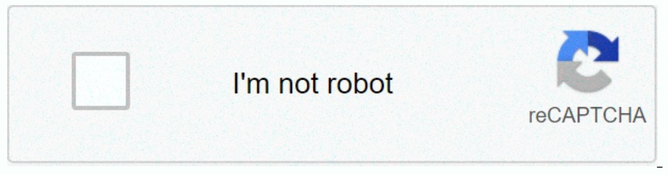

Continue

Compile c linux makefile

Programming Utilities Guide In previous cases you have seen how to compose a simple Program C from a single source file, using both explicit target entries and implicit rules. Most C programs, however, consist of several source files. Many include a library routine, either from one of the standard system libraries or from a user-submitted library. Although it may be easier to re-receive and link a single-source program with a single cc command, it is usually more convenient to migrate programs with multiple resources in stages -- first, by migrating each source file to a separate object file (.o), and then by linking the object files to form an executable (a.out) file. This method requires more disk space, but the resulting (repetitive) recompitions must only be performed on those object files for which the resources have changed, saving time. Simple Makefile Case Next makefile is not elegant, but it does the job. Table 4-4 Simple Makefile to compile C resources: All explicit # Simple makefile to compose the program from #two C source files. .KEEP_STATE features: main.o data.o cc -O -o functions main.o data.o main.o: main.c cc -O -c main.c data.o: data.c cc -O -c data.c clean: rm functions main.o data.o In this case, make products object files main.o i data.o, a executing function file: $make cc -o functions main.o data.o cc -O -c main.c cc -O -c data.c Using make's Predefined Macros Next example performs exactly themselves, but demonstrates the use of make's preefined macros for the indicated compilation commands. The use of predefined macros eliminates the need to edit makefils when the preparation environment changes. Macros also provide access to the CFLAGS macro (and other FLAGS macros) to provide the commy option from the command line. Predefined macros are also widely used under implicit makea rules. The predefined macros in the following makefile are listed below. [Predefined macros are used more extensively than in previous versions of the trade marks. Not all predefined macros shown here are available with earlier versions.] In general, they are useful for preparing C programs. Collect. C Command line cc; CC, CFLAGS and CPPFLAGS as follows, together with option -c. COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS) -c The root of the macro name, COMPILE, is a convention indicating that a macro represents a command line for compile (to create an object or .o file). .c is a mnemonic device indicating that the command line applies to the .c (C) of the file. Note - Macro names that end with FLAGS string options to the associated compiler command macro. It is good practice to use these macros for consistency and portability. It is also good practice to record the default values you want for them in makefile. Full list all predefined macros are shown in Table 4-9. LINK.c Basic command line cc to link object files, For example, COMPILE.c, or without option -c i with reference to macro LDFLAGS: LINK.c=$(CC) $(CPLAGS) $(LDFLAGS) CC Value cc. (Value cc. (Value is redefiniated so that path title is an alternative C coma. CFLAGS Options for the cc command; by default none. CPPFLAGS Options for CPP; by default, none. LDFLAGS Connection Editor options, ld; by default none. Table 4-5 Makefile for compiling C resources using predefined macros # Makefile to prepare two C sources CFLAGS= -O . KEEP_STATE: features: main.o data.o $(LINK.c) -o functions main.o data.o main.o: main.c $(COMPILE.c) main.c data.o: data.c$(COMPILE.c) data.c clean: rm functions main.o data.o Using implicit rules to simplify Makefile: Sufiks rules S'doing that su command lines for compiling main.o and data.o from their file .c now functionally equivalent to the rule .c.o sufiksa, their target unosi su redundant; performs the same preparation, whether shown in the makefile or not. The next version of makefile eliminates them, relying on the .c.o. to migrate individual object files. Table 4-6 Makefile for Compiling C Sources