The Ubuntourist Trap

Release 0.1

Kevin Cole ("The Accidental Ubuntourist")

Nov 10, 2023

CONTENTS:

CHAPTER

ONE

LINUX TIPS

1.1 Linux Tips

A collection dating back to 2001.08.23.

> The following tips are ones that I've become tired of looking up in the `man` and `info` pages, (or searching the Internet for) then whittling down to their barest essences.
>
> **NOTE**: «`text within French quotes`» indicates variable text – often a filename.

1.1.1 Redirecting Standard Error to Standard Out

The proper way to redirect `stderr` is to first decide where `stdout` is going, and **THEN** redirect the `stderr` to `stdout`. So, for example:

```
$ «yada-yada» | «pager» 2>&1
```

takes the output of «`yada-yada`» and pipes it into «`pager`», then tells the system to send `stderr` to wherever `stdout` is going. (A pager is a program that allows a user to scroll through long documents. I use one called `most`, but older, commonly used ones include `more` and `less`.)

1.1.2 Making a patch file

Assume we have a Red Hat source RPM:

```
$ rpm -Uvh «package».src.rpm
$ cd /usr/src/redhat/SPECS
$ rpm -bp «package».spec
$ cd /usr/src/redhat/BUILD
$ mv «package» «package».orig
$ cd /usr/src/redhat/SPECS
$ rpm -bp «package».spec
$ cd /usr/src/redhat/BUILD
  (edit to your heart's content)
$ diff -Naur «package».orig «package» > ../SOURCES/«package».patch
$ rm -rf «package».orig
```

(continues on next page)

(continued from previous page)

```
$ cd /usr/src/redhat/SPECS
$ emacs «package».spec
  ...
  Source: ...
  Patch: «package».patch
  ...
  %prep
  %setup
  %patch -p 1
  ^X^S^X^C
$ rpm -ba «package».spec
```

The idea is to create two directories with identical contents, then modify one of them. Create a diff file of the changes and save it. I **THINK** I got all the basic steps in there… However, it may be necessary to create the tarball too, in which case you need something like:

```
$ tar czvf «archive».tar.gz «directory_to_archive»
```

1.1.3 Copying directory trees

Often, it becomes necessary to copy entire directory trees from one directory to another. The method I saw somewhere was:

```
$ cd «olddir» ; tar cf - . | (cd «newdir» ; tar xpf -)
```

This creates a tarball that never actually becomes a file. The tarball is piped directly to a little script subroutine which changes to the new directory, and untars the tarball on the fly.

According to JP Abgrall, there's an optimized way to do this:

```
$ tar cf - -C «olddir» . | tar xpf - -C «newdir»
```

1.1.4 Getting landscape output

It looks like `mpage` will do the trick:

```
$ mpage -1lvH «filename» | lpr
```

The man page suggests that there's a way to pass `pr` switches to `mpage` (using `-p` instead of `-H`), but I've been unable to pass the line-numbering switch `-n` together with `pr` into `mpage`. (What I want is the headings a la `-H` together with line numbering.)

A fancier way, requiring a bit more study, is to use `enscript`. In fact, `enscript` is neat for LOTS of stuff – customizable layout, "Page X of Y", line numbers, etc.

1.1.5 Verifying Red Hat packages

You can verify each package with the following command:

```
$ rpm --checksig
```

If you only wish to verify that each package has not been corrupted or tampered with, examine only the md5sum with the following command:

```
$ rpm --checksig --nopgp
```

1.1.6 Security: Watching the watchers...

The `netstat` command is a handy tool for seeing who's poking around at any given moment:

```
$ netstat -v | most
```

1.1.7 Stripping comments:

Assuming you have a script that uses the number sign (a.k.a. pound symbol, hash mark) as a comment character, and you wish to examine only those lines containing "active" commands and options, the following will produce such a listing:

```
$ grep -v "^\#" «scriptfile» | grep -v "^[[:space:]]*$"
```

What's happening: The file is first stripped of lines beginning with #. Then, that result is stripped of any lines which have 0 or more whitespace characters (and nothing else) between the start and end of the line.

1.1.8 Verifying all RPM's

Here's a small script that constructs a list of all package names sans version numbers, then feeds the list to `rpm` with the `--verify` option. It also echoes the package name:

```
$ for i in $(rpm -qa --queryformat "%{NAME}\n" | sort)
$ do
$   echo $i":" >>verify.log 2>&1
$   rpm --verify $i >>verify.log 2>&1
$ done
```

The stuff enclosed in `$(...)` gets run as a script within a script, and the output of that is fed to the outer script. (See `man eval` and other stuff about evaluating.)

1.1.9 Viewing post-installation RPM scripts

Occasionally, after the files are dropped onto the system, hither and yon, RPM will run a script embedded in the package file. It's nice to see what the squirrels are doing under the hood:

```
$ rpm -q --scripts «package-name»
```

1.1.10 What are we listening to?

`lsof` shows which processes are listening on a given port. (Actually it stands for "list open files", which shows what files are currently open.) `lsof -i` will list which ports are open on the machine:

```
$ lsof -i
```

1.1.11 Finding duplicate files with identical contents

There's probably a better way, but this worked for me:

```
$ diff -qrs «directory1» «directory2» 2>&1 | \
  grep "identical$" > «unedited-shell-script.sh»
```

Then edit the file «`unedited-shell-script.sh`» to your heart's content.

1.1.12 Listing DNS stuff

Lots of different ways to do this, but I like:

```
$ nslookup
> ls -d gallaudet.edu
```

1.1.13 Who's been sleeping in MY bed?

Here's a more informative way to use the `last` command:

```
$ last -adf «wtmp_file»
```

1.1.14 Clip-clip. Taking care of really LONG lines

Lots of times, we only need to see the beginning of lines in a file to determine something. (For example, a subroutine that takes a single string argument may have a really long string literal.) To see just the first 100 characters on a line use the `cut` command. Like so:

```
$ cut -b -100 «FY2000.sql» | land
```

(`land` is an `alias` I've set up to print a file in landscape orientation using `enscript` command – a very nice printing program.)

1.1.15 Find and delete

Sometimes it's nice to do something (like delete) a bunch of files based on a searchable criteria, e.g. portion of the filename, size, date, etc. Here's how:

```
$ find . -empty -exec rm -v {} \;
```

This is a specific example that searches for empty files and directories from the current working directory down, and then it deletes them. The important parts are the `{}` which gets replaced with whatever has been found, and the `\;`, an escaped semicolon indicating the end of the command to be "exec'ed". (And the `-v` is the ubiquitous "verbose" option, to tell you what's happening.)

A better approach, I'm told, is:

```
$ find . -empty | xargs rm -v
```

1.1.16 Formatting and using a floppy

Without mounting anything, just pop a floppy in the drive and:

```
$ fdformat /dev/fd0H1440
$ mke2fs /dev/fd0H1440
$ mount /dev/fd0 /mnt/floppy ...
$ umount /mnt/floppy
```

1.1.17 Mounting an NSF device

To mount an NSF disk:

```
$ mount «server»:«remote_directory» /«local_mount_point» -t nfs -o ro
```

1.1.18 Searching for files and manipulating them

To find files in or beneath the current directory, of type "file", of size 800 KB or greater, and then pipe the results through the ls command:

```
$ find . -type f -size +800k -exec ls -l {} \;
```

As mentioned in an earlier tip, it's better with xargs. The command above could be improved as:

```
$ find . -type f -size +800k | xargs ls -l
```

To find all files that match the pattern `*.o`, print the full filespec (`%p`), the last-access time (`%a`) and the last-modified time (`%t`), and prompt for deletion:

```
$ find . -name "*.o" -printf "%p\nA: %a\nM: %t\n" -exec rm -i {} \;
```

To find files whose data has changed since midnight:

```
$ find . -daystart -mtime 0
```

(The `-mtime` can be replaced with the `-ctime` to show files whose status has changed.)

1.1.19 Starting a remote X windows program on a local screen

I must have had to do this at some point:

```
$ xon «remote_host»           \
      -access                 \
      -user «remote_username» \
      «remote_program_path»
```

1.1.20 Making a boot floppy

What the heck is a "floppy"? Well, if you have one:

```
$ dd bs=8192 if=/vmlinuz of=/dev/fd0
```

That allowed me to recover from a machine that someone infected with a boot sector virus. I still had to rebuild the kernel.

1.1.21 Switching parallel from printer to ZIP

Speaking of dead hardware… ZIP drives that connect to the parallel port:

```
$ modprobe -r lp
$ modprobe ppa
$ mount /dev/sdc4 /mnt/ZIP
```

1.1.22 More fun with RPM's

That `--queryformat` be some powerful voodoo. Convert the datetime tags to human readable form via the system command:

```
$ convdate -c `rpm -q --queryformat "%{INSTALLTIME}\n" «package name» `
```

1.1.23 Burning CD's

Unfortunately, with no CD-burner on any of the Linux boxes, you have to resort to M$ to do the actual burn. But just copying the files and trying to burn things didn't seem to work. So, on a Linux box, create a disk image, then move the image to a M$ machine. Like so:

```
$ mkisofs -vrTJV "«Volume Label»" -o «image filename».iso «root directory»/
$ mount -t iso9660 -o ro,loop=/dev/loop0 «image filename».iso /mnt/cdrom
```

The first line makes an ISO file system and write it to a file. The `.iso` just makes it easier for the Windoze software to recognize it as a CD image. The command line options used are:

- `-v` verbose,
- `-r` Rockridge extensions,
- `-T` make `TRANS.TBL` files,
- `-J` Joliett extensions,
- `-V` Volume label,
- `-o` output file.

The second line tests the image, by mounting it as though it were a real device.

Fancying the image creation up a bit, the following puts an abstract on the CD and hides the `TRANS.TBL` from systems that can handle long file names:

```
$ mkisofs -vrTJV "«Volume Label»"         \
          -abstract "«Short description»" \
          -hide-joliet-trans-tbl          \
          -o «image filename».iso «root directory»/
```

And a variation with some Macintosh options thrown in:

```
$ mkisofs -vrJV "«Volume Label»" \
          -hfs                   \
          -magic «magic file»    \
          -probe                 \
          -o «image filename».iso «root directory»/
```

The **magic file** helps `mkisofs` determine which `CREATOR` and `TYPE` to use so that a Macintosh knows how to open the files. It appears the magic file is only needed if the system cannot already determine what the file is by examining the first few bytes. (I used `/dev/null` for the magic file.)

