(MINI-UNIX) 1125

REGENERATING SYSTEM SOFTWARE

This document discusses how to assemble or compile various parts of the MINI-UNIX system software. This may be necessary because a command or library is accidentally deleted or otherwise destroyed; also, it may be desirable to install a modified version of some command or library routine. It should be noted that in the system as distributed, there are quite a few commands that depend to some degree on the current configuration of the system; thus in any new system modifications to some commands are advisable. Most of the likely modifications relate to the standard disk devices contained in the system. For example, the *df* ("disk free") command has built into it the names of the standardly present disk storage drives (e.g. "/dev/rk0", "/dev/rk1"). *Df* takes an argument to indicate which disk to examine, but it is convenient if its default argument is adjusted to reflect the ordinarily present devices.

The companion document "Setting up MINI-UNIX" discusses which commands are likely to require changes.

The greater part of the source files for commands resides in several subdirectories of the directory /usr/source. Each directory and subdirectory contains a "run" file which contains "shell" sequences for re-compiling all commands in that directory. These subdirectories, and a general description of their contents, are

s1 Source files for most commands with names beginning with "a" through "l".

s2 Source files for most commands with names beginning with "m" through "z".

s3 Source files for subroutines contained in the standard system library, "/lib/liba.a" (see below).

s4 Source files for the C library, "/lib/libc.a" (see below).

s5 Source files for more of the C library.

s7 Contains the source files for all the text formatters roff, nroff and neqn. They are separate because they overloaded the s2 directory.

as Source files for assembler.

c Source files for C compiler.

cref Source files for cross reference program.

fort Source files for Fortran Compiler.

iolib Source files for Portable C library.

m6 Source files for Macro Processor.

mdec Source files for utility and boot programs.

rat Source files for Ratfor.

salloc Source files for storage allocation routines.

sno Source files for Snobol Interpreter.

tmg Source files for TMG compiler-compiler.

yacc Source files for YACC compiler-compiler.

To regenerate most commands in the s1 and s2 directories is straightforward. The appropriate directory will contain one or more source files for the command. These will all have the suffix ".s" if the command is written in assembler language, or ".c" if it is written in C. The first part of the name begins with the name of the command. If there are several source files, the command name will be followed by a character which distinguishes the several files. it is typically "1", "2", ...; Sometimes the last is "x". For example, The "bas" command has source files (in s1) called "bas0.s", "bas1.s", ..., "bas4.s", "basx.s". In all cases, the lexicographical order of the distinguishing character is the order in which the source files should be compiled or assembled. Thus, for example, the way to reassemble a new "bas" is to say (in s1)

```
as bas?.s
```

Some of the assembly-language commands are completely stand-alone and require no inclusion of routines from system libraries. Unfortunately there is no *a priori* way of determining which need library routines. A simple *a posteriori* method is to assemble the command as discussed above, then say

```
nm -u a.out
```

which will list the undefined external symbols. If any appear, the loader should be called by saying

```
ld a.out -l
```

However *all* assembly-language programs require the application of the link editor *ld* (also loosely called the loader), since the link editor automatically relocates the object code to 060000 for a 12K MINI-UNIX system.

One important command which needs slightly special treatment is "tp" which has to be loaded with the C library:

```
as tp?.s
ld a.out -l -lc
```

because it calls the C-language ctime subroutine.

As it happens, there are no commands written in C (except those described below) which consist of more than one file. The command "com.c" can therefore be recompiled simply by saying

```
cc -O com.c
```

Here the "-O" indicates the desire to use the optimizer pass of the C compiler.

Some of the most important commands are considerably more complicated to regenerate, and these are discussed specifically below. The contents of libraries are also discussed.

AS

The assembler consists of two executable files: /bin/as and /lib/as2. The first is the 0-th pass: it reads the source program, converts it to an intermediate form in a temporary file "/tmp/atm0?", and estimates the final locations of symbols. It also makes two or three other temporary files which contain the ordinary symbol table, a table of temporary symbols (like n:) and possibly an overflow intermediate file. The program /lib/as2 acts as an ordinary two-pass assembler with input taken from the files produced by /bin/as.

The source files for /bin/as are named "/usr/source/s1/as1?.s" (there are 9 of them); /lib/as2 is produced from the source files "/usr/source/s1/as2?.s"; they likewise are 9 in number. Considerable care should be exercised in replacing either component of the assembler. Remember that if the assembler is lost, the only recourse is to replace it from some backup storage; a broken assembler cannot assemble itself.

C

The C compiler consists of four files: "/bin/cc", which expands compiler control lines and which calls the phases of the compiler proper, the assembler, and the loader; "/lib/c0", which is the first phase of the compiler; "/lib/c1", which is the second phase of the compiler ; and "/lib/c2", which is the optional third phase optimizer. The loss of the C compiler is as serious as that of the assembler.