Using Suffix Rules #Makefile for a program from two C sources # using suffix rules. CFLAGS= -O . KEEP_STATE: functions: main.o data.o $(LINK.c) -o functions main.o data.o clean: rm functions main.o data.o Note - Table 4-8 displays a complete list of connection rules. As if the dependency processes main.o and data.o, it does not find the target entries for them. Check whether the relevant implicit rule should be applied. In this case, select the .c.o. rule to build the .o file from an dependency file that has the same base name and .c. Note - Applies the appearance order in the admon list to determine which dependency file and prime rule to use. For example, if the directory has .c and main.s files, use the .c.o rule because it .c before .s in the list. First, scan the list of its flailes to see if a name for the destination file appears. In the case of main.o, .o appears in the list. Then do checks for the touch rule with when you want to build it and the dependency file from which you want to build it. The dependency file has the same base name as the destination, but a different name. In this case, while checking the .c.o rule, make sure it finds the dependency file with the .c, so it uses this rule. The list of plugs is the goal of a specific function called . Extensions. Different Primles are included in the definition for macro SUFFIXES; dependency list for . SUFFIXES is referenced to the following macro: Table 4-7 List of standard suffixes SUFFIXES= .o .c .c~ .cc .cc. C.C. .y.y~ .l.l~s.s~ .sh .sh. S. S~ .ln \ .h .h~ .f .f~ . F. F~ .mod .mod~ .sym .def .def~ .p.p~ .r .r~ \ .cps . Y. Y. . L. L . PRIDIHI: $(SUFFIXES) $(SUFFIXES) The following example shows makefile to assemble the entire set of executing programs, each with only one source file. Each executing program must be built from a source file that has the same base name and .c attached. For example demo_1 is built from demo_1.c. Note - Just as clean, all the target name is used by the convention. Builds all the goals on your dependency list. Normally, do and do all are usually equivalent. # Makefile for a set of C programs, one source # per program. Source file names have .c # . CFLAGS= -O . KEEP_STATE: everything: demo_1 demo_2 demo_3 demo_4 demo_5 in this case, the trade mark cannot find a suse that corresponds to any of the objectives (via demo_5). So, treat everyone like he has a zero touch. It then looks for a rule to attach and file dependencies with a valid attachment. In the case demo_2, I'd find a file named demo_2.c. Because there is a target entry for the .c rule, along with the corresponding .c file, use this rule for demo_2 from demo_2.c. This makefile program: zap zap: produced without output: $ make program $ When to use explicit target entries vs. Implicit rules Whenever you build a goal from multiple dependencies files, you must ensure that with an explicit target entry that contains a rule for it. When building a goal from a single dependency file, it is often convenient to use an implicit rule. As previous examples appear, make sure that the unique source file is immediately assembled into the corresponding object file or execute. However, it does not have built-in knowledge of how to link a list of object files to the executing program. Also, you can only mneal to those object files that they encounter when reviewing dependencies. It needs a baseline- a destination for which each file of an object in the list (and ultimately each source file) is an dependency. So, for a goal built from multiple dependencies files, you do that needs an explicit rule that ensures a colleague's order, along with a list of dependencies that represents your dependency files. If each of these dependencies files is built from only one source, you can rely on implicit rules for them. Implicit rules and dynamic macros maintain a set of macros dynamically, goal-by-goal. These macros are used quite extensively, especially in the definitions of implicit rules. It's important to understand what they mean. Note - Since they