If you **DO** have a burner on your box, you can add the command:

```
$ cdrecord -v -speed=«##» dev=«#,#,#» -data «image filename»
```

In my case the `-speed` is `10` and the `dev` is `2,1,0`. This burns the image file created by the `mkisofs` command onto your CD. If you don't know the `dev`, you can find it with one of the following two lines:

```
$ cdrecord -scanbus
$ cdrecord dev=ATA -scanbus
```

depending on your kernel version and your CD burner controller. The second version picks up an ATA CD-burner under kernel 2.6.

1.1.24 Turning off NetQUE broadcasts

NetQue boxes attached to dumb printers keep sending `RWHO` packets (`UDP/513`) all over campus. This is annoying. To turn it off:

1. `telnet` into the NetQue
2. Type `SU` at the prompt.
3. It will display a `Password>` prompt. Type a `Control-H` (ASCII backspace) and then `SYSTEM` and hit enter. (`SYSTEM` is the default password.)
4. If you get to the prompt, type `DEFINE SERVER ANNOUNCEMENT DISABLE`
5. Type `SYNC`
6. Type `LO`
7. Power-cycle the printer server for it to take affect.

1.1.25 Checking out from CVS

I don't yet understand what I've done, but apparently, I got it right. The following pulled the latest version of Boa Constructor:

```
$ cvs -z3 \
      -d:pserver:anonymous@cvs.Boa-Constructor.sourceforge.net:/cvsroot/boa-constuctor \
      co boa
```

1.1.26 Streaming with Icecast and Darwin

`Icecast` streams `MP3` and `Ogg Vorbis`, Darwin is Apple's `QuickTime` streamer. Again, I'm not certain of all the details, but I've got them both going.:

```
$ icecast -b
$ liveice -F ~/liveice.test 2> /dev/null
$ /usr/local/sbin/streamingadminserver.pl
```

The first line starts `Icecast` listening. The second sends a stream to `Icecast` for rebroadcast. The third starts the Darwin server. Be sure to check the configuration files in `/etc/icecast` and `~/liveice.test`.

1.1.27 Pretty-printing code as web pages

My favorite program lister `enscript`, can generate color-coded web pages, as well as color-coded Postscript. Separate colors are used for comments, keyowrds, functions, and quoted strings. To generate a page, complete with a table of contents, the magic is:

```
$ enscript -E -C -G -j -Whtml --color --toc -p«output».html \
          «program1» [«program2» «program3» ...]
```

1.1.28 md5sum

MD5 checksums are frequently distributed with files to be downloaded, in an effort to insure data integrity. The program `md5sum` for Linux is fairly easy to find, and may already be on your system.

To check a file, download the corresponding `MD5SUM` file (possibly named `«filename».md5`) to the same directory where the files to be checked live. Then issue the command:

```
$ md5sum -c «filename».md5
```

To create an MD5 checksum file for others to use against your files, issue the command:

```
$ md5sum «filenames» > «filename».md5
```

I donwnloaded a Microsoft DOS/Windows version of `md5sum` from https://etree.org/md5com.html. The web page suggests the directory in which to save the program. It needs to be run from the DOS command prompt.

1.1.29 Slow Hand

The problem: During a rescue operation, I needed to copy a HUGE file. Unfortunately, while booted up in Red Hat's rescue mode, memory management seems to have some problems. Every attempt to copy this would go for a while then run out of memory and force a reboot.

Solution: I hypothesized that if I could slow the machine down, I might give it time to recover its memory. (I know I forget things when I think too fast.)

So, how?

1. Break the file into chunks and copy a chunk at a time, with delays between chunks.
2. The file (a bzipped tarball) was 689459745 bytes long.
3. dd (a file copying program) writes nulls when it hasn't got any data. So, I couldn't write more data than was actually in the original file. Otherwise I'd end up with nulls at the end.
4. That means, the blocksize times the number of blocks had to exactly match the file size.
5. I found a web page with a factoring calculator, and learned that the prime factors of 689459745 are 3, 5, 13, 19, 379 and 491.
6. Armed with that info, I opted for 379 blocks of 1819155 bytes each.
7. Finally, I wrote a little script

```
#!/bin/sh
# SLOW DOWN! Copy slooooowly, and provide a running
# progress report comparing the file sizes of the
# two files in question. When done, compute the MD5
# checksum for each file, for comparison.
#
# NOTE: A 1-second delay wasn't enough. A 10-second
# delay was, but it was also probably overkill.

cd /mnt/sysimage/usr/share
rm /mnt/jaz/home.tbz
touch /mnt/jaz/home.tbz
ls -al home.tar.bz2
ls -al /mnt/jaz/home.tbz
sleep 1

for ((blk = 0; blk < 379; blk++))
do
  dd if=home.tar.bz2 of=/mnt/jaz/home.tbz \
     bs=1819155 count=1 \
     seek=$blk  skip=$blk
  ls -al home.tar.bz2
  ls -al /mnt/jaz/home.tbz
  sleep 10
done
md5sum home.tar.bz2 /mnt/jaz/home.tbz
```

UPDATE: Apparently, I misread or misunderstood the dd command. It doesn't pad its output with NULL's unless you explicitly ask it to by using the conv=sync option. So any reasonable block size should have worked above…

Bob Solomon <wogsol (at) bestweb (dot) net> wrote to me about a different way to split up a file: Given a list of files that you want to tar, but make into several "chunks":

```
$ tar -czf - «file1 file2 file3 ...» | split -b «###»m - «filename».tgz.
```

(where ### is a block size in megabytes.) This creates files filename.tgz.aa, filename.tgz.ab, filename.tgz.ac… and so on, with each file being ### MB long.

(Bob also uses an environment variable $DATE which he sets to the current date in the form yy-mm-dd, in the base filename of the split command.)

1.1.30 Synchronizing with rsync

To copy big directory trees across the entire universe, and maintain protections, user and group ID's etc, use `rsync`. It appears to have a few kinks–like it's slow as molassas on my machine, I think I caused a kernel panic the first time I used it, and now that it's finished a copy it appears to be hanging, but it gets the job done.

Important tip: It seems to do better at pushing files out to the remote machine, rather than pulling from the remote:

```
$ rsync -avz --rsh=ssh «local_directory_tree» \
                       «remote_machine»:«remote_destination»
```

1.1.31 Exploring binary RPM's without installing them

Sometimes it's nice to see what's in an RPM file without actually installing it. If you have the source RPM (`.src`) then it's easy. Just make the binary. But if you don't have it and don't want to bother getting it, you can extract the `CPIO` "heart" of the RPM and explore that. (`cpio` = copy in and out.):

```
$ rpm2cpio «package».rpm > «package».cpio
$ cpio -it --verbose < «package».cpio | most
$ cpio -id --verbose < «package».cpio
```

The first line pulls out the CPIO from the RPM file. The second gives a verbose listing of the contents of the file, and the third actually does the extraction, forcing the creation of directories that aren't already present.

1.1.32 Renaming all files in a directory

Sometimes you want to rename all the files in a directory, and the new names will in some way be based on the old names. Here's one way to tackle the problem (not necessarily the best way):

```
$ ls | grep -v "[on]names" > onames
$ ls | grep -v "[on]names" > nnames
# (edit nnames and change the old filenames to their new filenames.)
# (edit onames and insert "mv " at the start of each line.)
$ paste onames nnames > script.sh
$ bash script.sh
$ rm onames nnames script.sh
```

The first two lines make identical files containing all the filenames in the directory, sans the two files being created. The `paste` command in step 5 puts them together, line by line, so you end up with several lines of `mv old-name new-name`, which you just push through your shell.

If all you want to do is lowercase the names, this will do the trick:

```
$ for i in *; do mv $i $(echo $i | tr [A-Z] [a-z]); done
```

1.1.33 "rpm -qil" in Debian

To query a Debian package and obtain both a package description and a list of files within the package, use the command:

```
$ (dpkg -p «package» ; dpkg -L «package» ) | «pager»
```

To just get a list of ALL packages (installed and uninstalled, use:

```
$ dpkg -l '*'
```

1.1.34 Handling NULL's in PostgreSQL

The `COALESCE` function is your friend. It allows you to substitute a string for a `NULL` value. The example below shows how to combine several fields together into a single string:

```
SELECT  COALESCE(name,'')    || '-' ||
        COALESCE(version,'') || '-' ||
        COALESCE(release,'') || '\n\t' ||
        COALESCE(summary,'') || '\n\t' ||
        COALESCE(url,'')
FROM     gri
WHERE    name ~* '.*devel.*' and release ~* '.*ximian.*'
ORDER BY name;
```

When run on my database of installed RPMs, that produces output like this:

```
...
sane-backends-devel-1.0.8-1.ximian.1
       The SANE (a universal scanner interface) development toolkit.
       https://www.sane-project.org/
SDL-devel-1.2.4-1.ximian.3
       Files needed to develop Simple DirectMedia Layer applications.
       https://www.libsdl.org/
xmms-devel-1.2.7-1.ximian.4
       XMMS - Static libraries and header files.
       https://www.xmms.org/
(28 rows)
```

1.1.35 Using BitTorrent with RedHat

According to https://www.redhat.com/en, RPMS for Red Hat Linux 7.3 through 9 of BitTorrent are available from:

> https://torrent.dulug.duke.edu/btrpms/

Usage is simple:

```
$ btdownloadcurses.py --url «https://URL.torrent»
```

Allow incoming TCP 6881 - 6889 to join the torrent swarm.

1.1.36 Decoding base-64 encoded text

If you end up with a file that is base-64 encoded, fetch a copy of `uudecode` and add a line to the top of the base-64 encoded file (if it isn't there already) that looks like:

```
begin-base64 664 «ASCII-file»
```

Then issue the command:

```
$ uudecode -m «b64-file»
```

This should read in `b64-file` and create `ASCII-file`.

1.1.37 Optimizing SQL field lengths

Obvious, when one thinks about it, but... To obtain the lengths of all entries for a particular field, use:

```
SELECT LENGTH(«field_name») AS «column_name»
FROM «table_name»
GROUP BY «column_name»;
```

To list only the length of the longest entry in a field, use:

```
SELECT MAX(LENGTH(«field_name»)) AS «column_name»
FROM «table_name»;
```

1.1.38 A light at the end of the tunnel

I don't yet consider myself an expert by any means, but I'm making progress understanding tunneling. Here's an example:

```
$ ssh -L 7000:134.231.8.45:80 -l kevin.cole gallaudet.edu
http://localhost:7000/
```

The first line establishes a tunnel on the `localhost`, going out port `7000` to `134.231.8.45`, port `80`, via ssh logged in as `kjcole` on `gri.gallaudet.edu`.