The source for /bin/cc resides in "/usr/source/s1/cc.c". Its loss alone is not fatal. Provided that prog.c does not contain any compiler control lines, prog.c can be compiled by

```
/lib/c0 prog.c temp0 temp1
/lib/c1 temp0 temp1 temp2
as - temp2
ld /lib/crt0.o a.out -lc -l
```

If /bin/cc is lost, it can be recovered in this way, since it contains no compiler control lines.

The source for the compiler proper is in the directory /usr/c. The first phase (c0) is generated from the files c00.c, ..., c05.c, which must be compiled by C; c0t.s, which must be assembled; and c0h.c, which is a header file which should not be compiled but is a file *included* by the C programs of the first phase. The c0t.s program contains a parameter "fpp" which determines whether C is to be used on a machine which has PDP 11/45 floating-point hardware; it should be set to 1 if so, 0 if not. In the standard system fpp is 0. To make a new /lib/c0, assemble c0t.s, name the output c0t.o, and

```
cc c0t.o c0[0-5].c
```

Before installing the new c0, it is prudent to save the old one someplace.

The second phase of C (/lib/c1) is generated from the C source files c10.c, ..., c13.c, the assembly-language program c1t.s, the include-file c1h.c, and a library of object-code tables called tab.a. To generate a new second phase, assemble c1t.s, call it c1t.o, and

```
cc c1t.o c1[0-3].c tab.a
```

It is likewise prudent to save c1 before installing a new version. In fact in general it is wise to save the object files for the C compiler so that if disaster strikes C can be reconstituted without a working version of the compiler.

In a similar manner, the third phase of the C compiler (/lib/c2) is made up from the files c20.c and c21.c together with c2h.c. Its loss is not critical since it is completely optional.

The library of tables mentioned above is generated from the files regtab.s, sptab.s, cctab.s, and efftab.s. The order is not important. These ".s" files are not in fact assembler source; they must be converted by use of the *cvopt* program, whose source and object are located in the C directory. For example:

```
cvopt regtab.s temp
as temp
mv a.out regtab.o
ar r tab.a regtab.o
```

Refer to the *run* shell sequence in the C directory for more complete details.

FORTRAN

Probably because it is a very large subsystem written entirely in assembly language, Fortran is quite complicated to regenerate. On the other hand, Fortran is vital only to its own users; since none of the compiler nor any important part of the run-time system is written in Fortran, both can be regenerated in case of loss.

The *fc* command itself is essentially equivalent to a long shell command file; for a single source program *prog.f*, it amounts to saying

```
/usr/fort/fc1 prog.f
as - f.tmp1
ld /lib/fr0.o a.out /lib/filib.a -lf -l
```

Thus, /usr/fort/fc1 is the compiler proper; fc1 leaves its output in the current directory in the file "f.tmp1". /lib/fr0.o is the runtime startoff. Filib.a is the library of operators; Fortran is essentially interpretive, and operations such as "add floating variable to floating variable" are short routines loaded from the filib.a library.

/lib/libf.a (specified by the "-lf") is an archive file containing the language builtin functions plus a few others. The standard assembly language library (the "-l", or /lib/liba.a) is referenced by certain of the builtin functions (for routines like *sin*).

The source and object of the compiler are stored in subdirectories of the /usr/fort directory, named f1, f2, f3, f4, and fx. The first four represent putatively separable phases; the last contains subroutines used by several of the phases. Each directory contains an archive file with the object programs corresponding to the source programs in that directory; it is called f?_o.a where "?" is the last letter in the directory name. To reload Fortran from theses libraries, see the Shell command file /usr/fort/ld, which should contain

```
ld -u pass1 -u pass2 -u pass3 -u pass4 \
  f1/f1o.a f2/f2o.a f3/f3o.a f4/f4o.a fx/fxo.a -l
```

Each subdirectory should contain a Shell command file called "as" which assembles a particular file in that subdirectory; the one for f1 contains, for example,

```
as ../fx/fhd.s f1$1.s
mv a.out $1.o
ar r f1o.a f1$1.o
rm f1$1.o
```

so that the command

```
sh as 5
```

would assemble f15.s (preceded by the definition file /usr/fort/fx/fhd.s) and place it in the library for that subdirectory.

Actually we hope that no one will be required to make a new Fortran from the pieces, or fix it themselves. For those who are curious, we will say that phase 1 analyzes declarations, phase 2 does storage allocation, phase 3 code generation, and phase 4 puts out constants, code from format and data statements, and the actual storage-reserving code for variables.

MINI-UNIX

The source and object programs for MINI-UNIX are kept in */usr/sys* and three subdirectories therein. The main directory contains several files with names ending in ".h"; these are header files which are picked up (via "#include ...") as required by each system module. The files lib1 and lib2 are libraries (archives) of (almost) all the object programs in the system. Lib1 is made from the source programs in the subdirectory *mxsys;* lib2 is made from the programs in subdirectory *dev.* The latter consists mostly of the device drivers together with a few other things, the former is the rest of the system.

