Setting up UUCP for v7 Unix

1. Getting started with v7 Unix

I'll start with some preliminaries for those unfamiliar with the system. Start by logging in as root, password root.

If this is your first time logging in and you are not familiar with Unix/Linux, now is a good time to make the system yours by changing the root password.

At the # prompt, type “passwd”.

On login, the system will notify you that you have mail. You can read this by typing "mail", and delete it by pressing "d" at the “?” prompt, so that it doesn’t bug you on future logins.

The mail tells you that you have secret mail. v7 Unix has a "secret mail" facility that didn't survive in modern Unixes. You can read the secret mail message with xget, send one with xsend (not that that's really relevant to modern use cases for v7 Unix), and change your password for the secret mail system with enroll.

2. Recompiling the kernel

Now let’s get down to business with setting up UUCP. The first thing we need to do is recompile the kernel to be able to use the DZ11 terminal multiplexer simulated by simh. The DC11 lines included in the default boot.ini for v7 aren't 8-bit clean, so they'll do nasty things like mangling the high bit of characters and dropping control characters. Because the UUCP implementation for v7 Unix (which confusingly enough, is called "version 2 UUCP") doesn't support any of the protocols that were later developed for lines that weren't 8-bit clean, this will prevent UUCP on v7 from talking to Taylor UUCP on the Pi.

Figuring out that the DZ11 was needed consumed a good chunk of the time I spent working on this.

I used http://www.ljosa.com/~ljosa/v7-dz11 as a guide for setting up the dz11, however, some of his instructions assume a bit more knowledge of v7 Unix than even a fairly seasoned veteran of modern Linux is likely to have, let alone anyone who has come from Windows or DEC OSes, so I will go over them in detail.

Things you should type will be indented by one tab from the rest of my instructions and will include the prompt character (if any) provided by the Unix shell or by the program you are running. If there is a large amount for you to copy in, it will be between horizontal lines:

```
like this.
```

First, we set the time. The compile scripts check file timestamps against the current date to determine what has been modified and needs to be recompiled. As it doesn't have any real time clock, or an internet connection with ntp, the system boots up with the date set to sometime around the Unix epoch (Jan 1st 1970), and all the files on the system are from the late 1970s (in the future as far as the system is concerned).

Unfortunately, while v7 Unix itself is Y2K compliant, its program for *setting* the date isn't, and only takes 2-digit years. So set the time to sometime after 1980, but well before 2000, so that it doesn't roll over to 2000, causing files to be created with timestamps that are later than any date we can set the system date to. So our first command is:

```
# date yymmddhhmm
```

or, optionally:

```
# date yymmddhhmm.ss
```

Next, move to the configuration directory for the kernel sources:

```
# cd /usr/sys/conf
```

Now, remove all the *.o files in that directory, as well as the program mkconf, which needs to be rebuilt, to make sure we get a fresh mkconf executable and rather than using the old one.

```
# rm *.o
```

Now, open up the file mkconf.c in v7's text editor, ed:

```
# ed mkconf.c
```

When you load or save a file, ed will print the number of characters in the file. Ed does not print a prompt, but it will print a ? if it doesn't understand your input. This can happen if you accidentally bump an arrow key, which sends control characters that modern systems interpret as cursor movement commands, but that v7 Unix interprets as part of your command. On modern systems, when you accidentally send a control sequence to a program that doesn't understand it, you usually get something back that looks like ^]]A, but v7 Unix doesn't send anything back in this scenario, so you can get a random "command not found" from what looks like a validly typed command if you bump a key that sends control characters, like ESC, or an arrow key.

Now, on whatever computer you're logged into the Pi from, download the file

http://www.ljosa.com/~ljosa/mkconf.c.ed

or copy the script between the lines below. This is the set of commands that will cause ed to add the code we need to mkconf.c. You can just paste this into your terminal and hit "enter" to submit the last command. The write-up at ljosa.com warns:

"If you transfer the patch to your PDP-11 using cut and paste in an xterm or some similar method, verify that tab characters have not been translated into spaces. Mkconf generates the assembly file l.s, and the assembler distinguises between tabs and spaces."