The second line is the URL to make use of the above tunnel, effectively connecting to https://134.231.8.45/.

1.1.39 Paper size

To switch to 8.5 * 11 paper size:

```
$ cd /usr/share/libgnomeprint/.../printers/
```

Edit the files `GENERIC.xml` and `PDF-WRITER.xml`. Change the `PhysicalSize` from `A4` to `USLetter` (no spaces).

1.1.40 GNU Privacy Guard (GPG) tip

Apparently, the GPG `honor-http-proxy` `keyserver-option` is buggy. (Either that, or `privoxy` is.) So, in order to use commands like:

```
$ gpg -v --keyserver x-hkp://pgp.mit.edu --refresh-keys
```

or:

```
$ gpg -v --keyserver x-hkp://pgp.mit.edu --send-keys
```

remember to issue the shell command `unset http_proxy` first.

1.1.41 Searching for strings using grep the right way

I was constantly annoyed by `grep` hanging indefinitely when searching recursively. One possible reason for the trouble was that `grep` would encounter a "file" which was in fact a pipe or other weird beastie that doesn't really have a beginning or end. As a result, `grep` would search such a file indefinitely. So, instead, use find to guarantee that `grep` only searches actual normal files:

```
$ find «path» -type f | xargs \
  grep -H "«And I still haven't found what I'm looking for»"
```

And, an often related nusance: When find encounters a filename with spaces, what it pipes to `grep` ends up as several arguments. When it finds a file with a filename like "`Scholarly Work.txt`", `grep` interprets as a file named Scholarly and a second file named Work.txt. Not at all what I had intended. So, in the simple case, a solution is:

```
$ find «path» -type f -exec \
  grep -Hil "«And I still haven't found what I'm looking for»" {} \;
```

`grep` receives each filename supplied by find whole and intact.

1.1.42 Handling files with spaces in the name

Those nasty Mac OS X people, and later those nasty Windows folks have made life messy for us saints of Free / Libre Open Source Software. But, there is hope:

```
#!/bin/bash
#
# This script iterates through several file types and makes substitutions
# within them. It's done this way to get around funky directory and file
# names containing spaces. The IFS indicates an inter-file separator,
# in this case NUL. The "set -f" turns off pathname expansion, allowing
# the "htm*" and "php*" to be passesd as-is to the find command.  And
# finally, the ${1:-.} says to use a dot (current directory) if no
# directory path is supplied on the command line.

IFS="$(echo -ne '\000')"
set -f
```

(continues on next page)

(continued from previous page)

```
for filetype in "htm*" "php*" "py" "cgi" "c" "pl" "txt" \
                "js" "asp" "shtm*" "pm" "java" "lore" "kid"
do
  find ${1:-.}/ -iname "*.$filetype" -print0 | while read -d "$IFS" file
  do
    echo "\"$file\""
    perl -p -i -e "s|\<i\>|\<em\>|g;"       "$file"
    perl -p -i -e "s|\<I\>|\<em\>|g;"       "$file"
    perl -p -i -e "s|\</i\>|\</em\>|g;"     "$file"
    perl -p -i -e "s|\</I\>|\</em\>|g;"     "$file"
    perl -p -i -e "s|\<b\>|\<strong\>|g;"   "$file"
    perl -p -i -e "s|\<B\>|\<strong\>|g;"   "$file"
    perl -p -i -e "s|\</b\>|\</strong\>|g;" "$file"
    perl -p -i -e "s|\</B\>|\</strong\>|g;" "$file"
  done
done
```

1.1.43 Fancier apache protection

First, enable some apache modules: `auth_digest`, `dav`, and `ssl`. The `auth_digest` module enables the use of better-encrypted usernames and passwords. The `dav` module enables `WebDAV`, which allows those with the appropriate permissions to look at the files in a directory using a file browser / manager, with drag-n-drop, and the ability to add and delete files to the directory. And, finally, the `ssl` module gets down and dirty with data transfer encryption. SSL's a bitch, and therefore not covered here:

```
$ a2enmod dav
$ a2enmod auth_digest
$ a2enmod ssl
```

Now, edit the file containing directives for your web directories (in the case of Ubuntu, one of the files in `/etc/apache2/sites-available/` e.g. `default`). Add in a stanza for the URL you want to protect:

```
<Location «relative URL»>
    Order Allow,Deny
    Allow from all
    Dav On
    AuthType Digest
    AuthName "«realm»"
    AuthDigestDomain «relative URL»
    AuthDigestProvider file
    AuthUserFile «/path/to/password.file»
    Require valid-user
</Location>
```

Finally, create the password file:

```
$ cd «/path/to/password.file»
$ htdigest -c «password.file» "«realm»" «username»
```

The realm is a short description of the area to be protected. The realm in the `htdigest` command should match the realm specified with the `AuthName` directive in the apache configuration file. Ditto for the path to the password file. The `htdigest` command will create (`-c`) the password file password.file add username to it, prompting for a new password.

It appears that it's also a good idea to match up the argument in the `<Location>` directive with that in the `AuthDigestDomain` directive. This should be "relative" to the `DocumentRoot`. In other words, it's what appears after the https://host.domain.tld/

Addendum: Oh the perversity that is Micro$oft Winblows. Every variation of URL, username, etc. failed to create a mapped network drive. What finally worked… sort of? From inside Micro$oft Weird, opening a Network Place. But, not exactly. You see, it cannot create a new file directly. It needs to create a folder. So, it creates a new FOLDER inside the already shared WebDAV folder. That means all the files we expected to find were one level above the Network Place we created and had to be moved into the newly created directory.

1.1.44 Restoring files with cp

To preserve dates, permissions, links, etc. when copying files:

```
$ cp -rvP --preserve=all «source directory» «destination»
```

1.1.45 QR Codes as SVG

`qrencode` creates QR Code images as PNG files. If that's all you need, then the first line below will suffice. However, if you want to convert the PNG to an SVG, go through an intermediate step that makes a BMP:

```
$ qrencode -s 20 -o «filename».png "«blablabla.bla»"
$ convert «filename».png «filename».bmp
$ potrace -b svg -o «filename».svg «filename».bmp
$ rm «filename».bmp «filename».png
```

`qrencode` produces the PNG (Portable Network Graphic), `convert` converts it to a BMP (bitmap), and `potrace` converts the BMP to an SVG. (The `rm` removes the first two files.) Use the same filename extensions as shown above. "blablabla.bla" is just any text you want to encode. You can also pipe a file to the encoder. `-s 20` sets the dot size to 20 pixels. The default size is 3 pixels. `-o` indicates the name of the output file. `-b svg` specifies the "backend" which tells `potrace` the output format. (`qrencode` is part of the `qrencode` package, `potrace` is part of the `potrace` package, and `convert` is part of the `imagemagick` package.)

1.1.46 Inserting lines at the start of a file

`sed`, the stream editor, is our friend here. To insert a single line:

```
sed -i '1s/^/«line to be inserted»\n/' «filename»
```

But a more clever approach that handles more than one line, methinks:

```
printf '%s\n%s\n' "«text to be inserted»" "$(cat «filename»)" > «filename»
```

1.1.47 Floating image in reStructuredText

To have an image with text appearing beside it in reStructuredText:

In the `.rst` file:

```
.. container:: twocol

   .. container:: leftside

      .. figure:: /_static/illustrations/structure.svg

   .. container:: rightside

      Bla-bla-blah, and yada-yada.
```

In the custom CSS (I used a copy of `sphinxdoc.css` which I put in `./source//_static/`):

```
div.leftside {
    width: 414px;
    padding: 0px 3px 0px 0px;
    float: left;
}

div.rightside {
    margin-left: 425px;
}
```

Each `..container::` becomes a . In my case, I wanted a fixed width for the image and a variable width for the remainder. And, with a wee bit o' tweaking of the LaTeX produced by Sphinx, it also did a decent job of producing two-column output for that section.

1.1.48 Learning Assembler !!!

(We'll see how long this exercise in terror / futility lasts.)

Write some C code in a file "`scratchpad.c`" (or whatever name you'd like). Then:

```
$ gcc -S -fverbose-asm scratchpad.c
```

or:

```
$ gcc -g -c scratchpad.c
$ objdump -drwC -Mintel scratchpad.o
```

(The later is actually a dis-assembly from the object binary. The first form is MUCH more useful to me: It interleaves the original C code with the assembly language it generates, making it easy to see "Oh. That 'if' statement compiles into these four assembly language instructions.)

I have a feeling there's much room for improvement on the process (e.g. adding `-O3` to optimize the hell out of the compilation, adding `-g` to the first form to include more debugging info) but this is quite nice. There are more command-line options for the GCC compiler than I've seen for any other Unix-y / Linux-y thing ever. So, fine-tuning the output for learnability is for a future date.

Source: StackOverflow, naturally: Using GCC to produce readable assembly?

Continuing on the adventure, the **E**xecutable and **L**inkable **F**ormat, better known as ELF:

1.1.49 Rectangular blocks in emacs

I'm sure there was a time when I did this regularly, but it's been ages! In any case, the magic:

1. Move the cursor to the starting location.
2. Mark the current location with `Ctrl-x SPACE`.
3. Move the cursor to the end location.
4. Kill the region `Ctrl-x r k`
5. Move to the new location
6. Yank the region `Ctrl-x r k`

1.1.50 git "mirroring"

With a very small tweak, you can make your local git repositories push out to as many hosting sites as you like. Once per each hosting site that you would like to replicate to (though you have to have created a repository there first, even it's empty):

```
git remote set-url --add --push origin «clone URL»
```

After that, any time you push, it will go to all of the sites that you've used that line for. So now, many of my repositories are pushed out to Codeberg, GitBox, and GitLab with every "`git push`" command. ("`git pull`" will still pull from the one you initially cloned from, or did your initial setup with.)

NOTE: The remote repository has to exist before adding it to the list of push destinations.

1.1.51 Copying between two remote machines

It turns out that it is as simple as 1, 2, `-3`:

```
$ scp -3 «remote1:/path/to/sources» «remote2:/path/to/destination»
```

1.1.52 Sledgehammer "git pull"

Probably not the smartest thing I've ever done, but at least not as disasterous as doing `pip update` on everything in sight, thus trashing packages installed by `apt`, `yum` or `pacman`...

This little ditty goes through as many git repositories as it can find, and tries to issue a `git pull` for each of them:

```
$ for i in $(locate /.git/ | sed -e "s/\/\.git.*//;" | sort | uniq)
$ do
$   cd $i
$   git pull
$ done
```

(It presumes the `locate` package is installed and that `updatedb` has been run recently.)

