e.g., the dags constructed by the front end are printed, and other interface functions print their arguments. You can specify this target with the option `−Wf−target=symbolic`. For example,

```
% lcc −Wf−target=symbolic −S tst/8q.c
```

generates symbolic output for `tst/8q.c` in `8q.s`. Finally, the option `−Wf−target=null` specifies the 'null' target for which `lcc` emits nothing and thus only checks the syntax and semantics of its inputs files.

5. Reporting Bugs

`lcc` is a large, complex program. We find and repair errors routinely. If you think that you've found a error, follow the steps below, which are adapted from the instructions in Chapter 1 of *A Retargetable C Compiler: Design and Implementation*.

1. If you don't have a source file that displays the error, create one. Most errors are exposed when programmers try to compile a program that they think valid, so you probably have a

demonstration program already.

2. Preprocess the source file and capture the preprocessor output. Discard the original code.

3. Prune your source code until it can be pruned no more without sending the error into hiding. We prune most error demonstrations to fewer than five lines.

4. Confirm that the source file displays the error with the *distributed* version of `lcc`. If you've changed `lcc`, and if the error appears only in your version, then you'll have to chase the error yourself, even if it turns out to be our fault, because we can't work on your code.

5. Annotate your code with comments that explain why you think that `lcc` is wrong. If `lcc` dies with an assertion failure, please tell us where it died. If `lcc` crashes, please report the last part of the call chain if you can. If `lcc` is rejecting a program that you think valid, please tell us why you think it's valid, and include supporting page numbers in the ANSI Standard, Appendix A in *The C Programming Language*, 2nd edition by B. W. Kernighan and D. M. Ritchie (Prentice Hall, 1988), or the appropriate section in *C, A Reference Manual*, 3rd edition by S. B. Harbison and G. L. Steele, Jr. (Prentice Hall, 1991). If `lcc` silently generates incorrect code for some construct, please include the corrupt assembly code in the comments and flag the bad instructions if you can.

6. Confirm that your error hasn't been fixed already. The latest version of `lcc` is always available for anonymous `ftp` from `ftp.cs.princeton.edu` in `pub/lcc`. A `README` file there gives acquistion details, and a `LOG` file reports what errors were fixed and when they were fixed. If you report a error that's been fixed, you might get a canned reply.

7. Send your program by electronic mail to `lcc-bugs@cs.princeton.edu`. Please send only valid C programs; put all remarks in C comments so that we can process reports semi−automatically.

6. Keeping in Touch

There is an `lcc` mailing list for general information about `lcc`. To be added to the list, send a message with the 1−line body

```
subscribe lcc
```

to `majordomo@cs.princeton.edu`. This line must appear in the message body; 'Subject:' lines are ignored. To learn more about mailing lists served by `majordomo`, send a message with the 1−word body `'help'` to `majordomo@cs.princeton.edu`. Mail sent to `lcc@cs.princeton.edu` is forwarded to everyone on the mailing list.

There is also an `lcc-bugs` mailing list for reporting bugs; subscribe to it by sending a message with the 1−line body

```
subscribe lcc-bugs
```

to `majordomo@cs.princeton.edu`. Mail addressed to *lcc−bugs@cs.princeton.edu* is forwarded to everyone on this list.

Chris Fraser / cwf@research.att.com
David Hanson / drh@cs.princeton.edu
Thu Sep 8 10:14:42 EDT 1994