Subdirectory *source* contains the source code for all MINI-UNIX user programs which have been modified from the standard UNIX programs.

The *mxsys* subdirectory contains the progams which control the device configuration of the system. *Low.s* specifies the contents of the interrupt vectors; *conf.c* contains the tables which relate device numbers to handler routines. A third program, mch.s, contains all the machine-language code in the system. A fourth program, emul.s contains the software emulation package to handle the extended instruction set, i.e. those instructions which are not implemented in the PDP-11/20 and PDP-11/10 processor hardware.

To recreate the system, compile conf.c and move the output to /usr/sys/conf.o. Assemble low.s and move the output to /usr/sys/low.o. Then change to /usr/sys, and load the whole system:

```
ld -a -x low.o conf.o lib1 lib2
```

For convenience, this command line has been placed into /usr/sys/shld. Consult the "run" file and the companion document "Setting Up MINI-UNIX - Sixth Edition" for further details on creating a new system.

When the *ld* is done, the new system is present as *a.out.* It can also be tested by putting it on tape (tp-I) and using tboot or mboot, or directly using uboot (boot procedures-VIII). When you have satisfied yourself that it works, it should be copied to /mx so that programs like ps (I) can use it to pick up addresses in the system.

A word of caution is in order here. The size of a.out must be less than 055400 bytes for the system to run properly. If the system is bigger than this, its size can be reduced by removing one more system buffer (NBUF in param.h) and recompiling all of the system source using the "shs" shell sequence file in the *mxsys* subdirectory. If enough space cannot be achieved in this manner, the system size must grow beyond 12K words to the next convenient boundary. This requires major surgery; therefore think twice before you do it. To form a new system with the size greater than 12K words, the file "mxsys/param.h" must be edited to change the following three parameters:

UCORE
TOPSYS
SWPSIZ.

Re-compile the complete system using the "run" command as before. A new root file system must be made and all system command programs must be re-compiled. Before proceeding, change the value of the TOPSYS parameter in "sys/source/ld.c" to the appropriate value and re-compile the link editor *ld.* At some point the value of "uorg" in sys/source/db1.s must also be changed and the debugger *db* re-assembled and link-edited for the new root file system. The complete re-compilation of all user command programs is likely to take the better part of a day.

To install a new device driver, compile it and place the object in lib2 if necessary. (All the device drivers distributed with the system are already there.) The device's interrupt vector must be entered in low.s. This involves placing a pointer to a callout routine and the device's priority level in the vector. As an example, consider installing the interrupt vector for DC11 number 2. Its receiver interrupts at location 320 and the transmitter at 324, both at priority level 5. Then low.s has:

```
. = 320^.
dcin; br5+2
dcou; br5+2
```

First, notice that the entries in low.s must be in order, since the assembler does not permit moving the location counter "." backwards. The assembler also does not permit assignation of an absolute number to ".", which is the reason for the ". = 320^." subterfuge; consult the Assembler Manual for the meaning of the notation. If a constant smaller than 16(10) is added to the priority level, this number will be available as the first argument of the interrupt routine. This stratagem is used when several similar devices share the same interrupt routine.

At the end of low.s, add

```
        .globl  _dcrint
dcin:
        jsr     r0,call; _dcrint

        .globl  _dcxint
dcou:
        jsr     r0,call; _dcxint
```

The *call* routine saves registers as required and makes a C-style call on the actual interrupt routine (here _dcrint and _dcxint) named after the jsr instruction. When the routine returns, *call* restores the registers and performs an rti instruction.

To install a new device thus requires knowing the name of its interrupt routines. These routines are in general easily found in the driver; they typically end in the letters "int" or "intr." Notice that external names in C programs have an underscore "_" prepended to them.

The second step which must be performed to add a new device is to add it to the configuration table /usr/sys/mxsys/conf.c. This file contains two subtables, one for block-type devices, and one for character-type devices. Block devices include disks, DECtape, and magtape. All other devices are character devices. A line in each of these tables gives all the information the system needs to know about the device handler; the ordinal position of the line in the table implies its major device number, starting at 0. The appropriate editing must be done in conf.c and then it must be re-compiled and the object module moved to /usr/sys/conf.o.

There are four subentries per line in the block device table, which give its open routine, close routine, strategy routine, and device table. The open and close routines may be nonexistent, in which case the name "nulldev" is given; this routine merely returns. The strategy routine is called to do any I/O, and the device table contains status information for the device.

For character devices, each line in the table specifies a routine for open, close, read, and write, and one which sets and returns device-specific status (used, for example, for stty and gtty on typewriters). If there is no open or close routine, "nulldev" may be given; if there is no read, write, or status routine, "nodev" may be given. This return sets an error flag and returns.

The above discussion is admittedly rather cryptic in the absence of a general description of system I/O interfaces.

The final step which must be taken to install a device is to make a special file for it. This is done by mknod (VIII), to which you must specify the device class (block or character), major device number (relative line in the configuration table) and minor device number (which is made available to the driver at appropriate times).