In some cases, this leads to conflicts, merge problems, etc.

1.1.53 Sucking down a web site directory tree

(There's probably a more clever way with `curl` these days...)

When a URL reveals a directory of stuff you want to get, while avoiding the stuff you don't want:

```
$ wget -r -np -R "index.html*" -e robots=off «URL»
```

Option	Meaning
`-r`	recursive
`-np`	no parent (don't go to the root of the URL)
`-R "index.html*"`	don't include the index.html files
`-e robots=off`	ignore what robots.txt is telling you to do

1.1.54 Stop annoying animated GIFs

In **Firefox** go to the url `about:config` and change `image.animation_mode` from `normal` to `none`.

1.1.55 Platform-provided Python packages

It turns out, that while in the virtual environment, I don't have access to some of the Python packages installed via *apt*. In particular, the PyQt5 stuff. But, there's an answer:

```
$ cd dirname
$ rm -rf ~/.local/share/virtualenvs/dirname*
$ rm Pipfile.lock
$ pipenv --three --site-packages
$ pipenv shell
$ pipenv update
```

Passing `--site-packages` during the initial `pipenv` setup adds magic to `~/.local/share/virtualenvs/dirname...` or so it would appear. As near as I can determine, it adds an `include-system-site-packages` line to `~/.local/share/virtualenvs/dirname-.../pyvenv.cfg`. Like so:

```
home = /usr
implementation = CPython
version_info = 3.8.5.final.0
virtualenv = 20.0.23
include-system-site-packages = true
base-prefix = /usr
base-exec-prefix = /usr
base-executable = /usr/bin/python3.8
prompt = (dirname)
```

And that's where it gets the command line prompt as well.

1.1.56 Pretty-print XML

Sometimes one wants to read that billion-character single-line XML file in order to make sense of it. First, set the indentation.

For tab indentation:

```
export XMLLINT_INDENT=`echo -e '\t'`
```

For four space indentation:

```
export XMLLINT_INDENT=\ \ \ \
```

Then:

```
xmllint -format -recover nonformatted.xml > formated.xml
```

1.1.57 Reformatting XML a la Emacs

Actually, I suspect this works for a **lot** of different source material, provided Emacs recognizes the file type. In emacs parlance:

```
C-x h C-M-\
```

In more readable form:

```
Ctrl-X h    Ctrl-Alt-\
```

Explanation:

```
Ctrl-X h      runs the command    "mark-whole-buffer"
Ctrl-Alt-\    runs the command    "indent-region"
```

1.1.58 Adding an upstream repository to a forked repository

So, after forking a repository, it would be nice to be able to keep it synchronized. First, set up an upstream branch:

```
$ git clone git@github.com:kjcole/obs-midi.git
Cloning into 'obs-midi'...
remote: Enumerating objects: 106, done.
remote: Counting objects: 100% (106/106), done.
remote: Compressing objects: 100% (69/69), done.
remote: Total 5924 (delta 64), reused 69 (delta 37), pack-reused 5818
Receiving objects: 100% (5924/5924), 2.21 MiB | 6.04 MiB/s, done.
Resolving deltas: 100% (3778/3778), done.
$ cd obs-midi/

$ git remote -v
origin  git@github.com:kjcole/obs-midi.git (fetch)
origin  git@github.com:kjcole/obs-midi.git (push)

$ git remote add upstream git@github.com:cpyarger/obs-midi.git

$ git remote -v
origin    git@github.com:kjcole/obs-midi.git (fetch)
origin    git@github.com:kjcole/obs-midi.git (push)
upstream  git@github.com:cpyarger/obs-midi.git (fetch)
upstream  git@github.com:cpyarger/obs-midi.git (push)
```

Then at a later date, periodically lather, rinse, repeat:

```
$ git fetch upstream
$ git checkout main
$ git merge upstream/main
```

1.1.59 Searching for Unicode characters

My `unprintables` and `expletives` aliases (shown below) come in handy for finding everything that is not pure, printable, easy to type ASCII. But sometimes, just sometimes, I need to search for a specific Unicode character. For example, a long dash. `most` (or `hexdump` if you prefer) can reveal that at least one variant of a long dash is hexadecimal **E2B8BA**. To search for it with `grep` use:

```
$ grep $'\xE2\xB8\xBA' *
```

The aforementioned and very handy aliases are:

```
$ alias unprintable='grep --color="auto" -P -n "[\x00-\x1E]"'
$ alias expletives='grep  --color="auto" -P -n "[^\x00-\x7E]" '
$ alias decomment='egrep -v "^[[:space:]]*((#|;|//).*)?$" '
```

`unprintables` searches for any lines containing control characters (ASCII characters in the range 0 to 31). `expletives` searches for any characters beyond ASCII, i.e. **NOT** in the range 0 to 127. (And, because it is probably my most used alias `decomment` shows the contents of files sans any comment lines using `#`, `;` or `//` as the comment delimiter.)

1.1.60 Running multiple commands in Bash

There are three different ways to combine commands in the terminal:

```
;     Command 1 ;  Command 2    Run command 1 first and then command 2
&&    Command 1 && Command 2    Run command 2 only if command 1 ends sucessfully
||    Command 1 || Command 2    Run command 2 only if command 1 fails
```

1.1.61 Bash: for i in range()

Two different techniques:

- Method 1:

```
for i in {0..10..1}
do
  printf "%02d\n" $i   # Print as a 2-digit number with leading zeros
done
```

- Method 2:

```
export END=10
for i in $(seq 0 $END)
do
  printf "%02d\n" $i   # Print as a 2-digit number with leading zeros
done
```

1.1.62 Bash substrings

This little ditty goes through a directory tree looking for filenames ending in "~" and comparing them to the same filename **NOT** ending in "~". In other words it quickly compares backup version of each file with the current version of the file. Like so:

```
for i in $(find . -name "*~")
do
    diff -u $i ${i:0:-1} | most
done
```

The new tidbit for me was the substring syntax:

If `$i` is a variable, `${i:offset:length}` is the substring (e.g. `${i:1:2}` will give the 2nd and 3rd character), but using negative values for the length parameter indicates how many letters should be cut from the end. (Using negative values for the offset appears to have no effect and behaves the same as zero.) Either `offset` or `length` can be omitted:

```
$ x="This string"
$ echo ${x::5}
This
$ echo ${x:5}
string
```

Marco the Marvelous suggests an improvement: Just remove last character:

```
${i%?}
```

This one is just saying output `$i` except remove the matching pattern after `%`, which is a single any character. A more specific version of the previous one, since all your files end with a tilde:

```
"${i%\~}"
```

This one is just saying output `$i` except for the matching pattern `\~`. (When escaped is just the ending tilde.)

For more fancy variable stuff, see GNU Bash - Shell Parameter Expansion

---

1.1.63 Compare two files ignoring whitespace AND newlines

Suppose you have two files:

```
$ cat file-A
The quick brown fox jumped over the lazy dog's back. Now    is     the    time...

$ cat file-B
The quick brown fox jumped
over the lazy dog's back.
Now is the time.
```

The text is identical, but the spacing is different. How can you determine that? A search of the package repositories turns up:

```
dwdiff    - diff program that operates word by word
docdiff   - Compares two files word by word / char by char
icdiff    - terminal side-by-side colorized word diff
numdiff   - Compare similar files with numeric fields
rfcdiff   - compares two internet draft files and outputs the difference
wdiff     - Compares two files word by word
wdiff-doc - Documentation for GNU wdiff
```

all of which may be worth exploring. However, StackExchange provides, once more:

```
$ diff <( tr -d " \n" <file-A ) \
       <( tr -d " \n" <file-B )
```

The command uses `tr` to delete all spaces and newlines from each file, and then feeds them as inputs to `diff`.

NOTE: The whitespace in the command is necessary.

Better still, `git` provides! `git` has a `--word-diff` option:

```
git diff --word-diff file-A file-B
```

1.1.64 Convert JPG to SVG in one swell foop

Easy (using **ImageMagick** as the middle-app):

```
convert -channel RGB -compress None input.jpg bmp:- | \
potrace -s - -o output.svg
```

1.1.65 Convert MIDI to MP3 in one swell foop

Thanks to StackOverflow - Convert midi to mp3 we have:

```
for i in *.midi
do
  timidity $i -Ow -o - | \
  ffmpeg -i - -acodec libmp3lame -ab 64k ${i:0:-5}.mp3
done
```

1.1.66 Expand MP4 to PNG frames

Sometimes, one just wants a single frame from a video:

```
ffmpeg -ss «timestamp»       \
       -i «source_movie».mp4 \
       -t «duration»         \
       «destination_frames»_%04d.png
```

will produce a series of images, `«destination_frames»_0000.png` through `«destination_frames»_%9999.png`. The `-ss «timestamp»` indicates the point from which the first frame should be extracted (e.g. `00:00` to start at the beginning) and the `-t «duration»` is the number of minutes and seconds to continue extracting ((e.g. `01:04` to get one minute and four seconds of frames).

The `%04d` indicates that the generated frames should have filenames that contain consecutive four-digit, zero-padded decimal number.

1.1.67 Running two applications in parallel

For this specific example, I wanted to have a GUI-based audio player play several tunes with an option for me to adjust the speed of playback, while simultaneously displayng a window with the sheet music for the corresponding tune. The magic that worked:

```
for i in polkas/john_ryans_polka \
         polkas/peg_ryans_polka  \
         single_jigs-slides/denis_murphys_slide-no._1
do
    evince $i.pdf &   \
    vlc    $i.mp3 &&  \
    fg
done
```

1.1.68 Create a new file with lines matching several patterns deleted

Suppose you have a mailing list, and you want to create a new mailing list with several addresses excluded. Easy. `fgrep` is your friend:

```
cat > remove
user1@host1.tld
user2@host2.tld
...
^D

fgrep -v -f remove source.csv > destination.csv
wc -l source.csv destination.csv remove
```

The `fgrep` uses `remove` as a file full of patterns to match. (And the `wc` is only there as a sanity check: The first number should equal the sum of the second and third numbers.)

1.1.69 Making terminal command-line "movies"

The `script` command will start a second Bash shell and record all terminal I/O including user input, command output and ANSI escape sequences for colorizing, cursor positioning, screen clearing, etc. The default behavior is to save the data to a file named `typescript`.

However, viewing such a file using standard tools is problematic: Using `cat typescript` shows the correct output, but it scrolls off the screen too quickly. Using a pager such as `most` prevents scrolling but does not know how to interpret the ANSI escape sequences, showing them as raw data, which drastically decreases the readability of the script.