are not explicitly defined in makefile, the convention document dynamic macros with the prefix $-sign attached (in other words, with the display of the macro reference). They are: $@ Name of current target. $? The dependency list is newer than the destination. $< Dependency file name as if selected using an implicit rule. $* The base name of the current destination name that has been taken from the touch). $% For libraries, the name of the member being processed. For more information, see Building Objects Libraries. Implicit rules use these dynamic macros to supply the target or dependency file name at the command line within the rule itself. For example, in the .c.o. rule shown in the following example: .c.o:$(COMPILE.c) $< $(OUTPUT_OPTION) $< is replaced by the dependency file name (in this case, the .c file) for the current destination. Note - The macro OUTPUT_OPTION default has an empty value. Although similar to CFLAGS in the function, it is available as a separate macro designed to transfer an argument to the compiler option to force the compiler's output to a given file name. In the .c: .c: $(LINK.c) $< -o $@ $@ is replaced by the name of the current destination. Because the values for $< and $* macros depend on the order of the extensions in the extension list, you may get surprising results when you use them in the explicit target entry. See Suffix Replacement and Macro References for a strict deterministic method for derived a file name from the associated file name. You can change dynamic modificates of Dynamic Macros by including F and @D D @F in the reference. If there are no /characters in the destination name, $(@D) is assigned a dot character (.) as its value. For example, with the target name /tmp/test, $(@D) has a value /tmp; $(@F) has a value test. Dynamic Macros and Dependency List: Delayed macro references Dynamic macros are assigned to process all and all destinations. They can be used in the target rule, such as, or in the dependency list with the previous additional $ sign per reference. A reference starting with $$is called a belated reference to a macro. For example, the entry: x.o y.o z.o: $$@. BAK cp $@. BAK$@ could be used to derivate x.o from x.o.BAK, and so on for y.o and z.o. Dependency list Read twice This technique works because make reads the dependency list twice, once as part of the initial reading of the entire makefile, and again when processing the target dependencies. In each transition through the list, it performs a macro extension. Because dynamic macros are not specified in initial reading, unless references to them are delayed until the second gateway, they expand to zero strings. String $$ is a reference to a predefined macro '$'. This macro, convenient enough, has a value of $; when it is saved in initial reading, the string $$@ is resolved to $@. In the dependency review, when the result is $@ the macro reference has a value dynamically assigned, make a resolution reference to that value. A notification that evaluates only the target part of the target entry in the first given. Macro Delay as the target name, it generates incorrect results. Makefile: NONE= none all: $(NONE) $(NONE): @: the name of this target is not 'none' produces the following results. $ Mark: Fatal error: I do not know how to make a goal none Rules evaluated When you make an estimated rule part of the target entry only once per use of this command, at the time the rule is executed. Here, the delay in referencing a macro generates incorrect results. There is no transitional closure for suria rules There is no transitional closure for suria rules. If we had a rule about the construction flaff, for example. Y file from . X file, and another one to build . Using the file from . Y file make would not combine their rules to build . Using the file from . X file. You must define intermediate steps as goals, although their entries may have zero rules: trans. Z: trans. Y: In this case, trans. Z is made of trance. Y, if there is one. No-look trans. Y as the target