This was not a problem for me, but double check. There are some spaces in the file, and some tabs. Make sure none of them have been converted to the other type of whitespace. You can exit ed without saving by pressing "q" (for **Q**uit) on a new line if you run into this issue.

If everything copies in fine, type:

```
     w
```

to save (**W**rite) the file, and:

```
     q
```

to quit.

In the script below "249a" means "**A**ppend the following text after line 249" (likewise for 45 a), and a dot on a line by itself means "stop adding text".

The rest of the script is text to be added.

```
249a

        "dz",
        0, 300, CHAR+INTR,
     "    dzin; br5+%d.\n     dzou; br5+%d.",
        ".globl _dzrint\ndzin:  jsr     r0,call; jmp _dzrint\n",
     ".globl   _dzxint\ndzou: jsr  r0,call; jmp _dzxint\n",
     "",
        "       dzopen, dzclose, dzread, dzwrite, dzioctl, nulldev,
dz_tty,",
     "",
        "int    dzopen(), dzclose(), dzread(), dzwrite(), dzioctl();\
nstruct   tty  dz_tty[];",
.
45a
     "dz",
.
```

The next step is to rebuild mkconf

```
     # cc mkconf.c -o mkconf
```

Now, we need to create a new configuration file for the kernel. Copy "myconf" (*not* mkconf! There's only one letter of difference between the two!) which was used to create the kernel we're currently running, to a new file.

You can call this whatever you want, but so that you know what its for if you come across it later, it should end in "conf". I used the name I chose for my v7 system to use over UUCP, "anduin":

```
# cp myconf anduinconf
```

Modern practice would probably be to end the filename in .conf, but the rest of the conf files in this directory don't have a dot, so I'll stick with that convention, not that it matters.

Now edit your new configuration file:

```
# ed anduinconf
```

The command n,mp will **P**rint lines n through m in the file. $ means the last line, so 1,$p will print the whole file. Do that so that you can see the whole file.

```
1,$p
```

The output will look like this:

```
hp
root hp 0
swap hp 1
swplo 0
nswap 8778
tm
4dc
```

When setting this up, I found that the configuration mkconf reported generating seemed to have the DC11 and DZ11 lines configured in a way that caused them to interfere, which caused the kernel to crash. /Enclosing text between two slashes/ causes ed to search for it and go to that line. Search for the line /4dc/, and **D**elete it.

```
/4dc/
d
```

You can even do this as one command line if you want:

```
/4dc/d
```

Append the following to the file, and print the file again to make sure everything looks correct.

```
a
dz
pack
.
1,$p
```

You should get the following output:

```
hp
root hp 0
swap hp 1
swplo 0
nswap 8778
tm
dz
pack
```

pack adds the packet driver, which I'm not sure is needed, but the UUCP documentation (see http://a.papnet.eu/UNIX/v7/files/doc/36_uucpimp.pdf) mentions it and says that certain things need to be done if it's not present, so I figured it was better safe than sorry to include it. "dz" adds support for the DZ11.

As long as everything looks as shown above, save and exit:

```
w
q
```

Now we need to run mkconf. Similar programs on modern Unices generally take a filename as an argument, but mkconf just reads from standard input, so the syntax to read in our configuration file is a bit different than usual:

```
# mkconf < anduinconf
```

You should get the following back:

```
console at 60
clock at 100
clock at 104
parity at 114
tm at 224
hp at 254
dz at 300
```

Now we can rebuild the kernel. I use "make all" in addition to "make unix" here to be safe, as I don't know what's already built on the image we start with, but on subsequent invocations (if you ever feel the need to rebuild the kernel again), you should be able to just type "make unix". For those used to modern conventions for writing makefiles, note that "make all" does not really build everything, you still have to type "make unix" afterwards.

```
# make all
# make unix
```

Now there will be a file called "unix" in /usr/sys/conf. This is the kernel.
Copy it to the root directory ***under a new name***. I've tested this configuration, so I'm quite certain it will work, but don't believe me: it's best practice on any operating system not to delete your old kernel before you've tested that the new one works. If you overwrite the old kernel before testing the new one, the system will be unbootable.

```
#cp unix /anduinunix
```

Note the leading slash on the new name. If you don't include it, it will go to the current directory instead of the root directory, and the bootloader won't find the kernel when you try to boot it.