Enter `scriptreplay`. By adding a **timing file** option to the `script` command, and then using that file with `scriptreplay` the `typescript` file will be displayed as if one was watching a ghost user enter commands into a terminal: The timing file preserves the original pauses that a user has while typing:

```
script -T timecodes
...
exit
```

produces two files: `typescript` with the terminal I/O and `timecodes` which contains the timing data for the keystrokes. By later typing:

```
scriptreplay -t timecodes
```

you can watch the "movie" in realtime.

The caveat: After issuing the `scriptreplay` command, the first thing you will see is the command prompt, making it appear as if the `scriptreplay` command has failed. This is an illusion: You are, in fact, NOT seeing the command prompt: You are seeing the recording of a command prompt: It is the first output in the `typescript` file. Wait. Eventually, the "ghost user" will begin typing.

The above caveat can be mitigated by creating an **alias** for the `scriptreplay` command that adds in messages indicating that the movie is about to start (or has just ended) and putting it into `.bash_aliases`, optionally adding an alias for `script` as well. The following aliases make `script` automatically produce a `timecodes` file, and produce

colorized (bright / bold red on yellow) informative messages regarding the starting and ending of a script replay to the `scriptreplay` command:

```
alias script='script -T timecodes '
alias scriptreplay='clear ; echo -e "\e[1;43;31m### STARTING replay ###\e[0m\n" ;␣
↪scriptreplay -t timecodes ; echo -e "\e[1;43;31m### FINISHED replay ###\e[0m" '
```

(That second alias is a bit long, and may eventually be changed to a Bash function in `.bashrc`. A wag of my acquainance, the goode Mr. Flint suggested the name `scriptease`. Clever. I like it.)

Ultimately, functions that use a special directory and time-stamp both the I/O data file an the timing file that pairs with it might be a good way to keep multiple recordings, rather than overwrite the files on each use. For example:

```
mkdir -p ~/scripts/$(date --iso)
mv typescript ~/scripts/$(date --iso)/
mv timecodes  ~/scripts/$(date --iso)/
```

CHAPTER

TWO

PDP-11 ASSEMBLER MADE PAINLESS (I HOPE)

2.1 Assembler Made Painless (I Hope)

by Kevin Cole

March 14, 1984

Dedicated to

CHICKEN

my favorite student

2.1.1 PREFACE

Updated 2020.09.10: *This was written 36 years ago, and at a time when students had access to the Digital Equipment Corporation (DEC) manuals. Also, in an era before hypertext. New users stumbling onto this may be better served by the* PDP-11 Processor Handbook *from the University of Calgary, Department of Computer Science, which is a mere 19 years old as of this vintage.*

I intend to explain much of the workings of a computer by analogy. The style is somewhat loose and flippant at times, but for the most part, has quite a bit of useful information to be absorbed. I am concentrating on what I consider to be of great importance in learning assembly language, and on areas which most people whom I have tutored had the greatest difficulty. So let's get straight to work.

2.1.2 INTRODUCTION

At the present time, most computers store information as electro-magnetic impulses. Many things in the physical world are "bipolar" in nature. That is, they have two states. In the case of magnets, the way in which the electrons are aligned create "poles" and by using electric current these poles can be reversed. This creates a switch with two conditions which can be interpreted as any of the following: on/off, true/false, 0/1, +/-… You get the picture. It is helpful to remember all of the above, because, depending on the context, the switch can represent any of the above conditions, and more. For convenience, this two-state switch (or binary switch) is referred to as a **BIT**.

2.1.3 BITS, BYTES and WORDS

Computer memory is filled with millions of these tiny electromagnetic bits. But, organized as independent elements, they cannot represent much. Therefore, bits are organized into groups of 8, 16, or 32. That is to say the computer will never interpret bits on an individual basis, but rather as part of a group of consecutive bits. This group of bits is referred to as a **WORD**. The size of a word varies from computer to computer. Most microcomputers have an 8-bit word, the large computers have a 32-bit word, and the PDP-11 minicomputers have 16-bit words. There are other machines which use unusual word sizes, but I won't go into that.

Sometimes, it is not handy to grab 16 bits at once. Occasionally, (actually quite often), it is more useful to get only 8 bits at a time. 8 consecutive bits are referred to as a **BYTE**.

Table 1: Word sizes on various machines

Computer	Type	Word size in bytes	(bits)	Notes
APPLE II-e	Micro	1 byte	8 bits	
Commodore VIC20	Micro	1 byte	8 bits	
PDP-11	Mini	2 bytes	16 bits	
Apple Macintosh	Micro	4 bytes	32 bits	
IBM-360	Mainframe	4 bytes	32 bits	
VAX	Mid-size	4 bytes	32 bits	
DEC 10	Mainframe	? bytes	36 bits	•

- The DECsystem 10 uses a concept called variable length bytes. Not discussed here.

Let's look at some of the more frequent uses for bits, bytes and words.

Here's a group of 8 bits (1 byte):

```
0 1 1 0 1 1 0 1
```

This can be interpreted several different ways. Each bit could be set up as a flag or switch to indicate when a particular condition has occurred. For example, when a calculation is performed the leftmost bit (also referred to as the high order bit) might be set to 1 if the result is negative, and set to 0 for positive results. The second bit from the left might be set when there is an error in the mathematical operation being performed, for example division by zero.

Another way to interpret the byte is as a binary number. In this case:

$$(0 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$$
$$(0 \times 128) + (1 \times 64) + (1 \times 32) + (0 \times 16) + (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) =$$
$$0 + 64 + 32 + 0 + 8 + 4 + 0 + 1 =$$
$$109$$

2.1.4 ASCII

Other ways of interpreting it, involve using the value as a code. Perhaps, as a kid, you played around with the idea of codes for passing notes between you and someone else, that told your innermost feelings about a member of the opposite sex, or something you did not want read by parents or teachers… In computerland, the code is not exactly used for that reason, but is a way of storing textual information in memory. The standard code used today is referred to as **ASCII** (American Standard Code for Information Interchange), pronounced "ass-kee". In this code, every character on your terminal keyboard, or on a printer, or in a file on disk, is represented as an 8-bit code. Actually, only 7 bits are currently needed for the code. The high order bit is not used on most machines. There are 128 different characters in the ASCII code ranging in value from 0 to 127.

The first 32 codes are non-printing control-characters. These are generated by holding the control key as you strike some other key. Actually, some of these have a visible effect, however they are still referred to as non-printing. Some of these characters, because of their frequent use, are also located in a separate key that does not have to be struck in conjunction with the control key. The most useful of these to remember are:

Table 2: Control-key values and corresponding key name

Control character	Keyboard key name	Notes
Control-[	= ESCAPE	(sometimes marked ALTMODE)
Control-H	= BACKSPACE	
Control-I	= TAB	
Control-J	= LINE FEED	(sometimes mapped to ENTER)
Control-M	= CARRIAGE RETURN	(sometimes mapped to ENTER)

Try these. (For example, next time you are using the machine type `Control-I` instead of `TAB`.)

(I know it may seem that I am belaboring this point, but bear with me.)

The code for a Control-A is 00000001 binary. Control-B is 00000010. Control-C = 00000011… If you understand binary as well as you indicated, you should see the pattern developing. Believe me, it's important. If you are debugging a machine language program, and you don't know the ASCII code, you'll be at it more than twice as long as necessary.

The next group of 32 (codes 32-63) are your digits and special symbols; punctuation generaly speaking. **IMPORTANT:** The ASCII code for the digit 2 is NOT the same as the value 2 !!! The digits on you keyboard and printer are merely **SYMBOLS**. The ASCII code for the digit 2 is 00110010 binary (or 50 decimal). If you try to print the value 2 (binary 00000010) it will print as a Control-B. Just as the same sound spoken at different pitches has different meanings in Chinese, so too, two bytes which are identical can have different meanings, depending on the context in which they are used. This is one of the most difficult and most important thing to remember about assembler language.

Next are the upper case letters (and a few punctuation symbols). These are codes 64-95. And lastly, the lower case letters (and still a few more punctuation marks) codes 96-127.

See the cheat sheet included in this for a better understanding. I will refer to it often in here.

2.1.5 INSTRUCTIONS

The most common use for a byte is as part of an instruction to the computer. This too is a code. For instance the value 00000011 on the PDP says Branch when Equal to zero (`BEQ`). The codes for each instruction are so tedious to memorize, that it is not worth the trouble. For the most part, you will have access to some reference which allows you to find the code for an instruction quickly. And, since there is NO standard here, it does no good to memorize the instruction codes for a PDP-11 and then go work on an IBM. The codes are completely different.

Therefore, it is better to memorize the scheme used for the mnemonics for the various instructions. Like knowing that a mnemonic which starts with the letter "B" is most likely to be a Branch instruction. This generally holds true for all machines.

2.1.6 ADDRESSING

Ok, so now you have these millions of bits organized into consecutive groups. Now comes the problem of how to get to a particular group. Well, each byte is assigned an **ADDRESS** starting at 0 and increasing by 1 til all memory is exhausted. Unfortunately, here's where the problems start…

I mentioned earlier that the PDP-11 "thinks" in 16-bit words. This means it usually grabs two bytes at a shot. This makes things complicated when you start to think about how memory on the 11 is organized. But I will labor to make it a bit more accessible. (Ha ha. Get it? A BIT more ACCESSible… Never mind.)

Most machines tend to number their bytes consecutively, but because Digital Equipment Corporation wants to give students a hard time, they decided to do things differently… Actually there is a method to their particular brand of madness, and I hope it will become clearer as we continue. Above, I mentioned that the computer we are using here works in units of 16 bits, or 2 bytes. And each byte has an address. Because it uses a 16-bit word, it can work with positive integers between 0 and $2^{16} - 1$ (0000000000000000 to 1111111111111111 which is 1000000000000000 minus 1, take my word for it). Representing negative numbers becomes a bit more complicated. It uses a scheme called **2's-complement notation**. More on that subject later. Because the rightmost 8 bits hold the lower portion of the value, it is considered to be the low-order byte and has the lower address value. This is always the even-addressed byte. And the high-order byte is the odd-addressed byte.

Imagining it as boxes filled with values:

1	0	3	2	5	4	
00000011	11010000	10101010	11110000	00000000	00000000	…

The numbers below the boxes represent the address of each byte in memory. The values in the boxes are the contents of each byte. Confused yet? If not, I'll try harder to confuse you.