entry, make maybe fail with an error I do not know how to build, as there would be no file dependencies to use. Target entry for trans. Y ensures that he will try to build when he is forever or missing. Since there is no rule in makefile, makefile will apply the appropriate implicit rule, which in this case would mean . X.Y rule. If trance. X exists (or can be retrieved from SCCS), do restore both trances. Y and trance. Z, as necessary. Adding suffix rules Although it supplies you with many useful suffix rules, you can also add a new one of your own. However, when adding new implicit rules, priority should be given to the use of pattern matching rules (see Pattern matching rules: an alternative to the adhesion rule. Unless you need to write implicit rules that are compatible with earlier versions of stamps, you can skip the rest of this section, which describes the traditional way of adding implicit rules for makefiles. (The procedure for adding implicit rules is here for compatibility with previous versions of make.) Adding a touch rule is a two-step process. First, you need to add the hints of destination and dependency files to the touch list by providing them as an addict. A special target. Because dependency lists are piling up, you can add last names to the list by adding another entry for that goal, such as: . SUFFIXES: .ms .tr Other, you must add the target entry for the append rule: ms.tr: troff -t -ms $< > $@ Makefile with these entries can be used to create source document files Containing ms macros (.ms files) into troff output files (.tr files): $ make doc.tr troff -t-ms doc.ms > doc.tr Entries in the suffixs list are contained in the SUFFIXES macro. To insert an add-on to the list header, first clear its value by dobaving the entry for . An dependent SUFFIXES target. This is an exception to the rule that dependency lists accumulate. You can clear the definition for this objective by supplying a target entry without dependencies and rules such as this: . SUFFIXES: Then you can add another entry containing new last names, followed by a reference to the SUFFIXES macro as below screenshot shown. I'm going to need you to get a good time. SUFFIXES: .ms .tr $(SUFFIXES) Pattern-Matching Rules: Alternatives to Suffix Rules The pattern matching rule is similar to the implicit rule in the function. Pattern matching rules are easier to write and stronger because you can define a relationship between a goal and an addiction based on prefixes (including path names) and suffixes, or both. The pattern matching rule is the target form entry: tp%ts: the dp%ds rule, where TP and ts are optional prefix and suffix in the goal name, dp and ds are (optional) prefix and suffix in the name of dependency, and % is a wild card that represents a base name common to both. Note - Check the rules for matching patterns before the touch rules. Although this allows you to override standard implicit rules, this is not recommended. If there is no goal building rule, look for a rule that matches the pattern before verifying that there is a rule for a name. If you could use a pattern matching rule, do it. If the target entry for the pattern match rule does not contain any rule, make the destination file processes as if it had an explicit target entry without the rule; therefore, it looks for a pin rule, tries to retrieve a version of the destination file from the SCCS, and ultimately treats the goal as a zero rule (marking this goal as updated in the current startup). The rule that corresponds to the pattern for formatting the troffa source file into the troffa output file looks like: %.tr: %.ms troff -t -ms$< > $@ make's Default Suffix Rules and Predefined Macros The following tables show the standard set of appetizer rules and predefined macros that are attached to the default makefile, /usr/share/lib/make/make.rules. Table 4-8 Standard Rules suffix Using Suffix Rule Command Line Name .s.o$(COMPILE.s) -o$$$< s(COMPILE.s) -o $$< $$$.s.a.a.a.a.a.a.)