Now, as long as you're logged in from the console, and not any of the DC11 lines, remove the device files for the DC11s. This will render those lines inoperable, but we'll be replacing them with the DZ11s, and the device files will need to be set up a bit differently for the DZ11s.

```
#rm /dev/tty00
#rm /dev/tty01
#rm /dev/tty02
#rm /dev/tty03
```

You could type "rm /dev/tty0*" here to do it all in one command, but **be careful**, you don't want to accidentally remove /dev/tty, which is the device file that each individual program sees as its own terminal.

Whatever you do: **DO NOT** type "rm /dev/tty*".

Now, open c.c in ed

```
#ed c.c
```

Type /dz/ 3 times to go to the 3rd line containing the string /dz/

```
/dz/
/dz/
/dz/
```

The third time you type /dz/, ed should print this line:

```
     dzopen, dzclose, dzread, dzwrite, dzioctl, nulldev, dz_tty,
/* dz = 19 */
```

Make note of the number in the comment at the end of the line (in my case it's `/* dz = 19 */`). You'll need this number. When we recreate the /dev/tty0x files, this is what will tell the operating system which terminal driver to use for the device file.

We're not changing anything, so leave ed without saving.

```
     q
```

Now we (re)create the device files. Change to the /dev directory:

```
     # cd /dev
```

We apparently need to tell simh to give us 16 dz lines in boot.ini for our new kernel not to crash. However, we don't need to have device files for all of them, and we'll need to do a lot of typing for 16 lines. Be aware, however, that any lines that you don't set up will not respond if you telnet into them. If you set 16 up, you may want to type one instance of the mknod command below, copy and paste it to the number of required lines, then edit the lines to have the required numbers, and paste the whole thing into your terminal when you're done.

First of all, because /etc is not in your path, and the mknod command is located in /etc, add /etc to your path to save typing:

```
     #PATH=$PATH:/etc
```

The command "mknod filename c n m" tells the system to create a **C**haracter device file called "filename" with major number (device type) n, and minor number (the id for that specific device) m. So "mknod /dev/tty00 c 19 0" means "Create a character device of type 19, unit number 0".

Type one mknod line for every DZ11 line you want to use. Use the major number that **you** found in c.c, not the "19" that I have here (though your number may well be 19):

```
     # mknod /dev/tty00 c 19 0
     # mknod /dev/tty01 c 19 1
     # mknod /dev/tty02 c 19 2
     ...
```

Now open up the file /etc/ttys in ed. This file tells the system which devices it should run login prompts on, what speeds they should use, etc.

```
      # ed /etc/ttys
```

Display the whole file to get an idea of what it looks like.

```
      1,$p
```

You should see something like this:

```
      14console
      10tty00
      10tty01
      10tty02
      10tty03
      00tty04
      00tty05
      00tty06
      00tty07
      ...
```

Leave the console line as it is.

Change the first four lines, which are already set up for the DC11, as shown. The first character is either "1" to indicate that a login prompt should be sent on that line, or "0" to indicate that no logins should be accepted on that line. The second tells the system what types of modems and terminals to expect on that line.

The following command is a **S**ubtitution command. 2,5 means to act on lines 2 through 5. Within those lines, ed will replace all instances of "10" with "12". The "p" at the end causes the last line acted on to be printed once the substitution has been performed.

```
      2,5s/10/12/p
```

For the rest of the lines we've added, the system isn't yet configured to allow logons, so we need to add a one in the leading digit.

```
      6,17s/00/12/p
```

For now, I will not be considering the case of having UUCP dial out to other systems from v7, only to accept calls from other systems, as I do not at the moment have serial set up on my PiDP, and simh can only accept logins over telnet, not call out (though I can imagine a setup where you had a program that telnetted in to simh when it started up and then waited for the emulated system to send "dialing" commands, whereupon it would telnet or ssh to another system).

If you have a serial line connected to another system on your PiDP, or otherwise want to look into configuring UUCP for dialout yourself, you may want to leave a few lines free (with no login process running). In that case, instead of the "6,17s... " line above, you might want:

```
6,15s/00/12/p
16,17s/00/02/p
```

The file contains lines for ttys up to tty32, but since we'll only be adding 16 DZ lines to our simh configuration, leave everything after tty15 as-is.