2.1.7 OCTAL

After a while, it becomes very tedious work to look at everything in binary. So on most machines, **HEXADECIMAL** or **BASE 16** is used to represent groups of 4 bits conveniently. Thus, the contents (or the address) of a 16-bit word could be written as 4 hexadecimal digits. However, DEC chose **OCTAL** or **BASE 8**. There is no justifiable reason for this. It is one of very few things DEC did which is just plain stupid. You'll see why as we go on. Anyway, octal represents 3 bits as a single digit. Refer to the cheat sheet for an illustration of how to count in any of the useful computer numbering systems. I am going to assume you either are already sufficiently familiar with it from previous school or that your assembly language teacher has covered it in depth.

2.1.8 GENERAL PURPOSE REGISTERS (or ACCUMULATORS)

Because memory is usually quite large, and it takes time for a computer to reference a particular address and manipulate the contents, computer designers created special locations which can be used as a fast memory and can be used easily with most instructions. These are called **ACCUMULATORS** on some machines and **REGISTERS** on others. The PDP-11 has 8 such registers, which are numbered `%0` to `%7`. Two of these have highly specialized functions. `%6` is known as the **Stack Pointer (SP)** and `%7` is the **Program Counter (PC)**. More on both of these later.

2.1.9 ADDRESSING MODES

You probably skipped right to this section because you have no patience. Well, good luck… cause if you did you may have missed something very important.

Generally speaking, to manipulate data in the computer, information is obtained from some auxiliary device (INPUT) and moved into memory. Then it is moved from memory to an register, where it is bent, folded, spindled, and mutilated (i.e. added to, subtracted from, etc.), moved back into memory and finally moved to an auxiliary device again (OUTPUT). Often the information must undergo a transformation or conversion from some external form (such as the ASCII code) to an internal representation (such as binary integer). One word of memory only holds 2 characters, maximum. This means that if we enter a 5 digit number from the keyboard, it will take up 2 and 1/2 words of memory in its ASCII form. Remember, each character entered from the keyboard occupies 1 byte of memory.

So, let's say we entered the string `"98760"`. Each character would be stored in ascending bytes. The octal codes for each digit are: 071, 070, 067, 066, 060, respectively. In memory, represented in octal and binary this would be:

1	0	3	2	5	4
0 7 0	0 7 1	0 6 6	0 6 7	0 0 0	0 6 0
00111000	00111001	00110110	00110111	00000000	00110000

By crushing the two halves of each word together we get:

0 3 4 0 7 1	0 3 3 0 6 7	0 0 0 0 6 0
0011100000111001	0011011000110111	0000000000110000

(It is still the same binary value, but when shown as a 6-digit octal value representing the word as a single value, the distinction between the 2 bytes becomes a bit blurred. This would not have happened if DEC had chosen hexadecimal. If the above illustration is not clear, we will have to go over it together.)

There are several different ways to access the above bytes. The simplest, is to use **ABSOLUTE** addressing. This means when referencing a location in memory, you give the address of that particular word in the instruction. So, if the 3 words in the above example occupied memory locations 400 to 406 (octal), and we wish to move the first word to another location, we would specify the value 400 as the address. The format for this "`@#400`".

Absolute addressing, though simple, has several drawbacks. First of all, some instructions were implemented by the manufacturer in such a way that they cannot use absolute addressing. Secondly, the addresses can range from 000000 to 177777 (octal). This means that when using absolute addressing in an instruction, one word of computer memory is used for the instruction itself, and another for the absolute address. This eats up space quickly. Other addressing modes do not share this deficiency. And it does not have much flexability. If you change the location of your data area in memory, it must have all of your program's absolute addresses changed as well.

In both FORTRAN and COBOL you encountered the idea of an indexed table (also called an array or matrix). This is used when you have several locations which are all related in some way (for example a list of department names which you select from based on the department number):

```
COBOL:      MOVE DEPART-LIST (DEPT-NUMBER-IN) TO DEPT-OUT.
FORTRAN:    DEPOUT = DEPT(CODE)
```

The number in the variables in the parentheses are indexes into the arrays. Well in **INDEXED** addressing, the same concept applies. The index value is kept in an register. In the above example, the first byte would be referenced by placing a value of 0 into an register (for now lets use `%2`) and using an address like "`400(%2)`". The "**EFFECTIVE**" address, i.e. the actual address from which the computer fetches the data, is computed by adding the contents of

register %2 to the offset 400. The result is 400. Now if we add 1 to register %2 the effective address becomes 401 but the instruction did not have to be changed, only the contents of the index register. You can choose any register to be your index (except the PC and SP), as long as you are not using it for anything else at that point in your program.

In COBOL or FORTRAN you would have a separate instruction to add one to your index... The same is true for indexed addressing. However there is an addressing mode which will automatically increment the index after it is used in an instruction. As you have probably guessed by now, this is the **AUTO-INCREMENT** addressing mode and is specified as "(%2)+". With this mode, however, you are not allowed to use an offset. Therefore, for our example, you would have to set register %2 to a value of 400 before using this mode. Its' crude equivalent in COBOL would be:

```
MOVE DEPART-LIST (DEPT-NUMBER-IN) TO DEPT-OUT.
ADD 1 TO DEPT-NUMBER-IN.
```

Now the index is automatically set up for the next pass through the loop to reference the next element in the table.

Now it seems appropriate to introduce two topics at once. These are the **STACK** and **AUTO-DECREMENT** mode. Suppose you have part of a calculation completed, and you have another quantity to compute. You do not want to loose the first part while calculating the next part. One solution is to move it from the register where you've chosen to store it, to a temporary memory location, and remember where you've left it... The **STACK POINTER** remembers it for you, and the STACK is the temporary area where you keep it. And both auto-increment and auto-decrement are used to control this process.

Ok. This is a typical, overused analogy, but it still works well. When you eat at the cafeteria, you often pick up a tray from a STACK of trays. As you pick yours up, a spring beneath the stack pops the next tray up a little. And when you replace your tray, it pushes everything down slightly. Now imagine each tray has a piece of paper with a number on it. Each tray is a word of memory and the paper is the contents of that word. Auto-increment mode is analogous to PUSHing a tray onto the stack and auto-decrement corresponds to POPing the stack.

Auto-decrement mode works in reverse. It subtracts 1 from the index first, then it uses the result to calculate the effective address. So now, to go back to our example, we could go through our 5 bytes of memory backwards. Starting with a 406 in register %2 and using "-(%2)" would access the byte located at location 405. Using it again would get the byte at 404, and so on.

Look at the following sequence of code:

```
MOVE 400 TO STACK-POINTER.
   .
   .    (Calculate something called TOT-1 here)
   .
MOVE TOT-1 TO MEMORY (STACK-POINTER).
ADD  1 TO STACK-POINTER.
   .
   .    (Calculate something called TOT-2 here)
   .
SUBTRACT 1 FROM STACK-POINTER.
ADD MEMORY (STACK-POINTER) TO TOT-2.
```

That's roughly what you should think of auto-inc and auto-dec as doing. And that's a good approximation of the stack also.

I've been misleading you somewhat with these examples. Actually, auto-increment and auto-decrement can add/subtract 1 from a register or it can add/subtract 2 from a register. This depends on the kind of instruction you are using. I mentioned at the beginning of this text that the PDP-11 usually handles memory in 16-bit quantities called words. This means it usually uses only the even addresses and works with the even/odd pair of bytes at that location in memory. However, using the byte instructions, you can force counting by 1 and only access 8-bits at a time.

The stack is also used in subroutines to "remember" where the main program is to resume execution when the subroutine finishes.

REGISTER mode addressing is similar to absolute. It's simple. It does not work with memory at all. Use it when all of your data is in registers already and specify it using "`%0`" through "`%7`".

IMMEDIATE mode. In the previous examples, we were accessing variable information in memory. Occasionally in machine language, as in other high level languages, it is necessary to use a constant, for example multiplying some variable by 100 to calculate percentage. Well, it's silly to store the value 100 in memory somewhere, then when we need it, fetch it from memory for use with the multiply instruction and never use it again. This takes unnecessary time. Instead, it would be much better to have the value available to the instruction without going to some other memory location to get it. So you would have it IMMEDIATELY. Strictly speaking, it's not an address. But it is classified that way. To use it specify "`#100`".

RELATIVE addressing and the PC… This one is pretty complicated, so pay attention. (I know, you find them ALL complicated. Sorry 'bout that. Doing the best I can.) Register 7 is used by the system itself, to point to the location of the current instruction being executed in memory. Keep in mind that your program and data both reside in memory and there is no way for the machine to distinguish one from the other, except through the fact that the PC (PROGRAM COUNTER) is pointing to the next instruction to be executed. **IF YOU MESS THIS UP AND POINT IT TO YOUR DATA AREA THE COMPUTER WILL TRY TO EXECUTE YOUR DATA !!!** RELATIVE addressing involves using an offset from the PC. This is transparent to you when you are writing the program. The assembler will convert things properly for you. It will look like you are using absolute addressing. The difference is how the address is stored in memory, when it is assembled. Absolute addressing will use the actual address you type in, but relative does not. Consider the following illustration:

Suppose at location 300 (octal) you have an MOV instruction:

```
MOV     %2,@#400        will assemble the address as 400
MOV     %2,400          will assemble the address as  74
```

Really the second instruction will use an address of 74 because relative addressing is calculated by adding 4 to the PC (which would be 300 in this case) and subtracting that from the address you specified (all in octal).

The reason for using relative addressing is that it makes the program position-independent (**PIC** = Position Independent Code). This means, unlike absolute addressing, that the program can be moved to a different memory location and it will still run correctly.

All of the **DEFERRED** (sometimes referred to as **INDIRECT**) modes work the same way, so I will just explain the concept. The best example of indirect addressing is the idea of a "jump" table. Suppose you have a program which expects the user to enter a code for the operation he/she wishes to perform.

Table 3: Example user application menu codes

Code	Meaning
0	Exit
1	Add 2 numbers
2	Subtract 2 numbers
3	Multiply 2 numbers
4	Divide 2 numbers

You could set up a table in memory with the address of each subroutine. Then, if the user types a 1 you jump to the CONTENTS of the first ADDRESS in the table. You do NOT jump to the first address of the table, but you use the address of the table to point to another address.

If I tell you to go to the bedroom to get a book for me, and you went to get it and found a note saying the book is in the kitchen. That's indirect addressing. You went to one location which told you to go to a second location.

You can use indirect addressing in conjunction with most of the other modes, thus producing an auto-increment deferred, etc.