$.100.000000000000000000000000000000000000000000000000000000000000000. S.o$(COMPILE. S) -o $@ $< . S.a$(COMPILE. S) -o $% $$(AR) $(ARFLAGS) $@ $%$(RM) $% C Files (.c Rules) .c$(LINK.c) -o $@ $< $(LDLIBS) $.c.ln$(LINT.c) $(OUTPUT_OPTION) -i $< .c.o.$(COMPILE.c) $(OUTPUT_OPTION) $< .c.a.compile.c) -o $% $< $(AR) $(ARFLAGS) $@ $%$(RM) $% C++ Files .cc$(LINK.cc) -o $@$< $(LDLIBS) .cc.o$(COMPILE.cc) $. (OUTPUT_OPTION) $< .cc.a$(COMPILE.cc) -o $%$< $(AR) $(ARFLAGS) $@ $% $(RM) $% C++ Files (SVr4 style). C$(Link. C) -o $@$< $(LDFLAGS) $*.c . C.o$(COMPILE. C) $<. C.a$(COMPILE. C) $< $(AR) $(ARFLAGS) $@ $*.o $(RM) -f$*.o FORTRAN 77 Files .cc.o$(LINK.f) -o $@$< $(LDLIBS) .cc.a$(COMPILE.f) $(OUTPUT_OPTION) $< $(COMPILE.f) -o $%$< $(AR) $(ARFLAGS) $@$% $(RM) $% . F$(LINK. F) -o $@$< $(LDLIBS). F.o$(COMPILE. F) $ < . F.a$(COMPILE. F) -o $% $< $(AR) $(AR) $@ $% $(RM) $% lex Datoteke .l$(RM) $*.c $(LEX.l) $< > $*.c $(LINK.c) -o $@ $*.c $(LDLIBS) $(RM) $*.c .l.c$(RM) $@ $(LEX.l) $< > $@ .l.ln$(RM) $*.c $(LEX.l) $< > $*.c $(LINT.c) -o $@ -i $*.c $(RM) $*.c .l.o$. (RM) $*.c $(LEX.l) $< > $*.c $(COMPILE.c) -o $@ $*.c $(RM) $*.c . L.C$(LEX) $(LFLAGS) $< . L.o$(LEX)(LFLAGS) $< $(COMPILE. C) lex.yy.c . L.orm -f lex.yy.c mv lex.yy.o $@ Modula 2 Files .mod$(COMPILE.mod) -o $@ -e $@ $< .mod.o$(COMPILE.mod) -o $@ $< .def.sym$(COMPILE.def) -o $@ $< NeWS .cps.h$(CPS) $(CPSFLAGS) $*.cps Pascal Files .p$(LINK.p) -o $@ $< $(LDLIBS) .p.o$(COMPILE.p) $(OUTPUT_OPTION) $< Ratfor Files .r$(LINK.r) -o $@ $< $(LDLIBS) .r.o$(COMPILE.r) $(OUTPUT_OPTION) $< .r.a$(COMPILE.r) -o $% $< $(AR) $(ARFLAGS) $@ $% $(RM) $% Shell Scripts .sh $(RM) $@ cat $< >$@ chmod +x $@ yacc Files (.yc Rules) .y$(YACC.y) $< $(LINK.c) -o $@ y.tab.c $(LDLIBS) $(RM) y.tab.c .y.c$(YACC.y) $< mv y.tab.c $@ .y.ln$(YACC.y) $< $(LINT.c) -o $@ -i y.tab.c $(RM) y.tab.c .y.o$(YACC.y) $< $(COMPILE.c) -o $@ y.tab.c $(RM) y.tab.c yacc Files (SVr4) . Y.C$(YACC) $(YFLAGS) $< mv y.tab.c $@ . Y.o$(YACC) $(YFLAGS) $< $(COMPILE.c) y.tab.c rm -f y.tab.c mv y.tab.o $@ Table 4-9 Predefinirani i dinamički makroi Koristi makro Default Value Library Archive ARar ARFLAGSrv Assembler Commands ASas ASFLAGS COMPILE.s$(AS) $(ASFLAGS) COMPILE. S$(CC) $(ASFLAGS) $(CPPFLAGS) -target -c C Compiler Ukazi CCcc CFLAGS CPPFLAGS COMPILE.c$(CC) $(CFLAGS) $(CPPFLAG -c LINK.c $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) C++ Compiler Commands [Za kompatibilnost unaokolo, C++ makroi ima alternativne oblike. Za C++C lahko namesto tega uporabite CCC; namesto C++FLAGS lahko uporabljate CCFLAGS; za COMPILE. C, lahko uporabite COMPILE.cc; in LINK.cc lahko zamenjate za LINK.C. Upoštevajte, da bodo ti nadomestni obrazci izginili za prihodnje izdaje.] CCCCC CCFLAGS COMPILE.cc$(CCC) $(CCFLAGS) $(CPPFLAGS) -c LINK.cc$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS) C++ SVr4 Compiler Commands (C++C)CC (C++FLAGS)-O COMPILE. C$(C++C) $(C++FLAGS) $(CPPFLAGS) -c LINK. C$(C++C) $(C++FLAGS) $(CPPFLAGS) $(LDFLAGS) -target FORTRAN 77 Compiler Commands FC in SVr4f77 FFLAGS COMPILE.f$(FC) $(FFLAGS) -c LINK.f $(FC) $(FFLAGS) $(LDFLAGS) COMPILE. F$(FC) $(FFLAGS) $(CPPFLAGS) -c LINK. F $(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS) Link Editor Command LDld LDFLAGS lex Command LEXlex LFLAGS LEX.l $(LEX) $(LFLAGS) -t lint Command LINTlint LINTFLAGS LINT.c $(LINT) $(LINTFLAGS) $(CPPFLAGS) Modula 2 Commands M2Cm2c M2FLAGS MODFLAGS DEFFLAGS COMPILE.def$(M2C) $(M2FLAGS) $(DEFFLAGS) COMPILE.mod$(M2C) $(M2FLAGS) $(MODFLAGS) NeWS CPScps CPSFLAGS Pascal Compiler Commands PCPC PFLAGS COMPILE.p$(PC) $(PFLAGS) $(CPPFLAGS) -c LINK.p$(PC) $(PFLAGS) $(CPPFLAGS) $(LD FLAGS) Ratfor Compilation Commands RFLAGS COMPILE.r$(FC) $(FFLAGS) $(RFLAGS) -c LINK.r$(FC) $(FFLAGS) $(LDFLAGS) rm RMrm -f yacc Command YACCyacc YACCyacc YACC.y$(YACC) $(YFLAGS) Suffixes List SUFFIXES.o .c .c~ .cc .cc~ . C. C~ .y .y~ .l .l~ .s .s~ .sh .sh~ . S. S~ .ln .h .h~ .f .f~ . F. F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .cps .cps~ . Y. Y~ . L. L~ SCCS dobili poveljstvo . SCCS_GETsccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@ -G$@ SCCSGETFLAGS-s