Now ask ed to print the file to make sure everything looks ok

```
1,$p
```

You should get something like this:

```
14console
12tty00
12tty01
12tty02
...

...
12tty14
12tty15
00tty16
00tty17
00tty18
...

...
00tty32
```

If everything looks good, save and exit

```
w
q
```

Now it's time to boot with the new kernel.

First, bring down the system to single-user mode. This was more critical in the old days than it is running today on a PiDP, as you probably won't have a lot of users logged in who might decide to save a file at any moment while you're trying to shut down, but better safe than sorry.

```
# kill -1 1
```

This sends signal 1 "hangup" to process 1, which is "init", the first userpace program that the kernel starts at startup, which manages the rest of the system. On modern Unices, sending SIGHUP to init causes it to reload its configuration, not to go to single user mode, and what signal numbers do what has changed, but "kill" largerly works the same way.

Now that we're in single user mode and are sure that some random user (or automated background process) won't do something that writes to the disk while we're trying to shut down, we can flush pending writes to disk. Best practice, in the days before newfangled programs like "shutdown" that bring the system down automatically on modern systems, was to do this three times for paranoia's sake.

```
# sync
# sync
# sync
```

Then hit "CTRL-E" (or flip the Enable/Halt switch to "halt", if you're in reach of the panel) to stop the processor.

Now copy the boot.ini file for Unix7 on the pi to boot.ini.bak, and open up boot.ini in your prefered modern editor.

Take 5 minutes to write a post to comp.editors flaming all the other editors that are CLEARLY INFERIOR to your prefered editor (because our end goal is to set up UUCP, the protocol that USENET ran on, and what would USENET be without holy wars over text editors?).

cough emacs^H^H^H^H^H vim^H^H^H C-x M-c M-butterfly *cough*

Then delete your post, and breathe a huge sigh of relief, because whatever side you take in the holy wars, at least you don't have to use ed for this :-)

NB - ed is part of the POSIX standard, so you generally will find it on modern Unices, such as Raspbian. So the masochistic can use it if they want to.

So, back to editing boot.ini:

Delete the lines:

```
set dci en
set dci lines=8
```

And add the following lines:

```
set dz lines=16
att dz -m 4107
att dz -am line=15,4108;notelnet
set dz 8b
```

4107 can be replaced with whatever port you prefer for telnet. I'm tryng to keep distinct telnet ports for different emulated systems.

lines=16 and the -m option have to be there, according to the ljosa.com writeup. 8b is to make sure we're 8-bit clean.

The second “att dz” line also has to be there, and the “notelnet” bit is critical. This was something I had not yet discovered when the initial version of this guide was written. Without it, when simh receives a telnet connection, it will assume that the program it’s speaking to on the other end is telnet. Taylor UUCP does *not* use the telnet protocol on the port it connects to, it just blasts over the raw data, and expects raw data back. The practical effect of this is that UUCP *appears* to be working, but sooner or later you notice a high incidence of dropped transfers and corrupted files (and if a file causes a transfer to drop, it will do so reliably, because some specific in the file is interpreted by simh as having a telnet protocol meaning instead of being passed on to the guest system).

So we set aside a line for UUCP and give it its own telnet port (so that simh knows which line to send the incoming connection to), and leave the rest of the lines in telnet mode to accept human logins.

Now save the file.

You're now ready to boot v7 again.

When you get to the boot prompt, type:

```
boot
```

as directed, but instead of:

```
: hp(0,0)unix
```

type

```
: hp(0,0)anduinunix
```

Replacing "anduinunix" with whatever name you used for the new kernel.

When it boots to the # prompt, hit Ctrl-D to exit single user mode and boot to multi-user. If all goes well, you'll get a login prompt. If there's something wrong with your kernel, this is where it will probably panic (since we've been messing with terminals, and booting to multi-user is where other terminals beyond the console are brought up. If that happens, you can replace boot.ini with boot.ini.bak and use hp(0,0)unix at the boot prompt to boot into a working system.