2.1.10 SYMBOLS and MNEMONICS

In the early days of computing, machine language programs were written completely with numbers. Imagine writing a program that looks like the left-hand side of your list (.LST) file… Pretty awful, if you ask me. So, names for each operation were invented. These are known as **MNEMONICS**. For example, MOV is the mnemonic for the move instruction whose octal code (often called an opcode) is 01####. (The #### portion is part of the address.) So, now we can say:

```
MOV     %2,@#400        instead of     010237 000400
```

This still has limitations, cause you have to remember all the addresses you are using. It becomes much easier when you use a symbol to remember a location. A **SYMBOL** is essentially a name for a memory location. Thus, you can say:

```
...
        MOV     %2,@#ANSWER
            .
            .
            .
ANSWER: .BLKW   1
```

And assuming for the sake of example, that `ANSWER:` is at location 400 it is identical to the previous method but a heck of a lot clearer than `010237 000400`.

2.1.11 APPENDIX A: Addressing Mode Summary

Table 4: Addressing Mode Summary

Code	Mode	Source Code	Machine Code
0	Register	MOV %3,%4	010304
1	Register Deferred	MOV (%3),(%4)	011314
2	Auto-Increment	MOV (%3)+,(%4)+	012324
3	Auto-Increment Deferred	MOV @(%3)+,@(%4)+	013334
4	Auto-Decrement	MOV -(%3),-(%4)	014344
5	Auto-Decrement Deferred	MOV @-(%3),@-(%4)	015354
6	Indexed	MOV 10(%3),15(%4)	016364 000010 000015
7	Index Deferred	MOV 10(%3),15(%4)	017374 000010 000015

The following four modes use the Program Counter (PC) as the register portion of the addressing mode. (Note the forth digit in the first word of the machine code.)

Table 5: Addressing Mode Summary

Code	Mode	Source Code	Machine Code
2	Immediate	MOV #100,%4	012704 000100
3	Absolute	MOV @#400,@#500	013737 000400 000500
6	Relative	MOV 400,500	016767 000074 000170
7	Relative Deferred	MOV @400,@500	017777 000074 000170

2.1.12 APPENDIX B: Instruction Format Summary

Table 6: Instruction Format Summary

Bit Pattern	Mnemonic	Opcode	Group
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	HALT	000000	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	WAIT	000001	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	RTI	000002	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	BPT	000003	0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0	IOT	000004	0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1	RESET	000005	0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0	RTT	000006	0
0 0 0 0 0 0 0 0 0 1 d d d d d d	JMP	000100	1
0 0 0 0 0 0 0 0 1 0 0 0 0 r r r	RTS	000200	1
0 0 0 0 0 0 0 0 1 0 0 1 1 n n n	SPL	000230	4
0 0 0 0 0 0 0 0 1 0 1 0 n z v c	CL(N/Z/V/C)	000240	5
0 0 0 0 0 0 0 0 1 0 1 1 n z v c	SE(N/Z/V/C)	000260	5
0 0 0 0 0 0 0 0 1 1 d d d d d d	SWAB	000300	1
0 0 0 0 0 0 0 1 o o o o o o o o	BR	000400	2
0 0 0 0 0 0 1 0 o o o o o o o o	BNE	001000	2
0 0 0 0 0 0 1 1 o o o o o o o o	BEQ	001400	2
0 0 0 0 0 1 0 0 o o o o o o o o	BGE	002000	2
0 0 0 0 0 1 0 1 o o o o o o o o	BLT	002400	2
0 0 0 0 0 1 1 0 o o o o o o o o	BGT	003000	2
0 0 0 0 0 1 1 1 o o o o o o o o	BLE	003400	2
0 0 0 0 1 0 0 r r r d d d d d d	JSR	004000	3
b 0 0 0 1 0 1 0 0 0 d d d d d d	CLR(B)	005000	1
b 0 0 0 1 0 1 0 0 1 d d d d d d	COM(B)	005100	1
b 0 0 0 1 0 1 0 1 0 d d d d d d	INC(B)	005200	1
b 0 0 0 1 0 1 0 1 1 d d d d d d	DEC(B)	005300	1
b 0 0 0 1 0 1 1 0 0 d d d d d d	NEG(B)	005400	1
b 0 0 0 1 0 1 1 0 1 d d d d d d	ADC(B)	005500	1
b 0 0 0 1 0 1 1 1 0 d d d d d d	SBC(B)	005600	1
b 0 0 0 1 0 1 1 1 1 d d d d d d	TST(B)	005700	1
b 0 0 0 1 1 0 0 0 0 d d d d d d	ROR(B)	006000	1
b 0 0 0 1 1 0 0 0 1 d d d d d d	ROL(B)	006100	1
b 0 0 0 1 1 0 0 1 0 d d d d d d	ASR(B)	006200	1
b 0 0 0 1 1 0 0 1 1 d d d d d d	ASL(B)	006300	1
0 0 0 0 1 1 0 1 0 0 n n n n n n	MARK	006400	4
0 0 0 0 1 1 0 1 0 1 s s s s s s	MFPI	006500	1
0 0 0 0 1 1 0 1 1 0 d d d d d d	MTPI	006600	1
0 0 0 0 1 1 0 1 1 1 d d d d d d	SXT	006700	1
b 0 0 1 s s s s s s d d d d d d	MOV(B)	010000	3
b 0 1 0 s s s s s s d d d d d d	CMP(B)	020000	3
b 0 1 1 s s s s s s d d d d d d	BIT(B)	030000	3
b 1 0 0 s s s s s s d d d d d d	BIC(B)	040000	3
b 1 0 1 s s s s s s d d d d d d	BIS(B)	050000	3
0 1 1 0 s s s s s s d d d d d d	ADD	060000	3
0 1 1 1 0 0 0 r r r s s s s s s	MUL	070000	7
0 1 1 1 0 0 1 r r r s s s s s s	DIV	071000	7
0 1 1 1 0 1 0 r r r s s s s s s	ASH	072000	7
0 1 1 1 0 1 1 r r r s s s s s s	ASHC	073000	7

continues on next page

Table 6 – continued from previous page

Bit Pattern	Mnemonic	Opcode	Group
0 1 1 1 1 0 0 r r r d d d d d d	XOR	074000	3
0 1 1 1 1 0 1 0 0 0 0 0 0 r r r	FADD	075000	1
0 1 1 1 1 0 1 0 0 0 0 0 1 r r r	FSUB	075010	1
0 1 1 1 1 0 1 0 0 0 0 1 0 r r r	FMUL	075020	1
0 1 1 1 1 0 1 0 0 0 0 1 1 r r r	FDIV	075030	1
0 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0	MED	076600	0
0 1 1 1 1 1 0 1 1 1 d d d d d d	XFC	076700	1
0 1 1 1 1 1 1 r r r o o o o o o	SOB	077000	6
1 0 0 0 0 0 0 0 o o o o o o o o	BPL	100000	2
1 0 0 0 0 0 0 1 o o o o o o o o	BMI	100400	2
1 0 0 0 0 0 1 0 o o o o o o o o	BHI	101000	2
1 0 0 0 0 0 1 1 o o o o o o o o	BLOS	101400	2
1 0 0 0 0 1 0 0 o o o o o o o o	BVC	102000	2
1 0 0 0 0 1 0 1 o o o o o o o o	BVS	102400	2
1 0 0 0 0 1 1 0 o o o o o o o o	BCC	103000	2
1 0 0 0 0 1 1 0 o o o o o o o o	BHIS	103000	2
1 0 0 0 0 1 1 1 o o o o o o o o	BCS	103400	2
1 0 0 0 0 1 1 1 o o o o o o o o	BLO	103400	2
1 0 0 0 1 0 0 0 t t t t t t t t	EMT	104000	4
1 0 0 0 1 0 0 1 t t t t t t t t	TRAP	104400	4
1 0 0 0 1 1 0 1 0 0 s s s s s s	MTPS	106400	1
1 0 0 0 1 1 0 1 0 1 s s s s s s	MFPD	106500	1
1 0 0 0 1 1 0 1 1 0 d d d d d d	MTPD	106600	1
1 0 0 0 1 1 0 1 1 1 d d d d d d	MFPS	106700	1
1 1 1 0 s s s s s s d d d d d d	SUB	160000	3
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0	CFCC	170000	0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1	SETF	170001	0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0	SETI	170002	0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1	LDUB	170003	0
1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0	MNS	170004	0
1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1	MPP	170005	0
1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1	MAS	170007	0
1 1 1 1 0 0 0 0 0 0 0 0 1 0 0 1	SETD	170011	0
1 1 1 1 0 0 0 0 0 0 0 0 1 0 1 0	SETL	170012	0
1 1 1 1 0 0 0 0 0 1 s s s s s s	LDFPS	170100	1
1 1 1 1 0 0 0 0 1 0 d d d d d d	STFPS	170200	1
1 1 1 1 0 0 0 0 1 1 d d d d d d	STST	170300	1
1 1 1 1 0 0 0 1 0 0 d d d d d d	CLR(F/D)	170400	1
1 1 1 1 0 0 0 1 0 1 fdfdfdfdfdfd	TST(F/D)	170500	1
1 1 1 1 0 0 0 1 1 0 fdfdfdfdfdfd	ABS(F/D)	170600	1
1 1 1 1 0 0 0 1 1 1 fsfsfsfsfsfs	NEG(F/D)	170700	1
1 1 1 1 0 0 1 0 f f fsfsfsfsfsfs	MUL(F/D)	171000	7
1 1 1 1 0 0 1 1 f f fsfsfsfsfsfs	MOD(F/D)	171400	7
1 1 1 1 0 1 0 0 f f fsfsfsfsfsfs	ADD(F/D)	172000	7
1 1 1 1 0 1 0 1 f f fsfsfsfsfsfs	LD(F/D)	172400	7
1 1 1 1 0 1 1 0 f f fsfsfsfsfsfs	SUB(F/D)	173000	7
1 1 1 1 0 1 1 1 f f fsfsfsfsfsfs	CMP(F/D)	173400	7
1 1 1 1 1 0 0 0 f f fdfdfdfdfdfd	ST(F/D)	174000	3
1 1 1 1 1 0 0 1 f f fsfsfsfsfsfs	DIV(F/D)	174400	7
1 1 1 1 1 0 1 0 f f d d d d d d	STEXP	175000	3

continues on next page

Table 6 – continued from previous page

Bit Pattern	Mnemonic	Opcode	Group
1 1 1 1 1 0 1 1 f f d d d d d d	STCFI	175400	3
1 1 1 1 1 0 1 1 f f d d d d d d	STCFL	175400	3
1 1 1 1 1 0 1 1 f f d d d d d d	STCDI	175400	3
1 1 1 1 1 0 1 1 f f d d d d d d	STCDL	175400	3
1 1 1 1 1 1 0 0 f f fdfdfdfdfdfd	STCFD	176000	3
1 1 1 1 1 1 0 0 f f fdfdfdfdfdfd	STCDF	176000	3
1 1 1 1 1 1 0 1 f f s s s s s s	LDEXP	176400	7
1 1 1 1 1 1 1 0 f f s s s s s s	LDCIF	177000	7
1 1 1 1 1 1 1 0 f f s s s s s s	LDCID	177000	7
1 1 1 1 1 1 1 0 f f s s s s s s	LDCLF	177000	7
1 1 1 1 1 1 1 0 f f s s s s s s	LDCLD	177000	7
1 1 1 1 1 1 1 1 f f fsfsfsfsfsfs	LDCDF	177400	7
1 1 1 1 1 1 1 1 f f fsfsfsfsfsfs	LDCFD	177400	7