Now test the DZ11 lines by logging in to the appropriate telnet port. You'll get some garbage characters for the login prompt, but if you hit enter you'll get a clear prompt.

If everything is working at this point, we're ready to start working on UUCP proper.

3. The v7 end of UUCP

Our resource for this section is the Uucp implementation guide, which can be found at http://a.papnet.eu/UNIX/v7/files/doc/36_uucpimp.pdf

Login and change directory to /usr/src/cmd/uucp:

```
     # cd /usr/src/cmd/uucp
```

Section 8 of the installation guide, "Uucp Installation" tells us that we will want to configure several directories, as well as the system name (Unix v7 didn't have TCP/IP, so it didn't have a hostname, as such, so UUCP needed to be told what system name to use, as there wasn't a system-wide name for it to default to).

We'll leave the directory names at their defaults. To edit the system name, open uucp.h with ed (unfortunately, it's compiled in, not in a config file):

```
     # ed uucp.h
```

Search for "MYNAME":

```
     /MYNAME/
```

ed will print:

```
     #define MYNAME          "myname"
```

now we can change "myname" to our chosen system name:

```
     s/myname/anduin/p
```

Please note here that Version 2 UUCP on v7 Unix only accepts 7 character system names. This applies to both this system, and to any system that communicates with it. Fortunately, Taylor UUCP, which we will be using on the Pi, is flexible enough to be able to use a system name different from its normal one when logging in to specific systems, so we aren't restricted to a seven character hostname for the pi itself, just for the name it passes to this system over UUCP. As the PiDP community gains experience with UUCP, we may want to try changing this, or seeing if we can compile HDB UUCP (the version used on, for example, 2.11 BSD) on v7 Unix.

But for now, just save and exit:

```
     w
     q
```

The next step is to build UUCP:

```
     #make
     #make cp
```

make cp will copy the binaries to their final location.

Make the spool directory and execute directories for UUCP, as well as a public directory. According to the implementation guide, these should have global read, write, and execute permissions set. I'm not sure that fits with later best practice, but I'm also not sure that the structure of this early implementation of UUCP allows for anything else. Perhaps that's a reason to try to get HDB UUCP running on v7.

```
# mkdir /usr/spool/uucp
# mkdir /usr/spool/uucp/.XQTDIR
# mkdir /usr/spool/uucp/public
# chown uucp /usr/spool/uucp
# chown uucp /usr/spool/uucp/.XQTDIR
# chown uucp /usr/spool/uucp/public
# chmod a+rwx /usr/spool/uucp
# chmod a+rwx /usr/spool/uucp/.XQTDIR
# chmod a+rwx /usr/spool/uucp/public
```

Now it's time to edit our configuration files, starting with /usr/lib/uucp/USERFILE:

```
# cd /usr/lib/uucp
# ed USERFILE
1,$p
```

This will show that USERFILE contains one line,

```
, /
```

This will allow any machine, logging in as any user, to write to any directory that user would normally have permissions to access. This is not secure, so we'll delete that line.

```
1,$d
```

Now append the following text:

```
a
root, /
, /usr/spool/uucp/public
pi-login,piname /directories /of /your /choice
.
```

Now any machine logging in as root can access any file on the system, any machine logging in as any user can access /usr/spool/uucp/public, and the Rapberry Pi, logging in under a username we will create specifically for it, can access whatever directories we choose to give it access to.

Remember that the system name that the pi passes to this system must be 7 characters or less, but it does not have to be the pi's hostname (though that is the default). We can configure Taylor UUCP to pass any system name we choose here for the pi. Remember the username and system name you use here. That username will have to be added to /etc/passwd on this system, and the system name will be used in a configuration file for Taylor UUCP. If the pi's hostname is less than 7 characters, we can just use that.

Save and exit:

```
w
q
```

Even though we won't be calling out, we need an entry in L.sys for the pi, so that UUCP knows that the pi exists and might call in to request any files we have waiting for it.

```
# ed L.sys
```

delete the last line, which is an example line:

```
$d
```

Now add an entry for the pi:

```
a
piname None ACU 300 555-5555 "" ""
.
```

This tells UUCP that a system called "piname" exists, that we never call out to it, that if we did call out to it we'd use the ACU device at 300 baud, we'd call phone number 555-5555, and that to log in we wouldn't do anything (that is, we wouldn't log in).