Table 7: Instruction Argument Groupings

Group	Instruction format
0	opcode
1	opcode source opcode destination
2	opcode offset
3	opcode source,destination
4	opcode octal-number
5	opcode bit-pattern
6	opcode register,offset
7	opcode destination,source

Table 8: Meanings of abbreviations

Bits	Meanings	Number of bits
r	Register	3
s	Source Address	6 (3 for mode, 3 for register)
d	Destination Address	6 (3 for mode, 3 for register)
o	Offset	8 (except SOB uses 6)
n z v c	Condition Codes	4
t	Trap	8
f	Floating Point Register	2
fs	Floating Point Source	6
fd	Floating Point Destination	6
n	Octal Number	3 for SPL, 6 for MARK

2.1.13 APPENDIX C: The D.O.B.A.S.H. Cheat Sheet

Table 9: The D.O.B.A.S.H. Cheat Sheet

Decimal	Octal	Binary	ASCII	Sixbit	Hex
000	000	0 0 0 0 0 0 0	^@		00
001	001	0 0 0 0 0 0 1	^A	!	01
002	002	0 0 0 0 0 1 0	^B	"	02
003	003	0 0 0 0 0 1 1	^C	#	03
004	004	0 0 0 0 1 0 0	^D	$	04
005	005	0 0 0 0 1 0 1	^E	%	05
006	006	0 0 0 0 1 1 0	^F	&	06
007	007	0 0 0 0 1 1 1	^G	'	07
008	010	0 0 0 1 0 0 0	^H	(	08
009	011	0 0 0 1 0 0 1	^I	)	09
010	012	0 0 0 1 0 1 0	^J	*	0A
011	013	0 0 0 1 0 1 1	^K	+	0B
012	014	0 0 0 1 1 0 0	^L	,	0C
013	015	0 0 0 1 1 0 1	^M	-	0D
014	016	0 0 0 1 1 1 0	^N	.	0E
015	017	0 0 0 1 1 1 1	^O	/	0F
016	020	0 0 1 0 0 0 0	^P	0	10
017	021	0 0 1 0 0 0 1	^Q	1	11
018	022	0 0 1 0 0 1 0	^R	2	12
019	023	0 0 1 0 0 1 1	^S	3	13
020	024	0 0 1 0 1 0 0	^T	4	14
021	025	0 0 1 0 1 0 1	^U	5	15
022	026	0 0 1 0 1 1 0	^V	6	16
023	027	0 0 1 0 1 1 1	^W	7	17
024	030	0 0 1 1 0 0 0	^X	8	18
025	031	0 0 1 1 0 0 1	^Y	9	19
026	032	0 0 1 1 0 1 0	^Z	:	1A
027	033	0 0 1 1 0 1 1	^[	;	1B
028	034	0 0 1 1 1 0 0	^\	<	1C
029	035	0 0 1 1 1 0 1	^]	=	1D
030	036	0 0 1 1 1 1 0	^^	>	1E
031	037	0 0 1 1 1 1 1	^_	?	1F
032	040	0 1 0 0 0 0 0		@	20
033	041	0 1 0 0 0 0 1	!	A	21
034	042	0 1 0 0 0 1 0	"	B	22
035	043	0 1 0 0 0 1 1	#	C	23
036	044	0 1 0 0 1 0 0	$	D	24
037	045	0 1 0 0 1 0 1	%	E	25
038	046	0 1 0 0 1 1 0	&	F	26
039	047	0 1 0 0 1 1 1	'	G	27
040	050	0 1 0 1 0 0 0	(	H	28
041	051	0 1 0 1 0 0 1	)	I	29
042	052	0 1 0 1 0 1 0	*	J	2A
043	053	0 1 0 1 0 1 1	+	K	2B
044	054	0 1 0 1 1 0 0	,	L	2C
045	055	0 1 0 1 1 0 1	-	M	2D
046	056	0 1 0 1 1 1 0	.	N	2E

continues on next page

Table 9 – continued from previous page

Decimal	Octal	Binary	ASCII	Sixbit	Hex
047	057	0 1 0 1 1 1 1	/	O	2F
048	060	0 1 1 0 0 0 0	0	P	30
049	061	0 1 1 0 0 0 1	1	Q	31
050	062	0 1 1 0 0 1 0	2	R	32
051	063	0 1 1 0 0 1 1	3	S	33
052	064	0 1 1 0 1 0 0	4	T	34
053	065	0 1 1 0 1 0 1	5	U	35
054	066	0 1 1 0 1 1 0	6	V	36
055	067	0 1 1 0 1 1 1	7	W	37
056	070	0 1 1 1 0 0 0	8	X	38
057	071	0 1 1 1 0 0 1	9	Y	39
058	072	0 1 1 1 0 1 0	:	Z	3A
059	073	0 1 1 1 0 1 1	;	[	3B
060	074	0 1 1 1 1 0 0	<	\	3C
061	075	0 1 1 1 1 0 1	=	]	3D
062	076	0 1 1 1 1 1 0	>	^	3E
063	077	0 1 1 1 1 1 1	?	_	3F
064	100	1 0 0 0 0 0 0	@		40
065	101	1 0 0 0 0 0 1	A		41
066	102	1 0 0 0 0 1 0	B		42
067	103	1 0 0 0 0 1 1	C		43
068	104	1 0 0 0 1 0 0	D		44
069	105	1 0 0 0 1 0 1	E		45
070	106	1 0 0 0 1 1 0	F		46
071	107	1 0 0 0 1 1 1	G		47
072	110	1 0 0 1 0 0 0	H		48
073	111	1 0 0 1 0 0 1	I		49
074	112	1 0 0 1 0 1 0	J		4A
075	113	1 0 0 1 0 1 1	K		4B
076	114	1 0 0 1 1 0 0	L		4C
077	115	1 0 0 1 1 0 1	M		4D
078	116	1 0 0 1 1 1 0	N		4E
079	117	1 0 0 1 1 1 1	O		4F
080	120	1 0 1 0 0 0 0	P		50
081	121	1 0 1 0 0 0 1	Q		51
082	122	1 0 1 0 0 1 0	R		52
083	123	1 0 1 0 0 1 1	S		53
084	124	1 0 1 0 1 0 0	T		54
085	125	1 0 1 0 1 0 1	U		55
086	126	1 0 1 0 1 1 0	V		56
087	127	1 0 1 0 1 1 1	W		57
088	130	1 0 1 1 0 0 0	X		58
089	131	1 0 1 1 0 0 1	Y		59
090	132	1 0 1 1 0 1 0	Z		5A
091	133	1 0 1 1 0 1 1	[		5B
092	134	1 0 1 1 1 0 0	\		5C
093	135	1 0 1 1 1 0 1	]		5D
094	136	1 0 1 1 1 1 0	^		5E
095	137	1 0 1 1 1 1 1	_		5F
096	140	1 1 0 0 0 0 0			60

continues on next page

Table 9 – continued from previous page

Decimal	Octal	Binary	ASCII	Sixbit	Hex
097	141	1 1 0 0 0 0 1	a		61
098	142	1 1 0 0 0 1 0	b		62
099	143	1 1 0 0 0 1 1	c		63
100	144	1 1 0 0 1 0 0	d		64
101	145	1 1 0 0 1 0 1	e		65
102	146	1 1 0 0 1 1 0	f		66
103	147	1 1 0 0 1 1 1	g		67
104	150	1 1 0 1 0 0 0	h		68
105	151	1 1 0 1 0 0 1	i		69
106	152	1 1 0 1 0 1 0	j		6A
107	153	1 1 0 1 0 1 1	k		6B
108	154	1 1 0 1 1 0 0	l		6C
109	155	1 1 0 1 1 0 1	m		6D
110	156	1 1 0 1 1 1 0	n		6E
111	157	1 1 0 1 1 1 1	o		6F
112	160	1 1 1 0 0 0 0	p		70
113	161	1 1 1 0 0 0 1	q		71
114	162	1 1 1 0 0 1 0	r		72
115	163	1 1 1 0 0 1 1	s		73
116	164	1 1 1 0 1 0 0	t		74
117	165	1 1 1 0 1 0 1	u		75
118	166	1 1 1 0 1 1 0	v		76
119	167	1 1 1 0 1 1 1	w		77
120	170	1 1 1 1 0 0 0	x		78
121	171	1 1 1 1 0 0 1	y		79
122	172	1 1 1 1 0 1 0	z		7A
123	173	1 1 1 1 0 1 1	{		7B
124	174	1 1 1 1 1 0 0	|		7C
125	175	1 1 1 1 1 0 1	}		7D
126	176	1 1 1 1 1 1 0	~		7E
127	177	1 1 1 1 1 1 1	RU		7F

2.1.14 APPENDIX D: Powers of 2 Cheat Sheet

Table 10: Powers of 2 Cheat Sheet

Power	Decimal	Octal	Hex	K
0	1	1	1	0 K
1	2	2	2	0 K
2	4	4	4	0 K
3	8	10	8	0 K
4	16	20	10	0 K
5	32	40	20	0 K
6	64	100	40	0 K
7	128	200	80	0 K
8	256	400	100	0 K
9	512	1000	200	0 K
10	1024	2000	400	1 K
11	2048	4000	800	2 K
12	4096	10000	1000	4 K
13	8192	20000	2000	8 K
14	16384	40000	4000	16 K
15	32768	100000	8000	32 K
16	65536	200000	10000	64 K
17	131072	400000	20000	128 K
18	262144	1000000	40000	256 K
19	524288	2000000	80000	512 K
20	048576	4000000	100000	1024 K
21	097152	10000000	200000	2048 K
22	194304	20000000	400000	4096 K
23	388608	40000000	800000	8192 K

- The whole enchilada as a PDF

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search