Save and exit:

```
w
q
```

There are several other files in this directory used for configuring dialout. We will not touch them for now.

Now we add the username for the pi to /etc/passwd, and make a correction for the existing "uucp" user.

```
ed /etc/passwd
1,$s/\/uucico/\/uucp\/uucico/p
a
pi-login::5:5::/usr/spool/uucp:/usr/lib/uucp/uucico
.
1,$p
w
q
```

Now set the password for the username for the pi:

```
passwd pi-login
```

Remember this password, Taylor UUCP will need it.

And that's it for setting up v7!

4. The pi end of UUCP

Now we just need to set up Taylor UUCP on the Raspberry Pi. We begin with:

```
     $ sudo apt-get install uucp
```

Fortunately, the spool and public directories are created automatically, so we don't have to worry about them. As root, open up /etc/uucp/sys in your text editor of choice. Leave the defaults at the top of the file in place.

Add the following block of text:

```
     system v7name
     myname piname
     time any n
     max-retries n
     call-login pi-login
     call-password pi-passwd
     port v7-dz
     address localhost
     chat "" \d\d\r ogin: \d\L word: \P

     alternate v7name-2
     port v7-dc-pipe
```

The argument to "system" should be the name you gave to the V7 system when setting up UUCP. The argument to "myname" should be the same system name you set up for v7 above, and must be less than 7 characters. Call-login should use the same name you created in /etc/passwd on v7, and call-password should be the same password you set up with passwd in the last step of the v7 instructions. For "time", the "any" keyword means that Taylor UUCP can call v7 at any time, and the number "n" is the number of minutes you want it to wait between attempts to call v7. It can be ommitted if you don't want to have a minimum wait time. For "max retries", n is the number of times UUCP will try calling v7 before giving up until the next day.

The chat script is made up of pairs of strings to expect, and the response to send
to those strings. So first, we expect nothing (that is, we immediately send the
"respones"), wait two seconds with \d\d, and then send a carriage return, \r.
When "login" on v7 sees the carriage return, it will reprint the login prompt.
We then wait for the string "ogin:" (in case we miss the first letter of "login:"),
wait one second, and send our login name with \L. We then wait for "word:", and send
our password with \P.

The alternate name is just a name to identify an alternate means of calling the
v7 system.

Now edit /etc/uucp/port, and add the following two entries. v7-dz is the method we use to call the v7 system on the first attempt, which is just to have uucico itself try to contact the v7 system over port 4107, v7-dc-pipe is the alternate method we try if that doesn't work, which is to invoke telnet and use that to try to connect.

```
port v7-dz
type tcp
service 4107

port v7-dc-pipe
type pipe
command /usr/bin/telnet -8 4107
```

Now, find a convenient file to try to send to the v7 system. Send it to the public directory on the v7 system:

```
$ uucp filename anduin\!~/public
```

"Anduin" of course, should be replaced with the name you used for the v7 system. Note that the ! has to be escaped with a backslash, since ! has a meaning on many modern shells.

This queues up the file for sending. Back in the day, our system would generally make scheduled callouts to the other system (or receive scheduled callins from the other system). The uucp program would queue work to be sent during the scheduled call. Since we're just transferring between two systems that we completely control (for now), setting up a scheduled call would be overkill, and also would force us to wait for the call. Instead, we'll just tell UUCP to call immediately.

```
$ sudo uucico -s anduin
```

If all goes well, once uucico finishes executing, we should be able to find our file in
/usr/spool/uucp/public on the v7 system. On the v7 system, type:

```
# ls /usr/spool/uucp/public
```

which should display:

```
filename
```

Now send it back. On the v7 system, type:

```
# uucp /usr/spool/uucp/public/foo.c piname!~/
```

Note that since the v7 shell doesn't interpret ! in any special way, we don't need to
escape it here.

And then on the pi, type:

```
$ sudo uucico -s anduin
```

Once uucico is done, we should be able to list /var/spool/uucppublic, and see our file.

```
$ ls /var/spool/uucp/public
```

That's it! We've set up UUCP.